Efficient Soft Error Protection for Microprocessors

Sudhanva Gurumurthi

Students: Angshuman Parashar, Kristen Walcott, Blake Sutton, Taniya Siddiqua

University of Virginia

E-mail: gurumurthi@cs.virginia.edu
The Multicore Era

Future processors are expected to have more cores
RELIABLE HARDWARE
The Silicon Reliability Challenge

• **Soft Errors**
  – Random bit flips
  – Do not permanently damage the circuit
  – Latches and logic becoming more vulnerable

• **Hard Errors**
  – Affect the lifetime of the circuit
  – Can appear during fabrication or in the field
  – Examples: NBTI, Electromigration

• **Process Variation**
  – Affect circuit delay characteristics
Error Protection Techniques

• **Redundancy**
  – **Informational**
    • Parity, ECC
  – **Spatial**
    • Executing instructions in different hardware units
    • Spare structures
  – **Temporal**
    • Executing instructions multiple times in the same hardware unit

• **Dynamic Reliability Management**
  • Dynamic voltage/frequency scaling
Error Protection Tradeoffs

- Error protection entails overheads:
  - Performance degradation
  - Increased power consumption
  - Higher area
Error Protection Trade-offs

- Error protection entails overheads:
  - Performance degradation
  - Increased power consumption
  - Higher area

- **Goal:** Reduce the overheads while providing the required level of error protection
Redundant Multi-Threading (RMT)

Sphere of Replication

Input Replicator

Output Comparator

Rest of the System

Mukherjee et al., ISCA 2002
Redundant Multi-Threading (RMT)

Sphere of Replication

Input Replicator

Output Comparator

Rest of the System

Mukherjee et al., ISCA 2002
Redundant Multi-Threading (RMT)

Sphere of Replication

Input Replicator

Output Comparator

Rest of the System

Leading Thread +
Trailing Thread

Mukherjee et al., ISCA 2002
Redundant Multi-Threading (RMT)

Sphere of Replication

Input Replicator

Output Comparator

Rest of the System

Leading Thread + Trailing Thread
Redundant Multi-Threading (RMT)

Sphere of Replication

Input Replicator

Output Comparator

Rest of the System

Store Instructions

Leading Thread + Trailing Thread

Mukherjee et al., ISCA 2002
Talk Outline

• Partial Redundant Multi-Threading - SlicK
• Runtime AVF Prediction
• NBTI Recovery Boosting
• Conclusions
All vs. No Coverage

Only two performance points
RMT

Partial RMT

All vs. No Coverage
Only two performance points

Tunable Error Coverage
Multiple performance points
Partial RMT
[Parashar et al., ASPLOS 2006]

• Reduce the number of instructions in the redundant thread
  – “Knob”: Number of instructions removed
  – Need to preserve register dependences
  – Examples: Slipstream, DIE-IRB, ReStore
Partial RMT

[Parashar et al., ASPLOS 2006]

- Reduce the number of instructions in the redundant thread
  - “Knob”: Number of instructions removed
  - Need to preserve register dependences
  - Examples: Slipstream, DIE-IRB, ReStore
- **Slice Kill (SlicK)**
Partial RMT

[Parashar et al., ASPLOS 2006]

- Reduce the number of instructions in the redundant thread
  - “Knob”: Number of instructions removed
  - Need to preserve register dependences
  - Examples: Slipstream, DIE-IRB, ReStore
- **Slice Kill (SlicK)**
  - Execute at the granularity of dependence chains/Slices
Partial RMT

[Parashar et al., ASPLOS 2006]

• Reduce the number of instructions in the redundant thread
  – “Knob”: Number of instructions removed
  – Need to preserve register dependences
  – Examples: Slipstream, DIE-IRB, ReStore

• **Slice Kill (SlicK)**
  – Execute at the granularity of dependence chains/Slices
  – Both program dataflow and control flow exhibit predictable behavior
Partial RMT
[Parashar et al., ASPLOS 2006]

- Reduce the number of instructions in the redundant thread
  - "Knob": Number of instructions removed
  - Need to preserve register dependences
  - Examples: Slipstream, DIE-IRB, ReStore

- **Slice Kill (SlicK)**
  - Execute at the granularity of dependence chains/Slices
  - Both program dataflow and control flow exhibit predictable behavior

- Use high-confidence speculation as a substitute for redundant execution
The SlicK Mechanism

leading thread

trailing thread
The SlicK Mechanism

leading thread

backward slice

trailing thread
The SlicK Mechanism

leading thread

backward slice

trailing thread
The SlicK Policy

leading thread

trailing thread

Store Predictor
The SlicK Policy

leading thread

trailing thread

Store Predictor

prediction failed

prediction succeeded
The SlicK Policy

leading thread

trailing thread

prediction failed

prediction succeeded

prediction succeeded

Store Predictor
The SlicK Policy

leading thread

trailing thread

prediction failed

prediction succeeded

Dead

Store Predictor

prediction succeeded
The SlicK Policy

leading thread

trailing thread

prediction failed

prediction succeeded

Store Predictor
The SlicK Policy

Branches are Verified Too
Overview of SlicK Design

- Predictors Used:
  - Predictor Outputs: Prediction, No-Predict
  - Stores: Last-value predictor with saturating counters
  - Branches: Branch predictor + confidence estimator
Overview of SlicK Design

• Predictors Used:
  – Predictor Outputs: Prediction, No-Predict
  – Stores: Last-value predictor with saturating counters
  – Branches: Branch predictor + confidence estimator

• Slice extractor needs to provide a smooth flow of instructions through the pipeline
  – Slice Extraction Matrix (SliceEM)
Overview of SlicK Design

• Predictors Used:
  – Predictor Outputs: Prediction, No-Predict
  – Stores: Last-value predictor with saturating counters
  – Branches: Branch predictor + confidence estimator

• Slice extractor needs to provide a smooth flow of instructions through the pipeline
  – Slice Extraction Matrix (SliceEM)

• Evaluated for the RMT implementation of an SMT processor
  – Simultaneous and Redundant Threading (SRT)
    [Reinhardt and Mukherjee, ISCA 2000]
SlicK Performance Results

Benchmark

Normalized IPC

SRT  SlicK
SlicK Performance Results

10.2% improvement over SRT
50% reduction in IPC gap
SlicK Fault Coverage: The Common Case #1
SlicK Fault Coverage: The Common Case #1

**Fault-Free Execution**

*leading thread*

![Diagram showing fault-free execution with a leading thread and store predictor.]
SlicK Fault Coverage: The Common Case #1

Fault-Free Execution

leading thread

Store Predictor (warm)

1000

Match: Commit

S

1000

S

Erroneous Execution

leading thread

Store Predictor (warm)

1001

SEU

S

1000

MisMatch: Pull Trailing

trailing thread

S

1000
SlicK Fault Coverage: The Common Case #2

Fault-Free Execution

leading thread

Store Predictor (cold)

No Predict

Pull Trailing

trailing thread
SlicK Fault Coverage: The Common Case #2

Fault-Free Execution

- **Leading Thread**
  - Store Predictor (cold)
  - No Predict
  - 1000

- **Trailing Thread**
  - S
  - 1000

Erroneous Execution

- **Leading Thread**
  - SEU
  - Pull Trailing
  - 1001

- **Store Predictor (cold)**
  - No Predict

- **Trailing Thread**
  - S
  - 1000
SlicK Fault Coverage: The Uncommon Case

Fault-Free Execution

leading thread

Store Predictor (warm)

trailing thread

Mismatch: Pull Trailing

1000

1000

1001 (mispredict)
SlicK Fault Coverage: The Uncommon Case

Fault-Free Execution

leading thread

Store Predictor (warm) → S
1000

1001 (mispredict)

Mismatch: Pull Trailing

trailing thread

Erroneous Execution

leading thread

Store Predictor (warm) → S
1001

1001 (mispredict)

Match: Commit

Silent Data Corruption!

SEU
SlicK Fault Coverage: The Uncommon Case

Fault-Free Execution

leading thread

Store Predictor (warm) → S
1000

1001
(mispredict)

trailing thread

Erroneous Execution

leading thread

SEU

Match: Commit
Silent Data Corruption!

Store Predictor (warm) → S
1001
(mispredict)

The Store and its Backward Slice are UNGUARDED
Architectural Vulnerability Factor (AVF)

- The probability that a fault will result in an externally visible error
- Only a subset of the bits affect Architecturally Correct Execution (ACE bits)
- AVF = 0% for structures within Sphere of Replication in RMT
Architectural Vulnerability Factor (AVF)

- The probability that a fault will result in an externally visible error
- Only a subset of the bits affect Architecturally Correct Execution (ACE bits)
- AVF = 0% for structures within Sphere of Replication in RMT
- Unguarded instructions = ACE
Architectural Vulnerability Factor (AVF)

- The probability that a fault will result in an externally visible error
- Only a subset of the bits affect Architecturally Correct Execution (ACE bits)
- AVF = 0% for structures within Sphere of Replication in RMT
- Unguarded instructions = ACE
- AVF $\sim$ 0%-2% for RUU, ISQ, and LSQ
  - Single threaded AVFs $\sim$ 20%-30%

Mukherjee et al, MICRO 2003
Partial RMT
Partial RMT

Soft Error Measurement
Runtime AVF Prediction
Measuring Runtime AVF

[Walcott et al., ISCA 2007]

- Little’s Law AVF Estimate = \(\frac{(B_{ace})(L_{ace})}{\# \text{ Bits}}\)

  - \(B_{ace}\): average bandwidth of the ACE bits into the structure
  - \(L_{ace}\): average residence time of an ACE bit in the structure

- Direct measurement in hardware is difficult
Measuring Runtime AVF
[Walcott et al., ISCA 2007]

• Little’s Law AVF Estimate = \( \frac{(B_{\text{ace}})(L_{\text{ace}})}{\# \text{ Bits}} \)

\( B_{\text{ace}} \): average bandwidth of the ACE bits into the structure
\( L_{\text{ace}} \): average residence time of an ACE bit in the structure

• Direct measurement in hardware is difficult

• **Our Approach:** Calculate AVF from very few, easily measurable metrics
Experimental Setup

- SimpleScalar 3.0 with SRT model
- Structures for AVF Analysis
  - RUU, ISQ, LSQ
- 26 SPEC2000 Benchmarks
  - Simulate all 100-million instruction SimPoints
  - Checkpoint simulation state (160 μarch variables) and calculate AVF every 4 million instructions
Choosing the Right Metrics

- **IPC**
  - Easy to measure
  - Used to characterize program behavior

- Intuitively, High IPC could mean:
  - More ACE bits \((\sim B_{ace})\) => Higher AVF
  - Bits move faster through the pipeline \((\sim L_{ace})\) => Lower AVF
IPC vs. AVF

perlbmk

bzip2

art
Detecting Correlations

• Visual inspection is tedious
• Need a systematic way to identify the strongly correlated variables
Detecting Correlations

• Visual inspection is tedious
• Need a systematic way to identify the strongly correlated variables

• **Our Approach:** Use regression techniques
  – Chose 22 SPEC benchmarks for training
  – Remaining 4 used for testing the predictor
  – Used data from all the SimPoints
Linear Regression

• Iterative Process:
  – Include the single variable with the highest correlation
  – Consider each remaining variable one by one and compute regression of the 2-variable expression
Linear Regression

- **Iterative Process:**
  - Include the single variable with the highest correlation
  - Consider each remaining variable one by one and compute regression of the 2-variable expression
Predictor Testing

galgel - RUU AVF

Multi-Variable Linear Predictor

AVF

Measured

Predicted
An AVF-Aware Partial RMT Policy

- Estimate the AVF of all structures every 2 million cycles.
- Sum the individual AVF values
- If (Sum > threshold): enable RMT
- After 10 million cycles: disable RMT
IPC Variations for twolf

![Graph showing IPC variations for twolf with RMT Disabled and RMT Enabled.]
IPC Variations for \textit{twolf}

![Graph showing IPC variations with different thresholds for RMT-disabled and RMT-enabled scenarios.]

- **RMT Disabled**
  - Threshold = 20
- **Threshold = 28**
- **Threshold = 36**
  - RMT Enabled
Predicting AVF in RMT Mode
[Sutton et al., SELSE 2009]

• To determine when RMT can be disabled
• Developed predictors of single-threaded mode AVF using metrics measured in RMT mode
• Conducted regression analysis between single-threaded mode AVF and μarch metrics collected in RMT mode
RUU Predictor

- Linear regression
- Predictor with most number of common variables with single-threaded mode predictor.
Another Partial RMT Policy

- Toggle RMT as long as AVF of any structure is greater than threshold
- Compare to periodic toggling every 10 million cycles
Another Partial RMT Policy

- Toggle RMT as long as AVF of any structure is greater than threshold
- Compare to periodic toggling every 10 million cycles

AVF Threshold = 25
Another Partial RMT Policy

- Toggle RMT as long as AVF of any structure is greater than threshold
- Compare to periodic toggling every 10 million cycles

AVF Threshold = 25

AVF Threshold = 15
Quantized AVF (Q-AVF)  
[Biswas et al., SELSE 2009]

- Work with SPEARS Group, Intel
- **Goal:** Practical AVF predictor hardware
  - Fine-grained vulnerability tracking
  - Tracking vulnerability of groups of structures

**Average AVF:** Average of Q-AVF s over all quantas

**Q-AVF:** AVF of a bit over a short interval of time
Quantized AVF (Q-AVF)
[Biswas et al., SELSE 2009]

- Work with SPEARS Group, Intel
- **Goal:** Practical AVF predictor hardware
  - Fine-grained vulnerability tracking
  - Tracking vulnerability of groups of structures
- Use linear regression approach to predict Q-AVF
- Intel Core™-like ASIM performance model
- Benchmarks: Spec2000, Spec2006, TPC-C

**parser** - Store Buffer AVF x86 Pipeline

**Average AVF:** Average of Q-AVF over all quantas

**Q-AVF:** AVF of a bit over a short interval of time
Q-AVF Estimation Accuracy High
(7 or fewer parameters per structure)

<table>
<thead>
<tr>
<th>Structures</th>
<th>Mean Correlation across Benchmarks</th>
<th>Min Correlation across Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>IQ – Instruction Queue</td>
<td>0.87</td>
<td>0.81</td>
</tr>
<tr>
<td>MB – Memory Buffer</td>
<td>0.90</td>
<td>0.82</td>
</tr>
<tr>
<td>RS – Reservation Station</td>
<td>0.93</td>
<td>0.82</td>
</tr>
<tr>
<td>ROB – Reorder Buffer</td>
<td>0.95</td>
<td>0.86</td>
</tr>
<tr>
<td>STB – Store Buffer</td>
<td>0.98</td>
<td>0.94</td>
</tr>
<tr>
<td>LDB – Load Buffer</td>
<td>0.85</td>
<td>0.80</td>
</tr>
</tbody>
</table>
Aggregate Q-AVF Accuracy High
(8 total parameters)

- **Only 8 input parameters used for all aggregate blocks:**
  1. Stores Flushed before DTLB response (ST_Flush)
  2. STB Utilization (ST_Util)
  3. ROB Empty Cycles (ROB_Empty)
  4. ROB Utilization (ROB_Util)
  5. Branch Mis-predicts (Br_Miss)
  6. RSUtilization (RS_Util)
  7. IDQ Utilization (IDQ_Util)
  8. Total Front-End Instruction Killed Latency (FE_Kill)

<table>
<thead>
<tr>
<th>Aggregate Blocks</th>
<th>Mean Correlation across Benchmarks</th>
<th>Min Correlation across Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front End</td>
<td>0.86</td>
<td>0.80</td>
</tr>
<tr>
<td>Back End</td>
<td>0.90</td>
<td>0.81</td>
</tr>
<tr>
<td>Memory Order Buffer</td>
<td>0.93</td>
<td>0.92</td>
</tr>
</tbody>
</table>
NBTI Recovery Boosting
Negative Bias Temperature Instability

- Problem for the PMOS devices
- Increases $V_t$ and hence the device delay
Negative Bias Temperature Instability

- Problem for the PMOS devices
- Increases $V_t$ and hence the device delay
- **Stress phase:**
  - Negative bias ($V_{gs} = -V_{dd}$) at the gate of the PMOS
  - Leads to generation of interface traps
Negative Bias Temperature Instability

• Problem for the PMOS devices
• Increases $V_t$ and hence the device delay

• Stress phase:
  – Negative bias ($V_{gs}=-V_{dd}$) at the gate of the PMOS
  – Leads to generation of interface traps

• Recovery phase:
  – No bias ($V_{gs}=0$) at the gate of the PMOS
  – Eliminates some of the interface traps
Negative Bias Temperature Instability

- Problem for the PMOS devices
- Increases $V_t$ and hence the device delay
- **Stress phase:**
  - Negative bias ($V_{gs} = -V_{dd}$) at the gate of the PMOS
  - Leads to generation of interface traps
- **Recovery phase:**
  - No bias ($V_{gs} = 0$) at the gate of the PMOS
  - Eliminates some of the interface traps
- **Goal:** Enhance NBTI recovery for SRAM arrays
6T SRAM cell
Cell Holding a ‘0’
Cell Holding a ‘0’
Cell Holding a ’0’

WL

V_{dd}

BL

‘0’

BLB

Under Stress
Cell Holding a ‘1’

Under Stress

WL

BL

BLB

V_{dd}
Approaches to NBTI Mitigation

• Stress reduction techniques:
  – **Power Gating**: Disconnect Vdd/GND connections
  – **Facelift**: Temperature-based job-scheduling with Vdd and Vt control [Tiwari et al., MICRO’08]

• Recovery enhancement techniques:
  – **Penelope**: Balance the degradation of the two PMOS devices in the cell [Abella et al., MICRO’07]
Recovery Boosting
Recovery Boosting

Valid Data: CR = 1
Recovery Boosting

Valid Data: CR = 1
Recovery Boosting

Valid Data: CR = 1
Recovery Boosting

Valid Data: CR = 1
Recovery Boosting

Valid Data: CR = 1

Normal 6T cell
Recovery Boosting

Valid Data: CR = 0
Recovery Boosting

Valid Data: CR = 0

[Diagram of circuit with labels WL, BL, BLB, V_{dd}, and CR connected by lines and nodes]
Recovery Boosting

Valid Data: CR = 0
Recovery Boosting

Valid Data: CR = 0
Recovery Boosting

Valid Data: CR = 0

Both PMOS undergo recovery
Fine-Grained Recovery Boosting
Coarse-Grained Recovery Boosting
Example Design

• Issue Queue (ISQ)
  – 64-entry Non-collapsing ISQ with 4 read and 4 write ports [Folegnani and Gonzalez, ISCA’01]
  – Both CAM and RAM use modified bitcells
  – ISQ entries with invalid data put into recovery boost mode

• SPICE simulations (Cadence Spectre) + Architecture simulations (M5 and SPEC CPU2000 benchmarks) for 32nm process
Area overhead = 3%
Overhead is small since the cell is highly multiported
SPICE Results

Issue Queue Entry Power Consumption

- Normal Cell
- Modified Cell

Operations:
- read_match
- read_mismatch
- write_match
- write_mismatch
- hold_match
- hold_mismatch

Power = 102.3 nW

Recovery Boost
ISQ Entry State Time
Area Neutral wrt. Baseline – 2 Fewer ISQ entries
ISQ SNM Improvement
Initial $V_t=0.2 \, V$, Service Life=7 years

SNM Improvement wrt. baseline (%)

Balancing
Recovery Boosting

Benchmarks:
- gap
- gap
- bzip2
- parser
- perl
- eon
- vortex
- twolf
- galgel
- ammp
- mesa
- mgrid
- wupwise
- facerec
- art
- apsi
- average
Conclusions

• Error protection imposes performance, power, and area overheads

• Adapt error protection to the protection needs at runtime
  – Partial RMT (SlicK)
  – Runtime AVF Prediction

• Recovery boosting enhances NBTI recovery with little performance, power, or area overheads
Thank You

http://www.cs.virginia.edu/~gurumurthi