Managing Application Resilience: A Programming Language Approach

Saurabh Hukerikar, Pedro C. Diniz and Robert F. Lucas
{saurabh, pedro, rflucas@isi.edu}
Information Sciences Institute

Presentation at the 2014 Mardi Gras Conference
Louisiana State University
Feb. 28, 2014
Why are We Here?

Increasing Soft- or Transient Errors:
- Shrinking feature sizes
  - Voltage level and noise margins decrease
- Hostile environments: Space or altitude
  - Even sea-level!
- Processor ageing
  - Delay errors
  - Latches no longer function / Errors in logic

*Single Event Upsets* are common cause of *soft errors*
Not permanent / no hardware damage
Errors are not New

- **Basic Idea: Redundancy in Space and Time**
  - Software/Hardware Triple-Modular Redundancy (TMR)
  - Other Variants: DWC, Redundant Threading...

- **Software approaches are Slow**
  - Checkpoint and restart does not scale
    - MTBF < Checkpoint cost
  - Code bloat from fine-grained approaches

- **Hardware approaches are Expensive**
  - TMR has high area/power cost
  - Inflexible (“always on”)

---
Resilient Computing Techniques

- EDDI, ThOR, etc.
- Resiliency-aware Scheduling
- TMR, TR, DWC-CED
- SOI, DICE, Hardening-by-process

Checkpoint and Restart

Less time

Less hardware

Coarse Grain

Fine Grain

Coarse Grain

Fine Grain

Software

Hardware
Memory Errors? What Errors?
(Data Courtesy of Al Gueist, ORNL 2011)

What About ECC?
• Transient error bit flips in memory
• ECC fixes single bit errors, but not double bit errors.

How Serious?
• Jaguar has a lot of memory (362 TB)
• It endures a constant stream of single bit errors
• It has a double bit error about every 24 hours
• ChipKill allows the system to run through double bit errors but DRAM can not correct double bit errors
Memory Errors? What Errors?
(Data Courtesy of Al Gueist, ORNL 2006)

Is This a Problem at Exascale?
• Exascale system target is 128 PB of memory (354 times Jaguar)
• Translates into a double bit error about every 4 mins
• Frequent enough to need something better than ChipKill at Exascale

We need to be Smart about Errors
• Not all errors are Identical...
• Not all errors can be ignored...
Application: Where and When Do Errors Matter?
- Type declarations
- Code region selection
- Dynamic memory allocation

Operating System: Reasons about the severity and significance of memory faults due to soft errors to the address space
- Crash?
- Ignore?
- Correct!
Introspective Engine: Observe and React to Trends:
- Avoid Specific Memory Pages and/or Cores
- Migrate Pages and Processes
Provide Linguistic Mechanisms to Convey Criticality of Errors

• Emphasis on keeping things simple
• Existing libraries and application code base need not be entirely rewritten
• Complementary:
  • Checkpoint & Restart: increase checkpoint interval
  • Redundant Computation: potential energy savings
• Inherent Algorithmic Fault Tolerance knowledge of an application
  • still there are no convenient mechanisms to convey this knowledge to the lower levels of system abstraction layers
• Engage all software layers including the application
  • Compiler infrastructure,
  • Operating system
• Fault Model: SECDED Failures
• tolerant int rgb[XDIM][YDIM];

• tolerant<MAX.VALUE=...> unsigned int counter;

• tolerant<precision.6f> double low_precision;

S ← Exponent → Fraction
<type>* <var> = (<cast>) tolerant_malloc(sizeof(<type>));

<sizeof><type>*</sizeof> <var> = (<cast>) tolerant_malloc(NUM * sizeof(<type><MAX.VALUE=..>));
Detection and Recovery:
Robust Code Blocks

```c
#pragma robust sentinel <variable list>
{
    <code>
}
```

```c
#pragma robust sentinel <predicate list>
{
    <code>
}
```

• Possible Uses of Variable/Predicate Lists:
  - Use of Variables List as the only non-tolerant storage
  - Use of Predicate List for “soft-error checkers”
Detection and Recovery:
Robust Code Blocks

```c
#pragma robust sentinel <variable list>
{
    <code>
}
```

```c
#pragma robust sentinel <predicate list>
{
    <code>
}
```

- How to use this information?
  - Dual Threading uses predicate for selection of correct execution
  - Triple Treading uses variables for voting followed by checking of predicates
  - Can be Adaptive
    - Dual threading for simple detection
    - Roll-back and proceed with Triple-Threading
Robust Program Elements – the ‘critical’Pragma

• Source code #pragma recognized by ROSE Compiler

```c
#include "new_critical_var.h"

int main(int argc, char **argv) {
    #pragma critical
    int *A_j = NULL;
    int A_i[100];
    int y_data[100], x_data[100], A_data[100];
    int jj = 0, i = 0;

    //A_j = (int *) malloc (sizeof(int) * 100);

    for (jj = A_i[i]; jj < A_i[i + 1]; ++jj) {
        y_data[i] += A_data[jj] * x_data[A_j[jj]];
    }

    free(A_j);
}

#include "new_critical_var.h"

int main(int argc, char **argv) {
    #pragma critical
    int *A_j = (int *)((int *)((void *)0));
    int *A_j0 = (int *)((int *)((void *)0));
    int *A_j1 = (int *)((int *)((void *)0));
    int A_i[100UL];
    int y_data[100UL];
    int x_data[100UL];
    int A_data[100UL];
    int jj = 0;
    int i = 0;

    //A_j = (int *) malloc (sizeof(int) * 100);
    for (jj = A_i[i]; jj < A_i[i + 1]; ++jj) {
        y_data[i] += (A_data[jj] * x_data[((int *)(triplication(A_j, A_j0, A_j1))[jj]]);
    }

    free(((int *)(triplication(A_j, A_j0, A_j1))));
    return 0;
}

void* triplication(p1, p2, p3) {
    if (p1 == p2 || p1 == p3) {
        return p1;
    } else if (p2 == p3) {
        return p2;
    } else {
        exit(1);
    }
}
```
int main (int argc, char** argv) {
    #pragma critical(3)
    int* A_j = NULL;
    int A_i[100];
    int y_data[100], x_data[100], A_data[100];
    int jj = 0, i = 0;
    //A_j = (int *) malloc (sizeof(int) * 100);

    for (jj = A_i[i]; jj < A_i[i] + 1; ++jj) {
        y_data[i] += A_data[jj] * x_data[A_j[jj]];
    }

    free(A_j);
}

#include "new_critical_var.h"

int main(int argc, char **argv){
    #pragma critical 3
    int *A_j = (int *)((int*)((void *)0));
    int *A_j0 = (int *)((int*)((void *)0));
    int *A_j1 = (int *)((int*)((void *)0));
    int A_i[100UL];
    int y_data[100UL];
    int x_data[100UL];
    int A_data[100UL];
    int jj = 0;
    int i = 0;
    //A_j = (int *) malloc (sizeof(int) * 100);
    for (jj = A_i[i]; jj < A_i[i] + 1; ++jj) {
        y_data[i] += A_data[jj] * x_data[A_j[jj]];
    }

    free((int*)((int*)((void*)0)));

    return 0;
}

void* triplication(p1, p2, p3) {
    if (p1 == p2 || p1 == p3) {
        return p1;
    } else if (p2 == p3) {
        return p2;
    } else {
        exit(1);
    }
}
Simple Experiments

• Methodology:
  - “Inject” errors in memory
  - Check if storage has been labeled as tolerant
  - If so ignore; else continue (and possibly crash later)

• Simple Codes:
  - HPCC Random Access
  - Molecular Dynamics
  - Algebraic Multi-Grid Solver
  - Graph - BFS

• Amelioration:
  - Ignore in many cases (AMG will naturally converge...)
  - Remove offending data in other cases (MD code)
Simple Experiments: System Workflow
## Experimental Results

<table>
<thead>
<tr>
<th>Fault Injection Rate (minutes)</th>
<th>% Execution Runs to Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Random Access</td>
</tr>
<tr>
<td>15</td>
<td>99.5 %</td>
</tr>
<tr>
<td>10</td>
<td>99.2 %</td>
</tr>
<tr>
<td>5</td>
<td>99.1 %</td>
</tr>
<tr>
<td>2</td>
<td>97.4 %</td>
</tr>
<tr>
<td>1</td>
<td>96.1 %</td>
</tr>
</tbody>
</table>

*Fault injection: Multi-bit faults, non-recoverable by ECC*
Vulnerability Analysis: Execution Lifetime

- HPCC Random Access
- Molecular Dynamics Simulation
- Algebraic Multigrid Solver
- Graph Breadth-First-Traversal
Vulnerability Analysis: Execution Lifetime

Safely Ignore Errors:
Programmer knows best

If molecule outside box – ignore it - removed

Just keep computing (ex. NaN) will converge - slower

Just crashed – nothing you can do about it for the time being
Resiliency Techniques for Pointer-based Data Structures

• Duplicate/Triplicate Node Fields
  • Duplication: Allows for Detection (and in some instances correction – see next slide)
  • Triplication: allows for correction (TMR-like)
• Increase Storage Overhead
• Not Exclusive to Graphs...

(a) Original Data Structure.
(b) Transformed Data Structure with Pointer Duplication and Memory Errors in Pointer Fields.
Exploiting Alignment and Storage Bounds

• Field Duplication
  • Traditionally only used for Detection of Errors

• Disambiguating 1-of-2 bad Pointers (rule out bad pointers):
  • Exploiting Data Structure Properties
    • Alignment of Node Storage
    • Out-of-Bounds Pointers

(a) Graph with mis-aligned and out-of-bounds addresses
(b) Graph with recovered node addresses
Coping with Irrecoverable Pointers

• Some Traversals *do not require* ‘exact’ Results
  • User may be happy with partial searches
  • Randomized Computations

• How to Exploit this ‘Acceptance’ Criteria?
  • Detect Error & Skip Node

• Built-in Redundancy of Data Structure Helps
  • Extra Pointers or Confluence of Paths
Future Directions

- Application Specific Programming Extensions
- Resilience with Power and Energy
- Platform & System Heterogeneity
Future Dynamic Systems

Off-line (Compile-Time)
- Developer
  - Annotated Application Source Code (C, C++, UPC, Fortran)
  - Compiler (ROSE)
    - LLVM
    - Vendor Compilers

On-line (Run-Time)
- Adaptive Application Executable
- Introspective Runtime System

Knowledge about:
- Application Domain
- System Properties
- Strategic Goals
- Execution Behavior

Monitoring
Analysis
Recovery/Optimization Feedback
Prediction

OS/Hardware Infrastructure (Out of scope)
- Exascale Hardware
Conclusion & Perspectives

• The need for Reliability is not New:
  • Hardware / Software TMR in Space-borne & Safety-critical systems;
  • HPC Systems have a ‘different’ constraint – all simultaneously available and scalability issues

• Plenty of system and application knowledge about acceptable solutions – but no widely accepted mechanisms to convey it.