Introspective Resilience for Exascale High Performance Computing Systems

Saurabh Hukerikar
{saurabh}@isi.edu
Information Sciences Institute
University of Southern California

INTRODUCTION

System Resilience is a key challenge on the road to advanced exascale capability computing systems due to:

- Large number of processor cores, memory modules projected to be deployed; System MTTF scales inversely to number of components
- Deep scaling of process technologies and near-threshold voltage operation leading to increased soft error induced failures, aging effects etc.

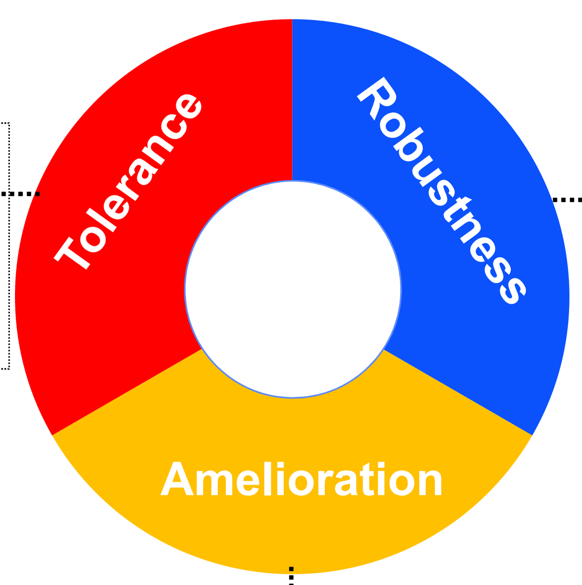
Proposed Approach: *Introspective Resilience:* HPC application programmers often have insights on application fault tolerance. But no convenient mechanisms in today's programming models to convey such information

- Knowledge leveraged by compiler & introspective runtime framework to build resilience knowledge base.
- Introspective runtime framework observes the rate and source of anomalous fault events and affects application execution and assignment of system resources to the application.

PROGRAMMING LANGUAGE EXTENSIONS

Simple Language Extensions based on familiar constructs to enable programmer to express their fault tolerance knowledge to the compiler & system runtime

Expression of *error tolerance* in application program state:
- Semantics that ignore certain errors
- Localized application-level recovery



Expression of *critical* regions in program state: application-level detection and correction semantics

Knowledge on how to *heal* specific data structures in the program state

- Tolerance: Qualifiers/Directives to specify tolerance: enables runtime to ignore errors at corresponding locations
- Preprocessor directives for roll-forward, roll-back recovery for structured code blocks

```
tolerant  int rgb [XDIM][YDIM]
tolerant  <MAX.VAL=1023> int rgb
tolerant  <precision=8> double low_precision
```

```
#pragma tolerant
{    ...    }
```

```
tolerant  int refine(int param1, int param 2… )
{       . . .
}
```

```
tolerant_malloc();
```

```
#pragma roll-back preserve<variable list . . . >
{    ...    }

#pragma roll-forward preserve<variable = . . . >
{    ...    }
```

- Robustness: Qualifiers to specify compiler driven data structure replication
- Preprocessor directives for redundant execution (DMR/TMR) for detection/correction

```
robust int* index;
robust_malloc()
```

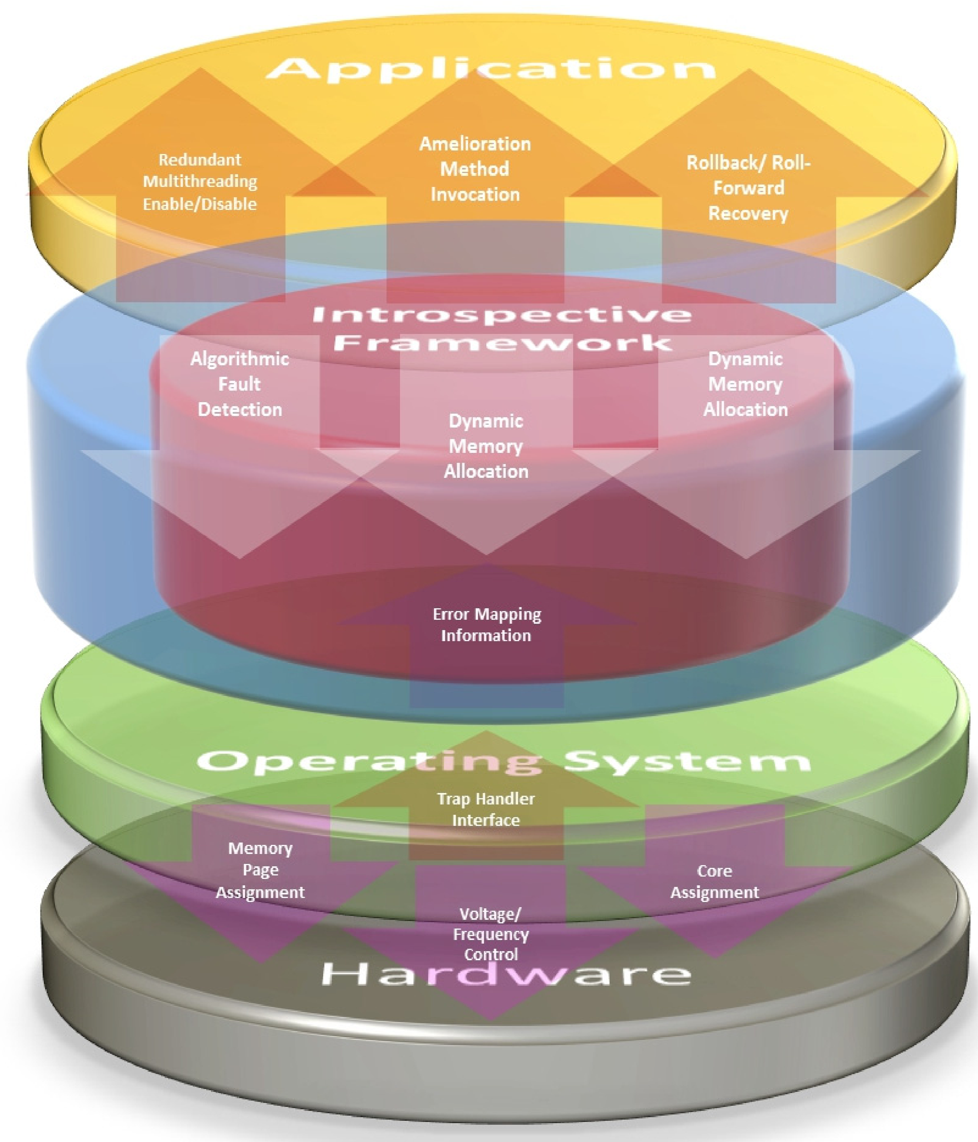
```
#pragma robust
{     ...     }
```

- Amelioration: Qualifiers to specify detection/healing functions for data structures
- Function can be asynchronously invoked; allows application state to be *repaired* before continuing execution

```
detect <method …> double matrix [][]
```

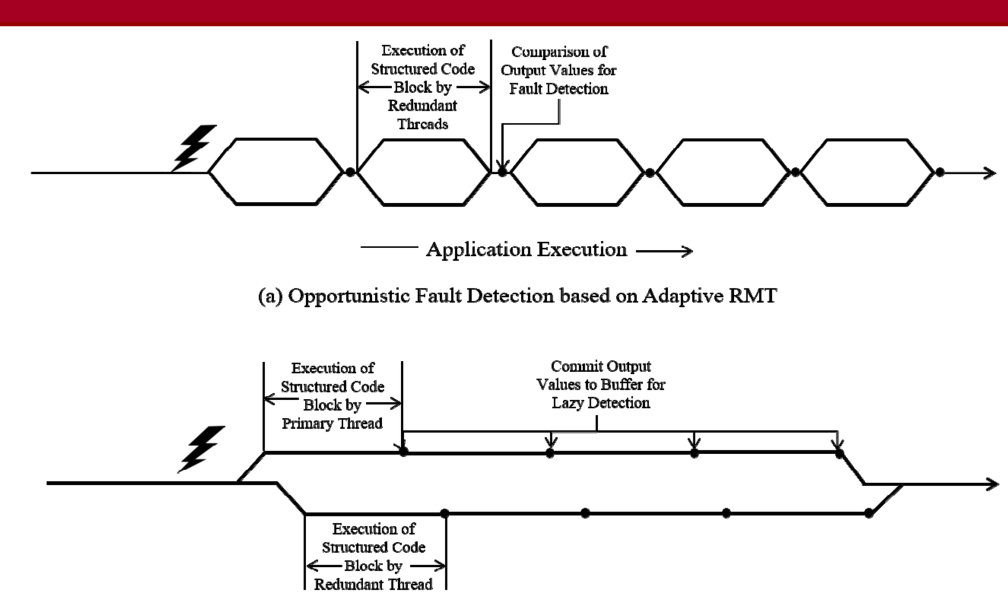
```
heal <method …> double matrix [ ][ ]
```

INTROSPECTION FRAMEWORK



- The runtime framework leverages the tolerance knowledge conveyed by the programmer through the programming language extensions, those exposed by compiler transformations
- Dynamic Resilience Map (DRM) is maintained as an active knowledge base of the addresses and offsets in the logical address space of the application marked tolerant.
- Observation of system 's anomalous events including corrected errors, thermal indicators etc. and trend analysis of rate and source of faults.

COMPILER TRANSFORMATIONS: OPPORTUNISTIC & LAZY FAULT DETECTION THROUGH REDUNDANT MULTITHREADING

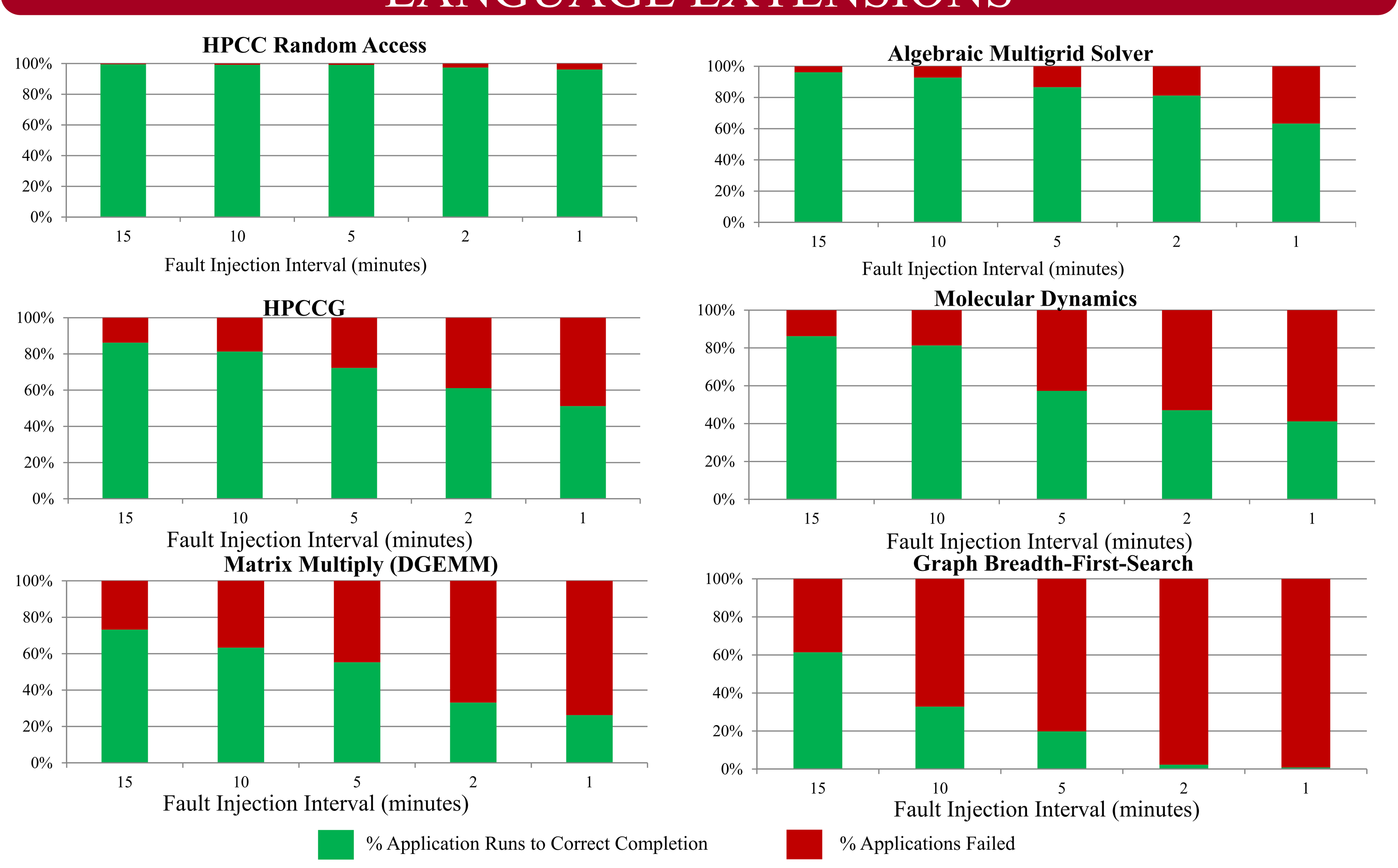


(a) Opportunistic Fault Detection based on Adaptive RMT

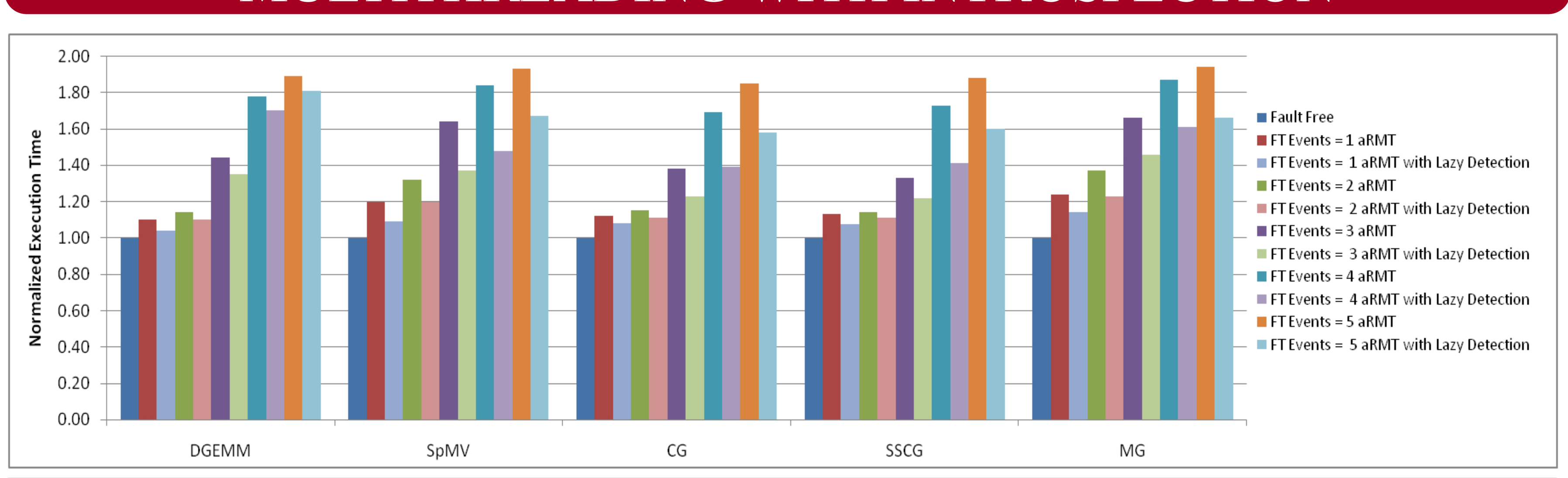
(b) Fault Detection based on Adaptive RMT with Lazy Evaluation

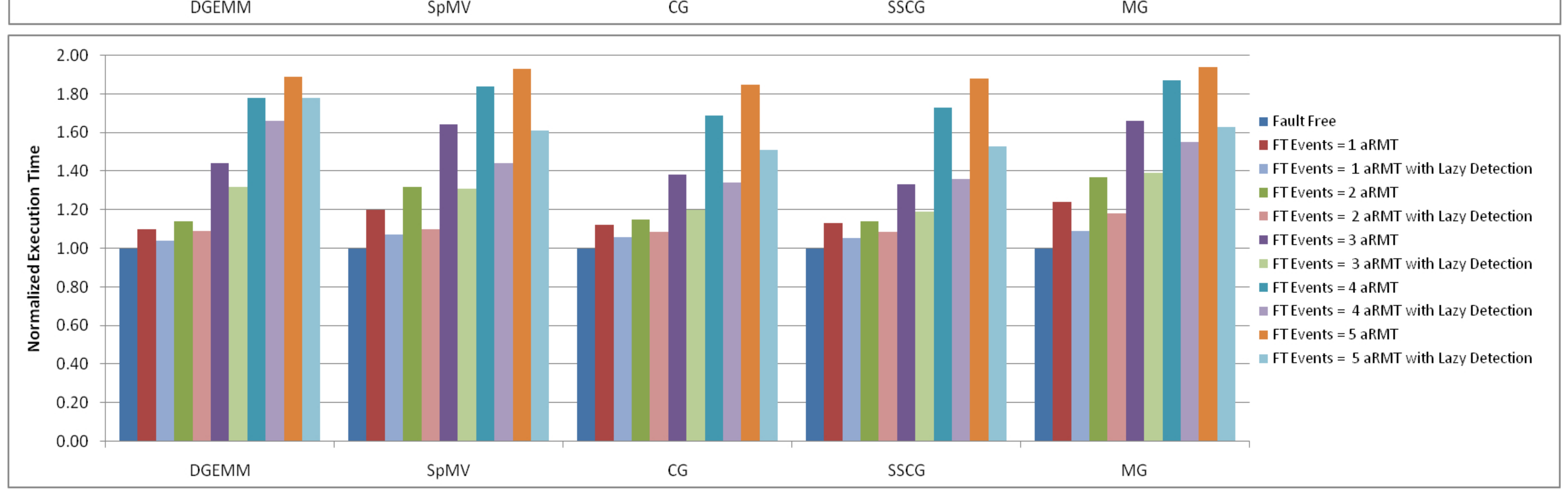
- The structured code blocks specified by programming directive are outlined by the compiler, but executed by a single thread until the runtime system intervenes.
- Introspective runtime system makes inferences about the vulnerability of the system resources and signals the application to dynamically enable/disable the redundant multithreaded execution of the outlined code blocks (Figure (a))
- *Optimization*: Lazy detection through relaxed comparison of values generated by duplicate threads (Figure (b))

RESULTS: APPLICATION SURVIVABILITY BASED ON LANGUAGE EXTENSIONS



RESULTS: PERFORMANCE EVALUATION OF REDUNDANT MULTITHREADING WITH INTROSPECTION





CONCLUSIONS

- Cross-layer approach based on introspection: programming model extensions capture programmer's fault tolerance insights which are leveraged by compiler and runtime framework
- Error detection/correction capabilities provided through source-level code transformations by compiler framework
- Runtime inference allows for reasoning about the context and significance of faults to the application outcomes.

USC Viterbi School of Engineering

Information Sciences Institute