Elzar
Triple Modular Redundancy using Intel AVX

Dmitrii Kuvaiskii
Oleksii Oleksenko  Pramod Bhatotia  Christof Fetzer  Pascal Felber
Hardware Errors in the Wild

• Online services run in huge data centers
Hardware Errors in the Wild

- Online services run in **huge data centers**
- **Hardware faults** are the norm rather than the exception
Hardware Errors in the Wild

- Online services run in **huge data centers**
- **Hardware faults** are the norm rather than the exception
- These faults can lead to **arbitrary state corruptions**
Online services run in **huge data centers**

**Hardware faults** are the norm rather than the exception

These faults can lead to **arbitrary state corruptions**

---

„...There were a handful of messages that had a single bit corrupted such that the message was still intelligible, but the system state information was incorrect...“
Hardware Errors in the Wild

- Online services run in **huge data centers**
- ** Hardware faults** are the norm rather than the exception
- These faults can lead to **arbitrary state corruptions**

"...There were a handful of messages that had a single bit corrupted such that the message was still intelligible, but the system state information was incorrect..."

"...Corruption can occur **transiently in CPU** or RAM. Guarding against such corruptions is an **important goal** in Mesa’s overall design..."
Protecting Against Data Corruptions

Principled approaches

Ad-hoc approaches
Protecting Against Data Corruptions

- Principled approaches
- Byzantine Fault Tolerance
- Ad-hoc approaches
Protecting Against Data Corruptions

Principled approaches

Byzantine Fault Tolerance

- Tolerates arbitrary faults
- Pessimistic fault model
- High resource overheads

Ad-hoc approaches
Protecting Against Data Corruptions

Principled approaches

Byzantine Fault Tolerance

- Tolerates arbitrary faults
- Pessimistic fault model
- High resource overheads

Ad-hoc approaches

Checksums / Assertions
Protecting Against Data Corruptions

Principled approaches

Byzantine Fault Tolerance

- ☺ Tolerates arbitrary faults
- ☹ Pessimistic fault model
- ☹ High resource overheads

Ad-hoc approaches

Checksums / Assertions

- ☺ Low performance overheads
- ☹ Only anticipated faults
- ☹ Manual and error-prone
Protecting Against Data Corruptions

**Principled approaches**

- Byzantine Fault Tolerance

**Ad-hoc approaches**

- Checksums / Assertions

**Triple Modular Redundancy**
Protecting Against Data Corruptions

Principled approaches

Byzantine Fault Tolerance

Ad-hoc approaches

Checksums / Assertions

Triple Modular Redundancy

better performance 🎉 Practical fault model
Protecting Against Data Corruptions

Principled approaches

Byzantine Fault Tolerance

Checksums / Assertions

Ad-hoc approaches

Triple Modular Redundancy

- Better performance
- Practical fault model
- Disciplined protection

better fault coverage
Triple Modular Redundancy

Source code → Triple Modular Redundancy (TMR) → Executable

CPU
Native

\[ z = \text{add} \ x, \ y \]
Triple Modular Redundancy (TMR)

Source code

Executable

CPU

**Native**

\[ z = \text{add} \ x, \ y \]

**TMR**

\[
\begin{align*}
z & = \text{add} \ x, \ y \\
z_2 & = \text{add} \ x_2, \ y_2 \\
z_3 & = \text{add} \ x_3, \ y_3
\end{align*}
\]
**Triple Modular Redundancy (TMR) with SIMD**

<table>
<thead>
<tr>
<th>Native</th>
<th>TMR</th>
<th>TMR with SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z = \text{add} \ x, \ y )</td>
<td>( z = \text{add} \ x, \ y )</td>
<td>( z_{\text{wide}} = \text{add} \ x_{\text{wide}}, \ y_{\text{wide}} )</td>
</tr>
<tr>
<td>( z_2 = \text{add} \ x_2, y_2 )</td>
<td>( z_3 = \text{add} \ x_3, y_3 )</td>
<td></td>
</tr>
</tbody>
</table>

**Source code** © 2016.
Triple Modular Redundancy (TMR) with SIMD

Source code → TMR with SIMD

<table>
<thead>
<tr>
<th>Native</th>
<th>TMR</th>
<th>TMR with SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z = \text{add} \ x, \ y )</td>
<td>( z = \text{add} \ x, \ y )</td>
<td>( z_{\text{wide}} = \text{add} \ x_{\text{wide}}, \ y_{\text{wide}} )</td>
</tr>
<tr>
<td>( z_2 = \text{add} \ x_2, \ y_2 )</td>
<td>( z_2 = \text{add} \ x_2, \ y_2 )</td>
<td>( z_3 = \text{add} \ x_3, \ y_3 )</td>
</tr>
</tbody>
</table>

Idea: SIMD uses less instructions → less perf overhead?
Can we use SIMD for efficient fault tolerance?
Can we use SIMD for efficient fault tolerance?

Implementation using Intel AVX
Source code \rightarrow \text{Elzar} \rightarrow \text{Executable} \rightarrow \text{CPU}
Elzar

Implementation

Intel AVX for triple modular redundancy
Elzar

Source code → Elzar → Executable

**Implementation**

Intel AVX for triple modular redundancy

**Outcome**

Mixed results 😞 (AVX not general-purpose)
Implementation

Intel AVX for triple modular redundancy

Outcome

Mixed results 😞 (AVX not general-purpose)

Investigation

Two bottlenecks in current AVX
☑ Motivation
☐ Intel AVX
☐ Design
☐ Evaluation
☐ Discussion
New registers

Intel AVX Background

YMM0
YMM1
YMM2
...
YMM15

256 bit

64-bit 64-bit 64-bit 64-bit
New registers

YMM0
YMM1
YMM2
...
YMM15

256 bit

64-bit 64-bit 64-bit 64-bit

New instructions

x_1  x_2  x_3  x_4  YMM0
y_1  y_2  y_3  y_4  YMM1
x_1+y_1  x_2+y_2  x_3+y_3  x_4+y_4  YMM2
AVX as free resource?

AVX is not actively used?
AVX as free resource?

AVX is not actively used?

- Histogram
- K-means
- Linear regression
- Matrix multiply
- PCA
- String match
- Word count
- Blacksholes
- Dedup
- Ferret
- Fluidanimate
- Streamcluster
- Swaptions
- X264
- Memcached
- SQLite3
- Apache
AVX is not actively used?

**AVX as free resource?**

- Phoenix
  - histogram
  - kmeans
  - linear regression
  - matrix multiply
  - pca
  - string match
  - word count
- PARSEC
  - blackscholes
  - dedup
  - ferret
  - fluidanimate
  - streamcluster
  - swaptions
  - x264
- Apps
  - memcached
  - sqlite3
  - apache

![Bar chart showing usage of AVX applications](chart.png)
AVX as free resource?

AVX is not actively used?

- Phoenix
  - histogram
  - kmeans
  - linear regression
  - matrix multiply
  - pca
  - string match
  - word count

- PARSEC
  - blackscholes
  - dedup
  - ferret
  - fluidanimate
  - streamcluster
  - swaptions
  - x264

- Apps
  - memcached
  - sqlite3
  - apache
AVX as free resource?

AVX is not actively used? → Reuse for fault tolerance!

<table>
<thead>
<tr>
<th>Phoenix</th>
<th>unused</th>
<th>used</th>
</tr>
</thead>
<tbody>
<tr>
<td>histogram</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kmeans</td>
<td></td>
<td></td>
</tr>
<tr>
<td>linear regression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>matrix multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pca</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string match</td>
<td></td>
<td></td>
</tr>
<tr>
<td>word count</td>
<td></td>
<td></td>
</tr>
<tr>
<td>blackscholes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dedup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ferret</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fluidanimate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>streamcluster</td>
<td></td>
<td></td>
</tr>
<tr>
<td>swaptions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x264</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Apps</th>
<th>unused</th>
<th>used</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcached</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sqlite3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>apache</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
☑ Motivation
☑ Intel AVX
☐ Design
☐ Evaluation
☐ Discussion
Fault Model

- Protect against *transient faults in CPU*
  - corruptions in CPU registers
  - miscomputations in CPU execution units

- Memory is *protected by other means*
  - DRAM protected by ECC
  - CPU caches protected by ECC and parity
Native

\[ x \, = \, \text{load} \, a \]
\[ z \, = \, \text{add} \, x, \, 1 \]
### Elzar by Example

<table>
<thead>
<tr>
<th>Native</th>
<th>TMR</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{load } a )</td>
<td>( x = \text{load } a )</td>
</tr>
<tr>
<td>( z = \text{add } x, 1 )</td>
<td>( z = \text{add } x, 1 )</td>
</tr>
<tr>
<td></td>
<td>( z_2 = \text{add } x_2, 1 )</td>
</tr>
<tr>
<td></td>
<td>( z_3 = \text{add } x_3, 1 )</td>
</tr>
<tr>
<td></td>
<td>( \text{majority}(z, z_2, z_3) )</td>
</tr>
</tbody>
</table>
Elzar by Example

<table>
<thead>
<tr>
<th>Native</th>
<th>TMR</th>
<th>Elzar</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = load a</td>
<td>x = load a</td>
<td>x = avx-load a</td>
</tr>
<tr>
<td>z = add x, 1</td>
<td>z = add x, 1</td>
<td>z = avx-add x, 1</td>
</tr>
<tr>
<td></td>
<td>z_2 = add x_2, 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>z_3 = add x_3, 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>majority(z, z_2, z_3)</td>
<td>avx-check(z)</td>
</tr>
</tbody>
</table>
Elzar by Example

<table>
<thead>
<tr>
<th>Native</th>
<th>TMR</th>
<th>Elzar</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{load } a )</td>
<td>( x = \text{load } a )</td>
<td>( x = \text{avx-load } a )</td>
</tr>
<tr>
<td>( z = \text{add } x, 1 )</td>
<td>( z = \text{add } x, 1 )</td>
<td>( z = \text{avx-add } x, 1 )</td>
</tr>
<tr>
<td>( z_2 = \text{add } x_2, 1 )</td>
<td>( z_2 = \text{add } x_2, 1 )</td>
<td>( )</td>
</tr>
<tr>
<td>( z_3 = \text{add } x_3, 1 )</td>
<td>( z_3 = \text{add } x_3, 1 )</td>
<td>( )</td>
</tr>
<tr>
<td>( \text{majority}(z, z_2, z_3) )</td>
<td>( \text{majority}(z, z_2, z_3) )</td>
<td>( \text{avx-check}(z) )</td>
</tr>
</tbody>
</table>

Common instructions introduce lower overhead 😊
<table>
<thead>
<tr>
<th>Native</th>
<th>TMR</th>
<th>Elzar</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = \text{load } a) [x = \text{load } a]</td>
<td>(z = \text{add } x, 1) [z = \text{add } x, 1]</td>
<td>(x = \text{avx-load } a) [x = \text{avx-load } a]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(z_2 = \text{add } x_2, 1)</td>
<td>(z = \text{avx-add } x, 1) [z = \text{avx-add } x, 1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(z_3 = \text{add } x_3, 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{majority}(z, z_2, z_3)) [\text{avx-check}(z)]</td>
</tr>
</tbody>
</table>

**Bottleneck 1:** Memory accesses require wrappers 😞
Elzar by Example

**Native**

\[
\begin{align*}
    x &= \text{load } a \\
    z &= \text{add } x, 1
\end{align*}
\]

**TMR**

\[
\begin{align*}
    x &= \text{load } a \\
    z &= \text{add } x, 1 \\
    z_2 &= \text{add } x_2, 1 \\
    z_3 &= \text{add } x_3, 1 \\
    \text{majority}(z, z_2, z_3)
\end{align*}
\]

**Elzar**

\[
\begin{align*}
    x &= \text{avx-load } a \\
    z &= \text{avx-add } x, 1 \\
    \text{avx-check}(z)
\end{align*}
\]

**Bottleneck 2: Checks are expensive 😞**
Bottleneck 1: Memory accesses

\[
\begin{align*}
x &= \text{avx-load } a \\
z &= \text{avx-add } x, 1 \\
\text{avx-check}(z)
\end{align*}
\]
Bottleneck 1: Memory accesses

\[ x = \text{avx-load } a \]
\[ z = \text{avx-add } x, 1 \]
\[ \text{avx-check}(z) \]

No such thing as avx-load! 😞
Bottleneck 1: Memory accesses

\[
x = \text{avx-load } a \\
z = \text{avx-add } x, 1 \\
\text{avx-check}(z)
\]
Bottleneck 1: Memory accesses

\[
x = \text{avx-load} \ a \\
z = \text{avx-add} \ x, \ 1 \\
\text{avx-check}(z)
\]
Bottleneck 1: Memory accesses

\[
\begin{align*}
x &= \text{avx-load } a \\
z &= \text{avx-add } x, 1 \\
\text{avx-check}(z)
\end{align*}
\]
Bottleneck 1: Memory accesses

\[ x = \text{avx-load } a \]
\[ z = \text{avx-add } x, 1 \]
\[ \text{avx-check}(z) \]
Bottleneck 1: Memory accesses

\[ x = \text{avx-load } a \]
\[ z = \text{avx-add } x, 1 \]
\[ \text{avx-check}(z) \]

Memory accesses require wrappers 😞
Bottleneck 2: Checks

\[
\begin{align*}
x &= \text{avx-load } a \\
z &= \text{avx-add } x, 1 \\
\text{avx-check}(z)
\end{align*}
\]
Bottleneck 2: Checks

\[
x = \text{avx-load } a \\
z = \text{avx-add } x, 1 \\
\text{avx-check}(z)
\]
\[ x = \text{avx-load } a \]
\[ z = \text{avx-add } x, \, 1 \]
\[ \text{avx-check}(z) \]
Bottleneck 2: Checks

\[
\begin{align*}
x &= \text{avx-load } a \\
z &= \text{avx-add } x, 1 \\
\text{avx-check}(z)
\end{align*}
\]
Bottleneck 2: Checks

\[ x = \text{avx-load } a \]
\[ z = \text{avx-add } x, 1 \]
\[ \text{avx-check}(z) \]
x = avx-load a
z = avx-add x, 1
avx-check(z)
Bottleneck 2: Checks

\[ x = \text{avx-load} \ a \]
\[ z = \text{avx-add} \ x, \ 1 \]
\[ \text{avx-check}(z) \]
Bottleneck 2: Checks

\[ x = \text{avx-load } a \]
\[ z = \text{avx-add } x, 1 \]
\text{avx-check}(z)

\[ z \]
\[ z \]
\[ z' \]
\[ z \]

\[ z \]
\[ z \]
\[ z \]
\[ z' \]

\[ z \oplus z \]
\[ z \oplus z \]
\[ z \oplus z' \]
\[ z' \oplus z \]

\text{shuffle}
\text{xor}
\text{ptest}
\text{jne}
\text{recover}

Checks introduce additional 55% overhead 😞
☑ Motivation
☑ Intel AVX
☑ Design
☐ Evaluation
☐ Discussion
Performance overheads
Performance overheads

Normalized runtime (w.r.t. native)

lower better
Performance overheads

![Bar chart showing performance overheads for various benchmarks. The chart compares normalized runtime (w.r.t. native) for different benchmarks, with some showing significant savings of 15-20x and 10-14x.](image)

Lower is better.
Performance overheads

Average performance overhead is $4\times$
Performance overheads

Amortized by very poor memory access pattern
Performance overheads

1. Native benefits from AVX vectorization
2. Most of the time spent in mem-intensive `bzero()`
Comparison with SWIFT-R (state-of-the-art TMR approach)
Comparison with SWIFT-R (state-of-the-art TMR approach)

Normalized runtime (w.r.t. native)

- SWIFT-R
- Elzar

lower better
Comparison with SWIFT-R (state-of-the-art TMR approach)

lower  better

Normalized runtime (w.r.t. native)

SWIFT-R
Elzar

hist  km  linreg  mmul  pca  smatch  wc  black  dedup  ferret  fluid  scluster  swap  x264  mean
+119%  -9%  +76%  +4%  +20%  +94%  -34%  +15%  +6%  -24%  +1%  +5%  +50%  +46%
Elzar performs 46% worse than SWIFT-R on average 😞
Comparison with SWIFT-R (state-of-the-art TMR approach)

Dominated by memory accesses, Elzar inserts many wrappers 😞
Comparison with SWIFT-R (state-of-the-art TMR approach)

Elzar produces 3x less instructions than SWIFT-R ☺
☑ Motivation
☑ Intel AVX
☑ Design
☑ Evaluation
☐ Discussion
Bottlenecks and Proposed Solution

Problem

**AVX** lacks certain instructions

- Need wrappers for memory accesses
- Need shuffle-xor-ptest for checks
### Bottlenecks and Proposed Solution

<table>
<thead>
<tr>
<th>Problem</th>
<th>AVX lacks certain instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>– Need wrappers for memory accesses</td>
</tr>
<tr>
<td></td>
<td>– Need shuffle-xor-ptest for checks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution</th>
<th>Offload to <strong>FPGA accelerator</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>– Intel's Xeon-FPGA pairing</td>
</tr>
</tbody>
</table>
Bottlenecks and Proposed Solution

**Problem**

AVX lacks certain instructions
- Need wrappers for memory accesses
- Need shuffle-xor-ptest for checks

**Solution**

Offload to **FPGA accelerator**
- Intel's Xeon-FPGA pairing

Diagram:
- **CPU**:
  - addr
  - addr
  - addr
  - addr
  - load
  - val
  - val
  - val
  - val
- **FPGA**:
  - majority voting
  - replication
- **Memory**:
  - addr
  - val
Bottlenecks and Proposed Solution

**Problem**

AVX lacks certain instructions
- Need wrappers for memory accesses
- Need shuffle-xor-ptest for checks

**Solution**

Offload to **FPGA accelerator**
- Intel's Xeon-FPGA pairing

**Diagram**

```
CPU
addr  addr  addr  addr
load  
val   val   val   val

FPGA
majority voting
replication

Memory
addr
val
```

Potentially **71% better** than SWIFT-R
Conclusion

Implementation

**Intel AVX** for triple modular redundancy
Conclusion

Implementation  Intel AVX for triple modular redundancy

Hypothesis  Less instructions $\rightarrow$ less perf overhead
Conclusion

Implementation  Intel AVX for triple modular redundancy

Hypothesis  Less instructions → less perf overhead

Outcome  46% worse than SWIFT-R
Conclusion

Implementation  Intel AVX for triple modular redundancy

Hypothesis  Less instructions $\rightarrow$ less perf overhead

Outcome  46% worse than SWIFT-R

Discussion  With FPGA, $\sim$71% better than SWIFT-R
Thank you!

dmitrii.kuvaiskii@tu-dresden.de

GitHub repo: https://github.com/tudinfse/elzar
backup slides
Intel Haswell CPU microarchitecture

Instruction Scheduler

port 0
- 64-bit ALU
- 256-bit MUL
- 256-bit FMA

port 1
- 64-bit ALU
- 256-bit FADD

port 5
- 64-bit ALU
- 256-bit ALU

port 6
- 256-bit Shuffle
- 64-bit ALU

ports 2-4, 7
- Memory access
Intel Haswell CPU microarchitecture

Instruction Scheduler

- Port 0:
  - 64-bit ALU
  - 256-bit MUL
  - 256-bit FMA

- Port 1:
  - 64-bit ALU
  - 256-bit FADD

- Port 5:
  - 64-bit ALU
  - 256-bit ALU
  - 256-bit Shuffle

- Port 6:
  - 64-bit ALU

- Ports 2-4, 7:
  - Memory access

Intel AVX units
Elzar: Check on Branch

Native

\texttt{cmp x, y}
\texttt{jne truebranch}
Elzar: Check on Branch

Native

```c
cmp x, y
jne truebranch
```

```plaintext
<table>
<thead>
<tr>
<th>x^1</th>
<th>x^2</th>
<th>x^3</th>
<th>x^4</th>
</tr>
</thead>
<tbody>
<tr>
<td>y^1</td>
<td>y^2</td>
<td>y^3</td>
<td>y^4</td>
</tr>
</tbody>
</table>
```

```plaintext
\[
\begin{align*}
\text{cmp} & \ x, y \\
\text{jne} & \ \text{truebranch}
\end{align*}
\]
Elzar: Check on Branch

Impact  Checks on branches introduce only 4% overhead 😊
Main Bottlenecks

**Problem**  
AVX instruction set lacks certain instructions

- Loads/stores require extracting AVX-replicated address
  - Elzar creates expensive wrappers around loads/stores
  - AVX-512 introduces promising *gather/scatter* instructions

- AVX has only one control-flow instruction `ptest`
  - Elzar has to insert `ptest` for each control-flow decision
  - AVX could add `cmp` instructions directly affecting control flow

- AVX misses integer division, integer truncation, etc…
  - Elzar produces very ineffective code in these cases
Estimation of Proposed Changes

Normalized runtime (w.r.t. native)

- SWIFT-R
- Elzar
- Estimated Elzar

lower better
Estimation of Proposed Changes

- SWIFT-R
- Elzar
- Estimated Elzar

Normalized runtime (w.r.t. native)

- hist
- km
- linreg
- mmul
- pca
- smatch
- wc
- black
- dedup
- ferret
- fluid
- scluster
- swap
- x264
- mean

16.3

lower better
Estimation of Proposed Changes

Normalized runtime (w.r.t. native)

16.3

SWIFT-R
Elzar
Estimated Elzar

lower better
Estimation of Proposed Changes

Estimated average overhead would be 48% (71% improvement over SWIFT-R)