Trading Fault Tolerance for Performance in AN encoding

Norman A. Rink and Jeronimo Castrillon
Technische Universität Dresden
norman.rink@tu-dresden.de

Computing Frontiers
15-17 May 2017
Siena, Italy
1. Introduction – hardware faults and software-based fault tolerance

2. AN encoding

3. Configurable compiler-implemented AN encoding

4. Fault experiments and results

5. Summary and outlook
1. Introduction – hardware faults and software-based fault tolerance

2. AN encoding

3. Configurable compiler-implemented AN encoding

4. Fault experiments and results

5. Summary and outlook
Hardware faults and soft errors

- Faults are a long-standing and recurring issue.
  - In safety-critical embedded devices.
  - In servers/data centers and HPC workloads.

- What causes hardware faults?
  - Cosmic radiation.
  - “Dim silicon” (near-threshold computing to save energy).
  - Temperature variations, process variations.

- Typically lead to transient hardware faults, aka soft errors.

- Non-negligible fault rates in emerging computing paradigms:
  - Silicon-/carbon-based nano-technologies, graphene.
  - Chemical information processing, bio-inspired/quantum computing, NVM.

---

The **trade-off** of computing with faulty hardware:

![Diagram](image)

- **Hardware-implemented fault tolerance is not flexible.**

<table>
<thead>
<tr>
<th>Cost (runtime, energy)</th>
<th>Error probability/od</th>
<th>Hardware faults and soft errors – cont’d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Safety-/security-critical applications, e.g. automotive, operating system (kernels)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-critical user applications, processes that can be restarted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Approximate computing applications, e.g. image processing, machine learning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Future and emerging HW (perhaps)</td>
</tr>
</tbody>
</table>
Error detection through redundant code

- Typical approach to detecting/correcting hardware faults:
  - Replicate data flow (duplication is sufficient for **error detection**).
  - Insert checks.

-damage

- Original code:
  
  ```
  %3 = add i64 %0, %1
  %4 = mul i64 %3, %2
  ```

- Fault-tolerant code:
  
  ```
  %3 = add i64 %0, %1
  %r3 = add i64 %r0, %r1
  %4 = mul i64 %3, %2
  %r4 = mul i64 %r3, %r2
  %f0 = icmp eq i64 %4, %r4
  br i1 %f0, label continue, label recover
  ```

- State of the art: EDDI ’02, SWIFT ’05.
  - Many variations and improvements exist.
  - It is usually assumed that the memory system is already protected, typically by ECC.

- Runtime overhead of duplicated dataflow typically < 2x
Outline

1. Introduction – hardware faults and software-based fault tolerance
2. AN encoding
3. Configurable compiler-implemented AN encoding
4. Fault experiments and results
5. Summary and outlook
AN encoding

- Correctness condition: **Integer values are multiples of a fixed constant A.**
  
  → Check for faults like this:
  ```c
  if (n % A != 0) { exit(AN_ERROR_CODE); }
  ```

- Advantages over replication:
  - Data in memory is encoded (automatically protected): no need to replicate loads/stores.
  - Suitable for multi-threaded and shared memory applications.

- Disadvantages: Large runtime overheads (up to and over several $10^x$).
  - Checking is expensive (modulo operation).
  - Decoding is expensive (division by A).
  - Decoding often required, e.g., for address operands.

Aside: more eager variants of AN encoding have even larger overheads.

Trade frequencies of these operations for runtime.
AN encoding – cont’d

- Make a program fault-tolerant by transforming it into an AN-encoded program.

1. Value encoding:

2. Operation encoding, e.g.:

3. Check insertion:

   - Where non-encoded values enter/leave the scope of AN encoding:
     - memory accesses, calls to external functions, return values

   → Often referred to as synchronization points.
Outline

1. Introduction – hardware faults and software-based fault tolerance
2. AN encoding
3. Configurable compiler-implemented AN encoding
4. Fault experiments and results
5. Summary and outlook
Configurable compiler-implemented AN encoding

- **optional pre-optimizations**
  - AN encoding increases the complexity of code.
  - Hence, after AN encoding, the compiler may fail to spot opportunities for optimization.

- **value encoding, operation encoding, check insertion**
  - As previously discussed.
  - Remember, checks are inserted as so-called *synchronization points*.

- **expansion**
  - Turn encoding, decoding and checking operations into (sequences) of native machine operations.
Encoding variants

- check insertion
  - Always check before values are stored to memory.
  - Always check before values are decoded.

<table>
<thead>
<tr>
<th>check insertion</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_enc = loadenc a_enc</td>
<td>v_enc = loadenc a_enc</td>
<td>v_enc = loadenc a_enc</td>
</tr>
<tr>
<td>check(v_enc)</td>
<td>an_assert(v_enc % A)</td>
<td>an_accumulate(v_enc)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>variant</th>
<th>load</th>
<th>check</th>
<th>pre-opt.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>A</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>A</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>B</td>
<td>x</td>
</tr>
<tr>
<td>po.1</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>po.2</td>
<td>B</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>po.3</td>
<td>B</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

one local accumulator per function

code generated for the variants during compilation
#### Runtime overheads

<table>
<thead>
<tr>
<th>label</th>
<th>test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-C</td>
<td>bubblesort</td>
</tr>
<tr>
<td>D</td>
<td>CRC (cyclic redundancy checker)</td>
</tr>
<tr>
<td>E</td>
<td>DES encryption</td>
</tr>
<tr>
<td>F</td>
<td>Dijkstra</td>
</tr>
<tr>
<td>G-I</td>
<td>lex, parse, eval</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>label</th>
<th>test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>Fibonacci</td>
</tr>
<tr>
<td>K</td>
<td>integer matrix multiply</td>
</tr>
<tr>
<td>L</td>
<td>memcopy</td>
</tr>
<tr>
<td>M-O</td>
<td>quicksort</td>
</tr>
</tbody>
</table>

Best results with pre-optimizations:

<table>
<thead>
<tr>
<th>Variant</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.9</td>
</tr>
<tr>
<td>2</td>
<td>7.2</td>
</tr>
<tr>
<td>3</td>
<td>6.5</td>
</tr>
<tr>
<td>po.1</td>
<td>4.3</td>
</tr>
<tr>
<td>po.2</td>
<td>3.8</td>
</tr>
<tr>
<td>po.3</td>
<td>3.6</td>
</tr>
</tbody>
</table>
1. Introduction – hardware faults and software-based fault tolerance

2. AN encoding

3. Configurable compiler-implemented AN encoding

4. Fault experiments and results

5. Summary and outlook
Fault injection

Assumptions:
- Only a single fault affects program execution.
- Only single bit flips occur.

Simulate symptoms of faults by …
- … flipping a random bit in a random register.
- … flipping a random bit in a random memory location (that is accessed).

Evaluate AN encoding on the conditional probability:

\[ p_{SDC} = P(\text{"silent data corruption"} \mid \text{"hardware fault (visible at the architecture level)"}) \]
Faults in the CPU – full results
Faults in the CPU and in memory – SDC

CPU:

- Memory accesses are relatively rare.
- A single vulnerable has a much stronger effect.
- Stack accesses have been found to be particularly vulnerable.

<table>
<thead>
<tr>
<th>fault type</th>
<th>plain</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>po.1</th>
<th>po.2</th>
<th>po.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>161.2</td>
<td>6.8</td>
<td>10.7</td>
<td>11.2</td>
<td>18.7</td>
<td>20.2</td>
<td>22.1</td>
</tr>
<tr>
<td>memory</td>
<td>480.0</td>
<td>8.8</td>
<td>26.8</td>
<td>32.9</td>
<td>19.4</td>
<td>20.5</td>
<td>32.3</td>
</tr>
</tbody>
</table>

$P_{SDC} \cdot 10^3$: 24x, 7x, 55x, 15x
1. Introduction – hardware faults and software-based fault tolerance
2. AN encoding
3. Configurable compiler-implemented AN encoding
4. Fault experiments and results
5. Summary and outlook
Summary and outlook

- On less reliable HW, some applications ...
  - ... can live with the occasional error (approximate computing).
  - ... may require additional measures to tolerate faults (e.g. OS kernels).

- Encoding is an interesting alternative to instruction duplication.
  - Protection of entire computing systems, multi-threaded and shared memory applications.

- AN encoding has large runtime overheads (several 10x to 100x).
  - Here: reduction from 9.9x to 3.6x – accompanied by an increase in $p_{SDC}$.
  - Further reduction to 2.1x if pointers are not encoded.

- Can we design HW that supports error detection by encoding?
  - Must understand better the interaction of encoding with compiler optimizations.
  - In assessments, would like to avoid long-running fault injection experiments.
Trading Fault Tolerance for Performance in AN encoding

Norman A. Rink and Jeronimo Castrillon
Technische Universität Dresden
norman.rink@tu-dresden.de

Work supported by the German Research Foundation (DFG) within the Cluster of Excellence ‘Center for Advancing Electronics Dresden’ (cfaed).

Thank you.
Trading Fault Tolerance for Performance in AN encoding

Norman A. Rink and Jeronimo Castrillon
Technische Universität Dresden
norman.rink@tu-dresden.de

Computing Frontiers
15-17 May 2017
Siena, Italy
Worked example of AN encoding

*p = foo( (*p) + 42 );

(A = 3)

%1 = load i64* %0
%2 = add i64 %1, 42
%3 = call i64 @foo(i64 %2)
store i64 %3, i64* %0

%t0 = ptrtoint i64* %0 to i64
%t1 = div i64 %t0, 3
%t2 = inttoptr i64 %t1 to i64*
%1 = load i64* %t2

%2 = add i64 %1, 126

%t3 = div i64 %2, 3
%t4 = call i64 @foo(i64 %t3)
%3 = mul i64 %t4, 3

%t5 = ptrtoint i64* %0 to i64
%t6 = div i64 %t5, 3
%t7 = inttoptr i64 %t6 to i64*

store i64 %3, i64* %t7

check insertion

call void @check(i64 %0)
%t0 = ptrtoint i64* %0 to i64
%t1 = div i64 %t0, 3
%t2 = inttoptr i64 %t1 to i64*
%1 = load i64* %t2

call void @check(i64 %1)
%2 = add i64 %1, 126

call void @check(i64 %2)
%t3 = div i64 %2, 3
%t4 = call i64 @foo(i64 %t3)
%3 = mul i64 %t4, 3

call void @check(i64 %0)
%t5 = ptrtoint i64* %0 to i64
%t6 = div i64 %t5, 3
%t7 = inttoptr i64 %t6 to i64*

call void @check(i64 %3)
store i64 %3, i64* %t7