Data-Flow-Sensitive Fault-Space Pruning for the Injection of Transient Hardware Faults

Oskar Pusz, Christian Dietrich, Daniel Lohmann

June 22, 2021
Motivation
“Toyota claimed the 2005 Camry's main CPU had error detecting and correcting RAM. It didn’t.”

The Fault Space

- Fault injection campaign for a given program (execution)
  - **FM**: Uniformly-distributed soft errors in registers and memory
  - **Goal**: Quantify the failure-behavior of a single program execution.

```
// initial r0=5, r1=11

// shift-left by 1
r0 := SHL r0, 1 //r0=10

// bit-wise XOR with 7
r1 := XOR r1, 7 //r1=12

// bit-wise AND
r1 := AND r0, r1 //r1=8

// move result to r0
r0 := MOV r1 //r0=8

// result in r0
r0
r1
```

Plan and inject!

- Record a fault-free execution of the program-under-test.
- Inject every memory location in each processor cycle.
- Wait.... (40 injections)
Fault injection campaign for a given program (execution)

- **FM**: Uniformly-distributed soft errors in registers and memory
- **Goal**: Quantify the failure-behavior of a single program execution.

```plaintext
// initial r0=5, r1=11

// shift-left by 1
r0 := SHL r0, 1 // r0=10

// bit-wise XOR with 7
r1 := XOR r1, 7 // r1=12

// bit-wise AND
r1 := AND r0, r1 // r1=8

// move result to r0
r0 := MOV r1 // r0=8

// result in r0
```

---

**Plan and inject!**

- Record a fault-free execution of the program-under-test.
- Inject every memory location in each processor cycle.
- Wait... (40 injections)
The Fault Space

- Fault injection campaign for a given program (execution)
  - **FM**: Uniformly-distributed soft errors in registers and memory
  - **Goal**: Quantify the failure-behavior of a single program execution.

```plaintext
// initial r0=5, r1=11

// shift-left by 1
r0 := SHL r0, 1 //r0=10

// bit-wise XOR with 7
r1 := XOR r1, 7 //r1=12

// bit-wise AND
r1 := AND r0, r1 //r1=8

// move result to r0
r0 := MOV r1 //r0=8

// result in r0
```

- Plan and inject!
  - Record a fault-free execution of the program-under-test.
The Fault Space

- Fault injection campaign for a given program (execution)
  - **FM**: Uniformly-distributed soft errors in registers and memory
  - **Goal**: Quantify the failure-behavior of a single program execution.

```c
// initial r0=5, r1=11

// shift-left by 1
r0 := SHL r0, 1 //r0=10

// bit-wise XOR with 7
r1 := XOR r1, 7 //r1=12

// bit-wise AND
r1 := AND r0, r1 //r1=8

// move result to r0
r0 := MOV r1 //r0=8

// result in r0
```

- **Plan and inject!**
  - Record a fault-free execution of the program-under-test.
  - Inject every memory location in each processor cycle.
The Fault Space

Fault injection campaign for a given program (execution)
- **FM**: Uniformly-distributed soft errors in registers and memory
- **Goal**: Quantify the failure-behavior of a single program execution.

```
// initial r0=5, r1=11
////////////////////
// shift-left by 1
r0 := SHL r0, 1 //r0=10
// bit-wise XOR with 7
r1 := XOR r1, 7 //r1=12
// bit-wise AND
r1 := AND r0, r1 //r1=8
// move result to r0
r0 := MOV r1 //r0=8
```

- **Plan and inject!**
  - Record a fault-free execution of the program-under-test.
  - Inject every memory location in each processor cycle.
  - Wait.... (40 injections)
Fault Space Pruning

- **Def-Use Pruning**
  - **Observation**: Faults between read/write events have equivalent behavior
  - Faults only become active on a read; a write makes it benign.
  - Select one *fault-injection pilot* for each equivalence interval

---

![Diagram showing fault space pruning with read event, fault equivalence, write event, and fault pilot intervals over time (t) with operations SHL, XOR, AND, MOV.]

- Significantly reduces number of injections (40 → 24), but...
- Equivalences are only formed horizontally, not vertically.
- Some instructions mask errors or only propagate them.
Fault Space Pruning

- **Def-Use Pruning**
  - **Observation**: Faults between read/write events have equivalent behavior.
  - Faults only become active on a read; a write makes it benign.
  - Select one *fault-injection pilot* for each equivalence interval.

  ![Diagram](image)

  - Significant reduces number of injections ($40 \rightarrow 24$), but...
    - Equivalences are only formed horizontally, not vertically.
    - Some instructions mask errors or only propagate them.
Basic principle

As long as a single-bit error does not escalate to a multi-bit error or becomes visible, we can extend the equivalence set.
In a nutshell: Data-flow-Sensitive Pruning

Basic principle

As long as a single-bit error does not escalate to a multi-bit error or becomes visible, we can extend the equivalence set.

Golden run is one path through the program

- Knowledge: instructions, register values, instruction semantic
- We can calculate masking and propagation behavior.
Step 1: Build a data-flow graph

- Directed graph of operations (blue) and operands (yellow)
  - All values and operations are known from the golden run
  - Artificial $\varepsilon$-nodes model the influence of read events
Step 1: Build a data-flow graph

- Directed graph of operations (blue) and operands (yellow)
  - All values and operations are known from the golden run
  - Artificial $\varepsilon$-nodes model the influence of read events
- Choosing read or final value nodes for injection leads to Def-Use pilots
Error propagation of a single instruction
- **Assumption**: Exactly one input bit is faulty
- Combine instruction semantic and operand values
Step 3: Propagate equivalences globally

- One FI symbol per operand bit
  - All occurrences are equivalent
  - Goal: Propagate symbols
Step 3: Propagate equivalences globally

- **One FI symbol per operand bit**
  - All occurrences are equivalent
  - **Goal**: Propagate symbols

- **Propagation Phase**
  - readers = 0 → mark benign
  - readers = 1 → propagate back
  - readers > 1 → do nothing
Step 3: Propagate equivalences globally

- One FI symbol per operand bit
  - All occurrences are equivalent
  - **Goal**: Propagate symbols

- Propagation Phase
  - readers = 0 → mark benign
  - readers = 1 → propagate back
  - readers > 1 → do nothing

- Mask and Plan
  - Operation can mask faults
  - One injection per symbol
## Evaluation: MiBench and Microbenchmarks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mi/BC</td>
<td>70.33</td>
<td>222.40</td>
<td>181.43</td>
<td>-18.42</td>
</tr>
<tr>
<td>mi/BFD</td>
<td>1894.82</td>
<td>331.38</td>
<td>295.95</td>
<td>-10.69</td>
</tr>
<tr>
<td>mi/BFE</td>
<td>1880.82</td>
<td>326.93</td>
<td>292.97</td>
<td>-10.39</td>
</tr>
<tr>
<td>mi/QSORT</td>
<td>1623.90</td>
<td>270.58</td>
<td>234.31</td>
<td>-13.40</td>
</tr>
<tr>
<td>mi/RDD</td>
<td>3506.17</td>
<td>397.60</td>
<td>345.37</td>
<td>-13.13</td>
</tr>
<tr>
<td>mi/RDE</td>
<td>3457.59</td>
<td>397.99</td>
<td>351.90</td>
<td>-11.58</td>
</tr>
<tr>
<td>mi/SHA</td>
<td>242.63</td>
<td>252.79</td>
<td>219.74</td>
<td>-13.07</td>
</tr>
<tr>
<td>µ/FIB</td>
<td>1.15</td>
<td>8.87</td>
<td>7.56</td>
<td>-14.78</td>
</tr>
<tr>
<td>µ/LSUM</td>
<td>0.02</td>
<td>0.26</td>
<td>0.26</td>
<td>0.00</td>
</tr>
<tr>
<td>µ/MIXED</td>
<td>0.03</td>
<td>0.45</td>
<td>0.40</td>
<td>-11.83</td>
</tr>
<tr>
<td>µ/QSort</td>
<td>0.18</td>
<td>1.27</td>
<td>1.22</td>
<td>-4.36</td>
</tr>
<tr>
<td>µ/QSortIter</td>
<td>1.20</td>
<td>4.23</td>
<td>3.88</td>
<td>-8.18</td>
</tr>
</tbody>
</table>
Def-Use Pruning is one-dimensional
- Equivalences are only formed along the time axis
- Instruction can mask errors benign or propagate them

DFPrune: Data-Flow-Sensitive Fault Space Pruning
- Faults are equivalent as long as the error does not escape!
- Propagate FI Symbols on the Data-Flow Graph
- Instruction-local Fault Equivalences

DFPrune reduces the number of required injections
- Between 10 and 18 percent reduction for MiBench
- Reductions across all failure classes
- At least as good as Def-Use Pruning