FaultHound: Value-Locality-Based Soft-Fault Tolerance

Nitin, Irith Pomeranz, and T. N. Vijaykumar

Purdue University

ISCA 2015
Soft errors

- CMOS scaling $\rightarrow$ higher susceptibility to soft faults
  - A major hurdle for voltage scaling
  - Memory can be ECC-protected
  - ECC for pipeline logic hard

- Nature of soft faults
  - Many faults are
    - masked (don’t matter) or
    - noisy (trigger exceptions)
  - Rest cause silent data corruptions (SDCs)
    - Main challenge and focus of this talk

SDCs due to soft faults are a growing problem
Previous schemes

• Redundancy-based [many papers]
  • Hardware or software
  • Replicate \textit{irrespective} of fault or not
  • Full redundancy - high energy/performance overhead
  • Partial redundancy - low SDC coverage

• Hint-based [many papers]
  • Faults often provide hints
    • e.g., branch mispredictions, TLB misses or value locality changes
  • Re-execution \textit{only} upon hints, not always \( \rightarrow \) lower overhead
  • Value-locality hints are a more direct indicator

\textbf{Hint-based schemes have lower overhead}
Value locality hints - previous work

• Perturbation-based Fault Screening (PBFS) [HPCA07]
  • Filters capture bit positions as changing or unchanging
  • A change in previously-unchanging bit $\rightarrow$ squash
  • Changing bits $\rightarrow$ no squash
  • True value changes lead to false positives (FPs)
    • Energy and performance overheads

• To cut FPs, PBFS filters use 1-bit sticky FSM
  • Catch first bit-change but no more until periodic clear
  • Low performance loss (1%) but low SDC coverage (30%)
  • Extending to 2-bit non-sticky FSM $\rightarrow$
    • better coverage but high performance loss (97%)

• Inherent tension between coverage and false positives

Fixing PBFS non-trivial: FPs and faults look similar
**FaultHound**

Even low FP rates $\rightarrow$ large performance and energy overheads
FaultHound contributions

1. **Observation:** PBFS’s PC indexing into table of filters
   - Same value in many entries
   - Each entry incurs false positive (FP) for same value change

   **Solution:** Use values themselves to index – *inverted index*
   - Clusters similar values in same filter, reinforces learning
   - Fewer FPs with only 16-32 entries

2. **Observation:** Delinquent bit positions cause repeated FPs

   **Solution:** Cut FPs further via a second-level filter

3. **Observation:** Most instructions depend on nearby instructions

   **Solution:** Replay instead of full squash to reduce FP penalty
   - Drastically cuts down FP penalty
FaultHound contributions

4. **Observation:** Replay doesn’t cover frontend $\rightarrow$ squash
   - Separate frontend faults and false positives
   - Rename faults $\rightarrow$ use of unusual but uncorrupted values
   - Unusual values $\rightarrow$ different-than-usual value neighborhoods
   **Solution:** Filter to identify such neighborhoods’ use

5. **Observation:** Replay doesn’t cover Load-Store Queue (LSQ)
   **Solution:** Check loads & stores against filters upon LSQ exit
   - Singleton re-execute load or store
Outline

• Introduction

➢ Contributions
  1. Value clustering via inverted index
  2. Suppress delinquent bit positions via a second-level filter
  3. Replay to avoid expensive squashes
  4. Cover rename faults
  5. Cover LSQ

• Methodology

• Results

• Conclusion
1. Clustering via inverted indexing

- Recall we have a table of filters
- Inverted index to place values in entries
  - Cluster similar values into an entry with most matches
1. Clustering via inverted indexing

- Recall we have a table of filters
- Inverted index to place values in entries
  - Cluster similar values into an entry with most matches

<table>
<thead>
<tr>
<th>New Value</th>
<th>Filter Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1</td>
<td>0 X X X X</td>
</tr>
<tr>
<td></td>
<td>0 X X 0</td>
</tr>
<tr>
<td>1 X 0 X</td>
<td>1 X 0 X</td>
</tr>
<tr>
<td></td>
<td>0 0 1 0</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
</tr>
</tbody>
</table>

Match
No fault
1. Clustering via inverted indexing

- Recall we have a table of filters
- Inverted index to place values in entries
  - Cluster similar values into an entry with most matches

<table>
<thead>
<tr>
<th>New Value</th>
<th>Filter Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1</td>
<td>0 X X X</td>
</tr>
<tr>
<td></td>
<td>0 X X 0</td>
</tr>
<tr>
<td></td>
<td>0 X 0 X</td>
</tr>
<tr>
<td></td>
<td>0 0 1 0</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
</tr>
</tbody>
</table>
1. Clustering via inverted indexing

• Recall we have a table of filters
• Inverted index to place values in entries
  • Cluster similar values into an entry with most matches

<table>
<thead>
<tr>
<th>New Value</th>
<th>Filter Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1</td>
<td>0 X X X X</td>
</tr>
<tr>
<td></td>
<td>0 X X 0</td>
</tr>
<tr>
<td></td>
<td>X X 0 X</td>
</tr>
<tr>
<td></td>
<td>0 0 1 0</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
</tr>
</tbody>
</table>

No match
Signal fault
1. Clustering via inverted indexing

- Recall we have a table of filters
- Inverted index to place values in entries
  - Cluster similar values into an entry with most matches

<table>
<thead>
<tr>
<th>New Value</th>
<th>Filter Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1</td>
<td>0 X X X</td>
</tr>
<tr>
<td>0 X X 0</td>
<td>X X 0 X</td>
</tr>
<tr>
<td>X X 0 X</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td></td>
</tr>
</tbody>
</table>

- Implemented as a Ternary CAM
- Only 16-32 entries

Inverted index clustering cuts false positives with just a few entries
2. Second-level filter to cut false positives

• Some delinquent bit positions cause repeated false positives
• Second-level filter to suppress such false alarms
  • Per-bit state machine
  • One global filter

Second-level filter suppresses repeated false false alarms
3. Replay instead of squash

1. Most values come from nearby preceding instructions
2. FaultHound checks every load and store
   → faulty value unlikely to propagate far
   → replay of a few instructions (6-8) sufficient
      • Instead of a full squash (100)
      • But existing replay needs modification
3. Replay instead of squash

1. Most values come from nearby preceding instructions
2. FaultHound checks every load and store
   → faulty value unlikely to propagate far
   → replay of a few instructions (6-8) sufficient
      • Instead of a full squash (e.g., 100 instructions)

• But existing replay needs modification – Delay Buffer

Replay drastically cuts down false-positive penalty
Other contributions

• Replay does not cover
  • Frontend (rename)
  • LSQ

• Details in the paper
Outline

• Introduction
• Contributions
  1. Value clustering via inverted index
  2. Suppress delinquent bit positions via a second-level filter
  3. Replay to avoid expensive squashes
  4. Cover rename faults
  5. Cover LSQ

➢ Methodology
• Results
• Conclusion
Methodology

• Full-system simulation
  • GEMS/Opal (out-of-order issue) + McPAT for energy
  • Fault framework to bin faults into masked, noisy, SDC
  • 15,000 faults in one run

• Fault Model
  • 15000 single-bit faults, randomly in register file, rename, LSQ
  • 72% register file, 20% rename table, 8% LSQ [McPAT]

• Workloads
  • 14 benchmarks – commercial (3), SPEC2006 (7), SPLASH (4)
SDC Coverage

FaultHound achieves 75% coverage, similar to PBFS-non-sticky
Due to 1st and 2nd level filters + covering LSQ & rename
False positives

FaultHound cuts down false positives over PBFS-non-sticky by >2x due to clustering + 2nd level filter.
FaultHound - 10%, PBFS-non-sticky – 97%, SRT iso - 13%
Due to fewer false positives and replay instead of squash
Core energy overhead

FaultHound - 25%, SRT-isol - 57%
Due to fewer false positives and replay instead of squash
Conclusion

• FaultHound – value locality based fault tolerance
  • Inverted index based clustering to reduce false positives (FP)
  • Second-level filter to further cut repeated FPs
  • Replay instead of full squash to reduce FP penalty
  • Separately cover rename and LSQ
    • Replay does not cover

• FaultHound – 75% SDC coverage
  • 10% and 25% performance and energy overheads

• PBFS (non-sticky) – 30% (75%) coverage
  • 1% (97%) performance overhead

• SRT-iso – 75% coverage
  • 13% and 57% performance and energy overheads

FaultHound with its coverage, performance and energy characteristics
An attractive option in future
FaultHound:
Value-Locality-Based Soft-Fault Tolerance

Nitin, Irith Pomeranz, and T. N. Vijaykumar