On Hardware Resource Consumption for Aspect-Oriented Implementation of Fault Tolerance

Ruben Alexandersson, Peter Öhman
Department of Computer Science and Engineering,
Chalmers University of Technology
Göteborg, Sweden

{ruben, peter.ohman}@chalmers.se
Overview of case study

Objective

- Assess hardware overhead associated with using aspect-oriented programming (AOP) for implementing fault tolerance in software

Control application

- Automotive Brake-By-Wire system
- Implemented in C, compiled using GCC

Transient hardware faults tolerated by means of:

- Time-redundant execution
- Control flow checking
Aspect oriented programming

Aspect

Fault tolerance code

Weaving directives

Target program source code

Complete program
Advantages of using AOP for implementing fault tolerance

- **Separation of cross-cutting concerns**
  - The source code for the primary functionality and the source code for fault tolerance can be developed separately

- **No need for special compiler**
  - Existing, thoroughly validated compilers can be used
Two implementations of control flow checking and time-redundant execution

- AOP implementation using an extended version of AspectC++

- A reference implementation manual programming in standard C
Joinpoints in AspectC++

- AspectC++ supports Call & Execution joinpoints
- Our extended version additionally supports Get & Set joinpoints
Implementation of fault tolerance: Function call

```c
if (Cfc_block!=0)
    Errorcode = 8;
Cfc_block = 1;
FT_run = 0;
apa();
FT_run = 1;
apa();
if (Cfc_block!=1)
    Errorcode = 8;
Cfc_block = 0;
```

- Control flow checking
- Redundant execution
Implementation of fault tolerance: Function body

```c
apa()
{
    if (Cfc_block!=1)
        Errorcode = 8;
    Cfc_block = 3;
    ...

    int i = Global;
    ...

    int i = Global[FT_run];
    ...

    if (Cfc_block!=3)
        Errorcode = 8;
    Cfc_block = 1;
}
```

Control flow checking
Time redundant execution
Compiler optimization levels

- Low compiler optimization
  - GCC ... -finline

- High compiler optimization
  - GCC ... -O3 -fno-strict-aliasing
## Initial results

<table>
<thead>
<tr>
<th></th>
<th>Low compiler optimization</th>
<th></th>
<th>High compiler optimization</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nr. of instructions</td>
<td>% overhead</td>
<td>Nr. of instructions</td>
<td>% overhead</td>
</tr>
<tr>
<td>No fault tolerance</td>
<td>790</td>
<td>0%</td>
<td>294</td>
<td>0%</td>
</tr>
<tr>
<td>Manual C</td>
<td>1865</td>
<td>136%</td>
<td>746</td>
<td>154%</td>
</tr>
<tr>
<td>AspectC++</td>
<td>5759</td>
<td>629%</td>
<td>1960</td>
<td>567%</td>
</tr>
</tbody>
</table>

Nr. of instructions = Number of executed machine code instructions in a complete control loop
Weaver optimizations

- Make the aspect class static when possible
- Remove the joinpoint struct
inline void __set__ZN4(a,b){
    Joinpoint tjp(a,b);
    ::Aspect::aspectof()->
    __a0_around (tjp); 
}

static Aspect *aspectof () {
    return &__instance; }

void __a0_around (JoinPoint *tjp){
    ... // The actual advice code }

inline void __set__ZN4(a,b){
    Joinpoint tjp(a,b);
    ::Aspect::aspectof()->
    __a0_around (tjp); 
}

static Aspect *aspectof () {
    return &__instance; }

static void __a0_around (JoinPoint *tjp){
    ... // The actual advice code }
Optimization #2: Remove joinpoint struct

```cpp
inline void __set__ZN4(a,b){
    Joinpoint tjp(a,b);
    ::Aspect::__a0_around (tjp); }

static void __a0_around (JoinPoint *tjp){
    ... // The actual advice code }
```

```
inline void __set__ZN4(a,b){
    Joinpoint tjp(a,b),
    ::Aspect::__a0_around (a,b); }

static void __a0_around (int* a, int b){
    ... // The actual advice code }
```
# Results for optimized weaver

<table>
<thead>
<tr>
<th></th>
<th>Low compiler optimization</th>
<th>High compiler optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nr. of instructions</td>
<td>% overhead</td>
</tr>
<tr>
<td>No fault tolerance</td>
<td>790</td>
<td>0%</td>
</tr>
<tr>
<td>Manual C</td>
<td>1865</td>
<td>136%</td>
</tr>
<tr>
<td>AspectC++</td>
<td>2396</td>
<td>203%</td>
</tr>
</tbody>
</table>

**Nr. of instructions** = Number of executed machine code instructions in a complete control loop
Conclusions of case study

- The use of AOP do not lead to prohibitively high overheads.
- When combined with compiler optimization AOP does not impose more overhead than manual programming in C.
- This, and its potential for reducing design and maintenance costs, makes AOP a promising approach for implementing software-based fault tolerance.

Future work
- Evaluate and compare error coverage for the manual C programming and AOP implementations by fault injection.
Thank You!

Questions?