

AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware

Ute Schiffel Christof Fetzer Martin Süßkraut

Technische Universität Dresden
Institute for System Architecture
ute.schiffel@inf.tu-dresden.de

SafeComp 2009
International Conference on Computer Safety, Reliability and Security

Introduction

Situation:

- commodity hardware is unreliable
- economic pressure
 - ⇒ commodity hardware in safety critical systems
- **unknown unknown:** silently corrupted output?

Introduction

Situation:

- commodity hardware is unreliable
- economic pressure
⇒ commodity hardware in safety critical systems
- **unknown unknown:** silently corrupted output?

Our Objective:

- *failure virtualization:*
silently corrupted output ⇒ fail-stop (crash)
- *practicability:*
 - ▶ hardware independence
 - ▶ completeness
 - ▶ ease of use → encoding compiler
 - ▶ justifiable performance impact

How Reliable is Commodity Hardware?

- [Borkar, 2005]: *“The reliability challenge”*
 - ▶ large fraction of unusable transistors
 - ▶ frequent soft-errors
 - ▶ faster aging transistors

How Reliable is Commodity Hardware?

- [Borkar, 2005]: *“The reliability challenge”*
 - ▶ large fraction of unusable transistors
 - ▶ frequent soft-errors
 - ▶ faster aging transistors
- [Dixit et al., 2009]: neutron beam testing revealed:
 - ▶ decreased memory error rates per bit, **but** larger memories
 - ▶ more multiple-cell upsets
 - ▶ more errors in logical circuits than in memory

How Reliable is Commodity Hardware?

- [Borkar, 2005]: *“The reliability challenge”*
 - ▶ large fraction of unusable transistors
 - ▶ frequent soft-errors
 - ▶ faster aging transistors
- [Dixit et al., 2009]: neutron beam testing revealed:
 - ▶ decreased memory error rates per bit, **but** larger memories
 - ▶ more multiple-cell upsets
 - ▶ more errors in logical circuits than in memory
- [Schroeder et al., 2009]: observed Google’s server fleet:
 - ▶ higher memory error rates than expected:
“25,000 to 70,000 errors per billion device hours per Mbit”
 - ▶ memory errors dominated by hard errors

Detection of Errors – How?

We chose **Arithmetic Code** because detection of:

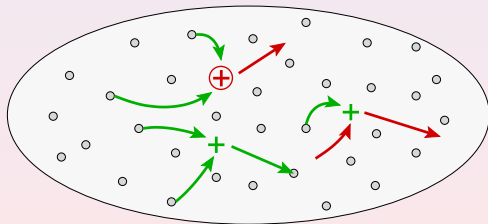
- data modifications, and
- computation errors

Presentation Outline

- introduction to Arithmetic Code: AN-code
- encoding problems and solutions
- performance impact and detection capabilities
- What next?

Arithmetic Codes

- redundant representation of numbers
- conserved by correct arithmetic operations
- destroyed by faulty arithmetic operations



○ domain of possible code words

○ valid code word

+ fault-free addition operation

⊕ faulty addition operation

AN-code

- encoding x_f : $x_c = A * x_f$
- decoding: $x_f = \frac{x_c}{A}$
- code checking: $x_c \bmod A == 0 ?$
- choice of A : large prime number
- probability of undetectable data modification:

$$p = \frac{\text{number of valid code words}}{\text{number of possible code words}} \approx \frac{1}{A}$$

Detectable Errors

without faults

Original source code:

$$x = y + z$$

Encoded version:

$$\begin{aligned}x_c &= y_c + z_c \\ &= A * y_f + A * z_f\end{aligned}$$

Code checking:

$$x_c \bmod A = 0 \quad ?$$

Detectable Errors

operation error: faulty addition

Original source code:

$$x = y + z$$

Encoded version:

$$\begin{aligned} x_c &= y_c + z_c + \text{err} \\ &= A * y_f + A * z_f + \text{err} \end{aligned}$$

Code checking:

$$x_c \bmod A = \text{err} \neq 0$$

Detectable Errors

modified operand: bitflip on z_c

Original source code:

$$x = y + z$$

Encoded version:

$$\begin{aligned} x_c &= y_c + z'_c \\ &= A * y_f + A * z_f + \text{err} \end{aligned}$$

Code checking:

$$x_c \bmod A = \text{err} \neq 0$$

Problems in Applying an AN-code

Our solutions for:

- overflow behavior according to C standard
→ *own set of encoded arithmetic operations*
- floating point operations
- encoded shifts
- encoded unaligned memory access
- encoded bitwise logical operations
→ *encodable software implementations*

Problems in Applying an AN-code

Our solutions for:

- overflow behavior according to C standard
→ *own set of encoded arithmetic operations*
- floating point operations
- encoded shifts
- encoded unaligned memory access
- encoded bitwise logical operations
→ *encodable software implementations*

Previous approaches:

- incomplete encoding, and/or
- less safe code

Problems in Applying an AN-code

Our solutions for:

- overflow behavior according to C standard
→ *own set of encoded arithmetic operations*
- floating point operations
- encoded shifts
- encoded unaligned memory access
- encoded bitwise logical operations
→ *encodable software implementations*

Previous approaches:

- incomplete encoding, and/or
- less safe code

⇒ **We are able to encode programs completely**

Encoded bitwise logical operations – example: not

```
uint32_t notTab[65536] = {0xFFFF, 0xFFFE, 0xFFFD, ...};

uint32_t not (uint32_t a){
    // divide parameter a into
    a1 = a / 0x10000; // upper and
    a2 = a % 0x10000; // lower 16 bit

    // fetch negated version for both parts
    r1 = notTab[a1];
    r2 = notTab[a2];

    // combine both 16-bit results
    return r1 * 0x10000 + r2;
}
```

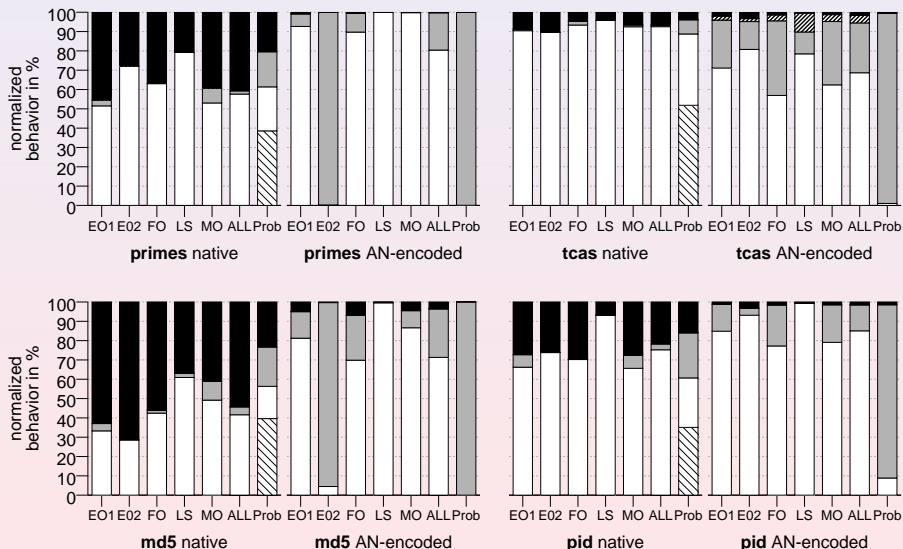
How slow is it?

application:	slowdown:
md5	238
tcas	137
pid	49
primes	8

Table 1: Slow down in x times slower than native execution.

- slowdown depends largely on workload
- **reason:** great differences between encoded operations' slowdowns
- programmers should avoid operations whose encoded version is slow

How many errors does it detect?



no error correct output failure detected performance failure incorrect output



Undetectable Errors

exchanged operand error: $z_c \rightarrow u_c$

Original source code:

$$x = y + z$$

Encoded version:

$$\begin{aligned}x_c &= y_c + u_c \\ &= A * y_f + A * u_f\end{aligned}$$

Code checking:

$$x_c \bmod A = 0 \Rightarrow \text{not detected}$$

Undetectable Errors

operator error: $+$ \rightarrow $-$

Original source code:

$$x = y + z$$

Encoded version:

$$\begin{aligned}x_c &= y_c - z_c \\ &= A * y_f - A * z_f\end{aligned}$$

Code checking:

$$x_c \bmod A = 0 \Rightarrow \text{not detected}$$

Make these Errors Detectable

AN-code with signatures

- first presented by [Forin, 1989]:
 - ▶ incomplete presentation
 - ▶ restricted applicability: dynamically allocated memory not supportable
- our previous work [Wappler and Fetzer, 2007]:
 - ▶ incomplete encoding
 - ▶ way too slow because of interpreter-based approach

Goal: ANB-encoding compiler

The End

Thank you very much for the attention.

Questions?

Email: ute.schiffel@se.inf.tu-dresden.de



Borkar, S. (2005).

Designing reliable systems from unreliable components: The challenges of transistor variability and degradation.

IEEE Micro, 25(6):10–16.



Dixit, A., Heald, R., and Wood, A. (2009).

Trends from ten years of soft error experimentation.

In *System Effects of Logic Soft Errors (SELSE)*.



Forin, P. (1989).

Vital coded microprocessor principles and application for various transit systems.

In *IFA-GCCT*, pages 79–84.



Schroeder, B., Pinheiro, E., and Weber, W.-D. (2009).

Dram errors in the wild: a large-scale field study.

In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 193–204, New York, NY, USA. ACM.



Wappler, U. and Fetzer, C. (2007).

Software encoded processing: Building dependable systems with commodity hardware.

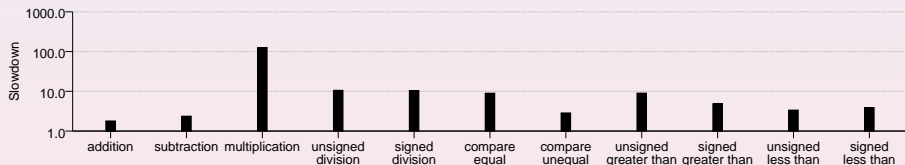
In *The 26th International Conference on Computer Safety, Reliability and Security (SafeComp 2007)*.

Workflow

Implementation using LLVM compiler framework:

- 1 replace unencodable instructions with their encodable version:
example: *shift right* \rightarrow *division by power of two*
- 2 replace all instructions with their AN-encoded version
- 3 replace all initialization values and constants with their AN-encoded versions
- 4 lower to binary code

Slowdown Encoded Arithmetic Operations



Slowdown Replacement Operations

