Enhancing the Dependability of Extreme-Scale Applications*

Hans P. Zima
Jet Propulsion Laboratory, California Institute of Technology
and
Institute for Scientific Computing, University of Vienna, Austria

Clusters, Clouds, and Grids for Scientific Computing
CCGSC 2010
Flat Rock, North Carolina, September 10th, 2010

*This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration and funded through the internal Research and Technology Development Program.
1. Introduction
2. Dependability
3. Introspection Framework for Fault Tolerance
5. Conclusion
Extreme Scale Systems

Aggressive Strawman Design (Bill Dally, Peter Kogge)

- 166 million cores, 742 cores/chip
- Cost of data access and synchronization (in terms of energy) by far dominates the cost of computation

**Linpack Study (Peter Kogge):**

475pJ data access (average) for 1 FLOPS (10pJ)

- Memory Capacity: 0.0036 B/FLOPS, 20 MB/core
- Hardware and software errors will become an issue—traditional checkpointing and recovery may become infeasible.
Key Challenges for Extreme-Scale Systems

- Concurrency
- Power/Energy / Locality
- High-Level Abstractions for Programming
- Dependability
A Short Note on High-Level Abstractions and Dependability

The size and complexity of the code for a given application problem *is* relevant for dependability (independent of language support for dependability)

For example…
Fortran+MPI Communication for 3D 27-point Stencil (NAS MG rprj3)
function rprj3(S,R) {
    const Stencil: domain(3) = [-1..1, -1..1, -1..1], // 27-points
        w: [0..3]real = (/0.5, 0.25, 0.125, 0.0625/), // weights
        w3d: [(i,j,k) in Stencil] = w((i!=0) + (j!=0) + (k!=0));

    forall ijk in S.domain do
        S(ijk) = sum reduce [off in Stencil] (w3d(off) * R(ijk + R.stride*off));
}
Contents

1. Introduction
2. Dependability
3. Introspection Framework for Fault Tolerance
5. Conclusion
Dependability*

The ability of a computing system to deliver service that can be justifiably trusted

Threats: the Fault-Error-Failure Chain

- **Fault**: defect in a system
- **Error**: invalid system state
- **Failure**: violation of system specification

System Boundary (Service Interface)

- **Fault**: activation
- **Error**: propagation
- **Error**: propagation to service boundary

- **external fault** (caused by external failure)
- **invalid system state**
Fault Prevention: via *quality control* during design and manufacturing of hardware and software
- *structured programming, modularization, information hiding; firewalls*
- *shielding and radiation hardening*
- ...

Fault Removal: Verification and Validation (V&V), model checking

In general, **fault prevention and removal cannot guarantee the absence of errors**—for theoretical as well as practical reasons (undecidability, NP-completeness, etc.). Even a perfectly correct program may be subject to hard and soft errors. This motivates the need for **fault tolerance** as a third category of fault protection.
Fault Tolerance: the ability to preserve the delivery of correct service (system specification) in the presence of active faults

- error detection
- recovery: error handling and fault handling
- fault masking: redundancy-based recovery without explicit error detection (e.g., TMR)
Fault Tolerance

Fault -> Error -> Failure

activation

propagation to service boundary

fault tolerance

elimination of detected errors

prevention of fault activations

well-defined system state
Issues in Dependability for Extreme-Scale Systems

- Extreme-scale systems will have less reliable components (due to smaller feature sizes) and a larger component count than current systems: as a consequence, errors are expected to become the norm, not an exception.

- Checkpointing and recovery may become intolerably expensive—and even infeasible, depending on MTTF.

- Dynamic power management may have a negative effect on hardware reliability (thermal stresses).
Issues in Multi-Core Fault Tolerance
Challenges

◆ Fault in a shared component of a multi-core chip may affect the whole chip
  – caches, memory controller, I/O circuitry, on-chip networks
  – fewer natural boundaries than for traditional architectures
  – example: failing cache controller for L2 cache in Sun Niagara

◆ Possible Solution: Hardware-Supported Isolation*
  – partition sets of components into independently configurable units
  – Tile64 chip supports “walling off” sets of cores

Issues in Multi-Core Fault Tolerance

Opportunities

- Cores are becoming an inexpensive resource

- Use of cores for:
  - **checkpointing**
  - **introspection support**
  - **spares**

Specification: Allocation of cores to tasks (RT Chapel)

```plaintext
const L:[n,n]locale = reshape(Locales)
on L(2..p1,2..q1) do task1
on L(p1+1..n-1,2..q2) do task2
on L(2..p1,q1+1..n-1) do task3
on L(p1+1..n-1,q2+1..n-1) do task4
on L(2..n-1,1) do Introspection
on L(1,2..n-1) do Checkpointing
allocate spares to L(1..n,n)
```
Contents

1. Introduction
2. Dependability
3. Introspection Framework for Fault Tolerance
5. Conclusion
Focus of Work

- Development of an *Introspection Framework* for adaptive fault tolerance
- Development of an API for expressing dependability requirements of applications
- Compiler analysis for the generation of redundant code and intelligent optimization of checkpointing
- Development of fault tolerance metrics
A Framework for Introspection

Introspection...

◆ provides dynamic monitoring, analysis, and feedback, enabling system to become self-aware and context-aware:
  – monitoring execution behavior
  – reasoning about its internal state
  – changing the system or system state when necessary

◆ exploits adaptively threads available in multi-core systems

◆ can be applied to a range of different scenarios, including:
  – fault tolerance
  – performance tuning
  – energy and power management
  – behavior analysis
Adaptive Fault Tolerance: the capability to provide dependability based on a fault model, application requirements, and system properties

Introspection provides functionality for error detection, analysis and recovery
The architecture of the Introspection Framework depends on the structure of the application, the mapping of application components to the hardware, and related dependability requirements.

*Introspection Modules*—the atomic components of this structure—are arranged into an *introspection graph*, which expresses a control relation in the set of modules.

Each introspection module is associated with application components and performs specialized functions related to these components.

This supports a capability for component-based fault tolerance, supporting *heterogeneity* as well as the capability to deal with errors locally, in parallel, and at the earliest possible time.
Introspection-Based Fault Tolerance Architecture: Example: PS3 Cluster
**Introspection Module**

„atomic“ component of introspection

**Fault Classes under Consideration**
- transient faults
- hard faults
- software design faults

**Sensors** provide input to the introspection system (e.g., state information, assertion values, hardware alarms)

**Actuators** provide feedback from the introspection system to the application (e.g., error analysis information, suggestions for recovery algorithms, modification of instrumentation)

**Introspection System**

- Inference Engine (SHINE)
- Monitoring
- Analysis
- Recovery
- Prognostics

**Application**

**System**

**Environment**

**Actuators**

**Sensors**
Introspection Complements V&V

- Introspection performs **execution time** monitoring, analysis, recovery

- Introspection can deal with transient errors, execution anomalies, performance problems
  - *this capability is inherently beyond the scope of V&V technology*
  - *and it can be used to deal with design errors*

- **Future Goal:** integration of introspection with V&V technology into a comprehensive program development scheme
Contents

1. Introduction

2. Introspection Framework for Fault Tolerance


4. Conclusion
Compiler- and Tool-Supported Fault Tolerance

- Fault Model
  - original source program
  - assertion generation
  - instrumentation
  - redundant code generation
  - extended source program

- User
  - Introspection Framework
  - FT Knowledge Base
  - TMR, NMR, ...

- Domain Knowledge Base

- Program KB
  - program analysis profiling

- exploiting results of automatic program analysis

- leveraging standard fault tolerance methods

- exploiting domain and application knowledge
Static and Dynamic Analysis

- **Static analysis and profiling** determine properties of dynamic program behavior before actual execution.

**Analysis of Sequential Threads**
- control & data flow analysis: solving flow problems over a program graph
- dependence analysis: determining read/write relationships
- slicing: determining the set of statements that affect a variable’s value

**Analysis of Parallel Constructs**
- data parallel loops: analysis of “independence” property
- locality and communication analysis
- race condition analysis
- safety and liveness analysis
- deadlock analysis

**Dynamic Analysis**
- program control and data flow, dynamic dependences
- performance, energy, and behavior analysis
Concluding Remarks

◆ Extreme-scale systems will need to deal with errors in a flexible and adaptive way

◆ Introspection
  – provides a generic framework for dynamic monitoring and analysis of program execution, together with a recovery scheme
  – can support fault tolerance, performance tuning, power management

◆ Analysis of program properties provides a basis for automatic generation of application-adaptive fault-tolerant code

◆ Integration of conventional V&V technology with introspection can provide a comprehensive approach to fault tolerance

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology