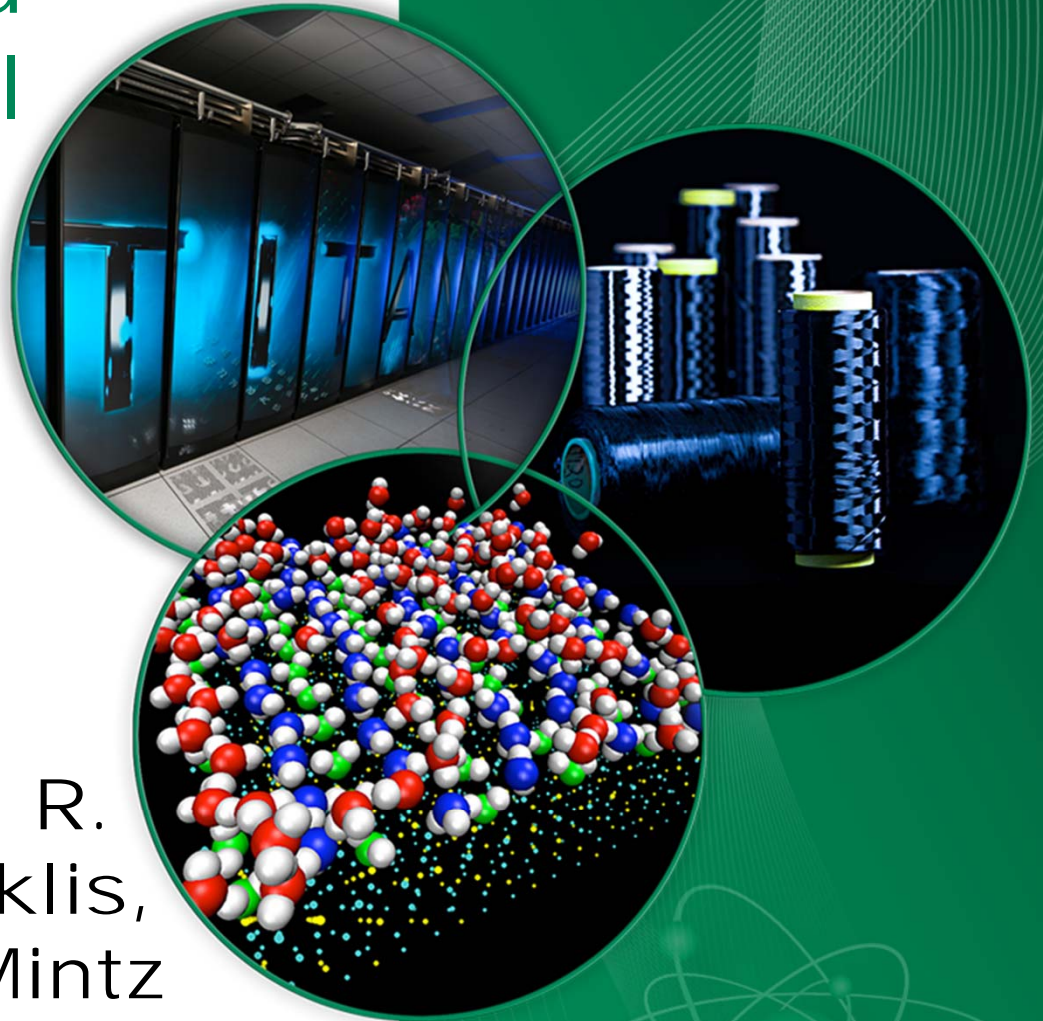


Programmer-Guided Reliability in Chapel

David E. Bernholdt, Wael R.
Elwasif, Christos Kartsaklis,
Seyong Lee, Tiffany M. Mintz
Oak Ridge National Laboratory



System Reliability at Extreme Scale

- All trends suggest increasing concerns about system reliability at extreme scales
 - Increasing node/component counts
 - Lithographic process shrinkage
 - Near-threshold voltage operation
 - Dynamic power management (thermal variability)
- Silent data corruption (SDC) is particularly insidious
 - Transient error causing bits to get flipped in storage, transmission, or computational logic
 - Typically due to cosmic ray strike, thermal or electrical fluctuation, etc.
 - Hard to get a handle on (they're "*silent*"!)

How to Address Reliability Concerns?

- To date, applications have generally relied on hardware to detect (and where possible correct) errors
- Hardware-only solutions cost \$, power, performance
 - Also tend to be blunt instruments
- Can we use software-based or HW+SW approaches to provide more tailored, more “efficient” solutions
 - Some parts of program are more vulnerable than others
 - Protecting key parts application may suffice
- Programmer generally knows much more about their code than the compiler can infer
 - Need ways to capture and communicate to compiler/system

Our Focus

- Understanding impact of and responses to transient errors at application level
 - Particularly silent data corruption
- Software-based techniques for error detection/correction
 - Potential for more flexible and tailored approach to reliability
 - Leverage programmer understanding of application
 - Can use special features of HW or lower SW layers, as available
- Understand efficacy of error detectors and their costs in energy and performance
 - (In time) identify patterns and automate, as possible
 - Locate application in R-E-P trade space and move around in controlled manner
- *Not addressing fail-stop errors in this project*
 - *Plenty of interesting R&D there too, but orthogonal*

Our Approach

1. Select demonstration applications
2. Instrument applications with various error detectors or correctors
 - Develop language extensions to capture such annotations and succinctly express common error detection patterns
3. Measure efficacy of error detectors, and their impact on performance and power through fault injection experiments
 - Develop models of resilience, energy, and performance (R-E-P) behaviors
4. Develop runtime back-end to dynamically move application in R-E-P trade space
 - Using Chapel as implementation language

Selected Demonstration Applications

Application	Description	Source	Chapel Port	Status
SSCA#1	Bioinformatics	Benchmark	Partial (1 st of 4 kernels)	Under study
SSCA#2	Graph analysis	Benchmark	Pre-existing	Under study
SSCA#3	Synthetic aperture radar and I/O	Benchmark	Except FFT, IO	Under study
LULESH	Shock hydrodynamics	LLNL co-design center mini-application	Pre-existing	Under study
HPCCG	Conjugate gradient solver	SNL Mantevo mini-application	Planned	Planned

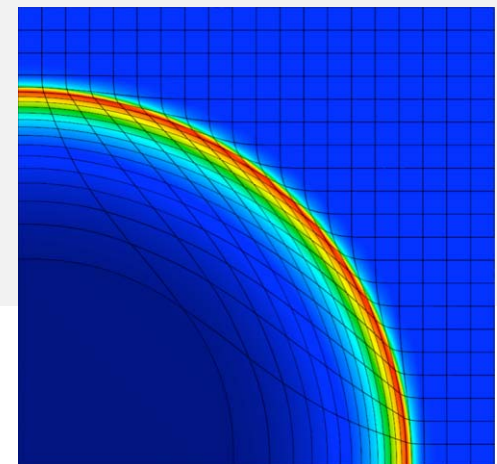
Error Detectors and Correctors

- Code provided by programmer to detect and possibly correct (data) errors
 - May utilize properties of algorithm, problem space, domain
 - Like assertions or contracts, see also containment domains
- Detectors will vary in efficacy (ability to detect errors), and have costs in both performance and energy usage
- Prefer detectors with “knobs” giving variable levels of protection (with different costs)
 - i.e. frequency of verifying checksums
- Core capability is *detection* of errors
 - *Correction* typically more complicated, requires more resources, may or may not be feasible

Error Detection Based on Problem Symmetry (LULESH)

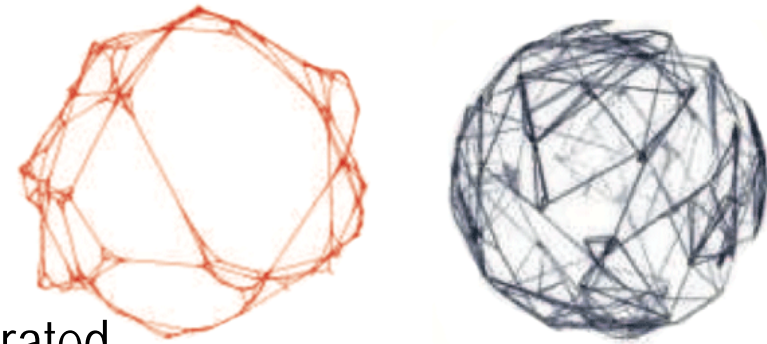
- LULESH is a shock hydrodynamics code that assumes a spherically-symmetric problem
 - Computation retains some symmetrically redundant elements
- Error detector exploits symmetry to detect and correct
 - Correction replaces with average (not the literally correct value)
 - Iterative algorithm eventually completes the “correction”
- Possible “knobs”
 - Frequency of verification
 - Density of sampling

```
void symmetry_errordetectorNrecovery() {  
    // Loop over 3d problem space  
    for (plane=0; plane<edgeNodes; ++plane) {  
        for (row=0; row<edgeNodes; ++row) {  
            for (col=0; col<edgeNodes; ++col) {  
  
                //Compare the current position vec.  
                //with three symmetric counterparts  
  
                if( asymmetry is found ) {  
                    //Update the current position vector  
                    //with average symmetric partners  
                }  
            }  
        }  
    }  
}
```



Error Detection Based on Known Ranges (SSCA#2)

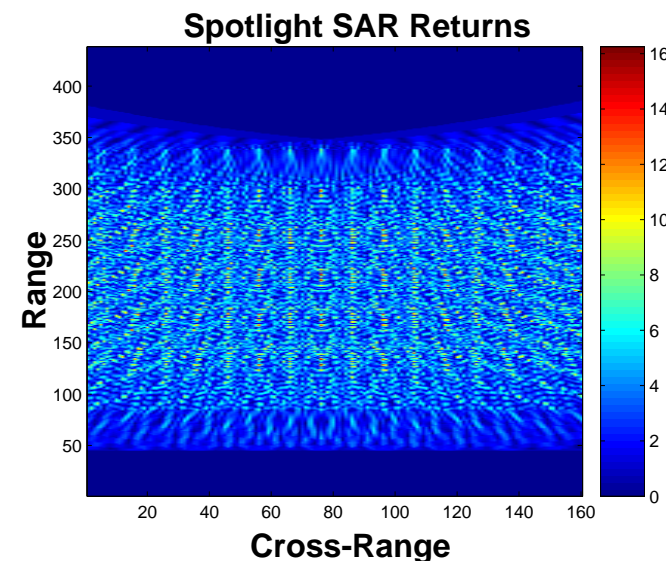
- Graph analytics application
 - Computes betweenness centrality metrics
 - Primary data structure is a table of vertices, each with a weight and set of edges, read-only after generated
- Error detector checks that edges connect to valid vertices
 - Would not detect an erroneous entry that pointed to a valid vertex
 - Does not correct errors
- Possible “knobs”
 - Frequency of verification
 - Density of sampling



```
proc checkEdges () {  
    return || reduce [ s in vertices ]  
        ( || reduce [ n in Neighbors(s) ]  
            ( n > 2**SCALE || n < 0 ) );  
}
```

Using Checksums to Detect Errors (SSCA#3)

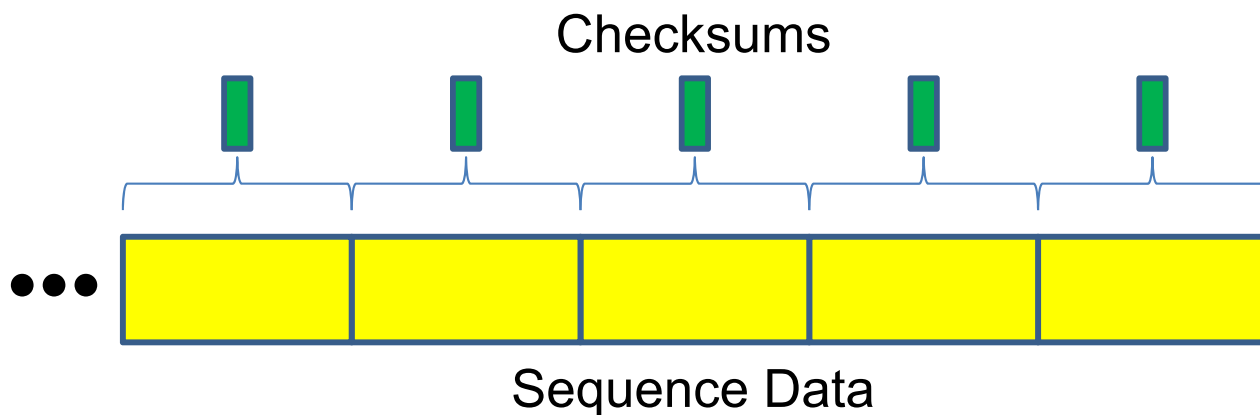
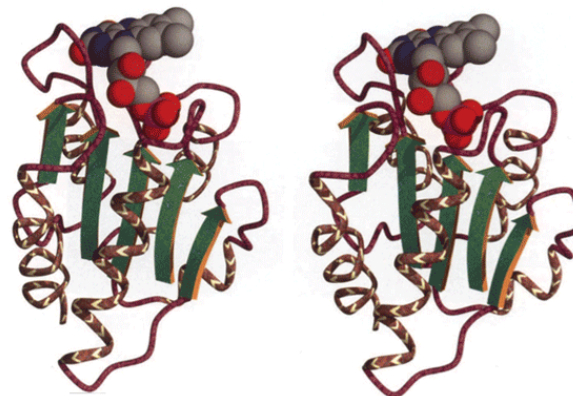
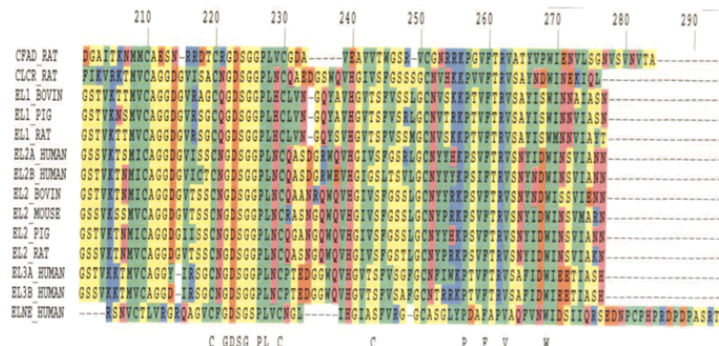
- Synthetic aperture radar processing application
 - Two stages process SAR data into images
 - Two stages compare images for target detection
- Error detector computes a checksum on a large “state” data structure which is read-mostly
 - Detection only
 - Correction would require redundant storage of state
- Possible “knobs”
 - Frequency of verification
 - Strength of checksum



```
if(!crc_cmp((crc_t *)state, sizeof(state_t), state_crc_chksum))
{
    fprintf(stderr, "\nPossible bit flip in 'state' struct\n");
    exit(1);
}
```

Blockwise Checksum with Rollback (SSCA#1)

- Bioinformatics optimal pattern matching application
 - Pairwise local alignment of sequences (Smith-Waterman)
- Error detector checksums large sequence data structures in blocks
 - Checksums can be verified as sequence is processed
- Possible “knobs”
 - Block size
 - Frequency of verification
 - Strength of checksum



TMR with Packed Data (SSCA#1)

- A key integer value is known to have a limited range (21 bits)
- Pack three copies into one 64-bit integer
 - Triply redundant storage

```
Vp = uint21_rel_unpack(V(j));  
V(j) = uint21_rel_pack(max(0, E, F(j), G));  
  
if (uint21_rel_unpack(V(j)) >= minScore &&  
    W>0.0 && uint21_rel_unpack(V(j))==G  
    && (j==m || i==n ||  
        weights(mainSeq(i+1), matchSeq(j+1))<=0.0))  
{ // core computation  
    considerAdding(V, goodEnds, goodScores,  
                  minScore, report, minSeparation, l,  
                  j, sortReports, maxReports);  
}  
  
E = max(E - gapExtend,  
        uint21_rel_unpack(V(j)) - gapFirst);
```

OO Methodology for Error Detectors in Chapel

- Construct classes to provide variable levels of protection to data and methods that provide different levels of protection/detection in processing
- Provide “quality of protection” weights for different approaches
- Provide methods to raise, lower, and reset (to highest or lowest) protection

```
class array_cnt_csum : array {  
  type t; var len: int; var data: [1..len] t; // arguments  
  var hash : int; // internal protection  
  
  proc plevel() { return (2); } // protection level  
  proc calculate() : int { return ((+ reduce data) : int); }  
  proc commit() { hash = calculate(); }  
  proc check() { assert(hash == calculate()); }  
  
  proc get(i) : t { return (data(i)); }  
  proc set(i,v) { data(i) = v; }  
  
  proc pup() : array { // switch to next protection level  
    var r = new array_tmr(t, len);  
    for i in {1..r.len} { r.data(i,1..3) = (get(i), get(i), get(i)); }  
    return (r);  
  }  
  
  proc pdown() : array { // switch to next protection level  
    var r = new array_bare(t, len);  
    r.data = data;  
    return (r);  
  }  
  ... // pre- and post- checks
```

OO Methodology (continued)

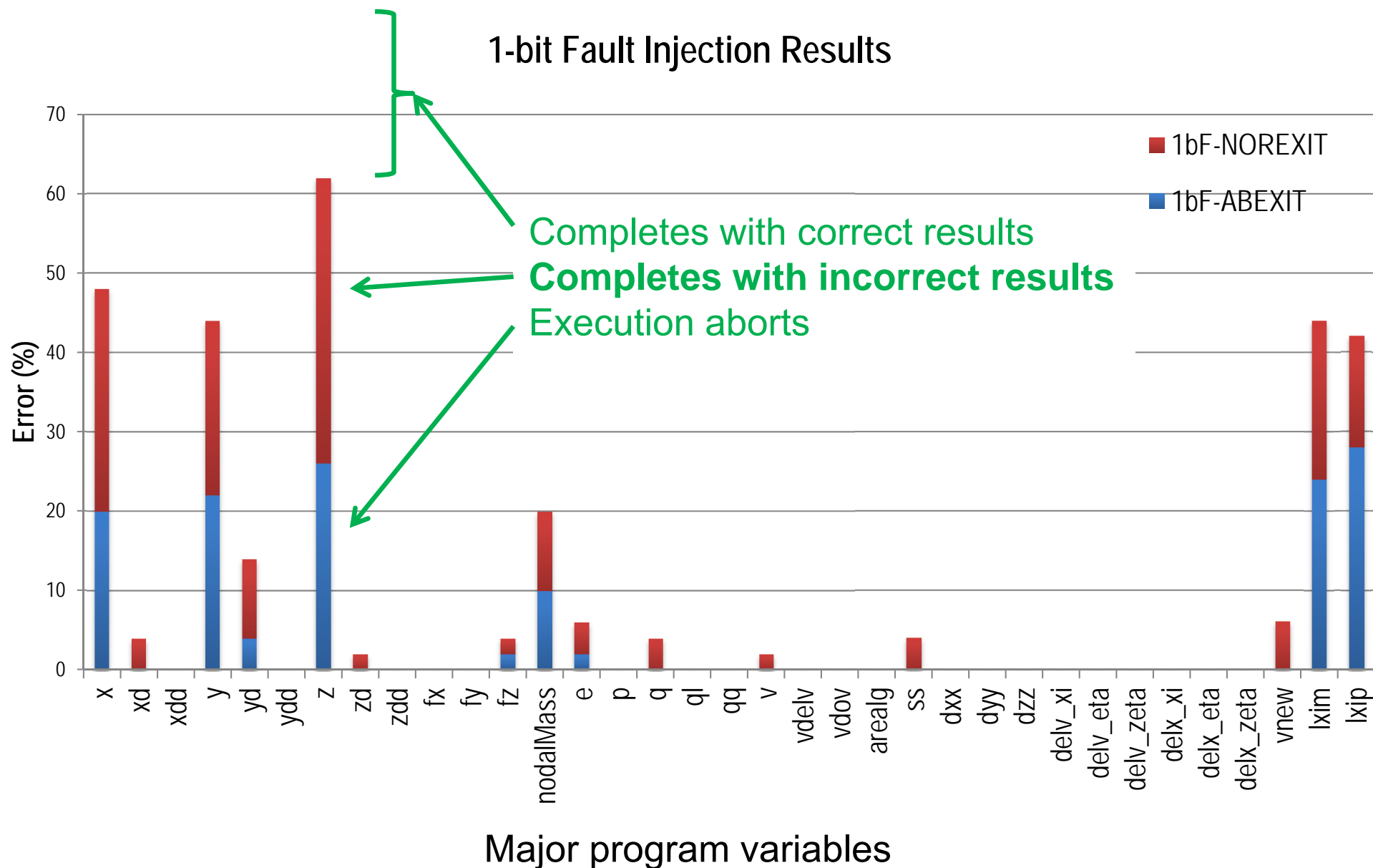
```
class dot_functor_dmr : dot_functor {  
  var d1, d2;  
  proc plevel() { return (2); }  
  
  proc run() : int { // Execute twice and compare  
    const r1 = d1.run();  
    const r2 = d2.run();  
    assert(r1 == r2);  
    return (r1);  
  }  
  
  proc pdown() : dot_functor { // next protection level  
    return (new dot_functor_default(d1.n, d1.x, d1.y));  
  }  
}
```

```
// create two protected arrays, level 1:  
var p1 : array = new array_bare(int, 3, v1);  
var p2 : array = new array_bare(int, 3, v2);  
var d : dot_functor = nil; var r : int = 0;  
  
d = new dot_functor_default(3, p1, p2);  
r = d.run();  
  
// increase level of p1: 1 -> 2  
pup(p1);  
d = new dot_functor_default(3, p1, p2);  
r = d.run();  
  
// reset p1 & p2's levels  
pmin(p1);  
pmin(p2);  
  
// increase the functor's level  
pup(d);  
r = d.run();
```

Fault Injection Studies

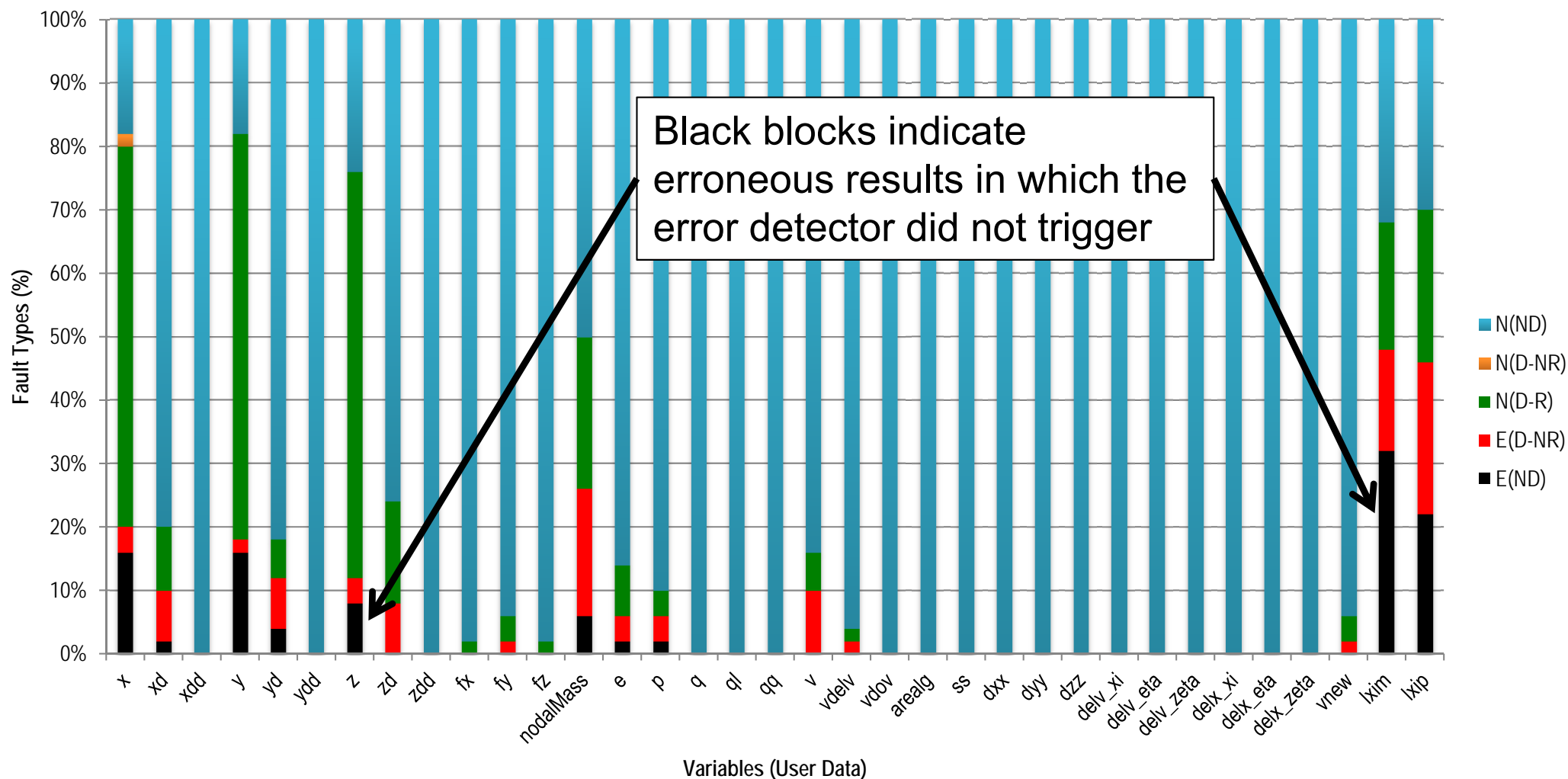
- Initially: exploratory, to help identify vulnerable code/data
- Then: characterize efficacy of detector as function of “knob” settings
 - Measure energy, performance costs

Vulnerabilities to Fault Injection (LULESH)



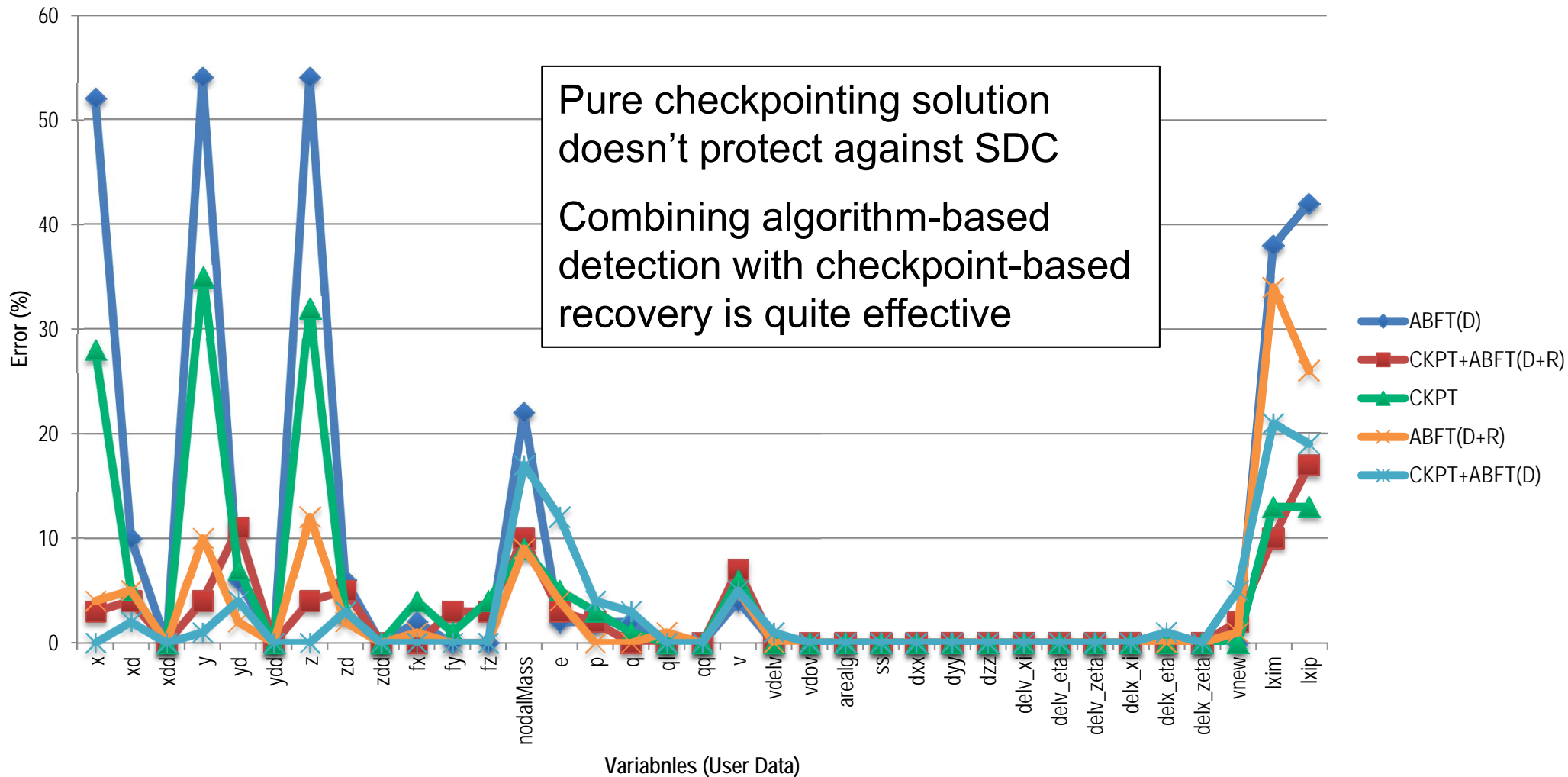
Efficacy of Symmetry-Based Error Detector/Corrector (LULESH)

Fault Behaviors of LULESH (1-bit Fault)
(Relative Error TH = 1.0E-13, 6-decimal-place outputs)



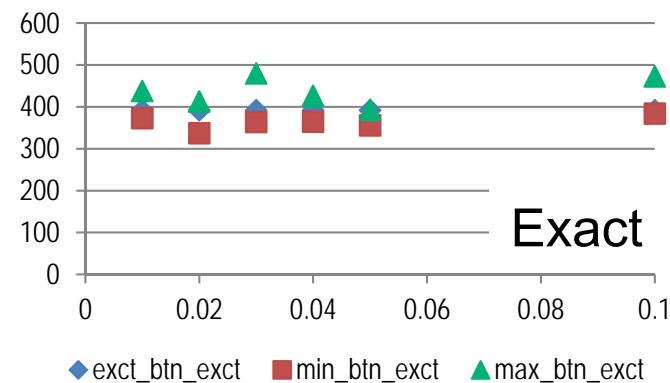
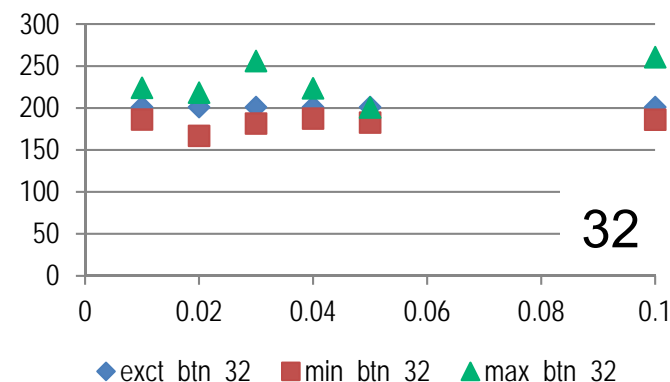
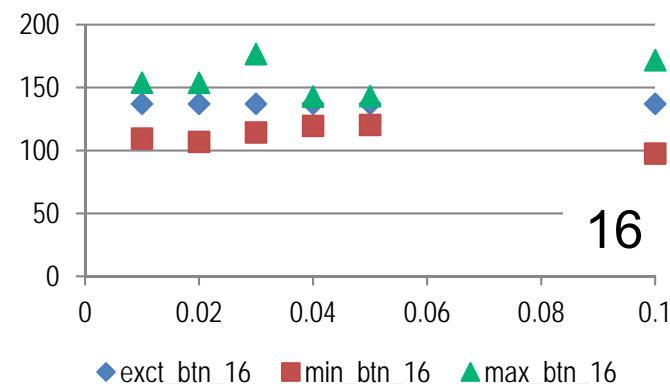
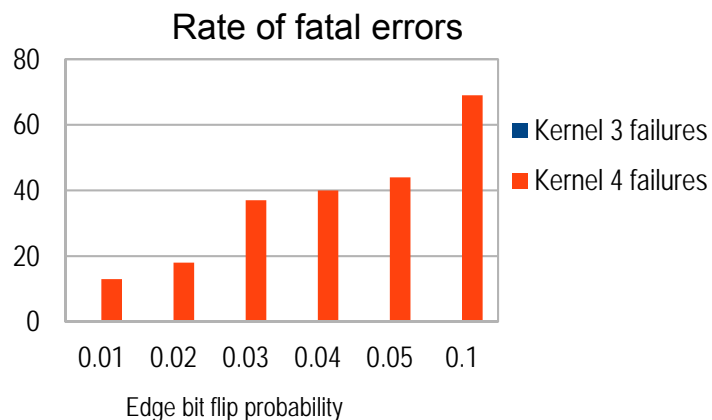
Combining Error Detection with Checkpoint/Restart (LULESH)

Fault Baviours of LULESH (1-bit Faults)
(Relative Error TH = 1.0E-13, 6-decimal-place outputs)



Magnitudes of Errors Observed (SSCA#2)

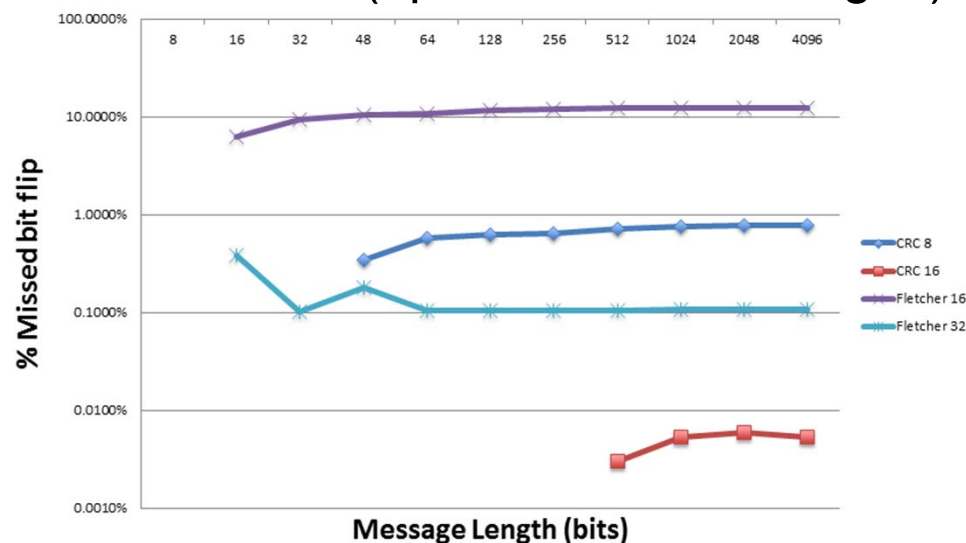
- Inject faults into edge lists only
- Inject only between computational kernels
 - In these examples, after kernel 3
- Look at results for betweenness centrality metric (Kernel 4)
 - Two approximate metrics (16, 32 starting vertices), exact metric
 - Variation due to errors significant larger in 16 metric than in 32 or exact



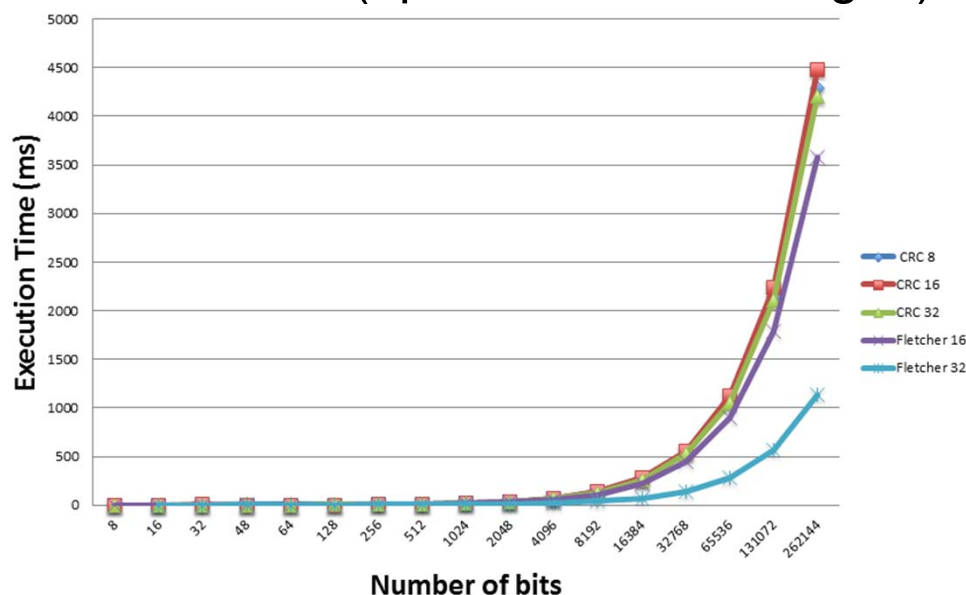
Comparing Different Checksums (SSCA#3)

- CRC and Fletcher checksums of data structure
 - Markedly different efficacies
 - CRC-32 catches all errors for these cases
 - Cost of all CRC variants is the same (< 6% variation)
 - Fletcher-16 more expensive than Fletcher-32

Errors Missed (up to 4096 b messages)



Performance (up to 256 kb messages)



Runtime Adaptation in R-E-P Trade Space

- Module in runtime to control “knobs” in error detectors
 - Informed by models of R-E-P behavior of detector
- Static settings (life of job) and dynamic control possible

Some approaches for dynamic control...

- Profile-based
 - Select error detectors based on execution phases in application profile
- Performance/energy-driven
 - Select best error detectors while staying within given E-P limits
- Symptom-based
 - Vary R depending on fault notifications
- Prediction-based
 - Choose R based on observed symptoms
 - Find best E-P point for chosen R

Extending Chapel to Support Programmer-Guided Reliability

- Initially
 - Programmer-provided code for error detection
 - May be intertwined with computational code
 - Can use OO techniques to “wrap up” a data structure with error detection
 - Need to be able to associate error detector control variable or reconfiguration routine with cost model
- Eventually
 - Identify reliability “patterns” that are common, reusable
 - Implement within module, or generate in compiler
 - Guide via annotations on target code
- Question
 - Try to cast as “regular code” or as directives/pragmas?

Possibilities for Registration of Detectors

As normal code?

```
pgr.register( detector_name, reconfig, cost);  
error detector // using R as parameter to define  
                // level of protection
```

As a pragma?

```
pragma pgr.register( detector_name, reconfig, cost);  
error detector // using R as parameter to define  
                // level of protection
```

As a structured comment?

```
//$pgr register( detector_name, reconfig, cost);  
error detector // using R as parameter to define  
                // level of protection
```

Possible “Automatic” Instantiations of Common Error Detection Patterns

Array declaration with “protect” attribute

```
const: D: domain(2) = [1..10, 1..10];  
var A: [D] real protect(checksum);
```

Declare variable with limited range of validity

```
var limited = float(-1.0, 1.0)
```

Task executed with triple redundancy

```
begin(tmr) result = important(stuff);
```

Iterator declaration with “monotonic” contract

```
iter squares(n: int): int monotonic {  
    for i in 1..n do  
        yield i*i;  
}
```

Summary

- Trends suggest that errors are going to get worse
 - Silent data corruption is particularly worrisome
- Applications will need to play an active role in detecting (and correcting) errors
- Programmers know much about what could go wrong and the impact it could have
- Give programmers tools to capture that information in the code
 - Automate common error detection patterns
- Give runtime capability to manage programmer-provided error detection
 - Need to connect detectors to back-end

This work has been supported by the DoD Advanced Computing Initiative and performed at Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.