Exploiting the User’s Knowledge of Resilience

Bob Lucas
USC – Lockheed Martin Quantum Computing Center
Information Sciences Institute
Viterbi School of Engineering
Collaborators

• USC
  Jacque Chame
  Pedro Diniz
  Saurabh Hukerikar
  Hans Zima
• JPL
  Mark James
• LLNL
  Chunhua Liao
  Dan Quinlan
• SNL CA
  Robert Clay
  Keita Teranishi
Acknowledgements

• SRC FCRP MuSyC
• DOE SciDAC-3 SUPER
• US Army Research Office Contract #63295-CS-ACI
Motivation

• HPCS Random Access Benchmark explicitly tolerates a finite number of errors in both memory and arithmetic.
• Iterative solvers like Algebraic Multigrid (AMG) can hill climb over errors in intermediate states.
• Post-processing such as iterative refinement after solving a linear system can correct for small errors, as a function of condition number.
• Users can add their own fault detection and correction techniques.
• You can often roll back.
Resilient Software Architecture

Off-line (Compile-Time)

- Annotated Application Source Code (C, C++, UPC, Fortran)
- Compiler (ROSE)
- LLVM
- Vendor Compilers

On-line (Run-Time)

- Adaptive Application Executable
- Monitoring
- Analysis
- Recovery/Optimization Feedback
- Prediction

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

Knowledge/Experience Database (KEX)

Exascale Hardware

OS/Hardware Infrastructure (Out of scope)

Developer

Graph500
SSCA2
max2sat

Vendor Compilers

Developer

Exascale Hardware

USC
School of Engineering

University of Southern California
RoLex: Resilience Oriented Language Extensions

• Language extensions that allow scientific application programmers to encode their fault resilience knowledge and expectations within their application codes.

• Application-level error management capabilities:
  • Error Detection
  • Error Containment
  • Error Recovery
  made intrinsic to application code.

• Tightly coupled compiler infrastructure and runtime inference system
  • Allows programmer knowledge to be used for error management during application execution.
RoLex: Design Objectives

- Maintain familiarity of current programming models
  - FORTRAN/C/C++ dominant choice of languages for HPC application programmers

- Concise and elegant syntax: small set of new language keywords
  - Seek to minimize the time and effort for programmers to learn and adopt RoLex

- Fair division of work between language extensions and the compiler & runtime framework

- Interoperability with other programming models/libraries
  - Message Passing Interface (MPI) as well as certain well-tuned productivity libraries such as BLAS and LAPACK

- Sensitive to performance features
  - Extensions and compiler transformations should not drastically alter code structure
RoLex Programming Model

Extensions to the C/C++ language

Type Qualifiers
• Specify how the program variables are dealt with by the runtime system when the associated variable is in error state

```c
<error-management-qualifier> int array[N];
<error-management-qualifier> float matrix[M][N];
```

Directives
• Specify error management knowledge on structured blocks of computation
• In C/C++ #pragma directive species program behavior

```c
#pragma <error-management-directive>
{
   /* structured blk: computation*/
}
```

Library Routines
• Extensions to existing standard library calls with error management

```c
resilience <error-management-routine> libroutine
```
RoLex: Tolerance Extensions

Type Qualifiers

```c
tolerant<precision.6f> float low_precision;
tolerant int rgb[XDIM][YDIM];
tolerant<MAX.VALUE=...> unsigned int counter;
```

Directives

```c
#pragma resilience recover-rollback share (variable_list) private(variable_list)
{ /* code block */ }
#pragma resilience recover-rollforward share (variable_list)
{ /* code block */ }
```

Library Routines

```c
<typename>* <var>  = (<cast>) resilience_malloc_tolerant (sizeof(<typename>));
int* array = (int*)
    resilience_tolerant_malloc (NUM*sizeof(int <MAX.VALUE=..>));
double* matrix = (double*)
    resilience_tolerant_malloc (NUM*sizeof(double <PRECISION=...>));
```
Evaluation: RoLex Tolerance Extensions

**Application:** Pseudo-random updates on large memory table (HPCC_Table)

**RoLex:** We allocate the HPCC_Table with the tolerant version of the malloc routine

**Faults:** Hardware uncorrectable multi-bit perturbations; raises SECDED violation to runtime system.
Evaluation: RoLex Tolerance Extensions

**Application:** Renders 3D model of a scene into a 2D screen representation. Final rendered scene written to frame buffer (declared as a 2-D array)

**RoLex:** We qualify frame buffer declaration tolerant type qualifier

**Faults:** Hardware uncorrectable multi-bit perturbations; raises SECDED violation to runtime system.

---

**3D Rendering**

<table>
<thead>
<tr>
<th>Fault Injection Interval (minutes)</th>
<th>15</th>
<th>10</th>
<th>5</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>% Execution Runs: Application Outcome</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application Completion with “correct” outcome</td>
<td>100.00%</td>
<td>90.00%</td>
<td>80.00%</td>
<td>70.00%</td>
<td>60.00%</td>
</tr>
<tr>
<td>Abnormal termination</td>
<td>0.00%</td>
<td>10.00%</td>
<td>20.00%</td>
<td>30.00%</td>
<td>40.00%</td>
</tr>
</tbody>
</table>

---

USC
School of Engineering
University of Southern California
Evaluation: RoLex Tolerance Extensions

Molecular Dynamics

Application: MD simulation

RoLex: We qualify the Floating-point array structures for the particle position, velocity and acceleration with tolerant <PRECISION=8>

Faults: Hardware uncorrectable multi-bit perturbations; raises SECDED violation to runtime system.
RoLex: Robustness Extensions

Type Qualifiers

```
robust (CORRECT) int* csr_matrix [ row_offsets ];
robust (DETECT) int* graph_edge_list [N];
```

Directives

```
#pragma resilience robust detect share(variable_list) private(variable_list)
compare( variable_list)
{
   /* code block */
}
#pragma resilience robust correct share (variable_list) private (variable_list) compare (variable_list)
{
   /* code block */
}
```

Library Routines

```
float* problem_matrix = ( float *) resilience_malloc_robust( N * sizeof (float), STRENGTH );
resilience_validate_detect ( void * problem_matrix );
resilience_validate_correct ( void * problem_matrix );
```
Evaluation: RoLex Robustness Extensions

Application: Algebraic Multigrid Linear Solver.

RoLex: We qualify the pointer references for the matrix references using the robust qualifier; intermediate solution grids allocated with tolerant version of malloc routine.

Faults: Silent Data Corruptions; no hardware interrupt raised to the runtime system.
RoLex: Amelioration Extensions

Type Qualifiers

heal (recovery_func()) float* matrix_A[N][N];

Directives

#pragma resilience recover-rollback reinitialize( variable_list )
{ /* code block */ }
#pragma resilience recover-rolloffward reinitialize( variable_list )
{ /* code block */ }
#pragma resilience recover-rollback ameliorate(recovery_func())
{ /* code block */ }
#pragma resilience recover-rolloffward ameliorate(recovery_func())
{ /* code block */ }

Library Routines

float * problem_matrix = (float*)
resilience_malloc_repairable(N * sizeof(float),checksum_func_pointer);
resilience_ameliorate_heal(void* problem_matrix);
Conjugate Gradients

// Solve a linear system with conjugate gradients

for(int k=1; k<max_iter && normr > tol; k++) {
    oldrtrans = rtrans;
    ddot(nrow, r, r, &rtrans, t4);
    beta = rtrans/oldrtrans;
    waxpby(nrow, 1.0, r, beta, p, p);
    normr = sqrt(rtrans);
    sparsemv(A, p, Ap);
    ddot(nrow, p, Ap, &alpha, t4);
    alpha = rtrans/alpha;
    waxpby(nrow, 1.0, x, alpha, p, x);
    waxpby(nrow, 1.0, r, -alpha, Ap, r);
    niters = k;
}
Sparse Matrix Structure

// Structure of sparse matrix A that will be protected with checksums,
// i.e., algorithm-based fault tolerance

HPC_Sparse_Matrix *A;
A->nrow = (int) nrow;
A->nnz = (int) nnz;
A->values = (double *) malloc ((nnz) * sizeof(double));
A->indices = (int *) malloc ((nnz) * sizeof(int));
A->ptr_to_rows= (int *) malloc ((nrow + 1) * sizeof(int));
Checksum-based Fault Tolerance
(Thanks Jack!)

// Checksum vectors for user-provided, checksum-based, repair functions.

c coef_row_sum = (unit64 *) malloc (A.nrow * sizeof(double));
c coef_col_sum = (uint64 *) malloc (A.nrow * sizeof(double));
ind_row_sum = (uint *) malloc (A.nrow * sizeof(int));
ind_col_sum = (uint *) malloc (A.nrow * sizeof(int));

// User-defined checksum functions

CreateChecksumD(* HPC_Sparse_Matrix, uint64* coef_row_sum, uint64* coef_col_sum);
CreateChecksumI(* HPC_Sparse_Matrix, uint * ind_row_sum, uint * ind_col_sum);
RepairChecksumD(* HPC_Sparse_Matrix, uint64* coef_row_sum, uint4* coef_col_sum);
RepairChecksumI(* HPC_Sparse_Matrix, uint * ind_row_sum, uint * ind_col_sum);
More Resilient, But Not Immortal, Conjugate Gradients

// Solve a linear system with conjugate gradients

for(int k=1; k<max_iter && normr > tol; k++) {
    oldrtrans = rtrans;
    ddot(nrow, r, r, &rtrans, t4);
    beta = rtrans/oldrtrans;
    waxpby(nrow, 1.0, r, beta, p, p);
    normr = sqrt(rtrans);
    sparsemv(A, p, Ap);
    ddot(nrow, p, Ap, &alpha, t4);
    alpha = rtrans/alpha;
    waxpby(nrow, 1.0, x, alpha, p, x);
    waxpby(nrow, 1.0, r, -alpha, Ap, r);
    niters = k;
}
**Evaluation: RoLex Amelioration Extensions**

**Application:** Self-Stabilizing Conjugate Gradient; Contains correction step that restores numerical stability.

**RoLex:** We define checksum routine for matrix A and vector B which are allocated using resilience_malloc_repairable() library routine; CG iteration steps are included in the #pragma resilience roll-forward which refers to correction step function in ameliorate clause.

**Faults:** Hardware uncorrectable multi-bit perturbations; raises SECDED violation to runtime system.

---

**Graph Details:**

- **Self-Stabilizing Conjugate Gradient**
- **X-axis:** Fault Injection Interval (minutes)
- **Y-axis:** % Execution Runs; Application Outcome
- **Legend:**
  - Green: Application Completion with “correct” outcome
  - Red: Abnormal termination

---

**Graph Data:**

- 15 minutes:
  - Application Completion: 100.00%
  - Abnormal termination: 0.00%

- 10 minutes:
  - Application Completion: 90.00%
  - Abnormal termination: 10.00%

- 5 minutes:
  - Application Completion: 80.00%
  - Abnormal termination: 20.00%

- 2 minutes:
  - Application Completion: 70.00%
  - Abnormal termination: 30.00%

- 1 minute:
  - Application Completion: 60.00%
  - Abnormal termination: 40.00%
Translation of RoLex:
Propagates fault resilience knowledge expressed by the programmer to the target code and execution environment through the runtime interface.

Compiler Front-end:
• Source-to-source Code Transformation: Generated program code uses only base language (C/C++) constructs
• Parses the qualifiers and directives to generate code equipped with the error detection, containment and repair capability
Compiler-Runtime Interface

RoLex runtime library routines (visible to compiler only)
• Inserted by front end compiler into source code

__rolex_initialize() : runtime initialization
__rolex_finalize() : runtime termination
__rolex_preserve_env() : save program environment
__rolex_restore_env() : restore program environment
__rolex_jmp_fwd() : move execution forward to set point
__rolex_rollback_fwd() : move execution forward to set point
__rolex_chkpoint() : save specified variable state
__rolex_restore_chkpoint() : restore previously saved variable state
RoLex based Execution Model: Workflow
Introspection based Runtime System: Components

- Introspection Framework
- Resilience driven Resource Management Strategies:
  - Resilience Aware Thread Scheduling
  - Resilience Aware DVFS
- RoLex Integration
Integration with RoLex Features: Resilience Aware Thread Assignment

- Policies for realignment of thread affinities based on fault resilience features of the application phase

Resists reconfiguration: allows tolerant computation to persist on vulnerable core

Allows modification of thread affinity mask depending on assignment of redundant computation

Resists thread migration until repair/heal routines are required to be invoked.
Integration with RoLex Features: Resilience Aware DVFS

- Policies for living dangerously/risk averse DVFS modes based on fault resilience features of the application phase

- Resists scaling back from living dangerously
- Allows living dangerously for either redundant computation
- Allows living dangerously for repairable phases; conservative DVFS state for heal routines
Summary

• Created a assertion language
  Allows users to communicate knowledge about their requirements with respect to resilience

• Created an Introspection Framework for Resilience
  Exploits user’s guidance to make run time decisions on how to respond to faults

• Leverage this automate a variety of resilience mechanisms
  Turn redundant threads on/off based on fault rate
  Migrate threads away from flaky nodes
  Aggressively adjust frequency independent of voltage