Probabilistic Fault Detection and Diagnosis in Large-Scale Distributed Applications

Ignacio Laguna
PhD’s Final Examination

Major Professor:
Prof. Saurabh Bagchi

Committee Members:
Prof. Samuel Midkiff
Prof. Y. Charlie Hu
Martin Schulz

Lawrence Livermore National Laboratory

Nov 27, 2012

---

Bugs Cause Million of Dollars Lost in Minutes

Amazon Web Services Outage Caused By Memory Leak And Failure In Monitoring Alarm

Amazon failure took ~6 hours to fix

Need for automatic problem-determination techniques to reduce diagnosis time
Failures in Large-Scale Applications are More Frequent

The more components the higher the failure rate

Faults come from:
- Hardware
- Software
- Network

Bugs from many components:
- Application
- Libraries
- OS & Runtime system

Multiple manifestations:
- Hang, crash
- Silent data corruption
- Application is slower than usual

Debuggers Need to Handle High Degree of Parallelism

- 100 million cores in Exascale HPC applications (in 2020)
  - 100 million different threads or processes executing simultaneously

- Most of the current parallel debuggers scale poorly
  - Bottleneck in handling data from many parallel processes
  - Data is analyzed in a central point (rather than distributed)
  - Generate too much data to analyze
Problems of Current Diagnosis/Debugging Techniques

- Poor scalability
  - Inability to handle large number of processes
  - Generate too much data to analyze
  - Analysis is centralized rather than distributed
  - Offline rather than online
  FlowChecker (SC’09), DMTRacker (SC’07), A. Vo (PACT'11)

- Problem determination is not automatic
  - Old breakpoint-based debugging (> 30 years old)
  - Too much human intervention
  - Requires large amount of domain knowledge
  TotalView®, DDT®, GDB, D3S (NSDI'08), model checking (Crystal ball – NSDI'09)

Focus of My Dissertation

- Detection
  - Detect that a problem exists

- Diagnosis
  - Root-cause analysis
  - Pinpoint faulty component

- Recovery
  - Checkpointing
  - Micro-rebooting
  - Redeployment

Prelim Exam
Fault Detection in HPC and Commercial Applications

Papers:
Supercomputing 2011
DSN 2010
Middleware 2009

Final Exam
Problem Localization in HPC and Commercial Applications

Paper:
PACT 2012
Remaining Agenda

Problem Localization

Distributed Applications

Scientific Applications
MPI, OpenMP

Commercial Applications
BigData, Java

Related Work and Conclusions

Some Failures Manifest Only at Large Scale

Molecular dynamics simulation code (ddcMD)

Failure Characteristics

- Application *hangs* with 8,000 MPI tasks
- Manifestation is intermittent
- Large amount of time spent on fixing the problem
- *Our technique isolated the problem origin in a few seconds*
Explanation for an Application’s Hang: 
*The Least-Progressed Task*

The Least-progressed task: 
*The task behind the others*

---

The Progress Dependence Graph

Tasks B and C can’t make progress because of task A

Tasks A doesn’t have any progress dependence
How do we define “Progress”? 

- Need notion that an MPI process is moving toward final state
  - Idea: keep track of executed states per process
  - States are executed “code regions”

Summarize Execution History Using a Markov Model

Sample code

```c
foo() {
  MPI_gather( )
  // Computation code
  for (…) {
    // Computation code
    MPI_Send( )
    // Computation code
    MPI_Recv( )
    // Computation code
  }
```

Finite State Machine with Transition Probabilities

- Gather call stack
- Create states in the model
What Tasks are Progress Dependent on other Tasks?

**Point-to-Point Operations**

Task X:

```c
// computation code...
MPI_Recv(..., task_Y, ...)
// ...
```

- X depends on task Y
- Dependency can be obtained from MPI-call parameters

**Collective Operations**

Task X:

```c
// computation code ...
MPI_Reduce(...)
// ...
```

- Multiple implementations (e.g., binomial trees)
- A task can reach MPI_Reduce and continue
- Task X could block waiting for another task (less progressed)

---

**Probabilistic Inference of Progress-Dependence Graph**

Sample Markov Model

```
Task A
```

```
1 1.0 0.7
2 0.3
3
```

```
Task B
```

```
4 1.0
5
6
```

```
Task C
```

```
7 1.0
8
9
```

```
Task D
```

```
10 1.0
```

```
Task E
```

```
```

Progress dependence between tasks B and C?

- Probability(3 -> 5) = 1.0
- Probability(5 -> 3) = 0

Task C is likely waiting for task B (A task in 3 always reaches 5)

C has progressed further than B
Resolving Conflicting Probability Values

Sample Markov Model

Dependence between tasks B and D?
- Probability(3 → 9) = 0
- Probability(9 → 3) = 0
The dependency is null

Dependence between tasks C and E?
- Probability(7 → 5) = 1.0
- Probability(5 → 7) = 0.9
**Heuristic: Trust the highest probability**

| C is likely waiting for E |

Distributed Algorithm to Infer the Graph

- All-reduction of current states
- All tasks know the state of others
- Build (locally) progress-dependence graph
- Reduction of progress-dependence graphs

**Reductions are O(log #tasks)**
### Examples of Reduction Operations: Dependence Unions

**X → Y: X is progress dependent on Y**

<table>
<thead>
<tr>
<th>Task A</th>
<th>Task B</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>X → Y</td>
<td>X → Y</td>
<td>X → Y (Same dependence)</td>
</tr>
<tr>
<td>X → Y</td>
<td>Null</td>
<td>X → Y (First dominates)</td>
</tr>
<tr>
<td>X → Y</td>
<td>Y → X</td>
<td>Undefined (or Null)</td>
</tr>
</tbody>
</table>

### Example of Distributed Algorithm to Infer Dependence Graph

1. Send only non-null dependencies:
   - 2 → 1
   - 3 → 1
   - 4 → 1
   - 2 → 3
   - 3 → 2
   - 4 → 2

2. Build progress-dependence graph:
   - 1 → 2
   - 1 → 3
   - 1 → 4
   - 2 → 1
   - 3 → 1
   - 3 → 2
   - 4 → 1

(1) Create dependencies locally
(2) Send only non-null dependencies
(3) Build progress-dependence graph
Progress Dependence Graph of Bug

Hang with ~8,000 MPI tasks in BlueGene/L

[3136] Least-progressed task

Our tool finds that MPI task 3136 is the origin of the hang

• How did it reach its current state?

Finding the Faulty Code Region: Program Slicing

done = 1;
for (...) {
  if (event) {
    flag = 1;
  }
}

if (flag == 1) {
  MPI_Recv();
  ...
}
...
if (done == 1) {
  MPI_Barrier();
}
Slice with Origin of the Bug

```c
int dataWritten = 0;
for (...) {
    MPI_Probe(..., &flag, ...);
    if (flag == 1) {
        MPI_Recv();
        MPI_Send();
        dataWritten = 1;
    }
    MPI_Send();
    MPI_Recv();
    // Write data
    if (dataWritten == 0) {
        MPI_Recv();
        MPI_Send();
    }
    Reduce();
    Barrier();
}
```

Dual condition occurs
- A task is a writer and a non-writer at the same time

**MPI_Probe** checks for source, tag and comm of a message
- Another writer intercepted wrong message

Programmer used unique MPI tags to isolate different I/O groups

---

Controlled Evaluation

- Used two Sequoia benchmarks (AMG, LAMMPS) and six NAS Parallel benchmarks

- Faults injected in two Sequoia benchmarks:
  - AMG-2006 and LAMMPS
  - Injected a hang in random MPI tasks
  - Only injected in executed functions (MPI and user functions)

- Perform slowdown and memory usage evaluation in all benchmarks
Accurate Detection of Least-Progress Tasks

- Least-progressed task detection recall:
  - Cases when LP task is detected correctly
- Imprecision:
  - % of extra tasks in LP tasks set

Example Runs: 64 tasks, fault injected in task 3

Example 1
[1,5,...] LP task detected Imprecision = 0
[2,4,...] [0,6,8,...]

Example 2
[1,9,...] [2,...] LP task detected Imprecision = 2/3
[3, 5, 4]

- Overall results:
  - Average LP task detection recall is 88%
  - 86% of injections have imprecision of zero

Performance Results

Least-Progress Task Detection Takes a Fraction of a Second
Performance Results:
Slowdown is Small For a Variety of Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Slowdown</th>
<th>Memory-usage Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAMMPS</td>
<td>1.50</td>
<td>6.11</td>
</tr>
<tr>
<td>AMG2006</td>
<td>1.46</td>
<td>10.36</td>
</tr>
<tr>
<td>BT</td>
<td>1.08</td>
<td>3.75</td>
</tr>
<tr>
<td>SP</td>
<td>1.67</td>
<td>5.14</td>
</tr>
<tr>
<td>CG</td>
<td>1.14</td>
<td>2.21</td>
</tr>
<tr>
<td>FT</td>
<td>1.05</td>
<td>1.01</td>
</tr>
<tr>
<td>LU</td>
<td>1.39</td>
<td>5.37</td>
</tr>
<tr>
<td>MG</td>
<td>1.04</td>
<td>1.04</td>
</tr>
</tbody>
</table>

- Tested slowdown with NAS Parallel and Sequoia benchmarks
  - Maximum slowdown of ~1.67
- Slowdown depends on number of MPI calls from different contexts

Remaining Agenda

Problem Localization

Scientific Applications
MPI, OpenMP

Commercial Applications
BigData, Java

Related Work and Conclusions
Commercial Applications Generate Many Metrics

How can we use these metrics to localize the root cause of problems?

Research Objectives

- Look for abnormal time patterns
- Pinpoint code regions that are correlated these abnormal patterns
Bugs Cause Metric Correlations to Break

- Hadoop DFS file-descriptor leak in version 0.17 (2008)

- Correlations are different when the bug manifests itself:
  - Metrics: open file descriptors, characters written to disk

![Comparison of Normal and Failed Runs](Image)

Approach Overview

1. Find Abnormal Windows
2. Find Abnormal Metrics
3. Find Abnormal Code Regions
Selecting Abnormal Window via Nearest-Neighbor (NN)

- Sample of all metrics
- Annotated with code region

Correlation Coefficient Vectors (CCV)
\[ [cc_{1,2}, cc_{1,3}, \ldots, cc_{n-1,n}] \]

Nearest-Neighbor to find Outliers

Selecting Abnormal Metrics by Frequency of Occurrence

- Contribution of correlation coefficient to the distance

Example:

<table>
<thead>
<tr>
<th>Steps</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Get abnormal windows</td>
<td>CC_{6,1} 0.1, CC_{5,1} 0.7, CC_{10,11} 0.2</td>
</tr>
<tr>
<td>2. Rank Correlation Coefficients (CC) based on contribution to the distance</td>
<td>CC_{5,2} 0.5, CC_{5,2} 0.05, CC_{15,16} 0.5</td>
</tr>
<tr>
<td>3. Select the most frequent metric(s)</td>
<td>Abnormal metric: 5</td>
</tr>
</tbody>
</table>
Selecting Abnormal Code-Regions

- Same technique as before:
  - Nearest neighbor approach
  - Focus only one metric (i.e., the abnormal metric)

Find abnormal windows
(using only one metric)

[Diagram showing windows X, Y, and Z ranked based on abnormality]

Select code regions that occur frequently in abnormal windows

Case 1: Hadoop DFS

- File-descriptor leak bug
  - Sockets are left open in the DFSClient Java class
  - 45 classes and 358 methods instrumented (as code regions)

Output of the Tool

2nd metric correlates with origin of the problem

Java class of the bug site is correctly identified
Case 2: HBase

- Deadlock in version 0.20.3 of Hbase (2010)
  - Incorrect use of locks
  - Bug site is the HRegion class

Output of the Tool

Abnormal metrics don’t provide much insight

HRegion appears as the abnormal code region

Remaining Agenda

Problem Localization

Distributed Applications

Scientific Applications
MPI, OpenMP

Commercial Applications
BigData, Java

Related Work and Conclusions

PACT 2012

Submitted to NSDI’12
Related Work

Logs and Metrics Analysis
- K. Ozonat (DSN’08)
- I. Cohen (OSDI’04)
- P. Bodik (EuroSys’10)
- K. Nagaraj (NSDI’12)

Debugging
Serial
- Relative debugging
- Memory checkers (Valgrind)
- Statistical debugging
- Dynamic invariants (DIDUCE)
- Delta debugging
Parallel
- STAT (SC’09)
- MPI correctness checkers
- TotalView, DDT
- FlowChecker (SC’09), DMTTracker (SC’07)

Model Checking
- C. Killian (NSDI’07)
- J. Yang - Modist (NSDI’09)
- H. Guo (SOSP’11)
- Cmc, M. S. Musuvathi (OSDI’02)

Failure Prediction
- I. Cohen (OSDI’04)
- Tiresias, (IPDPS ’07)
- A. Gainaru, prediction in HPC (SC’12)

Conclusion

• Fault detection and diagnosis can be scalable
  – Use of “computationally cheap” models
  – Can diagnose problems with 100,000 parallel tasks
  – Slowdown ~ 1.7 times application run time

• Techniques tested in real-world bugs and fault injections
  – Molecular-dynamics code bug @ LLNL
  – NAS Parallel benchmarks, Sequoia benchmarks
  – Commercial application bugs: Hadoop, Hbase, ITAP and IBM app.

• Diagnosis takes less time than traditional debuggers
  – Detection of least-progress task takes less than a second
  – Code regions where bugs manifest themselves are highlighted
Lessons Learned

- Different kinds of machine learning algorithms are good for different problems
  - Algorithms that are fast in testing phase are appropriate for HPC

- Finding the right kind of instrumentation is extremely important
  - Too much: not scalable and too much slowdown
  - Too few: not enough data to train statistical models

- Problem determination at a line-of-code granularity is challenging
  - But code-region granularity works well for many failures

Thanks to Contributors!

Purdue University

Prof. Saurabh Bagchi
Prof. Samuel Midkiff
Nawandol Theera-Ampompunt
Fahad A. Arshad

Lawrence Livermore National Laboratory

Bronis R. de Supinski
Greg Bronevetsky
Martin Schulz
Todd Gamblin
Dong Ahn
Thank you!

Backup Slides
Future Work

- Use of more complex dependencies between metrics
  - Non-linear dependencies
- Apply failure prediction techniques in HPC applications
  - Via analysis of metrics or system/application log analysis
- More general strategy for creating task’s state-machine (i.e., Markov model)
  - Sampling of user-level functions
- What metrics are useful in fault detection and diagnosis?
  - Are hardware metrics useful? (e.g., hardware counters)
- Handling failures in HPC systems from the application
  - Instead of killing all the processes, let the application continue with healthy processes

What if we have different Markov Models in different tasks?

- First, dependencies are built locally based on local information
- Second, dependence unions (in the distributed reduction) take care of null (or undefined) dependencies.

Global View of Markov Model

As seen from Task 2

As seen from Task 3

Dependencies:
2 → 1
2 X→ 3 (undefined)

Dependencies:
3 → 1
3 X→ 2 (undefined)

Result of Dependence Reduction:
2 → 1
3 → 1
Binomial Tree Implementation of MPI_Reduce

Code region 1
Task 5 blocks here

Code region 2: MPI_Reduce
Tasks 1, 6, 7 are progress dependent on 5

Code region 3
Tasks 2, 3, 4, 8 move to the next code region

Bug (Case Study)

R: reader
W: writer

Same message tags are used even in different groups

Bug: dual condition (a task is reader and a writer for different I/O groups)

BlueGene/L
Compute nodes perform I/O via dedicated I/O nodes

Linux cluster
Fault Injection Results for LAMMPS Application

- 88% of the time the least-progress task is detected.
- Every time is not detected, it's isolated.
- 86% of injections has imprecision of zero.

Sample Results of the Tool
Performance Results:
Least-Progress Task Detection Takes a Fraction of a Second

Correlation Coefficient Formula (Pearson)

\[ cc(X, Y) = \frac{1}{N-1} \sum_{k=1}^{N} \left( \frac{X_k - \bar{X}}{s_X} \right) \left( \frac{Y_k - \bar{Y}}{s_Y} \right) \]

- \( N \): number of samples
- \( \bar{X} \): mean
- \( s_X \): standard deviation

Slide 49/40

Slide 50/40
Hadoop’s Bug - Profile

Table 2: Average use of file descriptors per class in HDFS for the specific bug discussed in Section 4.1.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Class</th>
<th>Average # File Descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NamespaceInfo</td>
<td>6.0</td>
</tr>
<tr>
<td>2</td>
<td>INodeDirectory</td>
<td>1.31</td>
</tr>
<tr>
<td>3</td>
<td>INode</td>
<td>1.29</td>
</tr>
<tr>
<td>4</td>
<td>UnderReplicatedBlocks</td>
<td>1.25</td>
</tr>
<tr>
<td>5</td>
<td>DataNodeInfo</td>
<td>1.24</td>
</tr>
<tr>
<td>6</td>
<td>DataNode</td>
<td>1.21</td>
</tr>
<tr>
<td>7</td>
<td>DataNodeBlockInfo</td>
<td>1.2</td>
</tr>
<tr>
<td>8</td>
<td>DFSClent</td>
<td>1.16</td>
</tr>
<tr>
<td>9</td>
<td>DataBlockScanner</td>
<td>1.14</td>
</tr>
<tr>
<td>10</td>
<td>NameNode</td>
<td>1.13</td>
</tr>
</tbody>
</table>

Metrics Gathering: Multi-metric Profiling

Program
- Code Region 1
- Code Region 2
- Code Region 3
- Code Region 4

Collect metrics measurements: [0.5, 100, 34, 5.66, 3398, 2, ...]

**Synchronous**
- Sample at the beginning and end of code regions
- Granularity: Java class/methods calls
- Incur high overhead

**Asynchronous**
- Separate process sample metrics
- Do not interfere with application
- Inaccuracies in mapping samples to code regions
Case 3: IBM Mambo Health Monitor (MHM)

- Regression-test system for IBM Full System Simulator (Mambo)
  - **Mambo**: arch. simulator for systems based on IBM's Power(TM)

- Example of typical failures:
  - Problem with the simulated architecture
    - **NFS connection fails intermittently**
  - Failed LDAP server authentications
  - /tmp filling up

Focus of experiments:  
- **Fault injection**
Case 3: MHM Results

Abnormal metrics are correlated with the failure origin: *NFS connection*

- Abnormal code-region is selected *almost* correctly
  - Asynchronous profiling technique cause inaccuracies