Error Detector Placement for Soft Computation

Anna Thomas and Karthik Pattabiraman
University of British Columbia (UBC)
Soft Computing Applications

- Applications in AI, multimedia processing...
- Expected to dominate future workloads [Dubey’07]

Original image (left) versus faulty image from JPEG decoder
Egregious Data Corruptions

- Large or unacceptable deviation in output
- Based on fidelity metric (e.g., PSNR)

EDC image (PSNR of 11.37) of JPEG vs Non-EDC image (PSNR of 44.79)
Why Software Solutions?

- Device/Circuit Level
- Architectural Level
- Operating System Level
- Application Level

Impactful Errors

Overheads
Goal

Detect EDC causing faults

Pre-emptive

Selective

Application Execution

EDC

Non-EDC

Benign
Why detect EDC Causing Faults?

- Unacceptable outcome to the end user
- 92% : tolerable outcomes

Blindly detecting all faults is wasteful
Outline

- Motivation
- Approach
- Experimental Setup and Results
- Conclusion
Approach

- Step 1: Separate EDCs from Non-EDCs by fault injections
- Step 2: Heuristics identifying code regions prone to EDC causing faults
- Step 3: Automated algorithm for detector placement
Step 1: Initial Study

- Fault injection using LLFI [Thomas’13]
- Correlation between data type – fault outcome
Data Categorization of Fault Outcomes

High correlation between Control Non-Pointer and EDC/Non-EDC
Step 2: Heuristics

```c
void conv422to444 (char *src, char *dst, int height, int width, int offset) {
    for(j=0; j < height; j++){
        for(i=0; i < width; i++) {
            im1 = (i < 1) ? 0 : i – 1
            ...
            dst[im1] = Clip[(21*src[im1])>>8];
        }
        if( j + 1 < offset) { 
            src += w;
            dst += width; 
        }
    }
}
```
Step 2: Heuristics

Faults affecting branches with large amount of data within branch body, has a higher likelihood of resulting in EDC outcomes

```
void conv422to444 (char *src, char *dst, int height, int width, int offset) {
    for(j=0; j < height; j++){
        for(i=0; i < width; i++) {
            im1 = (i < 1) ? 0 : i – 1
            ...
            dst[im1] = Clip[(21*src[im1])>>8];
        }
        if( j + 1 < offset) {
            src += w;
            dst += width;
        }
    }
}
```
Step 2: Heuristics

Faults affecting branches with large amount of data within branch body, has a higher likelihood of resulting in EDC outcomes

```
void conv422to444 (char *src, char *dst, int height, int width, int offset) {
    for(j=0; j < height; j++) {
        for(i=0; i < width; i++) {
            im1 = (i < 1) ? 0 : i - 1
            ...
            dst[im1] = Clip[(21*src[im1])>>8];
        }
        if( j + 1 < offset) {
            src += w;
            dst += width;
        }
    }
}
```
Heuristics: Main Take Aways

- Program level, applied to soft computing applications
- Dependent on size of data affected
- Based entirely on static analysis of code
Step 3: Algorithm

Pre-emptive, Selective detection of EDC causing faults

Application Source Code

Performance Overhead

Execution Profile

Automated Detector Placement Algorithm

Data Variables or Locations to Protect
Step 3: Algorithm

Pre-emptive, Selective detection of EDC causing faults

Application Source Code

Compiler → IR → EDC Ranking Algorithm → Selection Algorithm → Data Variables or Locations to Protect → Backward slice replication

Performance Overhead
Execution Profile
Selection Algorithm

void F1(...) {
    for(i = k; i < num; i++)
        ...
}

void F2(void *src, void *dst, ...) {
    if(src < dst + offset)
        return;
    for(j = 0; j < width; j++)
        ....
}

void F3(void * src, void *dst, ...) {
    for(i =0; i < n; i++){
        if( k < r )
            src += k;
        ...
    }
}

void F4(...) {
    ...
    k = F5 (...);
}
Selection Algorithm

void F1(...) {
    for(i = k; i < num; i++)
        ...
}
void F2(void *src, void *dst, ...) {
    if(src < dst + offset)
        return;
    for(j = 0; j < width; j++)
        ....
}
void F3(void *src, void *dst, ...) {
    for(i = 0; i < n; i++){
        if( k < r )
            src += k;
        ...
    }
void F4(...) {
    ...
    k = F5 (...);
}
Selection Algorithm

void F1(...) {
    for(i = k; i < num; i++) B1
    ...
}
void F2(void *src, void *dst, ...) {
    if(src < dst + offset) B2
    return;
    for(j = 0; j < width; j++) B3
    ....
}
void F3(void *src, void *dst, ...) {
    for(i = 0; i < n; i++)
        if(k < r)
            src += k;
    ...
}
void F4(...) {
    ...
    k = F5 (...); C1
}

Initial Study

Heuristics

Algorithm

<table>
<thead>
<tr>
<th>Inst</th>
<th>Rank</th>
<th>P.O. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>B1</td>
<td>2.5</td>
<td>1</td>
</tr>
<tr>
<td>B3</td>
<td>2.5</td>
<td>2</td>
</tr>
<tr>
<td>C1</td>
<td>0.8</td>
<td>2</td>
</tr>
</tbody>
</table>

P.O. = 5%

Selection Window  Look-ahead Window

2%
4%

Inst

Rank

P.O. (%)
Selection Algorithm

void F1(...) {
    for(i = k; i < num; i++)
        ...
}
void F2(void *src, void *dst, ...) {
    if(src < dst + offset)
        return;
    for(j = 0; j < width; j++)
        ....
}
void F3(void *src, void *dst, ...) {
    for(i = 0; i < n; i++){
        if(k < r)
            src += k;
        ...
    }
void F4(...) {
    ...
    k = F5(...);
}
Selection Algorithm

```c
void F1(...) {
    for(i = k; i < num; i++)
        ...
}

void F2(void *src, void *dst, ...) {
    if(src < dst + offset)
        return;
    for(j = 0; j < width; j++)
        ....
}

void F3(void *src, void *dst, ...) {
    for(i = 0; i < n; i++)
        if(k < r)
            src += k;
        ...
}

void F4(...) {
    ...
    k = F5 (...);
}
```

<table>
<thead>
<tr>
<th>Inst</th>
<th>Rank</th>
<th>P.O. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B4</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>B5</td>
<td>2.5</td>
<td>1.5</td>
</tr>
<tr>
<td>P1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>C1</td>
<td>0.8</td>
<td>2</td>
</tr>
</tbody>
</table>

Selection Window
Look-ahead Window

P.O. = 5%
Selection Algorithm

void F1(...) {
    for(i = k; i < num; i++) B1
    ...
}

void F2(void *src, void *dst, ...) {
    if(src < dst + offset) B2
    return;
    for(j = 0; j < width; j++) B3
    ....
}

void F3(void *src, void *dst, ...) {
    for(i = 0; i < n; i++) {
        if(k < r) B4
        src += k; P1
    }
}

void F4(...) {
    ...
    k = F5 (...); C1
}

Detector Locations under 5% P.O.
B1: i < num
B3: j < width
B5: k < r

P.O. = 5%
Outline

- Motivation
- Approach
- Experimental Setup and Results
- Conclusion
Experimental Setup

- Six Benchmarks from MediaBench, Parsec Suite
  - Fidelity Metric: PSNR, scaled distortion [Misailovic’12]

- Performed fault injections using LLFI
  - 2000 fault injections, one fault per run (1.3% at 95% CI)

- Measured coverage under varying performance overhead bounds
LLFI Experiment Framework

Choose dynamic data instance at random

Inject Random Single bit flip

Execute Application

Compare faulty & fault-free outcome

Exception

Value Change

No Change

Benign

EDC

High deviation

Fidelity Metric

Low Deviation

Non-EDC

Exception

Value Change

No Change

Benign

EDC

High deviation

Fidelity Metric

Low Deviation

Non-EDC
Coverage Evaluation

\[
EDC \text{ Coverage} = \frac{\text{Number of Detected EDCs}}{\text{Total Number of EDCs}}
\]

Pre-emptive Detection

Average EDC Coverage of 82% versus 56% under 10% performance overhead
Coverage Evaluation

Lower is better
Coverage Numbers in Perspective

Initial Coverage: 95.95 %
Coverage With our Technique: 99.3 %
Outline

- Motivation
- Approach
- Experimental Setup and Results
- Conclusion
Conclusion

- **Soft Computing applications**
  - Egregious Data Corruptions (EDCs) are unacceptable outcomes

- **Detector Placement Technique: pre-emptive and selective**
  - Program level Heuristics – static analysis of code
  - Automated algorithm - varying performance overhead bounds

- **Future Work**
  - Actual detectors coverage
  - Compiler optimizations effect on technique


**Contact**: annat@ece.ubc.ca