Exploiting Task-Based Programming Models for Resilience

PhD defense 21/06/2019

Luc Jaulmes

Advisors: Marc Casas, Miquel Moretò
Resilience trends

Error rates increase
- Hard faults: ageing, manufacturing variability
- Soft faults: particle strikes, voltage noise
- Memory subsystem most subject to faults

Known error rates
- Cielo supercomputer\textsuperscript{1}: 1 error every 5h23min
- Mont-Blanc 1 prototype\textsuperscript{2}: 1 error every 10min

\textsuperscript{1}V. Sridharan et al. (2015). “Memory Errors in Modern Systems”. In: ASPLOS XX, pp. 297–310.
Resilience trends

Main techniques today

- Sparing, adjusting refresh rate
  - Mitigate hard faults
  - Requires profiling
- Error Correcting Codes (ECC)
  - Detect unforeseen errors
  - Correct transparently
- Checkpointing-Rollback
  - Enable recovery from crash

Limitations

- Yield too low
- Current ECC unable to address projected fault rates
- High checkpoint overhead

Emergence of runtime systems

Programmability Wall

- Due to complex architectures
  - E.g. large core counts, heterogeneous architectures, hybrid memory hierarchies, etc.
- Answer: new programming models
  - Take complexity away from programmer
  - Rely on supporting software: the runtime system

Runtime system role

- Initial focus: transparent task scheduling
- Opportunity for other optimizations
  - E.g. accelerating critical tasks, enforcing a power budget, partitioning caches, managing scratchpad memories, etc.
Goal: exploiting runtime systems for cross-level resilience

**Algorithm level (SC’15, TPDS 29:9)**
- Novel forward recovery for a class of iterative solvers
- Runtimes allow to overlap computation and recovery

**OS level (Multiprog 2019, IOLTS’19)**
- Define metric that improves correlation with error risk
- Delay reporting errors to ignore non-consumed errors

**Architectural level**
- Sampling-based methodology to estimate vulnerability online
- Implemented on real-world hardware with low overhead
- Dynamically adjustable ECC guided allows to trade resilience for redundancy
State of the Art: Algorithmic Recoveries

Checkpointing / Rollback


Algorithm-specific

• Restart methods

• Check invariants
  Z. Chen (2013). “Online-ABFT”. In: PPoPP’13, pp. 167–176
State of the Art: Vulnerability Metrics

Existing Metrics
- AVF, specialised for memory
- DVF
- LD/ST

Error injection studies
- M. Casas et al. (2012). “Fault resilience of the algebraic multi-grid solver”. In: ICS’12, pp. 91–100

Applications
- Reliability-driven memory mapping
- Selective protection (ABFT, pointer triplication, ...)

S. S. Mukherjee et al. (2003). “A Systematic Methodology to Compute the Architectural Vulnerability Factors”. In: MICRO 36, pp. 29–42
Y. Luo et al. (2014). “Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost”. In: DSN 2014, pp. 467–478
L. Yu et al. (2014). “Quantitatively Modeling Application Resilience with the Data Vulnerability Factor”. In: SC’14, pp. 695–706
State of the Art: Dynamically Adaptable ECC

**Two-level ECCs for DRAM**

- Static: VS-ECC, Odd-ECC

  D. H. Yoon and M. Erez (2010). “Virtualized and Flexible ECC for Main Memory”. In: ASPLOS XV, pp. 397–408
  A. Malek et al. (2017). “Odd-ECC”. In: MEMSYS’17, pp. 96–111

**Sampling-based memory access patterns analysis**

- Offline: bottleneck analysis...


- Online: miss rate curves on POWER5

Outline

Motivation

Algorithmic-based DUE Recovery

Measuring Vulnerability in Memory

Dynamically Adaptable ECC

Conclusion
Error model: memory DUE

Fault in memory
- ECC performs detection, attempts correction
- Detected Uncorrected Errors (DUE) may happen
  - 27% of node down events caused by DUE\(^1\)
- Reported in register: machine check exception

Application-level symptom
- Access DUE location $\Rightarrow$ OS kills application
- Replace physical page: data lost!

\(^1\)S. Levy et al. (2018). “Lessons Learned from Memory Errors Observed over the Lifetime of Cielo”. In: SC’18, 43:1–43:12.
Extract redundancy from operations

### Rewrite: 1 block per memory page

- Linear combination: \( u_i = \alpha v_i + \beta w_i \)
- Matrix-vector mult: \( q_i = \sum_j A_{ij} d_j \)
- 4KB page \( \Rightarrow \) 512 doubles

### Reuse or “invert” relations

- \( v_i = \frac{1}{\alpha}(u_i - \beta w_i) \)
- Factorize diagonal blocks:
  \[
  A_{ii} d_i = q_i - \sum_{j \neq i} A_{ij} d_j
  \]
fine-grain algorithm-based error correction for CG

```
d' ← 0
g ← b − Ax
for t in 0..t_{max}
    ε ← ||g||^2
    if ε < tol:
        break
    β ← ε / ε_{old}
    d ← βd' + g
    q ← Ad
    α ← ε / < q, d >
    x ← x + αd
    g ← g − αq
    ε_{old} ← ε
    swap (d, d')
```

initial x

g = b − Ax

d' = A^{-1}q  
g = b − Ax

d = βd' + g  
q = Ad     
d = A^{-1}q  
x = A^{-1}(b − g)

colors: constant, dynamic

Recover lost data

- Recompute from redundancy existing in algorithm
  - Recompute past operations
  - Identify invariants, e.g. \( g = b − Ax \)
- No overhead for adding redundancy
- Exact recovery: converge as well as without faults
- Forward recovery: no work reverted

Fully covers Krylov subspace iterative solvers

- Popular family of solvers: CG, GMRES, BiCGStab...
- Also works with preconditioners
Task-based CG implementation

- **OmpSs**
  - Asynchronous tasks
  - Dataflow dependencies

- **Adding recoveries**
  - Skip “corrupted” computations
  - Recovery computations in tasks
    - before scalar tasks
Strategies to schedule recoveries

- **Critical path**
  - Conservative approach
  - “Forward Exact Interpolation Recovery” (FEIR)

- **Overlap with work**
  - Reduce overhead
  - Sacrifice error coverage
  - “Asynchronous FEIR” (AFEIR)
Compared methods

- **Ideal**
  - No faults baseline

- **Trivial**
  - No recovery, erratic

- **Checkpoint-Rollback**
  - Exact, backwards

- **Lossy Restart**
  - Forward, lose some convergence guarantees

- **FEIR**

- **AFEIR**
  
  
  

---

Results with fault injection

Methodology

- Error injection: trigger signal on access
  - For application, strictly identical
  - MTBE scaled to baseline’s run time
- Single 8-core socket
- Geometric mean of overheads

CG and PCG evaluated

- 9 matrices\(^1\)
- Block-Jacobi preconditioner

---

Summary

**Algorithm level recovery**
- Fine-grain recovery
- Desirable properties: Forward & Exact
- Uses inherent redundancy, easy to identify

**Runtimes allow to overlap computation and recovery**
- Mask overhead
- Trade-off with error coverage

**Further results**
- Redundancy relations used for BiCGStab, GMRES, preconditioned solvers
- Strong scaling (up to 1024 cores)
- Impact of memory page size


Intuition: faults only matter if consumed

**Memory Vulnerability Factor (MVF)**

- Architectural Vulnerability Factor\(^1\) for memory
- Error overwritten by ST
  - \(\text{ST} \Rightarrow \text{data safe}\)
  - \(\text{LD} \Rightarrow \text{data vulnerable}\)
- \(\text{MVF} = \text{fraction of vulnerable time (before LD)}\)
- Upper bound on failure probability

---

\(^1\)S. S. Mukherjee et al. (2003). “A Systematic Methodology to Compute the Architectural Vulnerability Factors”. In: MICRO 36, pp. 29–42.
Errors in data fetched from memory, but not consumed

- Data fetched but *unused*
  - Loaded speculatively
  - Neighbour data (e.g. same cache line)
  - Write-allocate cache on ST miss

- Errors have no impact on the program
  - Reporting them causes *False Errors*
  - Solution: delay until data consumed

New Metric: False-Error Aware vulnerability (FEA)

\[
FEA = \frac{\text{time before data consumed}}{\text{total time}} = MVF - \frac{\text{time before “unused fetch”}}{\text{total time}}
\]

- Still upper bound on failure probability
**Vulnerability metrics in literature**

**Safe Ratio**\(^3\)
\[ sr = \frac{\text{safe time}}{\text{total time}} = 1 - MVF \]

**Store ratio (proxy for MVF)**\(^4\)
\[ \text{As } \frac{LD}{LD+ST} \text{ to fit in } [0, 1] \text{ and correlate positively} \]

**Data Vulnerability Factor**\(^5\)
\[ DVF = \sum_{d \in \text{data structures}} \text{error rate} \times \text{size}_d \times \text{execution time} \times \text{memory accesses}_d \]

- Detailed model for memory accesses, based on access pattern, cache sizes.

---

\(^3\) Y. Luo et al. (2014). “Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost”. In: DSN 2014, pp. 467–478


\(^5\) L. Yu et al. (2014). “Quantitatively Modeling Application Resilience with the Data Vulnerability Factor”. In: SC’14, pp. 695–706
Measure metrics and impact of errors

**TaskSim**

- Architectural simulator
  - Simple core model
  - Full cache hierarchy
  - Ramulator for memory
- Uses real runtime
  - Schedule tasks onto simulated cores
  - Simulate tasks in detail
- Based on task traces
  - Traced with DynamoRIO
  - Records memory accesses, basic blocks, runtime calls

**Native runs to measure impact of errors**

- Flip 1, 2, 3 bits or DUE (e.g. NaN)
- outcome crash, hang, wrong, slow or ok
- On Intel Xeon Platinum 8160

---

Comparing metrics over various benchmarks

Failure = crash + hang + wrong + slow outcomes, success = ok outcomes
Summary

**False-Error Aware metric**
- Identify errors that will not be consumed
- Best metric correlation with error risk
- Remains upper bound on error risk

**Further results**
- Metric comparison at memory page granularity
- Correlation coefficients
- DRAM refresh savings

L. Jaulmes et al. (2019b). “Memory Vulnerability: A Case for Delaying Error Reporting”. In: Multiprog 2019
L. Jaulmes et al. (2019a). “Memory Vulnerability for ECC-protected Memory”. In: IOLTS 2019
Outline

Motivation

Algorithmic-based DUE Recovery

Measuring Vulnerability in Memory

Dynamically Adaptable ECC

Conclusion
Dynamically adapting ECC online

Estimate FEA metric online: a methodology
- Identify memory access patterns in real time
- Compute fraction of time before LD
- Uses sampling of instructions

Dynamically adjustable ECC scheme
- ECC scheme with 2 protection levels
- Increase protection for pages identified as most vulnerable
Runtime vulnerability methodology

gather information from hardware sampling
- Sample LD and ST target address and time
  ▶ PMU randomly selects instructions
  ▶ Record 1 in $N$ selected instructions
- Only during “sampling phases”
  ▶ Control overhead: 0 cost when disabled

extrapolate memory access pattern from sampled instructions
- Aligned points $\Rightarrow$ streaming patterns
  ▶ Kernel Hough Transform$^1$
- Extrapolate streams to memory region
- Average time before LD as vulnerability

---

“Randomly” selected instructions

User-level interrupt to record
- Hand-written assembly routine
- Log to thread-local buffers

Technique overhead: sampling + analysis

Sampling period of 2, enabled 50% of the time: 3.47% overhead
Online vulnerability values

Varied Vulnerability Behaviour

• To be used for stronger protection targeting
**Core ideas**

- Apply strong ECC where needed
  - Trade-off reliability vs. redundancy
- Dynamically switch normal ↔ strong ECC
- Extra redundancy in addressable memory

**ECC Model**

- Baseline: $N$ bit flips ($0 \leq N < 3$)
- Selected pages: $N+1$ bit flips
WITSEC implementable in the memory controller

**Necessary extensions**
- Strong (N+1) EC codec
- Extended-protection pages
  - Physical address, size
  - Corresponding supplementary ECC blocks

**Supplementary extension**
- ECC blocks cache
### Measurement

- Inject single, double and triple bit flips
  - Vulnerability at injection time
  - Outcome at the end of the run
- Failure probability per vulnerability threshold:
  - Injection target above threshold ⇒ ok
  - Otherwise use outcome: failure or ok

### Trade-off applications

- Given a global reliability target, apply optimal level of redundancy
- Given a redundancy budget, get maximal reliability
Summary

**Sampling-based methodology to estimate vulnerability online**
- Generic: only requires sampling capability
- Uses real-time algorithms to identify streaming memory access patterns

**Evaluated on real-world hardware**
- Using POWER8 performance monitoring unit
- Low overhead: 3.47%

**WITSEC dynamically adjustable ECC**
- Implementable in memory controller
- Allows to dynamically trade resilience for redundancy

**Further results**
- CPU vs Memory Vulnerability Comparison
- Event-Based Branch details
  
  L. Jaulmes et al. (est. 2020). *Adapting ECC Protection Dynamically using Online Estimation of Memory Vulnerability*. under preparation
  
Outline

Motivation

Algorithmic-based DUE Recovery

Measuring Vulnerability in Memory

Dynamically Adaptable ECC

Conclusion
Conclusion

**Fine-grain algorithm-level recovery techniques**
- Take full advantage of OS and HW support for DUE
- Overlap work and recover: never stop solving

**False-Error Aware Metric**
- Consistent upper bound, correlates best with failure risk
- Highlights opportunities from dead data in memory

**WITSEC**
- Dynamically adjust ECC to protect more the most vulnerable regions
- Detect most vulnerable regions online using sampling-based instrumentation

Runtime systems help optimise recoveries, detect redundant data, and manage added redundancy.
Where to go from here

**Extend existing work**
- More algorithms, access patterns, different hardware...
- Using task dependency graphs for vulnerability estimation

**Use “dead data” insight from FEA**
- Skip DRAM refreshes, critical for future DRAM technologies
- Candidates for algorithmic optimisation, e.g. loop fusion for stencil codes

**Runtime-aided error detection**
- High DUE tolerance $\Rightarrow$ lower SDC rate
- Data dependency information, task profiling
  $\Rightarrow$ detect and contain errors on shared memory systems
- Programming-model support to express resilience
List of publications

First author publications:


- **L. Jaulmes, M. Moretó, M. Valero, and M. Casas (2019b).** “Memory Vulnerability: A Case for Delaying Error Reporting”. In: Multiprog 2019

- **L. Jaulmes, M. Moretó, M. Valero, and M. Casas (2019a).** “Memory Vulnerability for ECC-protected Memory”. In: IOLTS 2019

Other publications:


- D. Richards and **L. Jaulmes (2014).** “CoMD in Chapel: The Good, the Bad, and the Ugly”. In: Chapel Lightning Talks, Birds-of-a-Feather session at SC’14

Publicly available code:

