Protecting the Dynamic Dispatch in C++ by Dependability Aspects

Christoph Borchert
Horst Schirmeier
Olaf Spinczyk

TU Dortmund, Germany
Embedded System Software
Agenda

- A meteorologic case study
  - Dependability assessment by fault injection

- Dependability Aspects

- Evaluation
Simplified Weather-Station main loop:

repeat until hell freezes over:
  for each Sensor s:
    s.measure()
  for each Actor a:
    for each Sensor s:
      a.process(s)
Baseline Dependability Assessment

- **Fault model:** Transient single-bit errors in data memory
  - Assumption: Text / rodata stored in non-volatile memory

- **Fault injection:** *Fail* tool
  - Weather-station software modified to run in x86 emulator
    - Dummy sensor classes, new CGA display driver

- **Question:** Which data/BSS segment areas are the **most susceptible** to faults?
  - **Working hypothesis:** Temporary data inconsistencies are often far less important to the user than availability!
  - **Our definition of availability in this case study:** Weather station stays available if it continues executing its main loop
Dep. Assessment: Results

White: OK
Black: #loops not OK
Blue: Jump outside text segment
Red: Trap/MCE
Agenda

- A meteorologic case study
- Dependability Aspects
  - Generic protection of vptrs
- Evaluation
Background: OOP in C++

- Inheritance and *virtual* functions

```
Sensor
virtual measure()

Wind
virtual measure()
```

```
Sensor *s = new Wind;
s->measure();
```
Background: OOP in C++

- Inheritance and *virtual* functions

```cpp
Sensor
virtual measure()

Wind
virtual measure()

Sensor *s = new Wind;
s->measure();

Dynamic dispatch
```
Background: OOP in C++

- Inheritance and *virtual* functions

```
Sensor
virtual measure()
```

```
Wind
virtual measure()
```

```
Sensor *s = new Wind;
s->measure();
```

Dynamic binding at runtime
- Depending on the object's type
Dynamic Dispatch in C++

- Common compilers generate **vtables**
  - Objects contain **vptrs** that refer to vtables

**Diagram:**
- Pointer to object
  - **vptr**
  - Object's fields
  - RAM

- Virtual table
  - 1\textsuperscript{st} virtual method
  - 2\textsuperscript{nd} virtual method
  - ROM

- Virtual functions → **indirect jump instructions**
Dependability Mechanism

- Forward error correction
  - Replicate vptrs (triple-modular redundant)

- On virtual-function call:
  - Voting
    - Repair deviating vptr
  - Acceptance test (optional)
    - Does the vptr refer to memory where vtables reside?
    - Additional backward error correction, e.g. rebooting

- Modular implementation
  - Aspect-oriented Programming (AOP) with AspectC++
AspectC++ Compilation Process

Weather-Station Software

Dependability Aspect for Vptrs

Source-to-Source Compilation

Weather-Station Software
aspect VPTR_Protection {

};
aspect VPTR_Protection {
    pointcut critical() = "Actor" || "Sensor";
};
aspect VPTR_Protection { 
  
  pointcut critical() = "Actor" || "Sensor"; 

  advice critical() : slice class 
  {
    const void *redundant_vptr1, *redundant_vptr2;
    void init_vptr();
    bool check_vptr();
  };

};
aspect VPTR_Protection {

  pointcut critical() = "Actor" || "Sensor";

  advice critical() : slice class {
    const void *redundant_vptr1, *redundant_vptr2;
    void init_vptr();
    bool check_vptr();
  };

  advice construction(derived(critical())) : before() {
    tjp->target()->init_vptr();
  }

}; advice (what)
aspect VPTR_Protection {

pointcut critical() = "Actor" || "Sensor";

advice critical() : slice class {
    const void *redundant_vptr1, *redundant_vptr2;
    void init_vptr();
    bool check_vptr();
};

advice construction(derived(critical())) : before() {
    tjp->target()->init_vptr();
}

};

advice (what) advice type (when)
aspect VPTR_PROTECTION {
  pointcut critical() = "Actor" || "Sensor";

  advice critical() : slice class {
    const void *redundant_vptr1, *redundant_vptr2;
    void init_vptr();
    bool check_vptr();
  }

  advice construction(derived(critical())) : before() {
    tjp->target()->init_vptr();
  }
};

<table>
<thead>
<tr>
<th>aspect</th>
<th>pointcut expression (where)</th>
<th>slice introduction (structure)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aspect</td>
<td>VPTR_PROTECTION</td>
<td></td>
</tr>
<tr>
<td>pointcut</td>
<td>critical() = &quot;Actor&quot;</td>
<td></td>
</tr>
<tr>
<td>advice</td>
<td>critical() : slice class</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const void *redundant_vptr1, *redundant_vptr2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>void init_vptr();</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bool check_vptr();</td>
<td></td>
</tr>
<tr>
<td>advice</td>
<td>construction(derived(critical())) : before()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tjp-&gt;target()-&gt;init_vptr();</td>
<td></td>
</tr>
</tbody>
</table>
aspect VPTR_Protection {
    pointcut critical() = "Actor" || "Sensor";
    advice critical() : slice class {
        const void *redudant_vptr1, *redudant_vptr2;
        void init_vptr();
        bool check_vptr();
    };
    advice construction(derived(critical())) : before() {
        tjp->target()->init_vptr();
    }
    advice call(derived(critical())) &&
        call("virtual % ...:%(...)") : before() {
        if(!tjp->target()->check_vptr())
            vptr_panic();
    }
};
advice (what)  pointer to object (context)  advice type (when)
Agenda

● A meteorologic case study
● Dependability Aspects
● Evaluation
Evaluation

Faults in both vptrs and object pointers (linked-list heads and "next") are fatal.

Vptrs resilient against single-bit flips.

Object pointers (linked-list heads and "next") still highly susceptible.

Almost all faults in object pointers are detected (e.g., initiating a reboot).
Evaluation: Effectiveness

- Improvements: Unprotected → **Vptr protection variant**
  - Bit flips in Vptrs: 77.5% crash → 0% crash
  - Object pointers **still highly susceptible**: 99.0% crash → 78.7% crash, 20.3% detected (green)
Evaluation: Effectiveness

- Vptr protection → **acceptance test variant**
- Bit-flips in object pointers:
  78.7% / 20.3% crash/detection → **2.6% / 96.4% crash/detection**
Evaluation: Efficiency

- Memory overhead:
  - *text*: 64-96 bytes per call site
  - *BSS*: 8 bytes (modulo alignment changes) per protected object
  \[ \sum = 6.8\% / 9.4\% \]

- Runtime overhead:
  - +6.3\% / +7.7\% more instructions executed
Conclusions

- Dynamic Dispatch in C++ is a frequent point of failure
  - Not only the vptr is sensitive to bit errors, but also pointers to objects with virtual functions

- Our Dependability Aspect prevents:
  - 67.1% of all system crashes (in our case study)
    - 100% of bit errors in vptrs
    - 96.4% of bit errors in object pointers
  - At less than 10% overhead (runtime and memory)

- Good alternative to ECC memory: Only small fraction (2.4%) of program state “mission critical”