Fault Impact and Fault Tolerance in Heterogeneous Architectures

Michael Engel, Peter Marwedel
Design Automation for Embedded Systems, TU Dortmund

Outline

- A Heterogeneously Reliable Future
- Architecture Knowledge – Capturing Data Flow
- Assessing Heterogeneous Reliability
- The Semantic Gap
- Fine-Grained Resource Management
- Protecting the fault-tolerance components
Introduction

- Future semiconductor technologies pose new challenges
- Another free lunch will be over soon
  - Assuming computers will operate predictably no longer works
  - Errors will be the norm, not the exception
- Heterogeneity will be *forced upon developers*
  - Even in systems not designed as heterogeneous
- How can OS developers help here?
  - Fine-grained fault-tolerance fusing *architecture* and *application knowledge* to adapt software
- Idea: exploit cooperation of architecture, compilers, and OS

A Heterogeneously Reliable Future

- Future MPSoCs will be heterogeneous!
- Not necessarily *by design*, but in their *physical manifestation*
  - Different defects in each SoC of an MPSoC
  - Different defects in each fabricated MPSoC chip
- Defects can manifest themselves
  - Immediately after production
    - Reduce waste, increase yield by using *partially defective chips*
  - During lifetime
    - Keep system running: *graceful degradation & software adaptation*
Error Model

Failure

Data corruption

"No effect"

Error

Bit Flip

Radiation

Process variation

Faults

Jitter

Crosstalk

Physical sources

Temperature

Signal / Vdd noise

Electro-migration

Wrong CPU reg. value

Wrong branch decision

Single/multi

Temporal and Spatial correlated

Permanent/transient

Error Model

What might happen?

- Production defects and degradation can lead to:
  - Permanent errors
  - Increased susceptibility to transient errors
  - Probabilistic behavior
- In any case, we will observe **unexpected bit flips**
  - Manifestation of the Resilience Articulation Point
- Embedded systems constraints require low-cost adaptation to defects
- Design principle
  - Spend work at compile time to reduce work at run time
Protecting Applications Efficiently

- Classify possible errors as to their impact [1]
  - Reduce resource requirements: only correct critical errors
- Provide reliability annotations for data and code on C level
  - Automatic inference of reliability annotations for each C data object and source-level statement
- Safe annotations using compiler-based annotation inference to unannotated data, throwing errors in case of a mismatch
- First results: avoiding application crashes
  - “Black/white” classification
  - Differentiate between errors leading to crashes and others

```
reliable int r;
unreliable int u;
r = u + 42;
```

What’s happening to my Data?

- Problem: Data flows through different architecture components with different reliability characteristics while being processed
  - Annotations do not identify data flow
- Example: int s, d; d = s + 42;
Capturing Architectural Heterogeneity

- Assessment of “dangers” that data and instructions may encounter
  - Static assessment: test at fabrication / deployment time
  - Dynamic assessment: test at run time
- **Idea:** Introduce *dependability microsensors*
- Possible test methods:
  - Hardware microsensors: use parity bits, checksums, etc.
  - *Software microsensors: run frequent in-system tests*
- Microsensors deliver data on detected defects in registers, ALU, memory, caches, branch units, … [2]
  - Part of a future fault-tolerant OS infrastructure

Applying Microsensor Data

- Microsensors deliver fine grained information on the reliability of architecture components
  - What can we do with this information?
  - Reliability annotations use application semantics → source level
- **Problem:** Mapping from C code to architecture components is (mostly) unknown
  - Compiler backend is a black box
  - Control of memory assignment through linker script
  → *A Semantic Gap!*
- Annotations describe neither data flow nor architecture components
A Semantic Gap in the Error Model

Application
Behavior

Semantic
Gap

Microsensor
Data

Don’t
care 😞

Capturing Data Flow

- How can we handle errors in the data flow through the architecture?
- Back propagation of information extracted from compiler-generated assembler code to source code level
  - Successfully employed at DAES for WCET-aware compiler [9]
  - Use additional annotations to ensure data for *reliable operations* and data only uses known *reliable components*
- Virtualization of system architecture on hardware-level
  - Abstracting physical from logical resources in architecture
  - Rewriting instructions on the fly in case of defects
Managing Dependability Heterogeneity

- Tolerating a fault in an architecture component only possible if this component is *redundant* and the component can be substituted.

- What does “redundancy” mean?
  - Spatial redundancy:
    Provide multiple instances of a component (e.g., registers)
  - Temporal Redundancy:
    Replace functionality of one component (or instruction) by a combination of other components (or instructions)

- Components can be replaced using *micro-virtualization*
  - Well-known, e.g., in super-scalar CPUs (register renaming)
  - Binary rewriting for more complex replacements

Example: Exploit Dual ISA

- Replacement on CPU architecture level: ARM32 vs. Thumb
- Replace fixed translation with configurable logic or RAM
Example: Binary Rewriting

- Exchanging parts of an opcode works only for simple cases
  - What if an instruction has to be replaced by > 1 others?
- Binary instruction rewriting: JIT from native ISA to native ISA
- Example: defective bit zero in ALU, 8 Bit addition
  - Handle defect by shifting operands of ALU operation
  - Obviously, there are lots of optimization opportunities here…

```
add r1, r2, r3  ->  lsl r3, r3, #1
                  lsl r2, r2, #1
                  add r1, r2, r3
                  lsr r1, r1, #1
```

Redefining the OS for Reliability

- Traditionally, resource management is primary task of the OS
- Resource management is usually of coarse granularity
  - Complete CPUs in multicores
  - Complete RAM modules
- Defect in a component → disabling of complete coarse-grained unit
  - If only one CPU / one RAM chip: system failure
- Can we let the OS handle fine-grained resource handling?
  - Micro-virtualization/binary rewriting as basic components
  - How large is the overhead involved?
Making Fault Tolerance Fault Tolerant

- How can we ensure that the components required for remapping architecture components are reliable?
- **Classification of architecture components** as to their reliability
- Use only reliable components for fault-tolerance-preserving functions
  - If necessary, protect these components explicitly!
- Determine the **Reliable Computing Base (RCB)** of a system
  - Analogously to Trusted Computing Base in security research
- Classification idea is pervasive
  - From application to architecture level

Related Work and Ideas

- Exokernels and self-adapting kernels
  - MIT ExoPC [3], Synthesis OS [4]
- Binary Rewriting and JIT
  - HP Dynamo [5], Sun Walkabout [6], Transmeta Code Morphing [7]
  - Instruction detouring [8]
- Writable Microcode
  - Already used in the ‘70s to fix PDP11/S360 defects
  - Microcode updates supported in current Intel/AMD CPUs
- Register renaming and architecture virtualization
  - e.g., in superscalar CPUs
Summary

Semantic gap:
- high- vs. low-level code vs. architecture components
- Source analysis doesn’t capture data flow through arch. components
- How can the OS handle fault-tolerance here?
  - More efficient error handling: data flow information required
- Possible approaches:
  - Backpropagation of compiler backend information to source
  - Micro-virtualization
- **Challenge:** Build an OS infrastructure that is able to perform fine-grained resource replacement effectively and efficiently

References (1)


References (2)

“Synthesis: an efficient implementation of fundamental operating system services,”  

[5] Bala, Vasanth and Duesterwald, Evelyn and Banerjia, Sanjeev.  
“Dynamo: a transparent dynamic optimization system,”  

“Walkabout: A Retargetable Dynamic Binary Translation Framework,”  

“The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges,”  

References (3)

“Detouring: Translating software to circumvent hard faults in simple cores,”  
IEEE International Conference on Dependable Systems and Networks (DSN), Anchorage, AK, USA. IEEE, 2008

“Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems,”  