Runtime Verification and Validation
for
Multi-Core Based On-Board Computing

Hans P. Zima and Mark L. James

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA
{zima,mjames}@jpl.nasa.gov

High Performance Embedded Computing (HPEC) Workshop

MIT Lincoln Laboratory, September 23rd, 2009
1. Introduction

2. An Introspection Framework for Fault Tolerance

3. V&V versus Introspection


5. Concluding Remarks

The work described in this presentation has been funded by the JPL Research and Technology (R&TD) project 01STCR R.08.023.015 “Introspection Framework for Fault Tolerance in Support of Autonomous Space Systems”
Future missions and the limited downlink to Earth lead to two major new requirements:

1. Autonomy

2. High-Capability On-Board Computing

Missions will require on-board computational power ranging from tens of Gigaflops to Teraflops.

Emerging multi-core technology is expected to provide this capability.
Future Multi-Core Architectures: From 10s to 100s of Processors on a Chip

- **Tile64** (Tilera Corporation, 2007)
  - 64 identical cores, arranged in an 8X8 grid
  - iMesh on-chip network, 27 Tb/sec bandwidth
  - 170-300mW per core; 600 MHz – 1 GHz
  - 192 GOPS (32 bit)—about 10 GOPs/Watt

- **Maestro** (2010/11)
  - RHBD 7x7 grid SW-compatible version of Tile64 with FP
  - 270mW per core; 480 MHz; 70 GOPs; max 28W

- **Kilocore 1025** (Rapport Inc. and IBM, 2008)
  - Power PC and 1024 8-bit processing elements (125MHz)
  - 32X32 “stripes” dedicated to different tasks

- **80-core research chip from Intel** (2011)
  - 2D on-chip mesh network for message passing
  - 1.01 TF (3.16 GHz); 62W power—16 GOPS/Watt

The issue of dependability, and in particular fault tolerance, has to be addressed in this new context.
Dependability*

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

Threats: the Fault-Error-Failure Chain

- **Fault**
  - Activation: defect in a system
- **Error**
  - Propagation: invalid system state
- **Error**
  - Propagation to service boundary
- **Failure**
  - Violation of system specification

System Boundary (Service Interface)
Means (Fault Protection)

- **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, etc.

- **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
- Activation
- Propagation to service boundary
- Fault tolerance
  - Elimination of detected errors
  - Prevention of fault activations

Error → Failure
- Well-defined system state
Contents

1. Introduction
2. An Introspection Framework for Fault Tolerance
3. V&V versus Introspection
5. Concluding Remarks
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 the threads available in a multi-core system

- can be applied to a range of different scenarios, including:
  - fault tolerance
  - performance tuning
  - power management
  - behavior analysis
An Introspection Module

![Diagram of an Introspection Module]

- Sensors
- Inference Engine (SHINE)
- Monitoring
- Analysis
- Recovery
- Prognostics
- Application Knowledge
- System Knowledge
- Domain Knowledge
- Monitoring
- Analysis
- Recovery
- Prognostics

Application

actuators
Example: SHINE Diagnostics and Prognostics Architecture for DSN Health Management
Contents

1. Introduction
2. An Introspection Framework for Fault Tolerance
3. V&V versus Introspection
5. Concluding Remarks
Traditional V&V versus Introspection

Key Differences

◆ Verification and Validation (V&V)
  – is applied before actual program execution
    ∗ to static source program
    ∗ in the context of test executions for particular input set(s)
  – focuses on design errors

◆ Introspection
  – performs execution time monitoring, analysis, recovery
  – current approach focuses on transient errors
  – can, in principle, also address design errors
Limitations of V&V

- **Verification: theoretical limits**
  - **undecidability**: many problems are inherently unsolvable
    - halting problem
    - constant propagation
  - **NP-completeness**: many theoretically solvable problems are intractable due to exponential complexity
    - SAT problem
    - many graph problems

- **Model checking: subject to scalability challenge**
  - exponential growth of state space

- **Test**
  - tests can prove presence or absence of faults for specific input sets
  - but they cannot prove their absence for all inputs (Edsger Dijkstra)

- **V&V is inherently unable to deal with transient errors or execution anomalies**
Introspection Can Complement 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 extended to deal with design errors*

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

1. Introduction
2. An Introspection Framework for Fault Tolerance
3. V&V versus Introspection
5. Concluding Remarks
Effects of Single Event Upsets (SEUs)

SEUs and MBUs are radiation-induced transient hardware errors, which may corrupt software in multiple ways:

- instruction codes and addresses
- user data structures
- synchronization objects
- protected OS data structures
- synchronization and communication

Potential effects include:

- wrong or illegal instruction codes and addresses
- wrong user data in registers, cache, or DRAM
- control flow errors
- unwarranted exceptions
- hangs and crashes
- synchronization and communication faults
Design and Interaction Faults …

can result in errors similar to those that can be caused by transient faults
Fault Examples: Sequential threads

S1: \( a = \text{exp} \)

... \( \rightarrow \) fault

S1: \( a = \text{exp} \)

... \( \rightarrow \) undefined assignment to \( x \)

S2: \( x = f(a) \)

... \( \rightarrow \) corruption of assignment to variable \( a \)

S2: \( x = f(a) \)

... \( \rightarrow \) corruption of variable \( n \)

for \( i \) in \( 1:n \) \{ 

\( S(i) \) \}

... \( \rightarrow \) destruction of loop range

if \( \text{cond} \) then \( S1 \)

else \( S2 \)

end if

... \( \rightarrow \) corruption of condition \( \text{cond} \)

if \( ? \) then \( ? \)

else \( ? \)

end if

... \( \rightarrow \) branch to wrong or undefined target
**Example: Parallel Algorithms**

**Safety Violation**

```plaintext
P1: repeat forever
    ...
    wait(mutex)
    signal(mutex)
    ...
    end repeat

Critical Section 1: Exclusive access of P1 to R

P2: repeat forever
    ...
    wait(mutex)
    ...
    signal(mutex)
    ...
    end repeat

Critical Section 2: Exclusive access of P2 to R

Corruption of mutex destroys safety property of program
```
Example: Parallel Algorithms

Data Race

forall i in D, independent, A(f(i)) = exp(i) end

A(1)
...
A(f(i1))
A(j)
...
A(f(i2))
A(n)

Fault resulting in the existence of indices i1, i2, such that:
f(i1) = f(i2) = j

data race
Cyclic resource allocation graph results in deadlock (as a result of programming or runtime error)

Static approach/analysis: deadlock prevention
Runtime analysis: deadlock avoidance and detection
High-Performance Computing has produced a wealth of methods for parallel program analysis since the 1990s

- source program static analysis and restructuring
- dynamic performance and behavior analysis of program executions

Many of these methods can be adapted to or directly applied to fault tolerance

- static program analysis (variable use, dependences, call chains)
- dynamic analysis of program and data flow

This provides a basis for the generation of error-checking assertions and error correction and recovery
Analysis

- Static analysis and profiling determine properties of dynamic program behavior \textit{before} actual execution

- Analysis of Sequential Threads
  - control flow graph: represents the control structure in a program unit
  - data flow analysis: solves data flow problems over a program graph
  - dependence analysis: determines relationships between assignments of values to (possibly subscripted, or pointer) variables and their uses
  - program slice: the set of all statements that can affect a variable’s value
  - call graph: representing method/function/procedure calling relationships

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

- **Control Flow Graph** \( G=(N, E, n_0) \)
  - models the control structure in a program unit (control flow analysis)
  - \( N \): set of basic blocks
  - \( E \): control transfers between basic blocks
  - \( n_0 \): initial node

- **Monotone Data Flow Systems** \( M=(L, F, G, g) \)
  - \( L \) is a bounded semilattice representing the objects of interest, e.g., “definitions” reaching a basic block, variable-value associations, available expressions
  - \( F \): monotone function space over \( L \)
  - \( G=(N, E, n_0) \) flow graph; \( g: N \rightarrow F \)
  - under appropriate conditions, an optimal “meet over all paths (MOP)” solution can be determined by a general algorithm

\[ N = \{1, 2, 3, 4, 5, 6\} \]
\[ E = \{(1, 2), (2, 3), (2, 4), (3, 5), (4, 5), (5, 6), (6, 2)\} \]
\[ L: \mathcal{P}(\text{DEF}) \text{ powerset of all “definitions”} \]
\[ g(n) \text{ transformation function for } n \in N \]

For example, the transformation along path 12356 can be computed as \( g(6) \circ g(5) \circ g(3) \circ g(2) \circ g(1)(L_0) \).
Dependence Analysis

- **Dependence** is a relation in a set of statement executions that characterizes their access (R/W) to common variables.

- In general, dependence analysis may **not** be statically feasible—for example, in a 2D Euler solver performing a sweep over an irregular grid.

- “Optimistic parallelization” (or the correctness of an independent assertion) may have to be dynamically verified.

```plaintext
for e in all_edges do  [ independent ]
  ...
  delta = f(GRID(vx1(e)).V1, GRID(vx2(e)).V1)
  ...
  GRID(vx1(e)).V2 -= delta
  GRID(vx2(e)).V2 += delta
  ...
end
```
Generation of Assertion-Based Fault Tolerance

source program

program analysis profiling

instrumentation error detector generation

P

Assertions

P*

Examples:

assert (A(i)>B(i)) and (D<Epsilon) at L;  error (FT1,i,A(i),B(i),D,Epsilon)

assert (x < y+1000) invariant in (Loop1); error (FT2,...)

assert independence in (ParLoop);  error (FT3,...)

instrumented source program with error detectors

User

Knowledge Base
Concluding Remarks

- Future deep-space missions will require on-board high-capability computing for support of autonomy and science

Introspection
- provides a generic framework for dynamic monitoring and analysis of program execution
- can exploit multi-core technology
- a prototype framework for introspection supporting fault tolerance has been implemented for the Tile64 architecture

- Future work will address automatic analysis and generation of application-adaptive fault-tolerant code

- Integration of introspection with conventional V&V technology adds a new dimension 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.