Fast and Accurate Vulnerability Analysis of a RISC-V Processor

Joseph Paturel, Simon Rokicki, Olivier Sentieys

Univ. Rennes, Inria, IRISA
Why care about Fault Tolerance?

• Modern technologies
  – Lower node capacitances
  – Denser layouts
  – Increased frequencies

• Energy efficiency
  – Lower supply and threshold voltages

High SET sensitivity
Vulnerability Analysis

• Fault injection, simulation or emulation most often:
  – Only injects single-bit faults
  – Does not model the microarchitecture
  – Ignores combinational logic

• Memory/register fault injection is not enough
  – Need to model microarchitecture
  – Need to consider combinational logic [1]

• New technologies exhibit multi-bit error behaviors
  – Need to model MBUs as well as SEUs

Contributions

• MBUs are present and are here to stay

• Fault injection methodology and flow (Part II)
  – From gate to microarchitecture
  – MBU-aware
  – Fast and accurate

• Use case: Comet RISC-V processor core (Part I)
Part I: Comet
a HLS designed RISC-V Core
What You Simulate is What You Synthesize

• Traditional Processor Design Flow
  – Maintain two coherent models:
    • RTL and simulation (ISS) models
What You Synthesize is What You Simulate

- Traditional Processor Design Flow
  - Maintain two coherent models:
    - RTL and simulation (ISS) models

- Proposed Flow
  - Design the processor as well as its software validation flow from a single high-level model
Explicitly Pipelined Simulator (1/2)

- **Comet core**
  - 32-bit RISC-V instruction set RV32IM
  - In-order 5-stage pipeline micro-architecture
- Pipelined stages are explicit
- Main loop is pipelined (II=1)
- Explicit stall mechanism
- Explicit forwarding

```c
struct FtoDC ftodc;
struct DCtoEx dctoex;
struct ExtoMem extomem;
struct MemtoWB memtowb;

while (true) {
    ftodc_temp = fetch();
    dctoex_temp = decode(ftodc);
    extomem_temp = execute(dctoex);
    memtowb_temp = memory(extomem);
    writeback(memtowb);
    bool forward = forwardLogic();
    bool stall = stallLogic();
    if (!stall)
        ftodc = ftodc_temp;
        dctoex = dctoex_temp;
        extomem = extomem_temp;
        memtowb = memtowb_temp;
    end
    if (forward)
        dctoex.value1 = extomem.result;
    end
} /*
```

```c
```
Explicitly Pipelined Simulator (2/2)

```
struct FtoDC ftodc;
struct DCtoEx dctoex;
struct ExtoMem extomem;
struct MemtoWB memtowb;

while true do
    ftodc_temp = fetch();
    dctoex_temp = decode(ftodc);
    extomem_temp = execute(dctoex);
    memtowb_temp = memory(extomem);
    writeback(memtowb);
    bool forward = forwardLogic();
    bool stall = stallLogic();
    if !stall then
        ftodc = ftodc_temp;
        dctoex = dctoex_temp;
        extomem = extomem_temp;
        memtowb = memtowb_temp;
    end
    if forward then
        dctoex.value1 = extomem.result;
    end
end
```
Simulation performance
- 26 Millions cycles per sec.
- MiBench
- 8th-gen. Intel core i7

What about quality of the hardware?
Synthesis Results

- Target technology is STMicro 28nm FDSOI
- All cores are configured for rv32i

<table>
<thead>
<tr>
<th>Core</th>
<th>Language</th>
<th>Frequency Target (MHz)</th>
<th>Area ($\mu m^2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comet [1]</td>
<td>C++</td>
<td></td>
<td>8 476</td>
</tr>
<tr>
<td>PicoRV32 [3]</td>
<td>Verilog</td>
<td>700</td>
<td>7 830</td>
</tr>
</tbody>
</table>

Area and frequency results for different RISC-V cores.
Advantages and Limitations

Advantages

• Improves readability, productivity, maintainability, and flexibility of the design
• Fast simulation (~20.10^6 cycles/s)
• Object-Oriented processor model can be easily modified, expanded and verified

Limitations

• Pipeline stages and some features (e.g. multi-cycle operators) have to be explicit
• HLS tools may have trouble synthesizing large multi-core systems...
Part II: Vulnerability Analysis Flow
Proposed Approach to Vulnerability Analysis

- `.v/.vhdl`
- Gate-level Analysis
- Error Patterns
- Workload
- uArch Injection
- Vulnerability Metrics

HLS

C++ Model → C++ Compilation
1/ Gate-level Analysis

- Inject SETs in the gate-level netlist

![Diagram showing gate-level analysis process with error probability and bit position as parameters for SETs injection.]

**Parameters:**

- Resolution
- Duration
- Type
- \( N_{inj} \)
- \( N_{sim} \)

**Design Time**

- Fault injector
- Gate-level netlist
- Technology library

**Run Time**

- Error insertions
- Input generation
- Logging
- TestBench
1/ Gate-level Analysis

- Logging patterns and error probability (SEUs + MBUs)
Results: Comet Execution Stage

Number of erroneous bits in output register

- SEUs 94.9%
- MBUs 5.1%

Histogram of the number of erroneous register bits

1 Million injections
Influence of SET Width and Frequency on MBUs

• Fixed width (400ps)

<table>
<thead>
<tr>
<th>Freq.</th>
<th>200 MHz</th>
<th>300 MHz</th>
<th>400 MHz</th>
<th>500 MHz</th>
<th>600 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEU</td>
<td>9,308</td>
<td>15,592</td>
<td>23,613</td>
<td>26,489</td>
<td>30,919</td>
</tr>
<tr>
<td></td>
<td>93%</td>
<td>96.3%</td>
<td>94.1%</td>
<td>94.9%</td>
<td>95.5%</td>
</tr>
<tr>
<td>MBU</td>
<td>699</td>
<td>599</td>
<td>1,473</td>
<td>1,429</td>
<td>1,447</td>
</tr>
<tr>
<td></td>
<td>7%</td>
<td>3.7%</td>
<td>5.9%</td>
<td>5.1%</td>
<td>4.5%</td>
</tr>
</tbody>
</table>

• Fixed frequency (500MHz)

<table>
<thead>
<tr>
<th>SET</th>
<th>100 ps</th>
<th>200 ps</th>
<th>400 ps</th>
<th>500 ps</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEU</td>
<td>5,144</td>
<td>10,529</td>
<td>26,489</td>
<td>33,449</td>
</tr>
<tr>
<td></td>
<td>97.6%</td>
<td>95.3%</td>
<td>94.9%</td>
<td>95.9%</td>
</tr>
<tr>
<td>MBU</td>
<td>127</td>
<td>755</td>
<td>1,429</td>
<td>1,432</td>
</tr>
<tr>
<td></td>
<td>2.4%</td>
<td>4.7%</td>
<td>5.1%</td>
<td>4.1%</td>
</tr>
</tbody>
</table>
2/ Microarchitectural-Level Fault Injection

- Augmented simulator allows for injection of gate-level fault patterns
- Injection is guided by the area of the different pipeline stages
- Fault classes considered:
  - Crashes and Hangs
  - ISM, AOM, ISM & AOM

ISM: Internal State Mismatch
AOM: Application Output Mismatch
Comet Vulnerability Analysis Results

- Error class proportions
- Standard vs. proposed approach
Conclusion on Vulnerability Analysis

• MBUs are present and are here to stay

• MBUs significantly impact AVF
  – more than 50% critical errors (crashes & hangs)

• Fault injection methodology and flow
  – From gate to microarchitecture
  – Conscious of MBU patterns and error probability
  – Fast and accurate
Conclusion & Roadmap on Comet

• Efficient processor core design (HW μarch + SW simulator) from a single C++ code

• Current projects
  – Dynamic Binary Translation, Non-Volatile Processor, Fault-Tolerant Multicore, etc.

• Perspectives
  – Automatic source-to-source transformations for HLS
    • From ISS-like specification to HLS-optimized C code
  – Support for floating point extension
  – RTOS Support (process, interrupt controller, peripherals)
  – Multi-core system with cache coherency (Q4 2019)
  – Many-core system with NOC (2020)
Questions

Thank you for your attention!

https://gitlab.inria.fr/srokicki/Comet