Failures growing more common as systems become larger and more complex

- **Tera/Petascale High-Performance Systems**
  - ASCI Q: 26 radiation-induced errors/week
  - BlueGene/L: 3-4 L1 cache bit flips/day
  - Large scale codes encounter ~1 failures per day
  - Parallel file systems suffer frequent failures

- **Problem grows worse at Exascale**
Reliability a Significant Problem at Exascale

- Larger machines:
  - Sequoia system at LLNL: 1.6 million cores
  - Exascale: 200k compute chips, 3.5m DRAM chips, 300k disk drives

- Smaller circuit feature sizes (5-10nm)

- Predicted at Exascale:
  - 1 failure every 37 minutes or 3-26 minutes
  - 1 cache bit flip per minute

- At such high failure frequency, rollback isn’t an option (applications would roll back continuously)

Need tools to detect faults, identify causes

- Fault tolerance: requires fault detection
- System management: must know what failed
- Focus on complex system faults: reduction in capability of hardware resources (e.g. unexpected daemons, dropped packets, routing mis-configuration, CPU voltage scaling from overheating)

- Goal: detect fault and identify
  - Location – process most affected
  - Duration – starting and ending time
  - Fault type – e.g. CPU, Memory or Network
Need tools to detect faults, identify causes

- Fault tolerance: requires fault detection
- System management: must know what failed
- Focus on complex system faults: reduction in capability of hardware resources (e.g. unexpected daemons, dropped packets, routing mis-configuration, CPU voltage scaling from overheating)

Goal: detect fault and identify
- Location – process most affected
- Duration – starting and ending time
- Fault type – e.g. CPU, Memory or Network

Working on Statistical Techniques to Enable Resilient Applications and Systems

- Model application behavior with and without faults
- Use models to improve fault detection and identification of fault root causes
Detection and Localization of System Faults Requires Precise Models of Application Behavior

- Fault affects a fraction of application threads
- To localize fault
  - Divide execution into discrete code regions
  - Statistically model each region’s normal behavior
  - Abnormally-behaving regions indicate fault location and duration
- Detect fault type by training models on example runs with known injected faults

Detection and Localization of System Faults Requires Precise Models of Application Behavior

- Fault affects a fraction of application threads
- To detect fault type (CPU, Memory, Network)
  - Inject known fault types into application
  - Train statistical classifier on application behavior during each
  - Predict type based on application behavior in production
Application respond to faults in complex ways: makes fault analysis difficult

- **Contribution 1:** In reality faults have inconsistent effect on applications
  - CPU slowdowns only affect CPU-intensive code regions
  - Errant daemons primarily affect concurrent code
- **Contribution 2:** Automatic method to recover fault signatures from noisy application behavior data
  - Enables accurate fault localization and type prediction
Main Idea: Abstract Specification of Internal System Structure

- Statistical methods need context information about system state (features → observation)
- Relevant context often unknown to manager
- **Demonstrate:** can infer unknown context from observations given simple analytical model of system
- **Conclusion:** Division into known and inferred context very productive for modeling system behavior

Approach:
Instrument applications with behavior monitors

- **Automatically inject instrumentation points into application executable**
  - Present: Start and end of each MPI call
  - Measure system state at each point
    - Time
    - Values of performance counters
- **Application run divided into events**
  - (code regions)
  - Elapsed Time
  - Performance Counter Hits
Approach: Model Application Behavior Using Traditional Statistical Classification

- Run application
  - With no faults
  - With one of several fault types injected into execution
- Build model of application’s behavior in each case
- Identify each event’s fault location and type using model

![Diagram of Approach]

Application Run

Statistical Classifier

Train Classification Model

Event Time, Performance Counters

Duration of Memory Fault

Duration of CPU Fault

Training Runs

Non-Faulty

CPU Fault

Memory Fault

No Fault | CPU Fault | Memory Fault

Non-Faulty Training Runs

CPU Fault Training Runs

Memory Fault Training Runs
Approach: Build System Administration Tool From Single-Event Fault Predictions

- Single-event predictions too fine-grained to present to administrators
- Must be aggregated into simple fault identifications
  - On faulty runs classifier produces sequence of identical fault identifications (same location, type)
  - Aggregate entire sequence into single report to administrator

Experimental Setup Focuses on Static Applications and Multiple Fault Types

- 16-core NAS benchmarks runs (BT, CG, LU, MG, SP)
- Injected fault thread into one application core
  - Training: Three examples of each fault type
    - CPU-intensive: arithmetic operations
    - Memory-intensive: patterns of random memory updates
    - Socket-intensive: thread communicates with self
  - Evaluation:
    - NoFault: Application runs with no faults injected
    - KnownFault: Applications injected with faults used for training
    - UnknownFault: Applications injected with faults similar to those used in training
Traditional classification techniques fail for system faults

- Accuracy of Boosted Logit classifier in detecting and localizing faulty events
  (same results for other classifiers)

![Graph showing success rate for faulty and non-faulty runs for different classifiers]

- Traditional classification techniques fail for system faults
  - Accuracy of Boosted Logit classifier in detecting and localizing faulty events
    (same results for other classifiers)
  - Most faults missed
  - Majority of Fault Detections Incorrect
  - Majority of Fault Detections Incorrect
  - Low false-positive rate
  - Adding more additional features doesn’t improve accuracy
Accuracy of Naïve Method is Low Because System Faults Have Erratic Effects

- Can train probability distributions as above to measure probability each state

Can Improve Accuracy by Combining Probability and Classifier Models

- During fault only few events are abnormal
- These events truly representative of the fault
- Filter events in faulty runs to label only abnormal events as Faulty
- Other events labeled as Non-Faulty
Can Improve Accuracy by Combining Probability and Classifier Models

- During fault only few events are abnormal
- These events truly representative of the fault
- Filter events in faulty runs to label only abnormal events as Faulty
- Other events labeled as Non-Faulty

Filtered labeling
- Significantly improves data to noise ratio of training data
- Produces more accurate model
Abnormality-Filtered Classifiers Detect Faults More Accurately

Abnormality-Filtered Classifiers Have More False Positives

More Accurate Filtered Classifier Has More False Positive Detections
Improved Technique Detects Fault’s Location, Time and Type

Application behavior is a random process managed by hidden state

- Behavior of each code region depends on separate probability distribution
- Composition of multiple sub-behaviors
- Choice controlled by hidden process
e.g. region vulnerable/invulnerable to fault
To train accurate models we must extract hidden state and explicitly represent it.

By recovering the hidden state behind individual observations, the behavioral dynamics of code regions and faults are recovered!

Hidden variable inference: general approach to modeling system behavior

- Human manager: specifies analytical model for major dynamics of system behavior

- Infer system state from observed events

Observed events more constrained and predictable
Proposal: New Way for Administrators to Interact with Statistical System Models

- Known context: specify directly
  (e.g. node/core ID, code region, input data details)
  (code regions)

- Unknown context:
  (e.g. code region vulnerability, effective network load)
  - Specify implicitly via approximate analytic model of system behavior
  - Infer current state within model from observations

Developed statistical modeling technique focused on complexities of system faults

- Traditional modeling approaches fail on system faults because they can’t capture internal system state
- Developed filtering approach to infer hidden state from observed events
- General modeling approach:
  - System administrators develop simple analytical models of system behavior
  - Modelers infer current model parameters from observations
Developed statistical technique to localize and characterize complex system faults

- Traditional modeling approaches fail on system faults because they ignore internal system state
- Developed statistical modeling technique to accurately detect and localize system faults
- Developed filtering approach to infer hidden state from observed events
- Significant improvement in detection and localization accuracy over naïve classifier
  - Small increase in false alarm incidences