Pragma-controlled Source-to-Source Code Transformations for Robust Application Execution

Pedro C. Diniz, Chunhua Liao, Dan Quinlan and Bob Lucas

Presentation at the 9th Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids

in conjunction with the 22th Intl. European Conf. on Parallel and Distributed Computing (Euro-Par 2016)

August 23, 2016

Motivation: Trends & Projections

	2009	2012	2016	2020
System Peak Performance	2 Petaflops	20 Petaflops	~200 Petaflops	1 Exaflops
System Memory	0.3 PB	1.5 PB	~5 PB	~30 PB
System Node Count	8,000	18,000	~50,000	~100,000
Total Core Count	300,000	1,500,000	~ 50 million	~ 1 billion
Mean Time To Interrupt (MTTI)	1 day	20 hours	40 – 50 minutes	20 minutes
Power	7MW	8.2MW	~15MW	20MW

Petascale systems today already experience[1]:

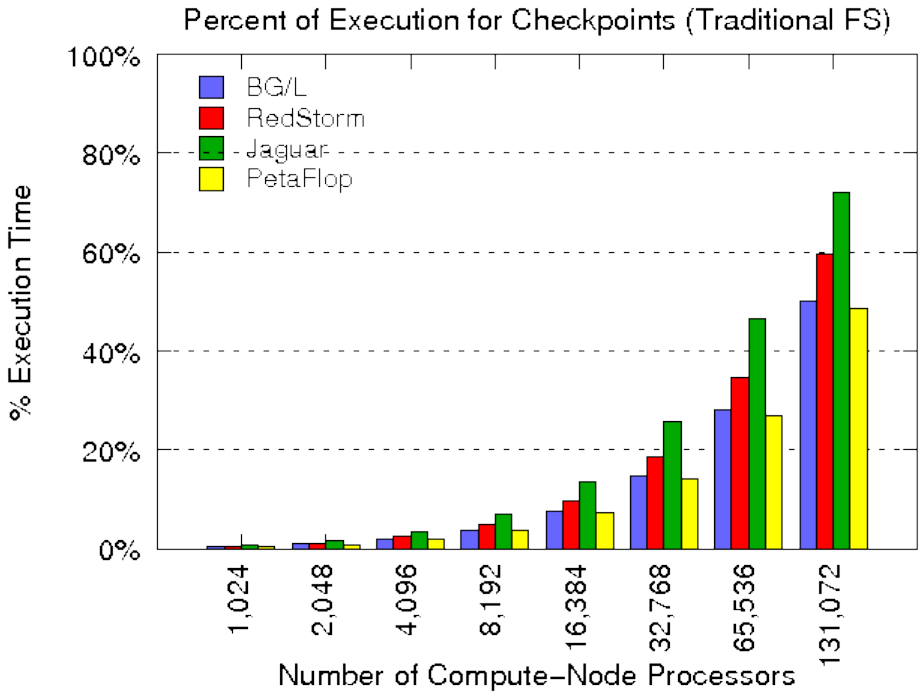
- ~20 faults/hour
- 1 double-bit DRAM error every 24 hours
- Constant stream of single bit memory errors

[1] Al Geist, "What is the monster in the closet?" Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in our Thinking

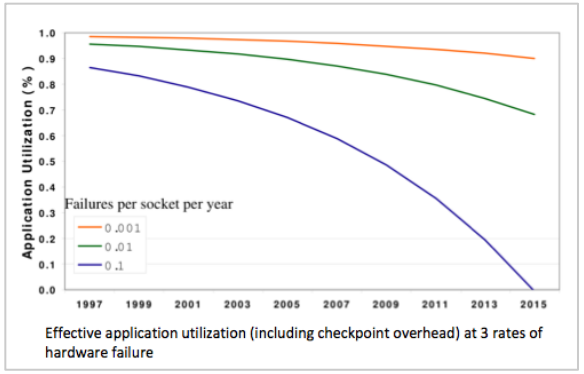
USC Viterbi School of Engineering

Lawrence Livermore National Laboratory

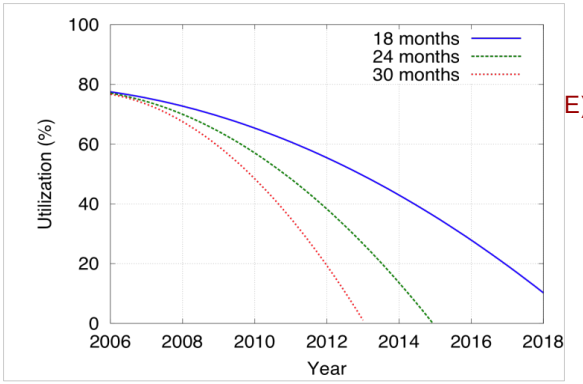
Check-Pointing and Restart



Reference:
Oldfield et al., *Modeling the Impact of Checkpoints on Next-Generation Systems*. MSST, 2007



Effective application utilization (including checkpoint overhead) at 3 rates of hardware failure



E)

Reference:
Schroeder and Gibson, *Understanding Failures in Petascale Computers*. Journal of Physics, 2007 (assuming that the number of cores per socket grows by a factor of 2 every 18, 24 and 30 months)

Not a Scalable Approach

USC Viterbi School of Engineering

Lawrence Livermore National Laboratory

Observations

- Many Algorithms are Inherently Resilient to Errors
- Programmers may have fault tolerance knowledge
- ... but no convenient mechanisms to convey this knowledge to system
  - System Layers are Inflexible
  - Programming Abstractions Very Rigid

Our Approach

Resiliency Oriented Programming Model Extensions

Evolutionary Approach: Based on current, familiar language constructs

Cross Layer Resilience

Can involve Compiler and Operating System
Introspection System to Manage State of Machine

Fault Model

Multi-bit memory errors uncorrectable by ECC schemes

Programming Extensions for Resilience

- **Previous Work**
  - Type Declarations
  - Dynamic Memory Allocation
- **This Work**
  - **#pragma** directives
  - Source-to-source Code Transformations

Programming Model Extensions

Tolerant Storage Declaration

#pragma failsafe tolerant (exp1 : exp2)

...

- Indicates:
  - Maximum number of tolerated errors (**exp1** , default: any)
  - Preferable (optional) storage assignment (**exp2** , default: none)

- Code Translation:
  - None
  - Auxiliary Storage and Resilience Map

Programming Model Extensions

Sentinel Value of Silent Data Corruption (SDC)

- **#pragma failsafe assert (predicate) error (function handler)**
  ...

- Indicates:
  - Predicate (simple, first order) to be Evaluated
  - Handler Function to be executed if Predicate Does not Hold True

Programming Model Extensions

Sentinel Value of Silent Data Corruption (SDC)

- Code Translation:

```
if(predicate(...) == 0){
    if(function handler (...) != 0){
        failsafe error++;
        FAILSAFEREPORTERROR(0,failsafe error);
         failsafe error flag = 0;
     } else {
       FAILSAFEREPORTCORRECTION(0,failsafe error);
     }
}
```

Programming Model Extensions

User-Controlled State Saving and Restoring with Retry

- **#pragma failsafe save restore (var list) retry (exp)**
  {/* code block */ }
  ...

- Indicates:
  - Set of Variables to be Saved/Restore at Each Iterations
  - Implicit Error Checking via Detected (uncorrected) Memory Error
  - Maximum Retry

Programming Model Extensions

User-Controlled State Saving and Restoring with Retry

- Code Translation (for retry = 2):

```
int fs num tries;
volatile int fs num errors ;
fs num tries = 0;
fs num errors = 0;
<code for saving data objects>
do {
  if (fs num tries != 0){
    <code for restore data objects>
  }
  fs num errors = 0;
    <original code block here>
  fs num tries++;
} while ((fs num errors != 0) && (fs num tries < 2));
if (fs num errors != 0){
  FAIL SAFE EXCEPTION ( )
}
```

USC Viterbi School of Engineering

Lawrence Livermore National Laboratory

Programming Model Extensions

Redundancy-based Fault Detection and Recovery

- **#pragma failsafe dual redundancy save restore (var list1) compare (var list2) retry (exp)**
  {/* code block */ }
  ...

- Indicates:
  - Set of Variables to be Saved/Restore at Each Iteration
  - Error Checking/Correction via Matching/Voting
  - Maximum Retry

Programming Model Extensions

Redundancy-based Fault Detection and Recovery

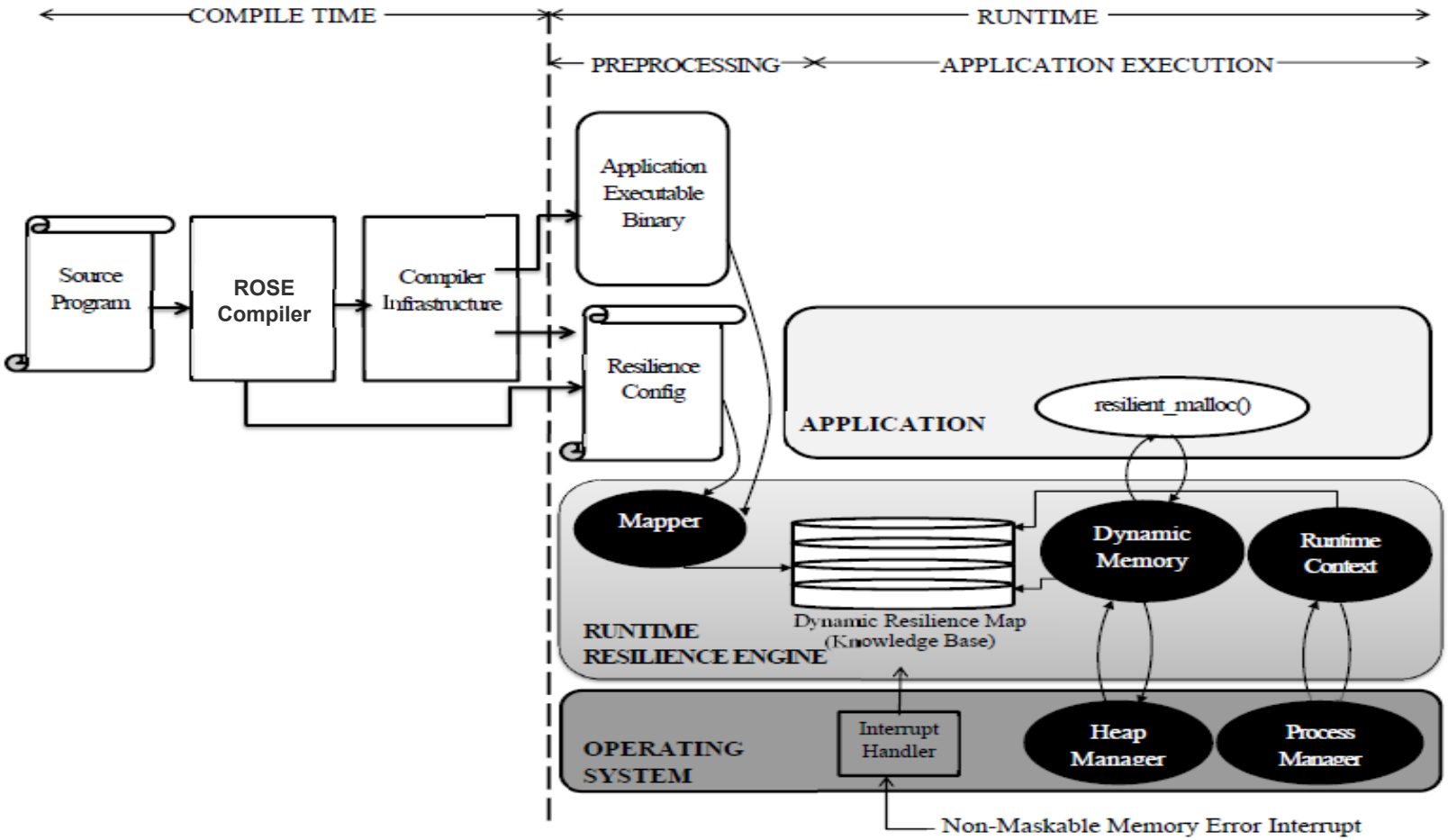
- Code Translation (for retry = 2):

```
int fs num tries ;
volatile int fs num errors ;
< declaration of duplicated of variables in var list2>
...
fs num tries = 0; fs num errors = 0;
<code for saving data in var list1> do
if (fs num tries != 0){
<code for restore data in var list1>
}
fs num errors = 0;
#omp parallel num threads(2)
{
<original code relabeling variables in var list2> }
<compare variables in var list2 for each thread>
if (mismatch( var list2 )) fs num errors++;
fs num tries++;
} while ((fs num errors != 0) && (fs num tries < 2));
if(fs num errors != 0){ FAIL SAFE EXCEPTION ( ) ;
}
```

Preliminary Results : CG kernel

- Simple Iterative Algorithm for Solving Ax = b (40 x 32)
- Main Matrix A is 4 MB and various vectors 0.23 MB
- Error Amelioration
  - in A, fixed using checksum (ABFT)
  - in vectors x, b and others, fixed by reloading saved state
- Errors
  - Restart of the previous iteration using saved state
  - Use of **#pragma save_restore directive** with **retry (2)**
- Total of 21 iterations (0.5 secs/iteration on Desktop)

System Workflow



Preliminary Results : CG kernel

- Methodology:
  - Force Memory Errors in Address Space
  - Restart iteration based on which data structure afflicted
  - Maximum Restart set to 2; additional storage for vectors

Error Internal (secs)	Checksum Recovery	Iteration Restart Recovery	Algorithm Iterations	Execution Time (secs)	Execution Overhead
2	12	1	34	23.761	122.5%
4	5	1	27	20.537	92.3%
5	4	1	26	18.569	73.9%
10	1	1	23	15.368	4.5%
20	1	0	22	11.310	0.6%

Table 1. Execution times vs. injected memory rates for CG simple solver.

  - All “Injected” Errors were Corrected
  - Majority of Errors in A, checksum Correction
  - Overhead not Excessive

USC Viterbi School of Engineering

Lawrence Livermore National Laboratory

Summary

- 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 opportunities for cross-layer resilience
- Interface to Indicating “dead regions” for data

Acknowledgment

Partial support for this work was provided by the US Army Research Office (Award W911NF-13-1-0219) and through the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under award number DE-SC0006844.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

USC Viterbi School of Engineering

Lawrence Livermore National Laboratory