Extending a Compiler Backend for Complete Memory Error Detection

Norman A. Rink and Jeronimo Castrillon
Technische Universität Dresden

Automotive – Safety & Security 2017
30 May 2017
Stuttgart
Outline

1. Motivation
2. Error detection, AN encoding
3. The extended compiler backend
4. Evaluation
5. Summary
Outline

1. **Motivation**

2. **Memory error detection, AN encoding**

3. **The extended compiler backend**

4. **Evaluation**

5. **Summary**
Motivation

- Frequency of transient HW faults (aka. soft errors) is increasing.
  - Traditional cause of faults: cosmic rays.
  - Vulnerability is increasing due to smaller feature sizes and lower operating voltages.
  - Dark/dim silicon in memory modules:
    - Extended refresh cycles for DRAM.
    - Lower supply voltage for SRAM.

- Memory errors: ECC memory modules have their limitations.
  - Typically SEC-DED codes (single error correction, double error detection).
  - Large fractions of memory errors cannot be handled by SEC-DED codes (Hwang et al., ASPLOS 2012).
  - ECC not necessarily extended to the entire memory hierarchy. (Load-store queues?)

For energy efficiency

Software-implemented error detection has the flexibility to detect also complex error patterns.

Software-implemented error detection

- Manual incorporation of integrity checks.
  - Laborious and cumbersome.
  - Mixes functional and non-functional requirements.
  - Requires expert knowledge.
  - Error detection limited to anticipated errors.

- Automated, disciplined approaches.
  - Enable comprehensive error detection.
  - Source-to-source transformation.
  - Aspects.
  - Compiler-based approaches:
    - Transformation of machine code.
    - Transformation of intermediate representation (IR).

\[
\begin{align*}
\text{check}(a_0, a_1); \\
... \\
\text{var}0 &= a_0 + b_0; \\
\text{var}1 &= a_1 + b_1; \\
\text{check}(\text{var}0, \text{var}1); \\
... \\
r_0 &= c_0 \times \text{var}0; \\
r_1 &= c_1 \times \text{var}1; \\
\text{check}(r_0, r_1);
\end{align*}
\]
Limitations of software-implemented error detection

To detect errors in memory …

- Which variables are kept in memory?
- When are variables kept in memory?
- Are there any hidden variables that are put into memory?

Ultimately, the compiler knows all this …

... but only very late!

Percentage of dynamic memory accesses (loads) that are present in the program IR or inserted by the compiler backend:

(Twelve test programs, labeled A-L)

In some cases (H, L) virtually all loads are inserted by the compiler backend!
1. Motivation

2. Memory error detection, AN encoding

3. The extended compiler backend

4. Evaluation

5. Summary
Memory error detection by DMR

- DMR (dual modular redundancy).
  - In the context of software-implemented error detection: duplication of data.

```assembly
store i64 %0, i64* %p
...  
%1 = load i64* %p
```

- DMR may introduce race conditions in multi-threaded applications.
  - State-of-the-art work usually assumes memory is protected by ECC (in hardware).
AN encoding

- AN encoding:
  - Fix an integer constant $A$.
  - Encode integer values by multiplying by $A$: $n_{\text{enc}} = n \times A$
  - Decode by dividing by $A$: $n = n_{\text{enc}} / A$
  - Check for errors: $n_{\text{enc}} \mod A = 0$

- Error-detecting capability varies with the constant $A$.
  - Generally, multi-bit errors can be detected by suitable $A$.
  - $A = 58659$ is known to have good properties; can detect up to 5 bit flips, Hoffmann et al., 2015.

- AN encoding introduces large overheads if used to protect operations: several $10x$-$100x$. 
Detection of multi-bit errors in memory, including caches, load-store queues.

Apply AN encoding only to values stored to memory → low overhead due to AN encoding.

AN encoding is applied at the LLVM IR level.

Common approach in software-implemented fault tolerance schemes.

Error detection at the IR level misses memory accesses that are inserted by the compiler backend.
Memory error detection by AN encoding (2)

- Remember this plot:

- Backend for the C programming language inserts memory accesses for:
  - Register spills (spill).
  - Callee-saved registers (csr).
  - Frame pointer (fptr).
  - Return address (return).
  - Function arguments (arg).
  - Jump tables (jt).

- Implement function calls

---

Norman Rink, norman.rink@tu-dresden.de
Outline

1. Motivation
2. Memory error detection, AN encoding
3. The extended compiler backend
4. Evaluation
5. Summary
The extended compiler backend

- Backend for the C programming language inserts memory accesses for:
  - Register spills (spill).
  - Callee-saved registers (csr).
  - Frame pointer (fptr).
  - Return address (return).
  - Function arguments (arg).
  - Jump tables (jt).

- Implement error detection in the compiler backend by DMR:
  - Faster than AN encoding.
  - Keeps function calls efficient.
  - Adds (almost) no register pressure.

- Duplicated store/load:
  - Additional memory accesses are "cheap".
  - Memory locations already in the cache.
  - (All) memory accesses are thread-local.
DMR for register spills

- Comparison memory/register is specific to x86 – more generally, CISC machines.
- RISC machines? → cmp mem/reg might be sensible ISA extension.

```assembly
mov eax, -0x34(ebp)  
mov eax, -0x30(ebp)  
...  
mov -0x30(ebp), eax  
cmp -0x34(ebp), eax  
jne <error_handler>  
add eax, (esi)
```
DMR for function arguments

- Requires co-operation between caller and callee (modified calling convention).
- Library calls still work. (Caller can ignore duplicated arguments).
- The number of arguments passed on the stack may be low (depending on the architecture).
- Modified calling convention: pass return address in register ebx.
- No modification required on, e.g., ARM or MIPS.
1. Motivation

2. Memory error detection, AN encoding

3. The extended compiler backend

4. Evaluation

5. Summary
Fault injection

- Assumptions:
  - Only a single fault affect program execution.
  - Only single bit flips occurs.

  Commonly justified by the rarity of faults.
  \((SEU \text{ – single event upset})\)

- Simulate symptoms of faults by …
  - … flipping a bit in a memory location that is loaded from.

- Perform exhaustive fault injections:
  - Flip a bit in all possible locations in all loads from memory

<table>
<thead>
<tr>
<th>letter</th>
<th>test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>array reduction</td>
</tr>
<tr>
<td>B</td>
<td>bubblesort</td>
</tr>
<tr>
<td>C</td>
<td>CRC-32</td>
</tr>
<tr>
<td>D</td>
<td>DES encryption</td>
</tr>
<tr>
<td>E</td>
<td>Dijkstra (shortest path)</td>
</tr>
<tr>
<td>F</td>
<td>expression evaluation</td>
</tr>
<tr>
<td>G</td>
<td>token lexer</td>
</tr>
<tr>
<td>H</td>
<td>expression parser</td>
</tr>
<tr>
<td>I</td>
<td>matrix multiplication</td>
</tr>
<tr>
<td>J</td>
<td>array copy</td>
</tr>
<tr>
<td>K</td>
<td>quicksort</td>
</tr>
<tr>
<td>L</td>
<td>switch</td>
</tr>
</tbody>
</table>
Full memory error detection

no error detection:

AN encoding and DMR in the backend:
Runtime overhead

AN encoding dominates the slow down.

Slow down dominated by register spills.

Test programs:

<table>
<thead>
<tr>
<th>:normalized runtime:plain</th>
<th>AN</th>
<th>fptr</th>
<th>ctsr</th>
<th>jt</th>
<th>return</th>
<th>arg</th>
<th>spill</th>
<th>all</th>
</tr>
</thead>
<tbody>
<tr>
<td>i386 (32bit)</td>
<td>4.0</td>
<td>3.5</td>
<td>3.0</td>
<td>2.5</td>
<td>2.0</td>
<td>1.5</td>
<td>1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>x86_64 (64bit)</td>
<td>1.6</td>
<td>1.4</td>
<td>1.2</td>
<td>0.8</td>
<td>0.6</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Subset of SPEC CINT2006:

<table>
<thead>
<tr>
<th>normalized runtime:plain</th>
<th>AN</th>
<th>fptr</th>
<th>ctsr</th>
<th>jt</th>
<th>return</th>
<th>arg</th>
<th>spill</th>
<th>all</th>
</tr>
</thead>
<tbody>
<tr>
<td>i386 (32bit)</td>
<td>1.0</td>
<td>0.8</td>
<td>0.6</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x86_64 (64bit)</td>
<td>1.2</td>
<td>1.0</td>
<td>0.8</td>
<td>0.6</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>
Outline

1. Motivation
2. Memory error detection, AN encoding
3. The extended compiler backend
4. Evaluation
5. Summary
Automatic code transformation that introduced memory error detection not comprehensive when applied above the level of machine code.
- Transformations at the level of source code or IR desirable for productivity.

Supporting memory error detection with DMR introduced by the compiler backend …
- ... leads to full memory error detection,
- ... incurs a runtime overhead of
  - 1.50 on i386 (SPEC CINT2006),
  - 1.13 on x86_64 (SPEC CINT 2006).

Absence of vulnerabilities introduced by the compiler backend required for …
- ... (reliable analysis/evaluation of) relaxed fault tolerance schemes.
- ... applications with strict safety and reliability requirements.

The stack has been found a major weakness.
Extending a Compiler Backend for Complete Memory Error Detection

Norman A. Rink and Jeronimo Castrillon
Technische Universität Dresden

Thank you.