COMET: Communication-Optimised Multi-threaded Error-detection Technique

Konstantina Mitropoulou, Vasileios Porpodas and Timothy M. Jones

Computer Laboratory

CASES 2016, Pittsburgh
Outline

- Redundant multi-threaded error detection
- Breakdown of overheads
- COMET optimisations
- Results
### Redundant Multi-threaded Error Detection

#### Original Code
- `r1 = r1 + 16`
- `load r2, (r1)`
- `r2 = r2 + 100`
- `r3 = r1 + 16`
- `store (r3), r2`

#### Software Redundant Multi-threading Error Detection Code
- `r1 = r1 + 16`
- `call enqueue(r1)`
- `load r2, (r1)`
- `call enqueue(r2)`
- `r2 = r2 + 100`
- `call enqueue(r2)`
- `r3 = r1 + 16`
- `call enqueue(r3)`
- `store (r3), r2`
- `r1' = r1' + 16`
- `r1 = call dequeue()`
- `cmp r1, r1'`
- `jmp`
- `r2 = call dequeue()`
- `cmp r2, r2'`
- `jmp`
- `r3 = call dequeue()`
- `cmp r3, r3'`
- `jmp`

#### Original Thread
- `r1 = r1 + 16`
- `load r2, (r1)`
- `r2 = r2 + 100`
- `r3 = r1 + 16`
- `store (r3), r2`

#### Checker Thread
- `cmp r1, r1'`
- `jmp`
- `cmp r2, r2'`
- `jmp`
- `cmp r3, r3'`
- `jmp`
Redundant Multi-threaded Error Detection

<table>
<thead>
<tr>
<th>Original Thread</th>
<th>Checker Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = r1 + 16</td>
<td>r1' = r1' + 16</td>
</tr>
<tr>
<td>call enqueue(r1)</td>
<td>r1 = call dequeue()</td>
</tr>
<tr>
<td>load r2, (r1)</td>
<td>cmp r1, r1'</td>
</tr>
<tr>
<td>call enqueue(r2)</td>
<td>jmp</td>
</tr>
<tr>
<td>r2 = r2 + 100</td>
<td>r2 = call dequeue()</td>
</tr>
<tr>
<td>r3 = r1 + 16</td>
<td>r2' = r2 + 100</td>
</tr>
<tr>
<td>call enqueue(r2)</td>
<td>cmp r2, r2'</td>
</tr>
<tr>
<td>call enqueue(r3)</td>
<td>jmp</td>
</tr>
<tr>
<td>store (r3), r2</td>
<td>r3 = call dequeue()</td>
</tr>
</tbody>
</table>

Software Redundant Multi-threading Error Detection Code
Redundant Multi-threaded Error Detection

**software redundant multi-threading error detection code**

- original thread
  - `r1 = r1 + 16`
  - `call enqueue(r1)`
  - `load r2, (r1)`
  - `call enqueue(r2)`
  - `r2 = r2 + 100`
  - `r3 = r1 + 16`
  - `call enqueue(r2)`
  - `call enqueue(r3)`
  - `store (r3), r2`

- checker thread
  - `r1' = r1' + 16`
  - `r1 = call dequeue()`
  - `cmp r1, r1'`
  - `jmp`
  - `r2 = call dequeue()`
  - `r2' = r2 + 100`
  - `r3' = r1' + 16`
  - `call enqueue(r2)`
  - `r2 = call dequeue()`
  - `cmp r2, r2'`
  - `jmp`
  - `r3 = call dequeue()`
  - `cmp r3, r3'`
  - `jmp`

**communication for checking**
- `r1 = r1 + 16`
- `call enqueue(r1)`
- `load r2, (r1)`
- `call enqueue(r2)`
- `r2 = r2 + 100`
- `r3 = r1 + 16`
- `call enqueue(r2)`
- `call enqueue(r3)`
- `store (r3), r2`

**communication for update**
- `r1' = r1' + 16`
- `r1 = call dequeue()`
- `cmp r1, r1'`
- `jmp`
- `r2 = call dequeue()`
- `r2' = r2 + 100`
- `r3' = r1' + 16`
- `call enqueue(r2)`
- `r2 = call dequeue()`
- `cmp r2, r2'`
- `jmp`
- `r3 = call dequeue()`
- `cmp r3, r3'`
- `jmp`
Redundant Multi-threaded Error Detection

The two threads communicate through a multi-section queue.

- Each section is exclusively used by one thread.
Redundant Multi-threaded Error Detection

The two threads communicate through a multi-section queue.

- Each section is exclusively used by one thread.
- Each thread cannot access the following section if the other thread still uses it.
Breakdown of Communication Overheads
Breakdown of Communication Overheads

- Frequent communication

<table>
<thead>
<tr>
<th>Communication for checking</th>
<th>Communication for update</th>
<th>Software redundant multi-threading</th>
<th>Error detection code</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = r1 + 16</td>
<td>r1’ = r1’ + 16</td>
<td>r1 = call dequeue()</td>
<td>r1 = call dequeue()</td>
</tr>
<tr>
<td>call enqueue(r1)</td>
<td>r2 = call dequeue()</td>
<td>cmp r1, r1’</td>
<td>cmp r2, r2’</td>
</tr>
<tr>
<td>load r2, (r1)</td>
<td>r2’ = r2 + 100</td>
<td>jmp r1, r1’</td>
<td>jmp r2, r2’</td>
</tr>
<tr>
<td>call enqueue(r2)</td>
<td>r3 = r1 + 16</td>
<td>r2 = call dequeue()</td>
<td>r3 = call dequeue()</td>
</tr>
<tr>
<td>r2 = r2 + 100</td>
<td>call enqueue(r3)</td>
<td>r2’ = r2 + 100</td>
<td>cmp r3, r3’</td>
</tr>
<tr>
<td>r3 = r1 + 16</td>
<td>store (r3), r2</td>
<td>r3’ = r1’ + 16</td>
<td>jmp</td>
</tr>
<tr>
<td>call enqueue(r2)</td>
<td></td>
<td>r2 = call dequeue()</td>
<td></td>
</tr>
<tr>
<td>call enqueue(r3)</td>
<td></td>
<td>r3 = call dequeue()</td>
<td></td>
</tr>
<tr>
<td>store (r3), r2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Breakdown of Communication Overheads

- Frequent communication
- Function call overhead
Breakdown of Communication Overheads

- Frequent communication
- Function call overhead
Breakdown of Communication Overheads

- Frequent communication
- Function call overhead
- inline enqueue/dequeue functions
Breakdown of Communication Overheads

- Frequent communication
- Function call overhead
- Overheads of inlining
Breakdown of Communication Overheads

- Frequent communication
- Function call overhead
- Overheads of inlining
  - control-flow overhead
Breakdown of Communication Overheads

- Frequent communication
- Function call overhead
- Overheads of inlining
  - control-flow overhead

enqueue

synchronisation code

enqueue function
dequeue function

synchronisation code

debueque
Breakdown of Communication Overheads

- Frequent communication
- Function call overhead
- Overheads of inlining
  - control-flow overhead

\[
\begin{align*}
\text{r1} & = \text{r1} + 16 \\
call & \text{enqueue(r1)} \\
\text{load} & \text{r2, (r1)} \\
call & \text{enqueue(r2)} \\
\text{r2} & = \text{r2} + 100 \\
\text{r3} & = \text{r1} + 16 \\
call & \text{enqueue(r2)} \\
call & \text{enqueue(r3)} \\
\text{store} & \text{(r3), r2}
\end{align*}
\]

original thread
Breakdown of Communication Overheads

- Frequent communication
- Function call overhead
- Overheads of inlining
  - control-flow overhead

Compared to no-ED

<table>
<thead>
<tr>
<th></th>
<th>no-ED</th>
<th>original-ED</th>
<th>inlined</th>
<th>proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Predictor Misses Compared to no-ED</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.0</td>
<td>112x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Breakdown of Communication Overheads

- Frequent communication
- Function call overhead
- Overheads of inlining
  - control-flow overhead
  - 6 instructions in the critical path
Reduce Communication Overhead

**Optimality:** enqueue and dequeue operations should have 2 instructions overheads:

- a store instruction for writing the data to the queue
- an addition to increment the enqueue or dequeue pointer
Reduce Communication Overhead

**Lynx** is a SP/SC queue which relies on memory protection system. In this way, each enqueue/dequeue operation has 2 instructions overhead.
Lynx is a SP/SC queue which relies on memory protection system. In this way, each enqueue/dequeue operation has 2 instructions overhead.
Reduce Communication Overhead

**Lynx** is a SP/SC queue which relies on memory protection system. In this way, each enqueue/dequeue operation has 2 instructions overhead.

- **SSRZ**: Section Synchronisation Red Zone
- **PRRZ**: Pointer Rotation Red Zone

Original pointer `original_ptr` and checker pointer `checker_ptr`
Lynx is a SP/SC queue which relies on memory protection system. In this way, each enqueue/dequeue operation has 2 instructions overhead.

- **SSRZ**: Section Synchronisation Red Zone
- **PRRZ**: Pointer Rotation Red Zone

original_ptr
checker_ptr
Reduce Communication Overhead

![Diagram showing sections and pointers]

- **section 1**
- **section 2**
- **SSRZ**
- **PRRZ**
- **original_ptr**
- **checker_ptr**

[http://www.cl.cam.ac.uk/~km647/]
Reduce Communication Overhead

- The exception is captured by Lynx’s handler
- Lynx’s handler does the job of the synchronization code
Reduce Communication Overhead

COMET optimises Lynx queue by applying the following compiler optimisations:

- Simplify Lynx design by using a fixed register (R15) for the enqueue/dequeue pointers.
- The queue has two fixed red-zones.
- The handler has less things to do.
- We can apply more optimisations.
Reduce Communication Overhead

COMET optimises Lynx queue by applying the following compiler optimisations:

- Simplify Lynx design by using a fixed register (R15) for the enqueue/dequeue pointers:
Reduce Communication Overhead

COMET optimises Lynx queue by applying the following compiler optimisations:

- Simplify Lynx design by using a fixed register (R15) for the enqueue/dequeue pointers:
  - The queue has two fixed red-zones
Reduce Communication Overhead

COMET optimises Lynx queue by applying the following compiler optimisations:

- Simplify Lynx design by using a fixed register (R15) for the enqueue/dequeue pointers:
  - The queue has two fixed red-zones
  - The handler has less things to do
Reduce Communication Overhead

COMET optimises Lynx queue by applying the following compiler optimisations:

- Simplify Lynx design by using a fixed register (R15) for the enqueue/dequeue pointers:
  - The queue has two fixed red-zones
  - The handler has less things to do
  - We can apply more optimisations

original_ptr

section 1

section 2

checker_ptr
Reduce Communication Overhead

COMET optimises Lynx queue by applying the following compiler optimisations:

- Simplify Lynx design

```assembly
original thread
r1 = r1 + 16
call enqueue(r1)
r2 = r2 + 100
call enqueue(r2)
store (r1), r2

error detection code with enqueue and dequeue calls
r1 = call dequeue()
cmp r1, r1'
jmp
r2' = r2' + 100
r2 = call dequeue()

original thread
r1' = r1' + 16

checker thread
r1 = call dequeue()
cmp r1, r1'
jmp
r2' = r2' + 100
r2 = call dequeue()

original thread
r1 = r1 + 16
load r1, (r15)
r15 = r15 + 8
r15 = r15 + 8

checker thread
r1' = r1' + 16
load r1, (r15)
r15 = r15 + 8
r15 = r15 + 8
```

```assembly
original thread
r1 = r1 + 16
store (r15), r1
r15 = r15 + 8
r15 = r15 + 8

error detection code with enqueue and dequeue calls
r2 = r2 + 100
store (r15), r2
r1 = r1 + 16
store (r1), r2
r15 = r15 + 8
r15 = r15 + 8
```

```assembly
original thread
r1' = r1' + 16
load r1, (r15)
r15 = r15 + 8
r15 = r15 + 8

checker thread
r1' = r1' + 16
load r1, (r15)
r15 = r15 + 8
r15 = r15 + 8
```
Reduce Communication Overhead

COMET optimises Lynx queue by applying the following compiler optimisations:

- Simplify Lynx design
- Address offset fusion optimisation

\[
\begin{align*}
\text{original thread} & \quad \text{original thread} \quad \text{checker thread} \\
\text{inlined code with COMET} & \quad \text{inlined code with COMET and offset optimisation}
\end{align*}
\]
Reduce Communication Frequency

Packed Checking: COMET packs the communication operations for each store instruction:

\[
\begin{align*}
\text{original thread} & : & r_1 &= r_1 + 16 \\
& & \text{store } (r_{15}), r_1 \\
& & r_2 &= r_2 + 100 \\
& & \text{store } (r_{15} + 8), r_2 \\
& & \text{store } (r_1), r_2 \\
\text{checker thread} & : & r_1' &= r_1' + 16 \\
& & \text{load } r_1, (r_{15}) \\
& & \text{cmp } r_1, r_1' \\
& & \text{jmp} \\
& & r_2' &= r_2' + 100 \\
& & \text{load } r_2, (r_{15} + 8) \\
& & \text{cmp } r_2, r_2' \\
& & \text{jmp} \\
\end{align*}
\]

inlined code with COMET and offset optimisation

COMET with packed checking optimisation

\[
\begin{align*}
\text{original thread} & : & r_1 &= r_1 + 16 \\
& & \text{store } (r_{15}), r_1 \\
& & r_2 &= r_2 + 100 \\
& & \text{store } (r_{15} + 8), r_2 \\
& & r_2 = r_1 \text{ XOR } r_2 \\
& & \text{store } (r_{15}), r_2 \\
\text{checker thread} & : & r_1' &= r_1' + 16 \\
& & \text{load } r_1, (r_{15}) \\
& & \text{cmp } r_1, r_1' \\
& & \text{jmp} \\
& & r_2' &= r_2' + 100 \\
& & \text{load } r_2, (r_{15}) \\
& & \text{cmp } r_2, r_2' \\
& & \text{jmp} \\
\end{align*}
\]
Experimental Setup

- **Compiler:**
  - GCC-4.9.0

- **Performance evaluation:**
  - Intel Core i5-4570 @ 3.2GHz desktop machine

- **Fault-coverage evaluation:**
  - in-house tool based on gdb

- **Benchmark suite:**
  - NAS benchmarks
Performance Evaluation

The graph shows the performance evaluation of different models and datasets. The x-axis represents different datasets: BT, CG, EP, IS, SP, Geo. The y-axis represents the performance metric ranging from 0.00 to 9.00. The models are represented by different colors: no-ED, simplify, MTED, COMET-U, COMET. The graph compares the performance of these models across the datasets.
Fault Coverage Evaluation

• Single-Event Upset (SEU) fault model
• Monte Carlo simulation:
  1. count the dynamic instructions
  2. randomly pick one instruction
  3. randomly pick one bit of the instruction’s output
  4. run the program
  5. repeat steps 2 to 4 for 300 times

• Type of errors:
  • detected errors are the ones that COMET detects
  • masked errors do not alter program’s output
  • exceptions can be detected by a specialised exception handler
  • corrupt errors change program’s output
  • timeout errors result in infinite execution of the program
Fault Coverage Evaluation

timeout  corrupt  exceptions detected  masked

BTCGEP IS SP  BTCGEP IS SP  BTCGEP IS SP

BTCGEP IS SP no-ED  BTCGEP IS SP COMET-U  BTCGEP IS SP COMET

http://www.cl.cam.ac.uk/~km647/
Summary

- The communication is the main performance bottleneck of redundant multi-threaded error detection
Summary

- The communication is the main performance bottleneck of redundant multi-threaded error detection
- COMET can potentially reduce the communication overhead down to one instruction
Summary

- The communication is the main performance bottleneck of redundant multi-threaded error detection
- COMET can potentially reduce the communication overhead down to one instruction
- COMET reduces the communication frequency
Summary

- The communication is the main performance bottleneck of redundant multi-threaded error detection
- COMET can potentially reduce the communication overhead down to one instruction
- COMET reduces the communication frequency
- COMET improves performance by 31.4% on average

http://www.cl.cam.ac.uk/~km647/
Summary

- The communication is the main performance bottleneck of redundant multi-threaded error detection
- COMET can potentially reduce the communication overhead down to one instruction
- COMET reduces the communication frequency
- COMET improves performance by 31.4% on average
- The proposed optimisations do not affect the fault-coverage