New challenges for designers of fault tolerant Embedded Systems based on future technologies

Carlos Arthur Lang Lisbôa          Luigi Carro

Instituto de Informática, Programa de Pós-Graduação em Computação
Universidade Federal do Rio Grande do Sul - Porto Alegre, RS, Brazil

IESS - Schloß Langenargen, Germany – September 15th, 2009
Outline

• **Introduction: concepts and definitions**
• Motivation: new challenges imposed by future technologies
• Radiation induced faults: the major challenges
• Existing mitigation techniques vs. the new scenario
• Desired properties of new radiation induced faults mitigation techniques
• Recent solutions working at different abstraction levels to deal with transient faults
• Conclusions
Concepts and Definitions

• Faults

• Errors

• Failures
Concepts and Definitions

• Duration of errors and faults
  - Permanent
  - Transient
  - Intermittent
Technology trends (1)

- Transistor size

Device size are decreasing

Nodes capacitances are decreasing

Source: ITRS 2004
Technology trends (2)

- Transistor Vth

![Graph showing the relationship between Technology Generation (μm) and Vcc or Vt.]
Single event upset

A transistor changes from OFF to ON state!
SEE and Technology trends (1)

- Consequences of C and V reduction

\[ \text{HIGH } C \ + \ \text{HIGH } V \rightarrow \text{HIGH } Q = C \cdot V \]
• Consequences of C and V reduction

LOW C + LOW V \rightarrow LOW Q = C.V
Concepts and Definitions

- Radiation Induced Faults
  - Single Event Effects – SEEs
  - Single Event Transients – SETs
  - Single Event Upsets – SEUs
  - Soft Error - SE
  - Multiple Bit Upsets – MBUs

- Soft Error Rate - SER
The Soft Error Problem

Single Event Upset (SEU)

```
CLK  D  Q  0
```
The Soft Error Problem

Transient Fault  Soft Error
Concepts and Definitions

• Masking of faults and errors
  o Logical
  o Latching window
  o Electrical
  o Architectural
  o Software
Example of Fault Masking in Microprocessors

- Logical: faulty value does not affect logical operation of the circuit

[Blome et al, CASES, 2006]
Example of Fault Masking in Microprocessors

- Latching-Window: the fault pulse does not reach a state element within the latching window

[Blome et al, CASES, 2006]
Example of Fault Masking in Microprocessors

- Electrical: the fault pulse is electrically attenuated by subsequent gates in the circuit

[Blome et al, CASES, 2006]
Example of Fault Masking in Microprocessors

- Architectural/Software: incorrect state is written before it is read

```plaintext
mov r2, 4
mov r5, 8
add r6, r2, r5
```

Register File

<table>
<thead>
<tr>
<th>decoder</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

[Blome et al, CASES, 2006]
Example of Fault Masking in Microprocessors

- Architectural/Software: incorrect state is written before it is read

```
mov r2, 4
mov r5, 8
add r6, r2, r5
```

![Register File]

[Blome et al, CASES, 2006]
Example of Fault Masking in Microprocessors

- Architectural/Software: incorrect state is written before it is read

```assembly
mov r2, 4
mov r5, 8
add r6, r2, r5
```

Register File

```
<table>
<thead>
<tr>
<th>decoder</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>9</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

[Blome et al, CASES, 2006]
Example of Fault Masking in Microprocessors

• Architectural/Software: incorrect state is written before it is read

```
mov r2, 4
mov r5, 8
add r6, r2, r5
```

Register File

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

[Blome et al, CASES, 2006]
Outline

• Introduction: concepts and definitions

• **Motivation: new challenges imposed by future technologies**

• Radiation induced faults: the major challenges

• Existing mitigation techniques vs. the new scenario

• Desired properties of new radiation induced faults mitigation techniques

• Recent solutions working at different abstraction levels to deal with transient faults

• Conclusions
Motivation: Future Technologies

• The good news:
  o Smaller devices  
    → Denser circuits, less area
  o Faster devices  
    → Higher performance
  o Less power consumption  
    → Longer battery life (portable systems)
Motivation: Future Technologies

• The bad news:

  o Higher defect rates
    → Lower yield

  o Higher sensitivity to radiation
    → Increased SER: combinational logic
    → Multiple simultaneous faults
    → Long duration transients
Outline

• Introduction: concepts and definitions
• Motivation: new challenges imposed by future technologies
• Radiation induced faults: the major challenges
• Existing mitigation techniques vs. the new scenario
• Desired properties of new radiation induced faults mitigation techniques
• Recent solutions working at different abstraction levels to deal with transient faults
• Conclusions
Major Challenges

• Long Duration Transients (LDTs)
  Different paces in transient widths vs. device speed scaling will lead to transient pulses lasting longer than cycle times of circuits. Temporal redundancy techniques will not cope.

• Multiple Simultaneous Faults
  Smaller distances between devices will allow a single particle to affect more than one device. The single fault model will fail.
Transient width studies

DODD, 2004

FERLET-CAVROIS, 2006
### Propagation delay (*) vs. Technologies

<table>
<thead>
<tr>
<th>Technology (nm)</th>
<th>180</th>
<th>130</th>
<th>90</th>
<th>32</th>
<th>180/32</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>10-inverter chain</strong></td>
<td>508.4</td>
<td>157.8</td>
<td>120.2</td>
<td>79.6</td>
<td><strong>6.39</strong></td>
</tr>
</tbody>
</table>

(*) simulated using parameters from PTM web site and HSPICE tool

Luigi Carro  
IESS - Schloß Langenargen, Germany, September 15th, 2009
Transient widths vs. Propagation delays

Cycle time and transient width scaling across technologies

Transient width scaling: max. 1.37 x

(*) 180, 130, and 100nm from [DODD, 2004], 70 nm from [Ferlet-Cavrois 2006]
Single event, multiple effects

[Rossi 2005 *]

[*] Multiple Transient Faults in Logic: An Issue for Next Generation ICs ?, Daniele Rossi et al, DFT 2005
Outline

• Introduction: concepts and definitions
• Motivation: new challenges imposed by future technologies
• Radiation induced faults: the major challenges
• **Existing mitigation techniques vs. the new scenario**
• Desired properties of new radiation induced faults mitigation techniques
• Recent solutions working at different abstraction levels to deal with transient faults
• Conclusions
LDT Effects on Temporal Redundancy

• Time Redundancy [Anghel et al, 2000]
LDT Effects on Temporal Redundancy

- **Time Redundancy** [Anghel et al, 2000]

Increase delay?  \[ \Rightarrow \text{Higher performance penalty!!!} \]
LDT Effects on Space Redundancy

- Space Redundancy [Nieuwland et al, 2006]
LDT Effects on Space Redundancy

- Space Redundancy [Nieuwland et al, 2006]

Can not cope with long duration transients !!!
LDT Effects on Space Redundancy

- DMR can cope with LDTs affecting one of the modules
- allows detection only, requires recomputation
- area and power overheads above 100% (too much for ES)
- weak point: comparator
LDT Effects on Space Redundancy

- TMR can cope with LDTs affecting one of the modules
- allows detection and correction
- area and power overheads above 200% (too much for ES)

weak point: voter
Multiple simultaneous errors [Sorin 2009 *]

- It is an interesting open problem.
- If forecasts of greatly increased fault rates come to pass, error detection schemes targeting single error scenarios may be insufficient.
- Most of current schemes assume a single error scenario.
- Some existing schemes may do well, but there are no results demonstrating that capability.

Multiple Effects vs. Space Redundancy

- DMR: what if a single particle affects two modules?

- different output bits affected \((O_{1i}, O_{2j})\) → OK

- same output bit affected \((O_{1k}, O_{2k})\) → PROBLEM! Comparator will not detect error
Multiple Effects vs. Space Redundancy

- TMR: what if a single particle affects two modules?
- different output bits affected \((O_{1i}, O_{2j})\) → no majority!
- same output bit affected \((O_{1k}, O_{2k})\)
  → EVEN WORSE → Voter will select erroneous output!
Outline

• Introduction: concepts and definitions
• Motivation: new challenges imposed by future technologies
• Radiation induced faults: the major challenges
• Existing mitigation techniques vs. the new scenario
• Desired properties of new radiation induced faults mitigation techniques
• Recent solutions working at different abstraction levels to deal with transient faults
• Conclusions
Analysis

• Currently known mitigation techniques based on temporal redundancy can not cope with LDTs.

• Space redundancy based mitigations techniques:
  - able to cope with LDTs;
  - may fail when subject to multiple faults;
  - impose very high area and power overheads;
  - not suited for the Embedded Systems arena.

• The development of new low cost techniques to face those new challenges is mandatory.
Desired properties of new approaches

- Tolerance to LDTs and multiple simultaneous faults.
- Error detection area overhead $\ll$ DMR
- Error correction area overhead $\ll$ TMR
- Low performance overhead
- Additional concern for Embedded Systems: low power consumption
Suggested approach

Work at higher abstraction levels with low cost

“Computer users do not notice if a transistor fails or a bit of SRAM is flipped by a cosmic ray; they notice when their programs crash” [Sorin, 2009]
Outline

• Introduction: concepts and definitions
• Motivation: new challenges imposed by future technologies
• Radiation induced faults: the major challenges
• Existing mitigation techniques vs. the new scenario
• Desired properties of new radiation induced faults mitigation techniques
• **Recent solutions working at different abstraction levels to deal with transient faults**
• Conclusions
Recently proposed solutions (1 of 6)

Working at circuit level with low cost to cope with increased SER in combinational logic

- System Level
- Algorithm Level
- Architecture Level
- Circuit Level
- Component Level
- Technology Level

Combinational Hamming
SER evolution\(^*\)

SER Trend: Latches & Chip impact

SER Trend: Full Chip

Source: Intel Barcelona
Combinational Hamming

Conventional Hamming applications:
- data storage and communications hardening
- number of inputs = number of outputs

Combinational logic: number of inputs ≠ number of outputs
Combinational Hamming

Hamming codeword for 4-output circuits

\[
\begin{array}{ccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
k_1 & k_2 & s_3 & k_3 & s_2 & s_1 & s_0 & P \\
\end{array}
\]

\[
k_1 = s_3 \oplus s_2 \oplus s_0 \\
k_2 = s_3 \oplus s_1 \oplus s_0 \\
k_3 = s_2 \oplus s_1 \oplus s_0 \\
P = k_1 \oplus k_2 \oplus s_3 \oplus k_3 \oplus s_2 \oplus s_1 \oplus s_0
\]
Combinational Hamming

Ripple carry adder: 7 inputs and 4 outputs
## Combinational Hamming: Experiments

### Sample circuits: adders and multipliers

<table>
<thead>
<tr>
<th>ID</th>
<th>I</th>
<th>O</th>
<th>Area ($\mu m^2$)</th>
<th>Power (mW)</th>
<th>Delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4+4</td>
<td>8</td>
<td>5</td>
<td>263.758</td>
<td>0.334</td>
<td>0.780</td>
</tr>
<tr>
<td>5+5</td>
<td>10</td>
<td>6</td>
<td>445.549</td>
<td>1.165</td>
<td>1.320</td>
</tr>
<tr>
<td>6+6</td>
<td>12</td>
<td>7</td>
<td>493.513</td>
<td>3.572</td>
<td>1.670</td>
</tr>
<tr>
<td>7+7</td>
<td>14</td>
<td>8</td>
<td>575.765</td>
<td>4.168</td>
<td>1.482</td>
</tr>
<tr>
<td>4+4+cin</td>
<td>9</td>
<td>5</td>
<td>296.758</td>
<td>0.394</td>
<td>0.830</td>
</tr>
<tr>
<td>5+5+cin</td>
<td>11</td>
<td>6</td>
<td>487.286</td>
<td>1.579</td>
<td>1.520</td>
</tr>
<tr>
<td>6+6+cin</td>
<td>13</td>
<td>7</td>
<td>590.279</td>
<td>3.712</td>
<td>1.130</td>
</tr>
<tr>
<td>4×4</td>
<td>8</td>
<td>8</td>
<td>2,993.088</td>
<td>8.357</td>
<td>2.940</td>
</tr>
<tr>
<td>5×5</td>
<td>10</td>
<td>10</td>
<td>6,993.088</td>
<td>8.357</td>
<td>2.940</td>
</tr>
<tr>
<td>6×6</td>
<td>12</td>
<td>12</td>
<td>27,865.910</td>
<td>29.278</td>
<td>5.600</td>
</tr>
<tr>
<td>7×7</td>
<td>14</td>
<td>14</td>
<td>121,649.969</td>
<td>112.609</td>
<td>13.250</td>
</tr>
</tbody>
</table>
## Combinational Hamming: Results

### Areas (µm²)

<table>
<thead>
<tr>
<th>ID</th>
<th>Standard</th>
<th>Hamming</th>
<th>Hamming overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>4+4</td>
<td>263.758</td>
<td>498.449</td>
<td>88.980%</td>
</tr>
<tr>
<td>5+5</td>
<td>445.549</td>
<td>924.943</td>
<td>107.596%</td>
</tr>
<tr>
<td>6+6</td>
<td>493.513</td>
<td>1,207.267</td>
<td>144.627%</td>
</tr>
<tr>
<td>7+7</td>
<td>575.765</td>
<td>1,408.478</td>
<td>144.627%</td>
</tr>
<tr>
<td>4+4+cin</td>
<td>296.758</td>
<td>516.449</td>
<td>74.030%</td>
</tr>
<tr>
<td>5+5+cin</td>
<td>487.286</td>
<td>938.179</td>
<td>92.532%</td>
</tr>
<tr>
<td>6+6+Cin</td>
<td>590.279</td>
<td>1,417.765</td>
<td>140.186%</td>
</tr>
<tr>
<td>4×4</td>
<td>2,993.088</td>
<td>3,796.460</td>
<td>26.841%</td>
</tr>
<tr>
<td>5×5</td>
<td>6,993.088</td>
<td>11,810.657</td>
<td>68.890%</td>
</tr>
<tr>
<td>6×6</td>
<td>27,865.910</td>
<td>48,609.331</td>
<td>74.440%</td>
</tr>
<tr>
<td>7×7</td>
<td>121,649.969</td>
<td>176,320.018</td>
<td>44.940%</td>
</tr>
<tr>
<td>Mean</td>
<td>14,786.815</td>
<td>22,495.272</td>
<td>91.608%</td>
</tr>
</tbody>
</table>
## Combinational Hamming: Results

### Power (mW)

<table>
<thead>
<tr>
<th>ID</th>
<th>Standard</th>
<th>Hamming</th>
<th>Hamming overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>4+4</td>
<td>0.334</td>
<td>0.697</td>
<td>108.692%</td>
</tr>
<tr>
<td>5+5</td>
<td>1.165</td>
<td>1.598</td>
<td>37.246%</td>
</tr>
<tr>
<td>6+6</td>
<td>3.572</td>
<td>6.990</td>
<td>95.658%</td>
</tr>
<tr>
<td>7+7</td>
<td>4.168</td>
<td>8.155</td>
<td>95.658%</td>
</tr>
<tr>
<td>4+4+cin</td>
<td>0.394</td>
<td>0.807</td>
<td>104.831%</td>
</tr>
<tr>
<td>5+5+cin</td>
<td>1.579</td>
<td>1.911</td>
<td>21.006%</td>
</tr>
<tr>
<td>6+6+Cin</td>
<td>3.712</td>
<td>7.812</td>
<td>110.427%</td>
</tr>
<tr>
<td>4×4</td>
<td>8.357</td>
<td>11.989</td>
<td>43.472%</td>
</tr>
<tr>
<td>5×5</td>
<td>8.357</td>
<td>11.989</td>
<td>43.472%</td>
</tr>
<tr>
<td>6×6</td>
<td>29.278</td>
<td>41.365</td>
<td>41.285%</td>
</tr>
<tr>
<td>7×7</td>
<td>112.609</td>
<td>97.835</td>
<td>87.120%</td>
</tr>
<tr>
<td>Mean</td>
<td>15.775</td>
<td>17.377</td>
<td>71.715%</td>
</tr>
</tbody>
</table>
## Combinational Hamming: Results

### Propagation Delays (ns)

<table>
<thead>
<tr>
<th>ID</th>
<th>Standard</th>
<th>Hamming</th>
<th>Hamming overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>4+4</td>
<td>0.780</td>
<td>1.120</td>
<td>43.590%</td>
</tr>
<tr>
<td>5+5</td>
<td>1.320</td>
<td>1.760</td>
<td>33.333%</td>
</tr>
<tr>
<td>6+6</td>
<td>1.670</td>
<td>2.170</td>
<td>29.940%</td>
</tr>
<tr>
<td>7+7</td>
<td>1.482</td>
<td>2.170</td>
<td>46.457%</td>
</tr>
<tr>
<td>4+4 +cin</td>
<td>0.830</td>
<td>1.200</td>
<td>44.578%</td>
</tr>
<tr>
<td>5+5 +cin</td>
<td>1.520</td>
<td>1.870</td>
<td>23.026%</td>
</tr>
<tr>
<td>6+6+ Cin</td>
<td>1.130</td>
<td>1.700</td>
<td>50.442%</td>
</tr>
<tr>
<td>4×4</td>
<td>2.940</td>
<td>3.690</td>
<td>25.510%</td>
</tr>
<tr>
<td>5×5</td>
<td>2.940</td>
<td>3.690</td>
<td>25.510%</td>
</tr>
<tr>
<td>6×6</td>
<td>5.600</td>
<td>6.900</td>
<td>23.214%</td>
</tr>
<tr>
<td>7×7</td>
<td>13.250</td>
<td>14.180</td>
<td>7.019%</td>
</tr>
<tr>
<td>Mean</td>
<td>3.042</td>
<td>3.677</td>
<td>32.056%</td>
</tr>
</tbody>
</table>
## Combinational Hamming vs. TMR

### Areas ($\mu m^2$)

<table>
<thead>
<tr>
<th>ID</th>
<th>TMR</th>
<th>Hamming</th>
<th>Reduction over TMR</th>
</tr>
</thead>
<tbody>
<tr>
<td>4+4</td>
<td>952.474</td>
<td>498.449</td>
<td>47.668%</td>
</tr>
<tr>
<td>5+5</td>
<td>1,530.087</td>
<td>924.943</td>
<td>39.550%</td>
</tr>
<tr>
<td>6+6</td>
<td>1,706.219</td>
<td>1,207.267</td>
<td>29.243%</td>
</tr>
<tr>
<td>7+7</td>
<td>1,985.216</td>
<td>1,408.478</td>
<td>29.052%</td>
</tr>
<tr>
<td>4+4+cin</td>
<td>1,051.474</td>
<td>516.449</td>
<td>50.883%</td>
</tr>
<tr>
<td>5+5+cin</td>
<td>1,655.298</td>
<td>938.179</td>
<td>43.323%</td>
</tr>
<tr>
<td>6+6+Cin</td>
<td>1,996.517</td>
<td>1,417.765</td>
<td>28.988%</td>
</tr>
<tr>
<td>4×4</td>
<td>9,237.184</td>
<td>3,796.460</td>
<td>58.900%</td>
</tr>
<tr>
<td>5×5</td>
<td>21,301.664</td>
<td>11,810.657</td>
<td>44.555%</td>
</tr>
<tr>
<td>6×6</td>
<td>83,984.610</td>
<td>48,609.331</td>
<td>42.121%</td>
</tr>
<tr>
<td>7×7</td>
<td>365,401.266</td>
<td>176,320.018</td>
<td>51.746%</td>
</tr>
<tr>
<td>Mean</td>
<td>44,618.364</td>
<td>22,495.272</td>
<td>42.366%</td>
</tr>
</tbody>
</table>
## Combinational Hamming vs. TMR

### Power (mW)

<table>
<thead>
<tr>
<th>ID</th>
<th>TMR</th>
<th>Hamming</th>
<th>Reduction over TMR</th>
</tr>
</thead>
<tbody>
<tr>
<td>4+4</td>
<td>1.103</td>
<td>0.697</td>
<td>36.788%</td>
</tr>
<tr>
<td>5+5</td>
<td>3.615</td>
<td>1.598</td>
<td>55.781%</td>
</tr>
<tr>
<td>6+6</td>
<td>10.858</td>
<td>6.990</td>
<td>35.628%</td>
</tr>
<tr>
<td>7+7</td>
<td>12.665</td>
<td>8.155</td>
<td>35.611%</td>
</tr>
<tr>
<td>4+4+cin</td>
<td>1.283</td>
<td>0.807</td>
<td>37.083%</td>
</tr>
<tr>
<td>5+5+cin</td>
<td>4.858</td>
<td>1.911</td>
<td>60.668%</td>
</tr>
<tr>
<td>6+6+Cin</td>
<td>11.278</td>
<td>7.812</td>
<td>30.735%</td>
</tr>
<tr>
<td>4×4</td>
<td>25.231</td>
<td>11.989</td>
<td>52.482%</td>
</tr>
<tr>
<td>5×5</td>
<td>25.271</td>
<td>11.989</td>
<td>52.557%</td>
</tr>
<tr>
<td>6×6</td>
<td>88.075</td>
<td>41.365</td>
<td>53.034%</td>
</tr>
<tr>
<td>7×7</td>
<td>338.110</td>
<td>97.835</td>
<td>71.064%</td>
</tr>
<tr>
<td>Mean</td>
<td>47.486</td>
<td>17.377</td>
<td>47.403%</td>
</tr>
</tbody>
</table>
## Combinational Hamming vs. TMR

### Propagation Delays (ns)

<table>
<thead>
<tr>
<th>ID</th>
<th>TMR</th>
<th>Hamming</th>
<th>Overhead over TMR</th>
</tr>
</thead>
<tbody>
<tr>
<td>4+4</td>
<td>1.090</td>
<td>1.120</td>
<td>2.752%</td>
</tr>
<tr>
<td>5+5</td>
<td>1.630</td>
<td>1.760</td>
<td>7.975%</td>
</tr>
<tr>
<td>6+6</td>
<td>1.980</td>
<td>2.170</td>
<td>9.596%</td>
</tr>
<tr>
<td>7+7</td>
<td>1.792</td>
<td>2.170</td>
<td>21.116%</td>
</tr>
<tr>
<td>4+4+cin</td>
<td>1.140</td>
<td>1.200</td>
<td>5.263%</td>
</tr>
<tr>
<td>5+5+cin</td>
<td>1.830</td>
<td>1.870</td>
<td>2.186%</td>
</tr>
<tr>
<td>6+6+Cin</td>
<td>1.440</td>
<td>1.700</td>
<td>18.056%</td>
</tr>
<tr>
<td>4×4</td>
<td>3.250</td>
<td>3.690</td>
<td>13.538%</td>
</tr>
<tr>
<td>5×5</td>
<td>3.250</td>
<td>3.690</td>
<td>13.538%</td>
</tr>
<tr>
<td>6×6</td>
<td>5.910</td>
<td>6.900</td>
<td>16.751%</td>
</tr>
<tr>
<td>7×7</td>
<td>13.560</td>
<td>14.180</td>
<td>4.572%</td>
</tr>
<tr>
<td>Mean</td>
<td>3.352</td>
<td>3.677</td>
<td>9.705%</td>
</tr>
</tbody>
</table>
Recently proposed solutions (2 of 6)

Working at algorithm level with low cost error detection for matrix multiplication algorithm

Matrix Multiplication Hardening

System Level
Algorithm Level
Architecture Level
Circuit Level
Component Level
Technology Level
Fault-Tolerant Matrix Multiplication

- MxM is a widely used algorithm:
  - signal and image processing,
  - weather prediction,
  - finite element analysis,
  - control systems, etc.
- Error correction $\leftrightarrow$ System performance
- Computational cost: $O(n^3)$
Alternative approaches

- Duplication With Comparison (DWC)
  - Detection only, > 100% overhead

- Triple Modular Redundancy (TMR)
  - Correction, > 200% overhead

- Freivalds, 1979
  - Detection only, probabilistic, overhead < 100%

- Subject technique (Lisboa, ETS 2007)
  - Detection only, deterministic, overhead <= 100%
Freivalds’ technique [*]

A_{11} \ldots A_{1n} \times B_{11} \ldots B_{1n} \Rightarrow C_{11} \ldots C_{1n} \times r_1 \Rightarrow C_{r_1}

A_{n1} \ldots A_{nn} \times B_{n1} \ldots B_{nn} \Rightarrow C_{n1} \ldots C_{nn} \times r_n \Rightarrow C_{r_n}

Vector r: random 0’s and 1’s

Freivalds’ technique

\[
\begin{array}{cccc}
A_{11} & \ldots & A_{1n} \\
\ldots & \ldots & \ldots \\
A_{n1} & \ldots & A_{nn}
\end{array}
\times
\begin{array}{cccc}
B_{11} & \ldots & B_{1n} \\
\ldots & \ldots & \ldots \\
B_{n1} & \ldots & B_{nn}
\end{array}
\Rightarrow
\begin{array}{cccc}
C_{11} & \ldots & C_{1n} \\
\ldots & \ldots & \ldots \\
C_{n1} & \ldots & C_{nn}
\end{array}
\times
\begin{array}{c}
r_1 \\
\ldots \\
r_n
\end{array}
\Rightarrow
\begin{array}{c}
Cr_1 \\
\ldots \\
Cr_n
\end{array}
\]

Vector \( r \): random 0’s and 1’s

\[
\begin{array}{cccc}
A_{11} & \ldots & A_{1n} \\
\ldots & \ldots & \ldots \\
A_{n1} & \ldots & A_{nn}
\end{array}
\times
\begin{array}{c}
r_1 \\
\ldots \\
r_n
\end{array}
\Rightarrow
\begin{array}{c}
Ar_1 \\
\ldots \\
Ar_n
\end{array}
\]

\[
\begin{array}{cccc}
B_{11} & \ldots & B_{1n} \\
\ldots & \ldots & \ldots \\
B_{n1} & \ldots & B_{nn}
\end{array}
\times
\begin{array}{c}
B_{r_1} \\
\ldots \\
B_{r_n}
\end{array}
\Rightarrow
\begin{array}{c}
ABr_1 \\
\ldots \\
ABr_n
\end{array}
\]
Freivalds’ technique

If $C_r = ABr$, OK, otherwise, ERROR
Basic subject technique [*]

- The main difference w. r. t. the Freivalds’ technique is that here the $r$ Vector has only 1’s.

- This means that to calculate $A_r$ and $C_r$ only additions are needed, no multiplications.

- The computational cost of verification is thereby significantly decreased.

Basic subject technique

\[ A_{11} \ldots A_{1n} \]
\[ \ldots \ldots \ldots \]
\[ A_{n1} \ldots A_{nn} \]

\[ B_{11} \ldots B_{1n} \]
\[ \ldots \ldots \ldots \]
\[ B_{n1} \ldots B_{nn} \]

\[ C_{11} \ldots C_{1n} \]
\[ \ldots \ldots \ldots \]
\[ C_{n1} \ldots C_{nn} \]

\[ Cr_{1} = \sum_{k=1}^{n} C_{ik}, \]

\[ k=1\ldots n \]

\[ A_{11} \ldots A_{1n} \]
\[ \ldots \ldots \ldots \]
\[ A_{n1} \ldots A_{nn} \]

\[ Ar_{i} = \sum_{k=1}^{n} A_{ik}, \]

\[ k=1\ldots n \]

\[ B_{11} \ldots B_{1n} \]
\[ \ldots \ldots \ldots \]
\[ B_{n1} \ldots B_{nn} \]

\[ Ar_{1} \]
\[ \ldots \]
\[ Ar_{n} \]

\[ A_{11} \ldots A_{1n} \]
\[ \ldots \ldots \ldots \]
\[ A_{n1} \ldots A_{nn} \]

\[ B_{11} \ldots B_{1n} \]
\[ \ldots \ldots \ldots \]
\[ B_{n1} \ldots B_{nn} \]

\[ Ar_{1} \]
\[ \ldots \]
\[ Ar_{n} \]

\[ A_{11} \ldots A_{1n} \]
\[ \ldots \ldots \ldots \]
\[ A_{n1} \ldots A_{nn} \]

\[ B_{11} \ldots B_{1n} \]
\[ \ldots \ldots \ldots \]
\[ B_{n1} \ldots B_{nn} \]

\[ Ar_{1} \]
\[ \ldots \]
\[ Ar_{n} \]

\[ A_{11} \ldots A_{1n} \]
\[ \ldots \ldots \ldots \]
\[ A_{n1} \ldots A_{nn} \]

\[ B_{11} \ldots B_{1n} \]
\[ \ldots \ldots \ldots \]
\[ B_{n1} \ldots B_{nn} \]

\[ Ar_{1} \]
\[ \ldots \]
\[ Ar_{n} \]

\[ A_{11} \ldots A_{1n} \]
\[ \ldots \ldots \ldots \]
\[ A_{n1} \ldots A_{nn} \]

\[ B_{11} \ldots B_{1n} \]
\[ \ldots \ldots \ldots \]
\[ B_{n1} \ldots B_{nn} \]

\[ Ar_{1} \]
\[ \ldots \]
\[ Ar_{n} \]

\[ A_{11} \ldots A_{1n} \]
\[ \ldots \ldots \ldots \]
\[ A_{n1} \ldots A_{nn} \]

\[ B_{11} \ldots B_{1n} \]
\[ \ldots \ldots \ldots \]
\[ B_{n1} \ldots B_{nn} \]

\[ Ar_{1} \]
\[ \ldots \]
\[ Ar_{n} \]

\[ A_{11} \ldots A_{1n} \]
\[ \ldots \ldots \ldots \]
\[ A_{n1} \ldots A_{nn} \]

\[ B_{11} \ldots B_{1n} \]
\[ \ldots \ldots \ldots \]
\[ B_{n1} \ldots B_{nn} \]

\[ Ar_{1} \]
\[ \ldots \]
\[ Ar_{n} \]

If $Cr = ABr$, OK, otherwise, ERROR
Extended Subject Technique [*]

- compute vectors $B_r$ and $B_r^T$ (only sums)

Extended Subject Technique

- compute vectors $Br$ and $Br^T$ (only sums)
- compute vectors $ABr = A \times Br$ and $ABr^T = A \times Br^T$
Extended Subject Technique

- compute vectors $Br$ and $Br^T$ (only sums)
- compute vectors $A Br = A \times Br$ and $A Br^T = A \times Br^T$
- compute vectors $Cr$ and $Cr^T$ (only sums)

\[
\begin{array}{cccc}
C_{11} & C_{12} & \cdots & C_{1n} \\
C_{21} & C_{22} & \cdots & C_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
C_{n1} & C_{n2} & \cdots & C_{nn}
\end{array}
\]

\[
\begin{array}{c}
\sum \\
\uparrow
\end{array}
\Rightarrow
\begin{array}{c}
\sum \\
\uparrow
\end{array}
\]

\[
\begin{array}{c}
Cr_1 \\
\cdots \\
Cr_n
\end{array}
\]
Extended Subject Technique

• Verification:
  • If $A Br = Cr$ AND $A Br^T = Cr^T$, then NO ERROR

• Otherwise:
Extended Subject Technique - Example

\[
\begin{align*}
C &= \begin{bmatrix}
-2082 & -3582 & 11793 \\
2160 & -61 & 2565 \\
2280 & 512 & -2565
\end{bmatrix} \\
Cr &= \begin{bmatrix}
6129 \\
15744 \\
2937
\end{bmatrix} \\
ABr &= \begin{bmatrix}
6129 \\
9637 \\
2937
\end{bmatrix}
\end{align*}
\][
## Results: Verification Cost

**Total Verification Cost (# of add equivalent operations)**

<table>
<thead>
<tr>
<th>$n$</th>
<th>Multiplication</th>
<th>Freivalds</th>
<th>Subject</th>
<th>Extended</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>36</td>
<td>58</td>
<td>26</td>
<td>52</td>
</tr>
<tr>
<td>4</td>
<td>304</td>
<td>244</td>
<td>116</td>
<td>232</td>
</tr>
<tr>
<td>8</td>
<td>2,496</td>
<td>1,000</td>
<td>488</td>
<td>976</td>
</tr>
<tr>
<td>16</td>
<td>20,224</td>
<td>4,048</td>
<td>2,000</td>
<td>4,000</td>
</tr>
<tr>
<td>32</td>
<td>162,816</td>
<td>16,288</td>
<td>8,096</td>
<td>16,192</td>
</tr>
<tr>
<td>64</td>
<td>1,306,624</td>
<td>65,344</td>
<td>32,576</td>
<td>65,152</td>
</tr>
</tbody>
</table>
## Results: Recomputation Cost

Subject (whole matrix) vs. Extended (single element)

<table>
<thead>
<tr>
<th>n</th>
<th>Subject</th>
<th>%</th>
<th>Extended</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>36</td>
<td>100</td>
<td>9</td>
<td>25.0</td>
</tr>
<tr>
<td>4</td>
<td>304</td>
<td>100</td>
<td>19</td>
<td>6.25</td>
</tr>
<tr>
<td>8</td>
<td>2,496</td>
<td>100</td>
<td>39</td>
<td>1.56</td>
</tr>
<tr>
<td>16</td>
<td>20,224</td>
<td>100</td>
<td>79</td>
<td>0.39</td>
</tr>
<tr>
<td>32</td>
<td>162,816</td>
<td>100</td>
<td>159</td>
<td>0.10</td>
</tr>
<tr>
<td>64</td>
<td>1,306,624</td>
<td>100</td>
<td>319</td>
<td>0.02</td>
</tr>
</tbody>
</table>
Minimizing the recomputation time

\[ C = \begin{bmatrix} -2082 & -3582 & 11793 \\ 2160 & -61 & 3122 \\ 2280 & 5122 & -2565 \end{bmatrix} \]

\[ Cr^T = \begin{bmatrix} 2358 \\ -421 \\ 22873 \end{bmatrix} \]

\[ ABr^T = \begin{bmatrix} 2358 \\ -6528 \\ 22873 \end{bmatrix} \]

\[ C[i,j] = \sum A[i,k] \times B[k,i], \quad k=1\ldots n \]

Single element recomputation:

\[ C[2,2] - (Cr[2] - ABr[2]) = -6,168 \]

or

\[ C[2,2] - (Cr^T[2] - ABr^T[2]) = -6,168 \]
Minimizing the recomputation time

Computational cost when an error occurs

<table>
<thead>
<tr>
<th>$n$</th>
<th>Multiplication $4n^3 + n^2(n-1)$</th>
<th>Verification $10n^2 + 6n(n-1)$</th>
<th>Recomputation $2$</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>36</td>
<td>52</td>
<td>2</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>304</td>
<td>232</td>
<td>2</td>
<td>538</td>
</tr>
<tr>
<td>8</td>
<td>2,496</td>
<td>976</td>
<td>2</td>
<td>3,474</td>
</tr>
<tr>
<td>16</td>
<td>20,224</td>
<td>4,000</td>
<td>2</td>
<td>24,226</td>
</tr>
<tr>
<td>32</td>
<td>162,816</td>
<td>16,192</td>
<td>2</td>
<td>179,010</td>
</tr>
<tr>
<td>64</td>
<td>1,306,624</td>
<td>65,152</td>
<td>2</td>
<td>1,371,778</td>
</tr>
</tbody>
</table>
Minimizing the recomputation time

Improvement over extended technique

<table>
<thead>
<tr>
<th>$n$</th>
<th>Extended Technique</th>
<th>Minimum cost technique</th>
<th>% Cost Reduced</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>36</td>
<td>2</td>
<td>94.44</td>
</tr>
<tr>
<td>4</td>
<td>304</td>
<td>2</td>
<td>99.34</td>
</tr>
<tr>
<td>8</td>
<td>2,496</td>
<td>2</td>
<td>99.92</td>
</tr>
<tr>
<td>16</td>
<td>20,224</td>
<td>2</td>
<td>99.99</td>
</tr>
<tr>
<td>32</td>
<td>162,816</td>
<td>2</td>
<td>99.99</td>
</tr>
<tr>
<td>64</td>
<td>1,306,624</td>
<td>2</td>
<td>99.99</td>
</tr>
</tbody>
</table>
Minimizing the recomputation time

Computational cost when an error occurs

<table>
<thead>
<tr>
<th>$n$</th>
<th>Multiplication $4n^3 + n^2(n-1)$</th>
<th>Verification $10n^2 + 6n(n-1)$</th>
<th>Recomputation $2$</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>36</td>
<td>52</td>
<td>2</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>304</td>
<td>232</td>
<td>2</td>
<td>538</td>
</tr>
<tr>
<td>8</td>
<td>2,496</td>
<td>976</td>
<td>2</td>
<td>3,474</td>
</tr>
<tr>
<td>16</td>
<td>20,224</td>
<td>4,000</td>
<td>2</td>
<td>24,226</td>
</tr>
<tr>
<td>32</td>
<td>162,816</td>
<td>16,192</td>
<td>2</td>
<td>179,010</td>
</tr>
<tr>
<td>64</td>
<td>1,306,624</td>
<td>65,152</td>
<td>2</td>
<td>1,371,778</td>
</tr>
</tbody>
</table>
Minimizing the recomputation time

Improvement over previous techniques

<table>
<thead>
<tr>
<th>N</th>
<th>Subject Technique</th>
<th>Extended Technique</th>
<th>% Cost Reduction</th>
<th>Minimum cost technique</th>
<th>% Cost Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>36</td>
<td>9</td>
<td>77.77</td>
<td>2</td>
<td>94.44</td>
</tr>
<tr>
<td>4</td>
<td>304</td>
<td>19</td>
<td>89.47</td>
<td>2</td>
<td>99.34</td>
</tr>
<tr>
<td>8</td>
<td>2,496</td>
<td>39</td>
<td>94.87</td>
<td>2</td>
<td>99.92</td>
</tr>
<tr>
<td>16</td>
<td>20,224</td>
<td>79</td>
<td>97.47</td>
<td>2</td>
<td>99.99</td>
</tr>
<tr>
<td>32</td>
<td>162,816</td>
<td>159</td>
<td>98.74</td>
<td>2</td>
<td>99.99</td>
</tr>
<tr>
<td>64</td>
<td>1,306,624</td>
<td>319</td>
<td>99.37</td>
<td>2</td>
<td>99.99</td>
</tr>
</tbody>
</table>
Recently proposed solutions (3 of 6)

Working at algorithm level with low cost for runtime error detection

- System Level
- Algorithm Level
- Architecture Level
- Circuit Level
- Component Level
- Technology Level

Using Invariants for Runtime Error Detection
Goal

- Achieve tolerance to long duration transient pulses
  - at algorithmic level
  - with low performance overhead
  - in an automatic fashion
  - generalized to other algorithms
Alternative approaches

• Software based error detection techniques
  
  • Duplication with Comparison: increases memory usage and execution time. [Rebaudengo et al, 1999]
  
  • Self Checking Block Signatures: imposes coding and performance penalties. [Goloubeva et al, 2003]
  
  • Use of object oriented languages and libraries in some approaches leads to increased memory footprint and requires source code modification. [Benso, 2005]
Alternative approaches

• An algorithm level technique is proposed in [Lisboa, 2007] for matrix multiplication hardening
  • Far less computational cost than recompute and compare (32x32 matrix – only 4.97% time increase).
  • Explores algorithm properties: conditions that hold after the execution of the algorithm - known as program invariants or post conditions - are checked.

IDEA

Use algorithm properties as a mean for run-time error detection.
Subject technique

• Invariants
  • Properties that always hold during program execution:
    • Pre-conditions
    • Post-conditions
    • Loop invariants
  • Usually used in the software engineering arena, to check if a program performs its tasks as expected after maintenance.
Subject technique

- **Daikon Tool [Ernst et al, 2001]**
  - Automatically detects potential invariants for a given program.
  - Identification of a testable set of invariants feasible for small programs.
  - Linear relationships between up to 3 variables.
  - Low support to complex data structures.
Methodology

• Fault injection campaigns
  • Main program is divided into smaller, less complex, pieces of code.
  • Daikon is used to extract the invariants of each part.
  • Verification code is appended after the algorithm code.
Methodology

• Fault coverage and performance evaluation

main()

Program Slice
Verification

Program Slice
Verification

Program Slice
Verification

Timing Report

Analysis Report

Check Detection

Program Slice
Verification

Random Fault Setup

Generate Reference

Fault Injection

Fault Coverage Evaluation

Modified Code

Performance Evaluation

F times?

Yes

No
Methodology

- Reference and execution results are compared.
- Comparison of results is confronted with verification flag.
- Statistical analysis with report generation.
Experimental results and analysis

- The subject methodology was applied to a test program, split into 5 code pieces:
  - Evaluation of the Baskara formula (domain).
  - Iterative integer multiplication.
  - Conditional statement execution.
  - Arithmetic expression evaluation.
  - Square root calculation.
Experimental results and analysis

Test case program

```c
/* baskara() */
x1=-1.1;
x2=-1.1;
if (a==0 && b!=0){
    x1=-c/b;
    x2=x1;
} else{
    delta= pow(b,2) - 4*a*c;
    if (a!=0 && delta>=0){
        x1=(-b + sqrt(delta) )/(2*a);
        x2=(-b - sqrt(delta) )/(2*a);
    }
}

/* mult() */
while(k2>0){
    if ((k2%2)==0 ){
        k2/=2;
        x2+=x2;
    } else{
        k2--;
        m2+=x2;
    }
}
/* biggerminus() */
if(m1>m2){
    bg=m1-m2;
} else{
    bg=m2-m1;
}
/* sum() */
s = a + b - c;
/* sqrt() */
if(s<0){
    sq=sqrt(-s);
} else{
    sq=sqrt(2*s);
}
/* biggerminus() */
if(sq>bg){
    r=sq-bg;
} else{
    r=bg-sq;
}
```

Luigi Carro
Experimental results and analysis

- Example of invariants inferred for the `mult()` algorithm which are used for verification

<table>
<thead>
<tr>
<th>inputs(x,y) &gt;= 0</th>
<th>inputs(x,y) &gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.mult():::EXIT</code></td>
<td><code>.mult():::EXIT</code></td>
</tr>
<tr>
<td><code>::y == orig(::z)</code></td>
<td><code>::y == 0</code></td>
</tr>
<tr>
<td><code>::y == 0</code></td>
<td><code>::y &lt; ::x</code></td>
</tr>
<tr>
<td><code>::z &gt;= 0</code></td>
<td><code>::y &lt; orig(``::y</code>)`</td>
</tr>
<tr>
<td><code>::y &lt;= ::x</code></td>
<td><code>::y &lt; orig(``::x</code>)`</td>
</tr>
<tr>
<td><code>::y &lt;= ::z</code></td>
<td><code>::x &lt;= ::z</code></td>
</tr>
<tr>
<td><code>::y &lt;= orig(``::y</code>)`</td>
<td><code>::x % orig(``::x</code>) == 0`</td>
</tr>
<tr>
<td><code>::y &lt;= orig(``::x</code>)`</td>
<td><code>::z % orig(``::y</code>) == 0`</td>
</tr>
<tr>
<td><code>::x &gt;= orig(``::x</code>)`</td>
<td><code>::z &gt;= orig(``::y</code>)`</td>
</tr>
<tr>
<td><code>::x &gt;= orig(``::x</code>)`</td>
<td><code>::z % orig(``::x</code>) == 0`</td>
</tr>
<tr>
<td><code>::z &gt;= orig(``::x</code>)`</td>
<td><code>::z &gt;= orig(``::x</code>)`</td>
</tr>
</tbody>
</table>
Experimental results and analysis

- Fault injection campaigns
  - 2000 samples (saturation) for each slice and complete program.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Correct detections</th>
<th>Detection rate*</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult( )</td>
<td>1141</td>
<td>57,05 %</td>
</tr>
<tr>
<td>baskara( )</td>
<td>394</td>
<td>19,70 %</td>
</tr>
<tr>
<td>sum( )</td>
<td>388</td>
<td>19,40 %</td>
</tr>
<tr>
<td>biggerminus( )</td>
<td>539</td>
<td>26,95 %</td>
</tr>
<tr>
<td>square( )</td>
<td>288</td>
<td>14,40 %</td>
</tr>
</tbody>
</table>

* (Reference ≠ Result) AND (verification = error)
Experimental results and analysis

- Fault injection campaigns
  - 2000 samples (saturation) for each slice and complete program.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Correct detections</th>
<th>Detection rate**</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult( )</td>
<td>1963</td>
<td>98,15 %</td>
</tr>
<tr>
<td>baskara( )</td>
<td>1621</td>
<td>81,05 %</td>
</tr>
<tr>
<td>sum( )</td>
<td>1729</td>
<td>86,45 %</td>
</tr>
<tr>
<td>biggerminus( )</td>
<td>1630</td>
<td>81,50 %</td>
</tr>
<tr>
<td>square( )</td>
<td>1031</td>
<td>51,55 %</td>
</tr>
</tbody>
</table>

** verification = error
Experimental results and analysis

- Fault injection campaigns
  - 2000 samples (saturation) for each slice and complete program.

<table>
<thead>
<tr>
<th>Function</th>
<th>(*)</th>
<th>(**)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult()</td>
<td>57.05%</td>
<td>98.15%</td>
</tr>
<tr>
<td>Baskara()</td>
<td>19.70%</td>
<td>81.05%</td>
</tr>
<tr>
<td>sum()</td>
<td>19.40%</td>
<td>86.45%</td>
</tr>
<tr>
<td>biggerminus()</td>
<td>26.95%</td>
<td>81.50%</td>
</tr>
<tr>
<td>sqrt()</td>
<td>14.40%</td>
<td>51.55%</td>
</tr>
</tbody>
</table>

* (Reference ≠ Result) AND (verification = error)

** verification = error
Experimental results and analysis

- Performance overhead

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Execution time</th>
<th>Verification time</th>
<th>Time increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult( )</td>
<td>190,00 ns</td>
<td>5,00 ns</td>
<td>2.63 %</td>
</tr>
<tr>
<td>baskara( )</td>
<td>207,33 ns</td>
<td>104,83 ns</td>
<td>50.56 %</td>
</tr>
<tr>
<td>sum( )</td>
<td>90,16 ns</td>
<td>00,67 ns</td>
<td>0.74 %</td>
</tr>
<tr>
<td>biggerminus( )</td>
<td>87,50 ns</td>
<td>12,66 ns</td>
<td>12.65 %</td>
</tr>
<tr>
<td>square( )</td>
<td>169,33 ns</td>
<td>3,50 ns</td>
<td>2.02 %</td>
</tr>
<tr>
<td>complete program</td>
<td>493,20 ns</td>
<td>68,80 ns</td>
<td>13.95 %</td>
</tr>
</tbody>
</table>
Analysis

• Provides a low cost error detection mechanism, when invariants are detected.

• Better performance using program slices.

• Coverage still low.
  • Coding style to enhance detection.

• Lack of automatic tools to handle complex data structures.

• Automatic generation of invariants is still a bottleneck.
Recently proposed solutions (4 of 6)

Working at software level

- System Level
- Algorithm Level
- Architecture Level
- Circuit Level
- Component Level
- Technology Level

SIFT
Software Implemented Fault Tolerance
Data-oriented Approaches

• Provide a solution for tolerating the effects of faults affecting the data program manipulates

SWIFT

• Introduced by Rebaudengo, Politecnico di Torino, Italy
• Used for hardening any operation among variables
• Based on automatic algorithm-level modifications that introduce information (duplication code) and time redundancies

[Violante, M. Politecnico di Torino, 2006]
SWIFT

Basic principle:

• Each variable must be replicated two times
• Each operation among variables must be replicated two times
• After every usage of a variable, its two replicas must be checked for consistency
The approach detects every transient fault affecting the program’s data stored in:
- External memory
- Registers
- Data cache

The approach is independent from the processor used to run the modified software.

The redundancy can be inserted automatically.

Three overheads are introduced:
- The data segment is doubled
- The code segment is increased
- The execution time is increased

[Violante, M. Politecnico di Torino, 2006]
SWIFT

Success-stories:
• Motorola 68040
• Intel 8051
• IBM PowerPC
• Gaisler LEON1/LEON2

Fault models:
• SEUs
• SETs

[Violante, M. Politecnico di Torino, 2006]
ED\textsuperscript{4}I

- Introduced by McCluskey, Stanford University, USA
- Used for hardening any operation among variables
- Based on algorithm-level modifications that introduces \textit{time redundancies} (replicated with shifted operands)

Basic principle:
- Compute one solution $S = f(x)$
- Compute a shifted solution $S' = f(x.k)$
- Verify whether $S$ and $S'$ are consistent

[Violante, M. Politecnico di Torino, 2006]
ED$^4$I

Original code:

```plaintext
x := 1;
y := 5;
i := 0;
while (i < 5) {
    z := x + i * y;
i := i + 1;
}
i := 2 * z;
```

RESO code (k=-2):

```plaintext
x := -2;
y := -10;
i := 0;
while (i > -10) {
    z := x + i * y / (-2);
i := i + (-2);
}
i := (-4) * z / (-2);
```

[Violante, M. Politecnico di Torino, 2006]
The approach detects every transient or permanent fault affecting the program’s data stored in:
- External memory
- Registers
- Data cache

The approach is independent from the processor used to run the modified software

[Violante, M. Politecnico di Torino, 2006]
The redundancy cannot be inserted automatically: manual intervention is needed to select k

It is not general: bit-wide operations can’t be hardened

Three overheads are introduced:
- The data segment is doubled
- The code segment is increased
- The execution time is increased

[Violante, M. Politecnico di Torino, 2006]
Control-oriented Approaches

- Provide a solution for tolerating the effects of faults affecting the programs’ execution flow

Control Flow Errors

A program may be partitioned in basic blocks obtaining the program’s graph

Example:

```c
0:  i = 0;
    while( i < n ) {
1:      if( a[i] < b[i] )
2:          x[i] = a[i];
3:      else
        x[i] = b[i];
4:          i++;
}
```

Diagram of control flow errors.
Control Flow Errors

Errors modifying the execution w.r.t. the program’s graphs are control-flow errors

[Violante, M. Politecnico di Torino, 2006]
ECCA

- Introduced by Abraham, University of Texas, USA
- Used for detecting contro-flow errors

Based on:
- Modifications to the program source code
- Trigger of division-by-zero exception for error detection

Basic approach:
- Assign an odd signature to each program’s basic block
- Maintain run-time signature with the currently executed basic block
- While entering a basic block, set the run-time signature according to the current basic block and check the correctness of the flow
- While exiting a basic blocks, set the run-time signature according to the next basic block

[Violante, M. Politecnico di Torino, 2006]
Program graph:

Modified code:

... id = BID2/(!((id&BID1)*(id&2)))

x[i] = a[i];
id = BID4+!(id-BID2);

...
Program graph:

Modified code:

```
id = BID2/((!(id&BID1))*(id&2));
x[i] = a[i];
```

- Set the ID to the current basic block’s signature
- Check if it is entered correctly
- A division-by-zero exception is executed in case of error

[Violante, M. Politecnico di Torino, 2006]
ECCA

Program graph:

Modified code:

\[ \text{id} = \text{BID2}/((!\text{id}\&\text{BID1})\&\text{BID2}); \]
\[ \text{x[i]} = \text{a[i]}; \]
\[ \text{id} = \text{BID4}+!!(\text{id}-\text{BID2}); \]

Set the ID to the next basic block’s signature

[Violante, M. Politecnico di Torino, 2006]
- Very effective in dealing with control-flow errors

- Very expensive:
  - Divisions are very time consuming
  - Many operations are needed to set and test the run-time signature: high code penalty

```c
id = BID2/(((id%BID1))*(id%2));
x[i] = a[i];
id = BID4+!!(id-BID2);
```

[Violante, M. Politecnico di Torino, 2006]
CFCSS

- Introduced by McClusckey, Stanford University, USA
- Used for detecting control-flow errors

Based on:
- Modifications to the program source code
- Use logic operations to track control-flow execution

Basic approach:
- Assign a signature to each program’s basic block
- During program execution, a run-time signature is continuously updated
- While entering a basic block:
  - The run-time signature is updated
  - The consistency of the run-time signature with a pre-defined one is evaluated

[Violante, M. Politecnico di Torino, 2006]
Program graph:

```

\[ \text{Modified code:} \]

\[
\ldots
G = G \land d2;
if (G \neq s2) \text{ error()};
x[i] = a[i];
\ldots
\]

[Violante, M. Politecnico di Torino, 2006]
CFCSS

Program graph:

1

2

3

4

5

b_{0,1}

b_{1,2}

b_{1,3}

b_{2,4}

b_{3,4}

b_{4,1}

b_{4,5}

G = G \wedge d2;
if (G != s2) error();
x[i] = a[i];

Modified code:

Set the run-time signature to that of the current basic block

[Violante, M. Politecnico di Torino, 2006]
Check whether the run-time signature corresponds to the expected one

[Violante, M. Politecnico di Torino, 2006]
CFCSS

• Low-cost techniques:
  • Logic operations are not time consuming
  • Few operations are added, resulting in low code penalty

• Error detection is very critical: it changes the program’s graph by introducing a jump

[Violante, M. Politecnico di Torino, 2006]
YACCA

• Introduced by Massimo Violante, Politecnico di Torino, Italy
• Used for detecting control-flow errors

Based on:
• Modifications to the program source code
• Use logic operations to track control-flow execution

[Violante, M. Politecnico di Torino, 2006]
Basic principle:

- **Two signatures** are assigned to each program’s basic block (enter and exit signatures, Bx1, Bx2)
- A run-time signature is constantly updated
- When entering a basic block:
  - Check the correctness of the execution
  - Set the run-time signature to the enter one
- When exiting a basic block:
  - Check the correctness of the execution
  - Set the run-time signature to the exit one

[Violante, M. Politecnico di Torino, 2006]
YACCA

Program graph:

1. $b_{0,1}$
2. $b_{1,2}$
3. $b_{1,3}$
4. $b_{2,4}$
5. $b_{3,4}$
6. $b_{4,1}$
7. $b_{4,5}$

Modified code:

```plaintext
... 
ERR|=code^B12 
code = code^M21;
x[i] = a[i];
ERR|=code^B21;
code=code^M22;
...
```
[Violante, M. Politecnico di Torino, 2006]
Program graph:

Modified code:

\[
\ldots
\]

```
code = code^M21;
```

Set the run-time signature to the enter signature (B21) of B2

[Violante, M. Politecnico di Torino, 2006]
Program graph:

Modified code:

...  
ERR|=code^B12  
code = code^M21;  
x[i] = a[i];  
ERR|=code^B21;  
code=code^M22;  

Check whether B2 was correctly executed

[Violante, M. Politecnico di Torino, 2006]
Program graph:

Modified code:

... 
ERR| = code^B12 
code = code^M21;  
x[i] = a[i];  
ERR| = code^B21;  

\[
\text{Set the run-time signature to the exit signature B22 of B2}
\]

[Violante, M. Politecnico di Torino, 2006]
YACCA

- Low-cost techniques:
  - Logic operations are not time consuming
  - Few operations are added, resulting in low code penalty
  - The program’s graph is not modified

[Violante, M. Politecnico di Torino, 2006]
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Impl. cost</th>
<th>Fault model</th>
<th>Generality</th>
<th>Code increase</th>
<th>Data increase</th>
<th>Time increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>N version</td>
<td>😞</td>
<td>😄</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😄</td>
</tr>
<tr>
<td>programming</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recovery blocks</td>
<td>😞</td>
<td>😄</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😄</td>
</tr>
<tr>
<td>ABFT</td>
<td>😞</td>
<td>😄</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😄</td>
</tr>
<tr>
<td>SWIFT</td>
<td>😄</td>
<td>😄</td>
<td>😄</td>
<td>😞</td>
<td>😞</td>
<td>😄</td>
</tr>
<tr>
<td>ED^4I</td>
<td>😄</td>
<td>😄</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😄</td>
</tr>
<tr>
<td>ECCA</td>
<td>😄</td>
<td>😄</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😄</td>
</tr>
<tr>
<td>CFCSS</td>
<td>😄</td>
<td>😄</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😄</td>
</tr>
<tr>
<td>YACCA</td>
<td>😄</td>
<td>😄</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😄</td>
</tr>
</tbody>
</table>

[Violante, M. Politecnico di Torino, 2006]
Some figures

• Experimental setup
  • Matrix multiplication program
  • Intel 8051 processor
  • Hardware-accelerated fault injection in:
    • Code segment
    • Data segment
    • Processor’s registers
  • SEU fault model

[Violante, M. Politecnico di Torino, 2006]
Some Figures

- System failures due to SEUs in the code segment:
  - Un-hardened program: 1.0
  - ABFT: 4x better
  - ED4I: 4x better
  - SWIFT+YACCA: 6x better

[Violante, M. Politecnico di Torino, 2006]
Some Figures

- System failures due to SEUs in the data segment:
  - Un-hardened program: 1.0
  - ABFT: 6x better
  - ED4I: 29x better
  - SWIFT+YACCA: $\infty$ better (0 system failures observed)

[Violante, M. Politecnico di Torino, 2006]
Some Figures

- System failures due to SEUs in the processor’s registers:
  - Un-hardened program: 1.0
  - ABFT: 9x better
  - ED4I: 13x better
  - SWIFT+YACCA: 15x better

[Violante, M. Politecnico di Torino, 2006]
Some Figures

- Time increase:
  - Un-hardened program: 1.0
  - ABFT: 3.8x
  - ED4I: 1.9x
  - SWIFT+YACCA: 3.5x

- Code increase:
  - Un-hardened program: 1.0
  - ABFT: 2.3x
  - ED4I: 1.6x
  - SWIFT+YACCA: 3.9x

[Violante, M. Politecnico di Torino, 2006]
Some Figures

• Data increase:
  • Un-hardened program: 1.0
  • ABFT: 2.0x
  • ED4I: 1.9x
  • SWIFT+YACCA: 2.2x
Hybrid SIFT

- Software-only SIFT may introduce unacceptable time penalty
- Moving in hardware some tasks may reduce this overhead
- Masking, detection, location, and recovery implemented in software and in hardware
- Possible approaches:
  - Lockstep execution
  - Watchdogs
  - Lightweight watchdogs
Recently proposed solutions (5 of 6)

Working at system (software and hardware) level

Main concepts

- Detection of errors when they affect software behavior is preferable to detection at hardware level.
- SWAT exploits this concept to achieve low cost error detection for cores at software level, by checking:
  - Fatal exceptions
  - Program crashes or hangs
  - Unusually high amount of operating system activity
- Some hardware errors that do not manifest themselves in software behaviors are not detected by SWAT.
- SWAT suffers from the drawbacks of high level error detection mechanisms that will be discussed later.
Recently proposed solutions (6 of 6)

Working at lower levels to detect errors and at higher system levels to correct them.

Application Layer
Middleware/Architectural Layer
Configurable/Programming Layer
Register/Logic Layer
Technology Layer

Main concepts

• SoC is divided into several layers

• Each layer has specific fault tolerance mechanisms:
  
  o Detection is cheaper at lower layers
  
  o Correction is better performed at higher layers

• Lower layers notify upper layers when error is detected

• Upper layers send reconfiguration information to lower layers according to application requirements

• Key issue: interfaces between layers to report errors and inform about needed level of reliability according to application
Sample roles of layers

• Technology layer
  o Built-in current sensors detect transient upsets
  o Upper layer can configure detection capabilities

• Register/Logic layer
  o EDAC used to harden memories
  o TMR used to harden logic
  o Upper layer can enable/disable detection mechanisms

• Configuration/Programming layer (in reconfigurable platforms)
  o Reconfiguration can be used to disable faulty modules
  o Periodical relocation of active modules reduces degradation
Sample roles of layers

• Middleware/Architectural layer
  o Applies well-known redundancy techniques such as TMR at component level
  o Redundant modules designed independently to allow SEU and design errors detection
  o Test mechanisms can be used to check modules at run time
  o Checkpoints can be used to allow error recovery

• Application layer
  o Almost everything can be used to improve reliability at this level
  o Software implemented TMR, EDAC and other techniques can be used
Outline

• Introduction: concepts and definitions
• Motivation: new challenges imposed by future technologies
• Radiation induced faults: the major challenges
• Existing mitigation techniques vs. the new scenario
• Desired properties of new radiation induced faults mitigation techniques
• Recent solutions working at different abstraction levels to deal with transient faults
• Conclusions
Conclusions

• New low cost mitigation techniques, providing error detection and error correction must be developed.

• Circuit level approaches can be better than TMR, but still impose significant area and power overheads.

• Algorithm level mitigation is a better approach, but it is hard to generalize and automate.
High level error detection: pros and cons

[Sorin, 2009]

• Checking at a higher level:
  • reduces hardware costs
  • reduces the number of false positives
  • is necessary anyway for certain types of errors

• However:
  • provides little diagnostic information (type and location)
  • longer and potentially unbounded error detection latency
  • recovery process may be more complex
Final Remark

• There is NO silver bullet!
• Combine hardware and software based techniques at different levels
• Leverage on specific strengths of each technique at each level.
Thank You!

Questions?

Contact: carro@inf.ufrgs.br, calisboa@inf.ufrgs.br

Copy of slides available at
http://www.inf.ufrgs.br/~calisboa/IESS2009
References (in order of appearance)


References (in order of appearance)


References (in order of appearance)

• LISBOA, C. A., ERIGSSON, M. I., and CARRO, L. System level approaches for mitigation of long
duration transient faults in future technologies. In: IEEE EUROPEAN TEST SYMPOSIUM, 12., ETS
2007, Freiburg, DEU. Proceedings... Los Alamitos, USA: IEEE Computer Society, 2007, p. 165-
170.

• LISBOA, C.; ARGYRIDES, C.; PRADHAN, D.; and CARRO, L. Algorithm level fault tolerance: a
technique to cope with long duration transient faults in matrix multiplication algorithms. In: IEEE

• LISBOA, C. et al. Invariant checkers: an efficient low cost technique for run-time transient errors
detection. In: IEEE INTERNATIONAL ON-LINE TESTING SYMPOSIUM, 15., IOLTS 2009,
Sesimbra, POR. Proceedings... [S.l.: s.n.], 2009.

• REBAUNDENGO, M. et al. Soft-error detection through software fault-tolerance techniques. In:
IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI

• GOLOUBEVA, O. et al. Soft error detection using control flow assertions. INTERNATIONAL
SYMPOSIUM ON DEFECT AND FAULT TOLERANCE, 18., 2003, Boston, USA. Proceedings...
References (in order of appearance)


