Containment Domains Resilience
Mechanisms and Tools Toward Exascale Resilience

Mattan Erez
The University of Texas at Austin
Yes, resilience is an exascale concern

– Checkpoint-restart not good enough on its own
– Commercial datacenters face different problems
– Heterogeneity keeps growing
– Correctness also at risk (integrity)
Containment Domains (CDs)

- **Isolate** application resilience from system
- **Increase** performance and efficiency
- **Simplify** defensive (resilient) codes
- **Adapt** hardware and software

**Portable**  **Performant**  **Resilient**  **Proportional**
Efficient resilience is an exascale problem
Failure rate possibly too high for checkpoint/restart
Correctness also at risk
Energy also problematic

Energy Overhead

2.5PF 10PF 40PF 160PF 640PF 1.2EF 2.5EF

- CDs, NT
- h-CPR, 80%
- gCPR, 80%

(c) Mattan Erez
Something bad every ~minute at exascale

Something bad every year commercially
- Smaller units of execution
- Different requirements
- Different ramifications
Rapid adoption of new technology and accelerators
  – Again, potential mismatch with commercial setting
So who’s responsible for resilience?

Hardware?
Software?
Algorithm?
Can **hardware alone** solve the problem?
Yes, but **costly**
- **Significant** and **fixed/hidden** overheads
- Different tradeoffs in commercial settings
Fixed overhead examples (estimated)
Both energy and/or throughput

- Up to ~25% chipkill correct vs. chipkill detect
- 20 – 40% for pipeline SDC reduction
- >2X for arbitrary correction
- Even greater overhead if approximate units allowed
Relaxed reliability and precision

- Some lunacy (rare easy-to-detect errors + parallelism)
- Lunatic fringe: bounded imprecision
- Lunacy: live with real unpredictable errors

Rough estimated numbers for illustration purposes
Can **software** do it alone?

- Detection likely very costly
- Recovery effectiveness depends on error/failure frequency
- Tradeoffs more limited
Locality and hierarchy are key
  – Hierarchical constructs
  – Distributed operation

Algorithm is key:
  – Correctness is a range
Containment Domains elevate resilience to **first-class abstraction**
- Program-structure abstractions
- Composable resilient program components
- Regimented development flow
- Supporting tools and mechanisms
- Ideally combined with adaptive hardware reliability

**Portable Performant Resilient Proportional**
CDs help bridge the gap

– Help us figure out exactly how
– Open source: lph.ece.utexas.edu/public/CDs
  bitbucket.org/cdresilience/cdruntime
CDs Embed Resilience within Application

Express resilience as a tree of CDs
- Match CD, task, and machine hierarchies
- Escalation for differentiated error handling

Semantics
- Erroneous data never communicated
- Each CD provides recovery mechanism

Components of a CD
- Preserve data on domain start
- Compute (domain body)
- Detect faults before domain commits
- Recover from detected errors
Mapping example: SpMV

void task<inner> SpMV(in M, in Vi, out Ri) {
    cd = GetCurrentCD()
    ->CreateAndBegin();
    cd->Preserve(matrix, size, kCopy);
    forall(...) reduce(...)
        SpMV(M[...],Vi[...],Ri[...]);
    cd->Complete();
}

void task<leaf> SpMV(...) {
    cd = GetCurrentCD()
    ->CreateAndBegin();
    cd->Preserve(M, sizeof(M), kRef);
    cd->Preserve(Vi, sizeof(Vi), kCopy);
    for r=0..N
        for c=rowS[r]..rowS[r+1]
            resi[r]+=data[c]*Vi[cIdx[c]];
    cd->CDAssert(idx > prevIdx, kSoft);
    prevC=c;
    cd->Complete();
}
Mapping example: SpMV

void task<inner> SpMV(in M, in Vi, out Ri) {
    cd = GetCurrentCD()
        ->CreateAndBegin();
    cd->Preserve(matrix, size, kCopy);
    forall(...) reduce(...)
        SpMV(M[…], Vi[…], Ri[…]);
    cd->Complete();
}

void task<leaf> SpMV(...) {
    cd = GetCurrentCD()
        ->CreateAndBegin();
    cd->Preserve(M, sizeof(M), kRef);
    cd->Preserve(Vi, sizeof(Vi), kCopy);
    for r=0..N
        for c=rowS[r]..rowS[r+1]
            resi[r]+=data[c]*Vi[cIdx[c]];
    cd->CDAssert(idx > prevIdx, kSoft);
    prevC=c;
    cd->Complete();
}
Mapping example: SpMV

```c
void task<leaf> SpMV(...) {
    cd = GetCurrentCD()
    ->CreateAndBegin();
    cd->Preserve(M, sizeof(M), kRef);
    cd->Preserve(Vi, sizeof(Vi), kCopy);
    for r=0..N
        for c=rowS[r]..rowS[r+1]
            resi[r]+=data[c]*Vi[cIdx[c]];
    cd->CDAssert(idx > prevIdx, kSoft);
    prevC=c;
    cd->Complete();
}
```
void task<leaf> SpMV(...) {
    cd = GetCurrentCD()
        ->CreateAndBegin();
    cd->Preserve(M, sizeof(M), kRef);
    cd->Preserve(Vi, sizeof(Vi), kCopy);
    for r=0..N
        for c=rowS[r]..rowS[r+1]
            resi[r]+=data[c]*Vi[cIdx[c]];
    cd->CDAssert(idx > prevIdx, kSoft);
    prevC=c;
    cd->Complete();
}
Concise abstraction for complex behavior

void task<leaf> SpMV(...) {
    char cd = GetCurrentCD()
        ->CreateAndBegin();
    cd->Preserve(M, sizeof(M), kRef);
    cd->Preserve(Vi, sizeof(Vi), kCopy);
    for r=0..N
        for c=rowS[r]..rowS[r+1]
            resi[r]+=data[c]*Vi[cIdx[c]];
    cd->CDAssert(idx > prevIdx, kSoft);
    prevC=c;
    cd->Complete();
}
General abstractions – a “language” for resilience
Replicate in space or time or none?

Local copy or regen

Sibling

Parent (unchanged)
CDs natural fit for:
- Hierarchical SPMD
- Task-based systems

CDs still general:
- Opportunistic approaches to add hierarchical resilience
- Always fall back to more checkpoint-like mappings
Reminder of why you care
CDs enable per-experiment/system “optimality”

- **(Portable)** Use same resilience abstractions across programming models and implementations
  - MPI ULFM? MPI-Reinit? OpenMP? UPC++? Legion?
    - Don’t keep rethinking correctness and recovery
  - CPU, GPU, FPGA accelerator, memory accelerator, ... ?

- **(Performant)** Resilient patterns that scale
  - Hierarchical / local
  - Aware of application semantics
  - Auto-tuned efficiency/reliability tradeoffs

- **(Resilient)** Defensive coding
  - Algorithms, implementations, and systems
  - Reasonable default schemes
  - Programmer customization

- **(Proportional)** Adapt hardware and software redundancy
CD Runtime System Architecture

- Annotations, persistence, reporting, recovery, tools
CD usage flow

- Annotate
- Profile and extrapolate CD tree
- Supply machine characteristics
- Analyze and auto-tune
  - Flexible preservation, detection, and recovery
- Refine tradeoffs and repeat
- Execute and monitor
  - CD management and coordination
  - Distributed and hierarchical preservation
  - Distributed and hierarchical recovery
CD annotations express intent

- **CD hierarchy** for scoping and consistency
- **Preservation** directives and hints exploit locality
- **Correctness** abstractions
  - Detectors and tolerances
- **Recovery** customization
- **Debug/test** interface

Work in progress: http://lph.ece.utexas.edu/users/CDAPI
State preservation and restoration API

curCD->Preserve(ptr, size, method_mask, byref_name, name, regenObj);

- Hierarchical
  - Per CD (level)
  - Match storage hierarchy
  - Maximize locality and minimize overhead

- Proportional
  - Preserve only when worth it (skip preserve calls)
  - Exploit inherent redundancy
  - Utilize regeneration
Local copy or regen

Parent (unchanged)

Sibling
LULESH CD mapping example

- **32k x 32k x 32k Mesh (PB)**
  - Global System Checkpoint

- **2k x 2k x 2k Mesh (TB)**
  - To Buddy Cabinet

- **500 x 500 x 500 Mesh (GB)**
  - To Buddy Module

- **100 x 100 x 100 Mesh (MB)**
  - To DRAM
  - Recover Ghosts from Sibling

- **Communication Barrier (Relaxed Variant)**

- **Preservation Depends on phase (MB)**
  - 3 primary phases separated by barrier

- **Preservation Depends on Thread (B-KB)**
  - ~30 independent, multithreaded for loops

- **Heterogeneous CDs**

- **Per Thread**

- **Per Socket**

- **Per Module**

- **Per Cabinet**
Autotuned CDs perform well

- **NT**
  - CDs, NT: h-CPR, 80% - g-CPR, 80%

- **SpMV**
  - CDs, SpMV: h-CPR, 50% - g-CPR, 50%

- **HPCCG**
  - CDs, HPCCG: h-CPR, 10% - g-CPR, 10%
CDs improve energy efficiency at scale

Peak System Performance

Energy Overhead

CDs, NT
h-CPR, 80%
g-CPR, 80%

CDs, SpMV
h-CPR, 50%
g-CPR, 50%

CDs, HPCCG
h-CPR, 10%
g-CPR, 10%
10X failure rate emphasizes CD benefits

- **CDs, NT** h-CPR, 80%
- **CDs, SpMV** h-CPR, 50%
- **CDs, HPCCG** h-CPR, 10%

*Energy Overhead vs. Performance Efficiency*
Can be implicit with right programming model
  – For example, **Legion**
Use Legion copies for CD preservation

Optimize for efficiency

– When to add copies
– Where to put copies to survive failures
– When to free copies

Account for different failure modes and rates
Preservation to more reliable medium

Preservation

(a) Single Legion Task

(b) Markov chain model of (a)

\[ E_{T_0} = \sum_{i=0}^{\infty} p_i (1 - p_i) \times (T_0 + T_{0,p}) \times (i + 1) \]

(c) Expected execution time of Task 1

(d) Sequential tasks

(e-1) Markov chain model of (d-1)

(e-2) Markov chain model of (d-2)

(e-3) Markov chain model of (d-3)

(f) three-successor Legion Tasks

(g) three-predecessor Legion Task

(h) Markov chain model of (f) and (g)
Portable correctness
– Resilience perspective
Correctness abstractions

– Detectors
– Requirements
– Recovery
What can go wrong?

- Application crash
- Process crash
- Process unresponsive
- Failed communication
- Hardware
  - Cache error
  - Memory error
  - TLB error
  - Node offline
  - ...

(c) Mattan Erez
What can go wrong?

- Lost resource
- Wrong value
  - Specific address?
  - Specific access?
  - Specific computation?
- Degraded resource

Who detects?
How reported?
System-provided detectors

- `curCD->Detect();`
  - Control response granularity

User-specified detectors

- `curCD->CDAssert(test, error_to_report);`

Consistent and unified reporting & analysis
Express correctness intent

- `curCD->`
  ```
  RegisterDetection(errors_reported);
  ```
  - Notifies auto-tuner of detection capability
  - Enables error elision

- `curCD->RequireErrorProbability(
  error_type, num_errors,
  probability,detect_or_fail_over);
  ```
  - Auto- add redundancy to meet requested level of reliability

- `curCD->GetErrorProbability(
  error_type, num_errors);
  ```
  - Customize action
Bounded Approximate Duplication
Bounded Approximate Duplication

![Diagram]

Estimated Efficiency (Relative to 32-bit Conventional)

- Conventional
- Approximate

Input Width

- 32
- 64
- 128

Graph

2^{k_2+1}

N_2

-0.02

-0.03

-0.05

-0.02

-0.02

-0.01

-0.04

-0.02

N_1

2^{k_1}

2^{k_1+1}
Analysis/decision-support/tuning
Example: integrity tools

- Selective injection by CD and error type
- Integrate with CD-level detectors
  - Only inject “SDCs”
- “Fuzzing” tools for completeness
- Analytical modeling for tradeoffs
  - Energy, memory, performance
Straightforward bottom-up analysis
- Analytical solutions for simple trees
- More computation for complex graphs
Example: what-if reliability/resilience tradeoffs

Should all memory be heavily ECC protected?
– Much cheaper to recognize anomalies in some apps
– Much cheaper to do detection only
– …

Mechanisms for adapting ECC scheme are known
– Though not implemented in any product
Another application example
TOORSES fault-tolerant hierarchical solver
– Brian Austin, Eric Roman, and Xiaoye Li (LBNL)
– Hierarchical semi-separable representation
Add CDs at different granularities
  – Hierarchical and partial preservation
Add algorithmic and cheap detection
Compare to:
  – Algorithmic recovery with redundant computation
Bottom line – expected benefits significant:
- Isolate application correctness from system
- Use same resilience abstractions across programming models
- Enable efficient resilience patterns that scale
  - Reasonable default schemes
  - Programmer customization
- Auto-tune efficiency/reliability tradeoffs
- Adapt to system and experiment dynamics

All open source
- lph.ece.utexas.edu/public/CDs and bitbucket.org/cdresilience/cdruntime
- lph.ece.utexas.edu/users/hamartia and bitbucket.org/lph_tools/hamartia_suite
Status

- Mostly-sequential functional CD runtime released
- MPI implementation mostly done (some merging left)
  - Bitbucket.org/cdresilience/cdruntime
- cdCUDA prototype
- Abstractions, for now, seem sufficient
  - But, not enough users yet
- Rudimentary implementation only
  - Lots of opportunities for improvement
    - Storage (object stores across hierarchy?)
    - OS/R (better isolation mechanisms, automation)
    - Communication (reduce recovery overhead)
- Other programming models coming along
  - UPC++ prototype in progress
  - Legion integration
Obviously, I’m just the figure head

- **Former UT students:**
  - Jinsuk Chung, Ikhwan Lee, Minsoo Rhu, Michael Sullivan, Doe Hyun Yoon

- **Current UT students:**
  - Chun-Kai Chang, Seong-Lyong Gong, Chanyong Hu, Tommy Huynh, Dong Wan Kim, Yongkee Kwon, Kyushick Lee, Sangkug Lym, Song Zhang

- **Collaborators:**
  - LBNL: Brian Austin, Dan Bonachea, Paul Hargrove, Eric Roman
  - Cray: Larry Kaplan and team
  - NVIDIA: Siva Hari, Tim Tsai

- **Funding (overall, not just CDs):**
  - DOE ASCR: ECRP, FastForward, X-Stack, Resilience, PSAAP II
  - DARPA: UHCP
  - NSF: CAREER
  - DOD: Fellowship
  - TACC and NERSC compute facilities
Containment Domains

- **Abstract** resilience constructs that span system layers
- **Hierarchical and Distributed** operation for locality
- **Scalable** to large systems with high energy efficiency
- **Heterogeneous** to match disparate error/failure effects
- **Proportional** and effectively balanced
- **Tunable** resilience specialized to application/system
- **Analyzable** and auto-tuned

- **Open source:** lph.ece.utexas.edu/public/CDs
  bitbucket.org/cdresilience/cdruntime

**Portable**  **Performant**  **Resilient**  **Proprtional**
Backup
An aside on error modeling and injection
High-fidelity modeling in Veracity
Multiple, detailed, low-level fault models to explore faults at circuit and microarchitectural levels

- Focused on particle strikes and voltage droop
- Circuit-level simulation and analysis is performed both statically (pre-characterization) and dynamically (runtime)

Low-level models combined with hierarchical injection and simulation frameworks

- Simulate the effect of faults at points in an application throughout the entire microarchitecture
- Multiple levels of abstraction, providing speedup at higher levels

Finally, simulation results will enable production of high-level models

- Characterize error patterns, fault rates, and portions of the microarchitecture vulnerable
- Models will provide insight into software resiliency
Accurate Low-level Injection and Modeling

Modeling OpenSPARC T1 microprocessor
- Free, open source
- Existing FPGA version of project
- High-performance, multithreaded

Model low-level fault propagation
- Full fidelity with fast hierarchical simulation
  - Circuit-level
  - RTL / microarchitectural-level
  - ISA-level
- Inject errors at the circuit-level
- Data and control fault injection
Fast Hierarchical Low-level Inject / Simulation

Transition between levels
- Circuit, RTL, ISA levels
- Speed / granularity benefits
- Maintain fidelity across levels

Novel, accurate RTL → ISA switching algorithm\(^1\)
- Maintains full fidelity
- Detect if can terminate early

Framework enables:
- Fast, accurate analysis of fault propagation
- Detect masked / unmasked faults
- Fault injection in any logic (data, control)
- Improvement over timeout-based detection

---

Hierarchical Simulation Advancements

Replace RTL simulation with FPGA emulation

- Fast, natural target for RTL
- OpenSPARC includes FPGA support
- Tool NIFD (created by our group)
  - Reads/writes FPGA register state
  - Used to inject errors into FPGA
  - No FPGA recompilation for different tests

Support multi-fault / multi-cycle errors

- Already supports particle strike fault injection
- Can mimic voltage droop
- Greater variety of fault scenarios
High-Fidelity Hardware Faults Modeling

On-demand transistor-accurate fault injection with workload-specific distributional properties

Use model for fault injector (FI):

- Higher level tool injects input vector to FI
- Returns an error-output for inputs with non-zero probability of error

Initial fault model

- Voltage droop

Possible methodologies

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Two-phase with pre-characterization</th>
<th>Runtime simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1) Error profiling, (2) Look-up</td>
<td>Full run-time error evaluation</td>
</tr>
<tr>
<td>Simulation speed</td>
<td>Fast</td>
<td>Slow</td>
</tr>
<tr>
<td>Memory usage</td>
<td>*Potentially High</td>
<td>Low</td>
</tr>
</tbody>
</table>

*High: O(n) where n is the number of critical paths
Fault Injection Methodology

Phase 1: Pre-characterization

- Builds a model of possible error-outputs and their probabilities for every possible outputs

- Model outputs
  1. Error-outputs and corresponding input pairs
  2. Probability producing any error-outputs
  3. Conditional probability of error-outputs

Phase 2: Runtime error-outputs generation

- For an input with a non-zero probability of producing an error, generates one out of possible set of error-outputs
  - With probability relative to total
Pre-characterization

RTL

\( V_{\text{nominal}} \text{ lib.} \)

Synthesize

\( V_{\text{min}} \text{ lib.} \)

STA

\( \text{Gate-level netlist (.v)} \)

STA

\( \text{Slack} \)

\( \text{Slack} \)

Paths with \( S < 0 \)

Critical \( V^* \) for \( N \) paths \( \pi_i \), using interpolation, Output error \( O(\pi_i) \)

Identify input patterns that generate each path using ATPG

\( \pi_i: I^A_A \times I^B_B \)

\begin{tabular}{|c|c|c|}
\hline
\( I^A_A \times I^A_B \) & \( \pi_{1,1}: V^*(\pi_{1,1}): O(\pi_{1,1}) \) & \( \pi_{1,2}: V^*(\pi_{1,2}): O(\pi_{1,2}) \) & \( \pi_{1,3}: V^*(\pi_{1,3}): O(\pi_{1,3}) \) \\
\hline
\( I^A_A \times I^B_B \) & \( \pi_{2,1}: V^*(\pi_{2,1}): O(\pi_{2,1}) \) & & \\
\hline
\( I^A_A \times I^B_B \) & \( \pi_{1,1}: V^*(\pi_{1,1}): O(\pi_{1,1}) \) & & \\
\hline
\end{tabular}

Error profile database .edb
Medium-fidelity fast error modeling and injection in AEDAM
Instruction-level Injection

Pin-based Fault Injection
- Generic Error/Detector API for reusable models across injectors and languages
- Instruction-driven injection, lower fidelity

Hierarchical fault model execution

Selective injection
- Filters based upon instruction & code region

Comprehensive coverage
- Open-source tools allows for distributed and large-scale error simulations

Allows for high-level model generation
- Fast, application injection
- Monte-Carlo injection methodology
Gate-Level Injection

Injection into synthesized, gate-level netlists
- Models transient and stuck-at faults
- Simulates faults at a unit-level (e.g. ALU)
- Filter eligible fault locations by gate/latch

Verilog simulation
- Iteratively find unmasked error patterns
- Optional detector support for modeling fault detection hardware

Connects with Pin Injector for hierarchical simulation and injection

Higher-level Injector (e.g. Pin)
Error Mgmt Framework
Gate-level Injector
Iterate till Unmasked
Modeling DRAM errors
- Aware of ECC options
- Aware of memory architecture
- Aware of memory fault modes

*redbox: fault injection point