Error Resilient Systems and Approximate Computing: Conjoined Twins Separated at Birth?

Karthik Pattabiraman,

Anna Thomas, Jiesheng Wei, Bo Fang, Guanpeng Li and Qining Lu
University of British Columbia (UBC)
http://blogs.ubc.ca/karthik/
What this talk is about

• **Approximate computing**: Exact results don’t matter - compromise on correctness

• **Error-Resilient Systems**: Can we produce correct results in the presence of hardware faults?
What this talk is about

• **Approximate computing**: Exact results don’t always matter - compromise on 100% correctness

• **Error-Resilient Systems**: Can we produce correct results in the presence of hardware faults?

• **TLDR**: The two fields have evolved independently. Can they be bridged, and learn from each other?
Error Resilient Systems: History

• Long history in fault tolerant systems (1960s)
  – Targeted mission-critical or safety-critical systems such as space missions, finance and healthcare
  – Costs (power/performance) not very important

• Around 2005, marked change in paradigm
  – Commodity systems where cost matters
  – Mostly software driven (with hardware support)
  – Error Detectors (UIUC), SWAT (UIUC), SWIFT (Princeton), Shoestring (Michigan) etc.
Approximate Computing: History

• Long history going back to Von Neumann’s notion of probabilistic computing in 1960s
  – Confined to the circuits or electronic devices level
  – Confined to selected application classes

• Starting around 2009, adoption of approximate computing at higher levels of the system stack
  – ERSA (Stanford), Flikker (UBC/Microsoft), Relax (U. Wisconsin), and EnerJ (U. Washington)
  – Main motivation: Energy savings
Similarities between the two fields

• Both consider hardware faults, primarily transients
  – In error resilient systems, faults are introduced by natural phenomena (e.g., cosmic rays, alpha particles)
  – In approximate computing, faults are introduced by deliberate lowering of voltages/frequencies etc.

• Both fields have notions of partial correctness
  – In error resilient systems, these are defined by SDCs or Silent Data Corruptions (e.g., SDC rate, coverage of SDCs)
  – In approximate computing, these are defined by application specific fidelity metrics (e.g., Peak Signal to Noise Ratio)
Differences between the two fields

• Approximate computing assumes that it is possible to control when & where faults occur
  – E.g., Only in low-refresh DRAM, only in some ALUs
  – Resilient systems assume that the number of fault occurrences is bounded (e.g., one fault per execution)

• Approximate computing assumes programmers will explicitly mark low (high)-fidelity data/code
  – Typically through type systems or annotations
  – Resilient systems automate this based on heuristics
Can we bridge this gap?

Example of a system that I have worked on

Egregious Data Corruptions (EDCs) in Approximate Computing Applications [DSN’13]
EDCs: What are they?

- Large or unacceptable deviation in output

EDC image (PSNR 11.37) Vs. Non-EDC image (PSNR 44.79)
EDCs: Goal

➢ Selectively detect EDC causing faults, but not others
EDCs: Main Idea

Our prior work: EDCs are caused by corruption of a small fraction of program data [Flikker - ASPLOS’11]

This work: Critical data can be identified using static and dynamic analysis, without any programmer annotations
EDCs: Fault model

- Transient hardware faults
  - Caused by particle strikes, supply noise

- Our Fault Model
  - Assume one fault per application execution
  - Processor registers and execution units
  - Memory and cache protected with ECC
  - Control logic protected with other methods
EDCs: Initial Study

Correlation between program data use & fault outcome

- Monitor Control/Pointer Data
- Instrument code
- Fault Injection

Performed using LLFI fault injector [DSN’14], at the LLVM IR code level
EDCs: Initial Study

Outcome %

- CRASH: 23%
- EDC: 6%
- Non-EDC: 43%
- BENIGN: 28%
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
      ...
      ...
    }
    if (j + 1 < offset) {
      src += w;
      dst += width;
    }
  }
}

Faults affecting branches with large amount of data within their bodies have a higher likelihood of resulting in EDC outcomes.
EDCs: Algorithm

Application Source Code

Compiler

IR

EDC Ranking Algorithm

Performance Overhead

Representative inputs

Selection Algorithm

Data Variables or Locations to Protect

Backward slice replication
EDCs: Detection Coverage

Average EDC Coverage of 82% at 10% performance overhead

Higher is better
EDCs: Selectivity

Average Benign and Non-EDC Coverage of 10 to 15% for overheads from 10 to 25%

Lower is better
How can the two fields learn from each other?

- **Approximate computing applications**
  - More generic fault models (more resilient)
  - More automation based on heuristics (say)

- **Error Resilient Systems**
  - Precisely quantify acceptable & unacceptable outcomes using the programmer’s help
  - Use of type theory or model checking to reason about whether the protection is adequate or not
Can we bridge this gap?

Challenges

1. Different constraints
2. Different methodologies