



# PROGRAMMING MODEL EXTENSIONS FOR RESILIENCE AT EXTREME SCALE

*Saurabh Hukerikar, Pedro C. Diniz, Robert F. Lucas*

Workshop on Productive Programming Models for Exascale

Portland, Oregon

August 14-15, 2012



# Introduction

- In supercomputing systems today hardware errors account for up to 60% of the total failures;
  - 40% attributed to memory related failures
- Computing platforms abstract lower layers of computing stack to provide higher layers with correct execution
- Many algorithms are inherently resilient to errors in computation
- Programmers may have fault tolerance knowledge but no convenient mechanisms to convey this knowledge to system



# Overview

## Resiliency Oriented Programming Model Extensions

Evolutionary Approach: Based on current, familiar language constructs

### Cross Layer Resilience

Engage fault tolerance knowledge from all software layers including

- Application
- Compiler infrastructure
- Programming language features
- Operating system

### Fault Model

Multi-bit memory errors uncorrectable by ECC schemes

# Programming Extensions for Resilience



- Type Declarations
- Dynamic Memory Allocation
- Code Sections
- Looping Constructs
- Procedure Calls

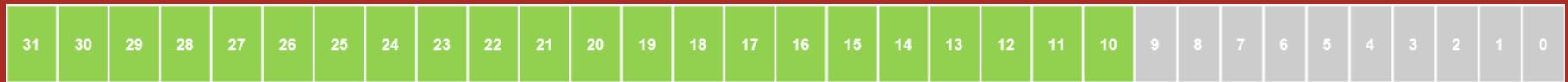


# Programming Model Extension

## Tolerant Type Qualifiers

```
tolerant int rgb[XDIM][YDIM];
```

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



```
tolerant<precision.6f> double low_precision;
```





# Programming Model Extension

## Tolerant Malloc

```
<type>* <var> = (<cast>) tolerant_malloc  
(sizeof(<type>));
```

```
<type>* <var> = (<cast>) tolerant_malloc (NUM *  
           sizeof(<type><MAX.VALUE=..>));
```



# Programming Model Extension

## Tolerant Preprocessor Directives

```
#tolerant begin
```

```
<code>
```

```
...
```

```
...
```

```
...
```

```
#tolerant end
```

- Tolerant Regions of Code



# Programming Model Extension

## Tolerant Looping Construct

```
void pi() {  
    int i ;  
    double step, x, sum = 0.0;  
    double pi;  
    step = 1.0/N;  
    tolerant_for( i =0; i < 100000000; i++; N/3 )  
    {  
        x = ( i + 0.5 ) * step ;  
        sum += 4.0/(1.0 + x*x) ;  
    }  
    pi = sum* step ;  
}
```



# Programming Model Extension

## Tolerant Procedures

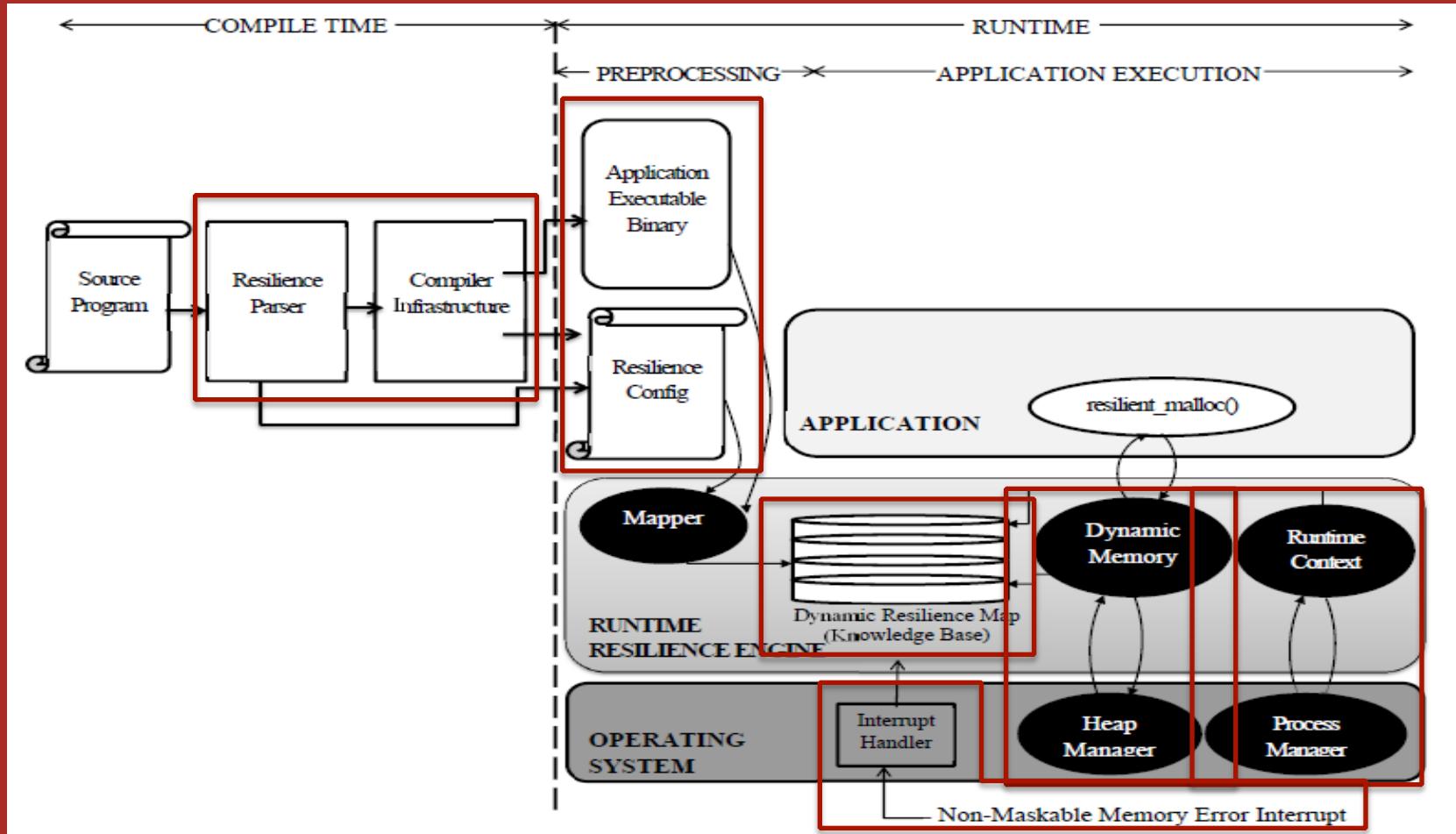
```
tolerant void residual ( tolerant<MAX.VAL=32> int arg1,  
                        tolerant int arg2)  
{  
    tolerant int local_var1;  
    tolerant int local_var2;  
    ...  
}
```

Allows programmer to specify tolerance on the stack addresses

- Iterative refinement
- Deep recursion algorithm



# System Workflow





# Results : HPCC Random Access

Table Size		Random Updates	Faults Injected	Faults Survived	% Faults Survived
Entries	Bytes				
$2^{18}$ entries	262,144	262,144	6	5	83.33
$2^{20}$ entries	4,194,304	4,194,304	26	24	92.31
$2^{22}$ entries	16,777,216	16,777,216	99	97	97.97
$2^{24}$ entries	67,108,864	67,108,864	398	395	99.24

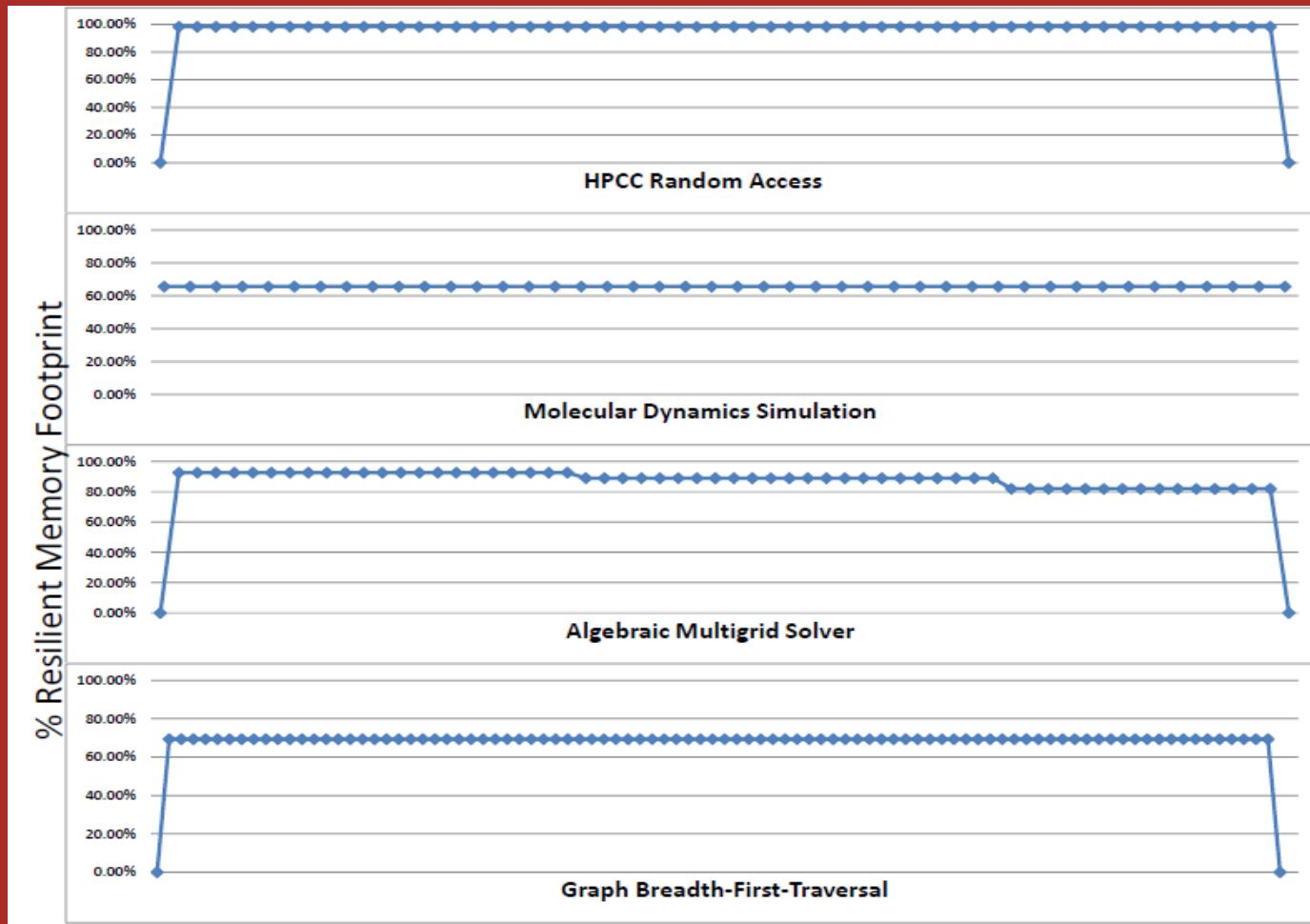


# Results : HPC Applications

Fault Injection Rate (minutes)	% Execution Runs to Completion			
	Random Access	Molecular Dynamics Simulation	Algebraic Multigrid Linear Solver	Graph Breadth-First-Search Traversal
15	99.5 %	66.2 %	96.1 %	61.4 %
10	99.2 %	61.3 %	92.7 %	32.8 %
5	99.1 %	36.3 %	86.6 %	19.8 %
2	97.4 %	7.1 %	81.2 %	2.3 %
1	96.1 %	1.2 %	63.3 %	0.8 %



# Sensitivity Analysis





# Summary

- Our approach highlights the benefits of programmer interfaces to express fault tolerance knowledge to the lower levels of system abstraction

## *Future Directions and Ongoing Work*

- Richer set of interfaces and opportunities for cross-layer resilience
- Explore more error models for extreme scale systems



# *Information Sciences Institute*

**Saurabh Hukerikar, Pedro C. Diniz, Robert F. Lucas**  
**{saurabh, pedro, rflucas}@isi.edu**