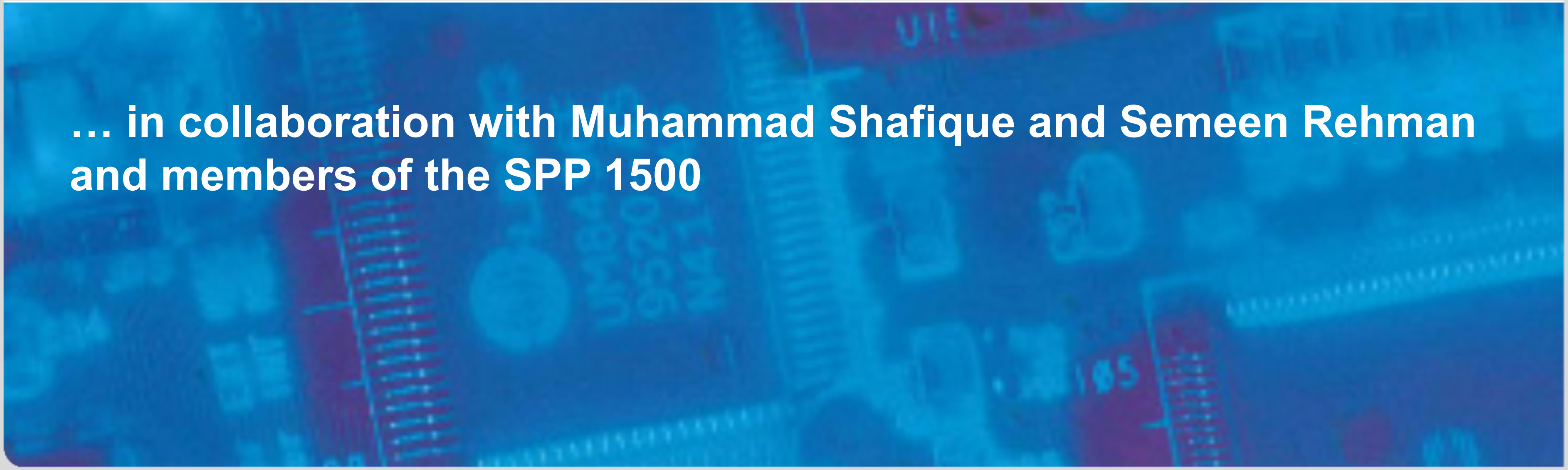
Dependable Software for Undependable Hardware

by Jörg Henkel

SPP1500.itec.kit.edu

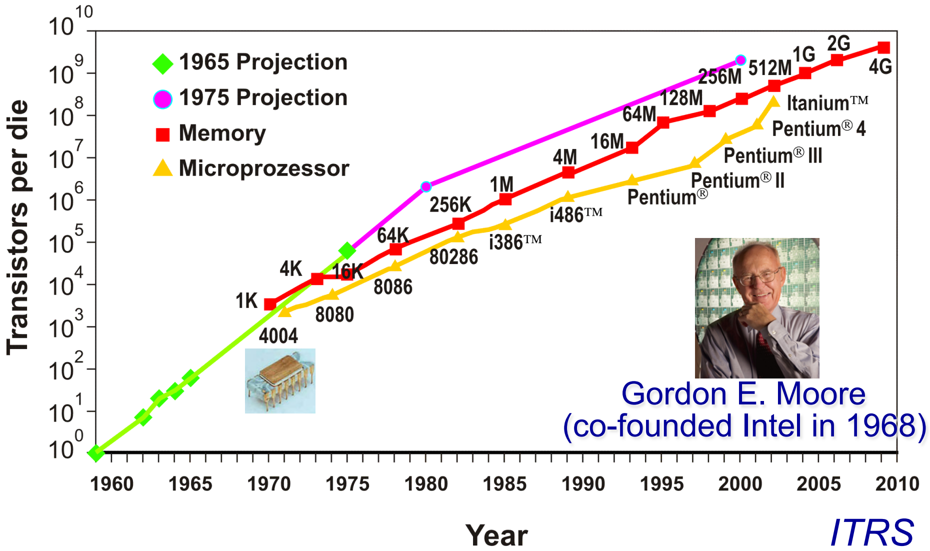


… in collaboration with Muhammad Shafique and Semeen Rehman and members of the SPP 1500

Overview

- Technology Induced Dependability Problems
- Solutions at System-Level with focus on Software

In the Past …

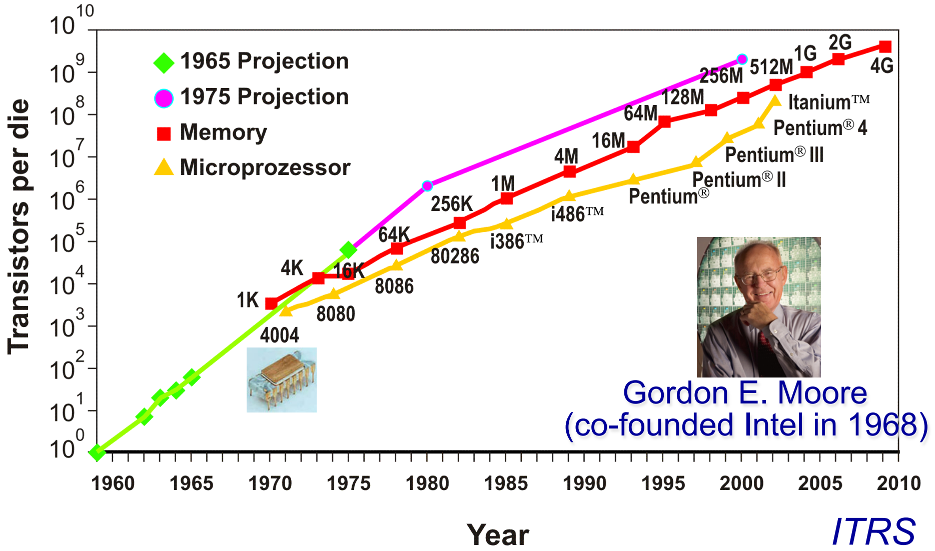


Gordon E. Moore
(co-founded Intel in 1968)

ITRS

- … Moore's Law provided a win-win situation:
  - Smaller feature size
  - Higher integration density, more functionality
  - Lower power consumption
  - Higher speed (performance)
  - Less cost (per-transistor costs)
  - …

In the Future …



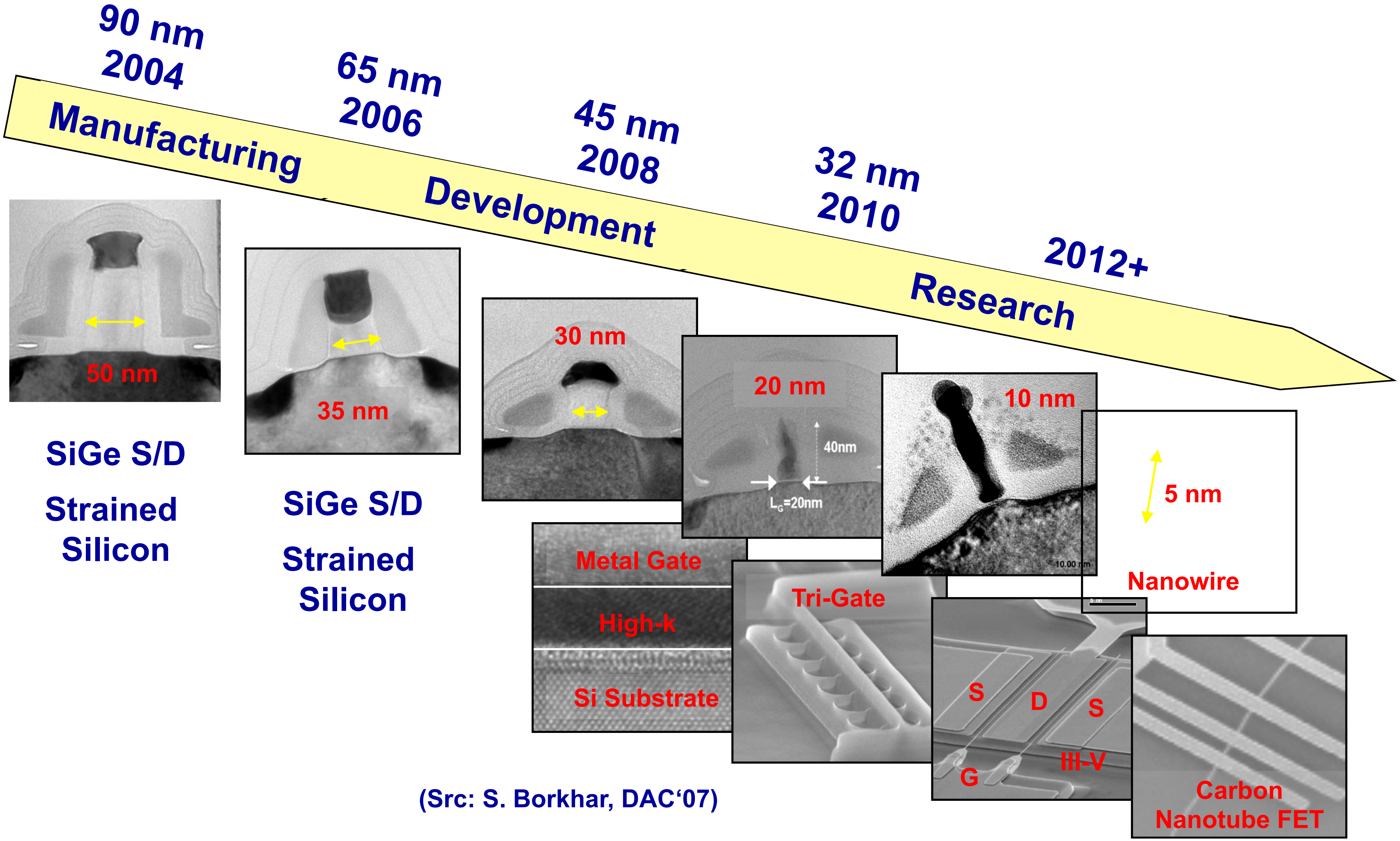
Quelle: Intel Corp., 2005

❑ ... Problems

- □ Complexity: In 2017 100 Billion Transistors on chip
- □ Productivity gap
- □ Thermal problems
- □ Increasing relevance of aging effects
- □ Manufacturing defects, process varation
- □ Stochastic effects since physical limits are reached
- □ Decreasing yield

Reliability

Technology Scaling



90 nm 2004

65 nm 2006

45 nm 2008

32 nm 2010

2012+

Manufacturing

Development

Research

SiGe S/D

Strained Silicon

SiGe S/D

Strained Silicon

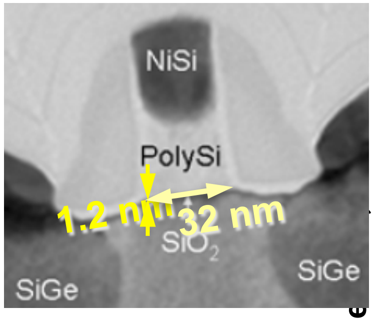
(Src: S. Borkhar, DAC'07)

Variabilities

- Variability of transistor structures
  - Channel Length
  - Isolators thickness (gate oxid) gate <-> transistor channel
  - Randomized Dopant Fluctiations (RDF) -> Threshold voltage

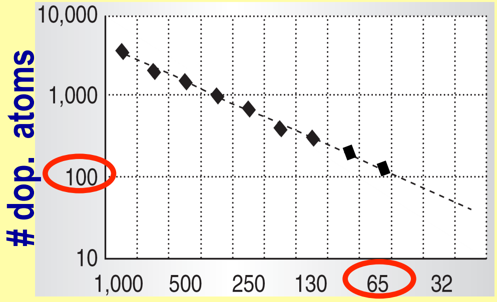
  => Decreasing mobility

  => Increasing leakage
- Counter Measures
  - Strained Silicon Engineering
    - Strain channel to increase mobility
  - „High-K“ materials for gate isolation (e.g. Hafnium)
    - May increase aging
- …

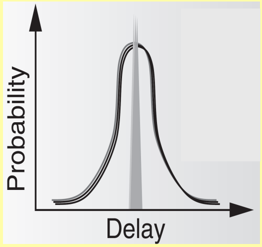


Quelle: Intel, 65 nm

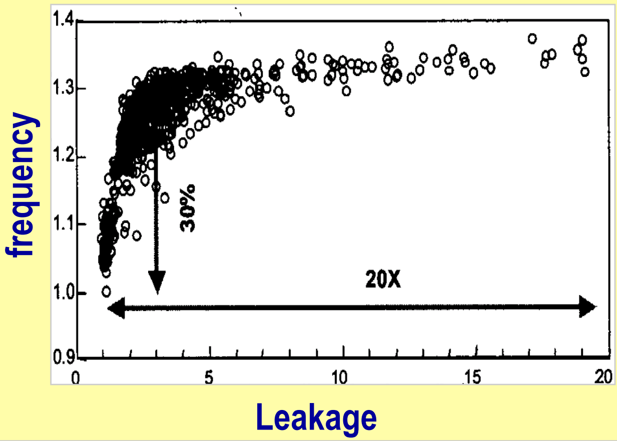
# of dop. atoms in T-channel



latency of an Inverter



Variations for processors an single die

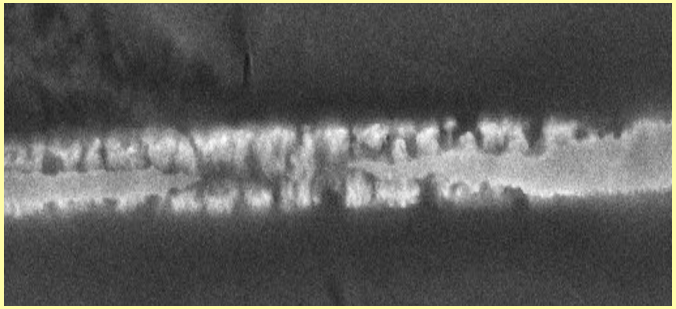


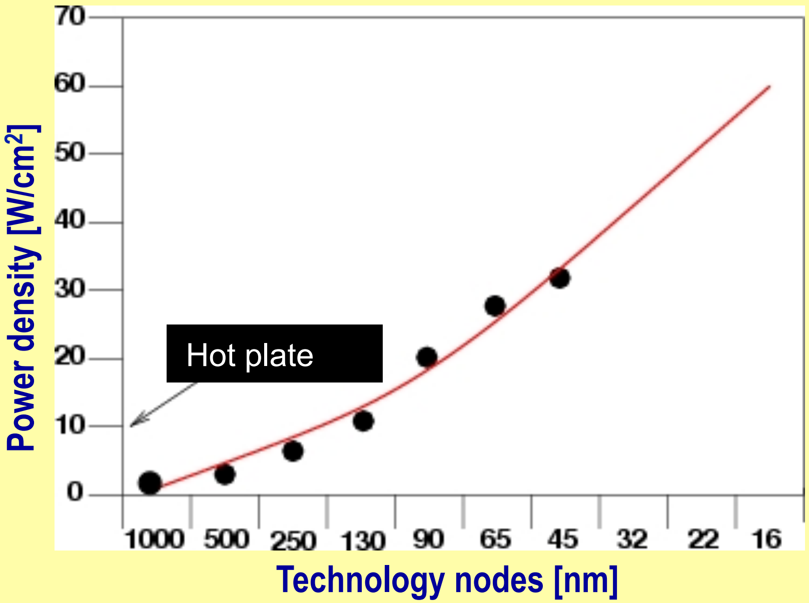
Aging Effects

- Elektromigration (EM)
- Stress Migration
- Time-dependent Dielectric Breakdown
- …

=> dependent upon operating temperature !

Wire affected by electro migration





Increasing Susceptibility to Soft Errors

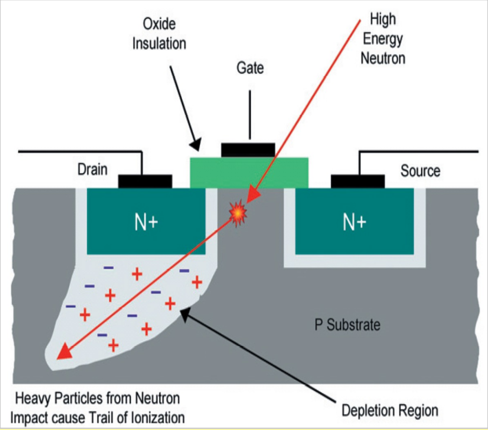
- Ionizing rays may change charge concentration
  - (like He^{2+})

  => may lead to bit flips
- α-rays
  - Radioactive decompostion of non-pure chip material

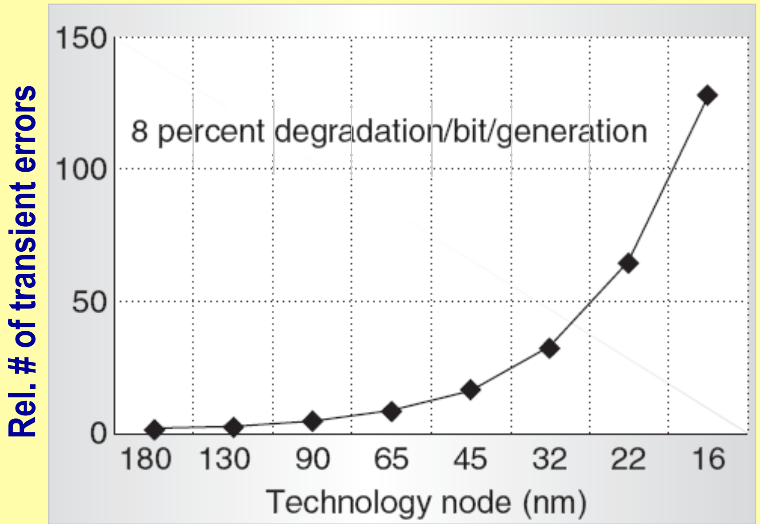
$${}^{A}_{Z}X \rightarrow {}^{A-4}_{Z-2}Y + {}^{4}_{2}He$$

- Cosmic rays (e.g. neutrons)
- accelerated through technolgy advancements
  - Low voltage and capacitances
  - Representation of bits through smaller and smaller charges

Transient errors through neutrons

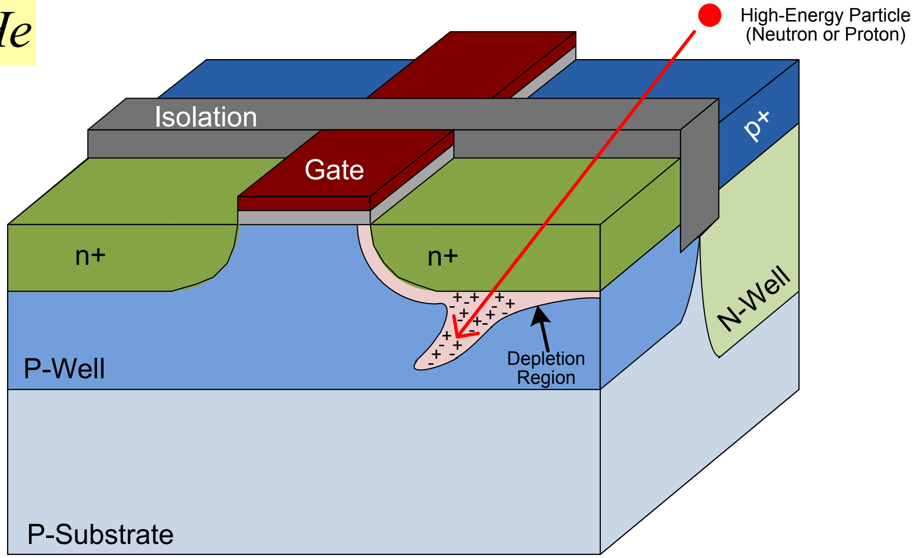


Transient errors

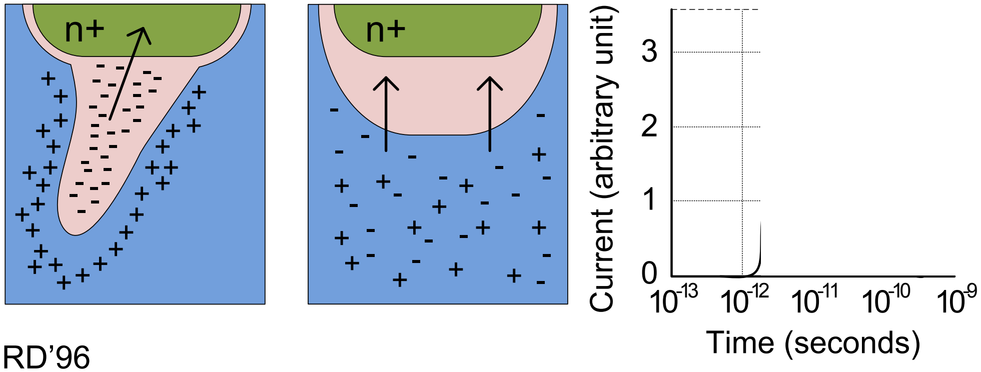


Soft Errors through Radiation

- radiation effects on semiconductor devices→Soft Errors
  - alpha particles $^{A}_{Z}X \rightarrow {}^{A-4}_{Z-2}Y + {}^{4}_{2}He$
  - low-energy neutrons
  - high-energy neutrons/protons
- radiation event
  - ion track formation
  - ion drift
  - ion diffusion



- **Sensitive areas:**
  - Channel region of NMOS
  - Drain region of PMOS
  - “off” state is more sensitive

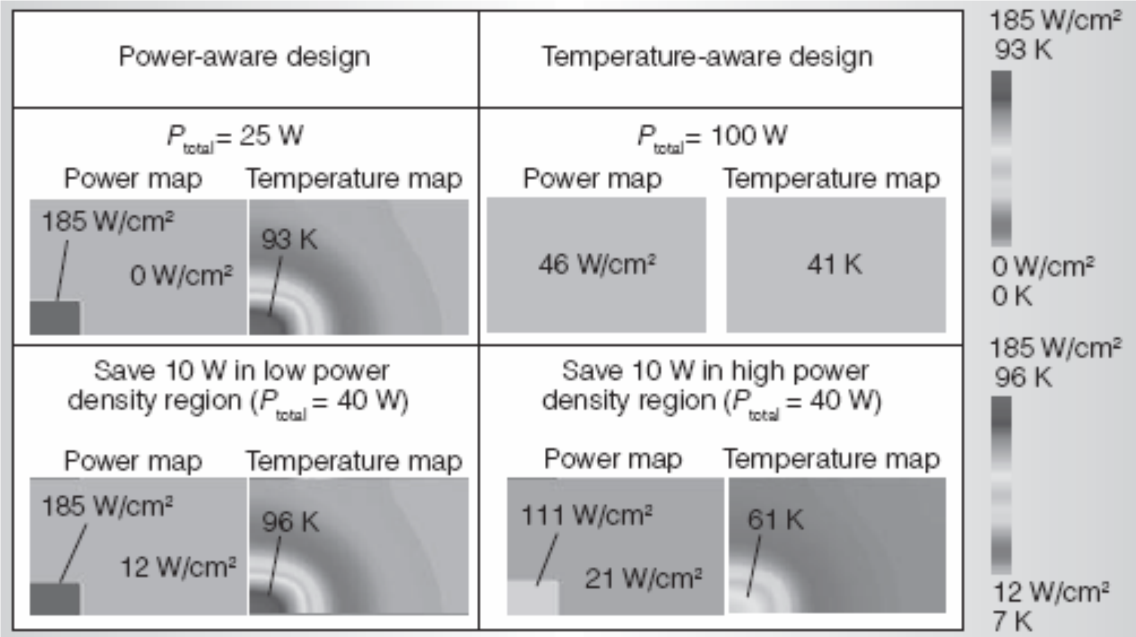


Source: Baumann, TI@Design&Test’05, Ziegler, IBM@IBM JRD’96

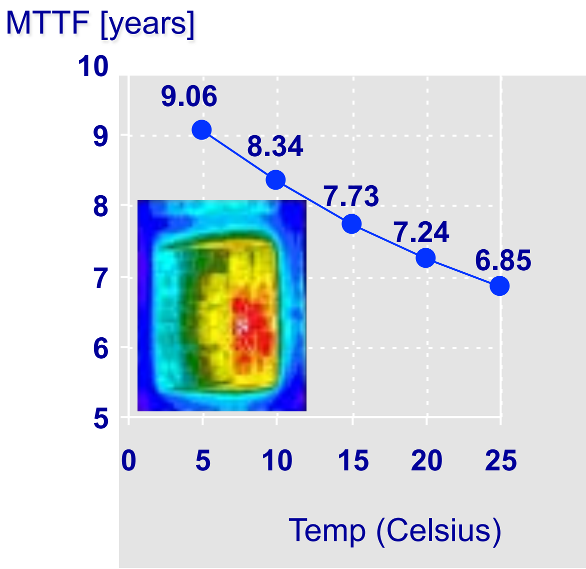
Heat Remains a Problem …

“Circuit heat generation is the main limiting factor for scaling of device speed and switch circuit density”

By Jeff Welser, Director SRC Nanoelectronics Research Initiative, IBM, Opening Keynote Address ICCAD 2007



(Src: K. Skadron: Low-Power Design and Temperature Management; IEEE Micro, Vol. 27, No. 6, 2007)



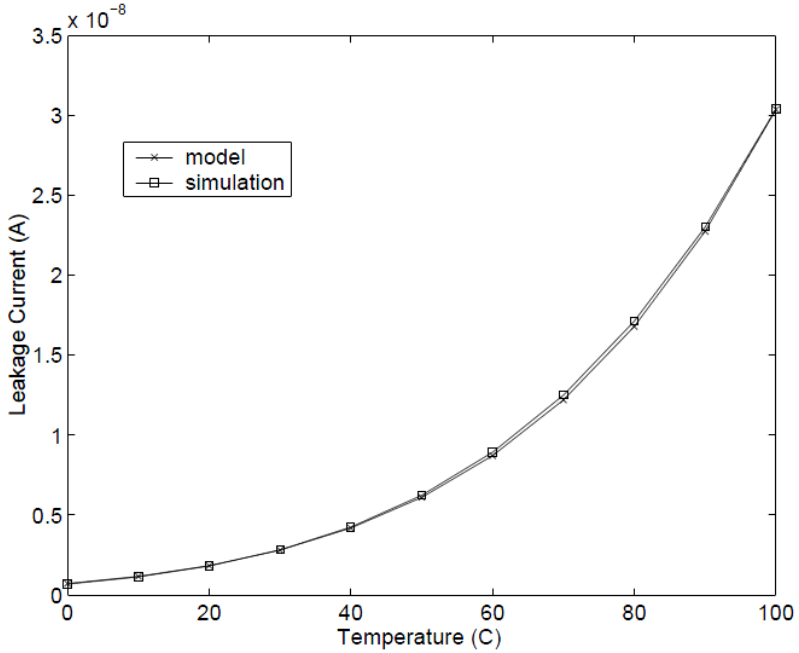
K. Skadron et al., ICCAD 2004

Thermal “Runaway” Problem

- Temperature and leakage: thermal “runaway” problem:
  - Increase in temperature leads to increase in leakage power → feedback loop possible!
  - Sub-threshold leakage approximated by

$$I_{sub} \approx A \cdot e^{-\frac{B}{T}}$$

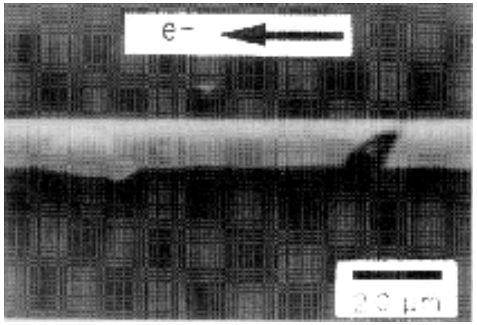
where *A* and *B* are constants → exponential growth!



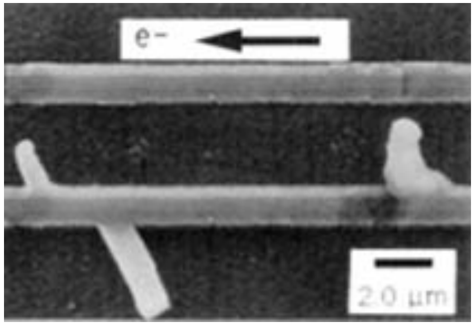
[Zhang 2003]

Temperature-Dependent Effects

- Process variations and electromigration can result in hillocks and holes
  - Lead to open failures or short circuit failures respectively
  - Failures may be temperature dependent due to material expansion
    - Holes may function normally at high temperatures but fail at low temperatures
    - Hillocks may function normally at low temperatures but short circuit at high temperatures



Hole/crack



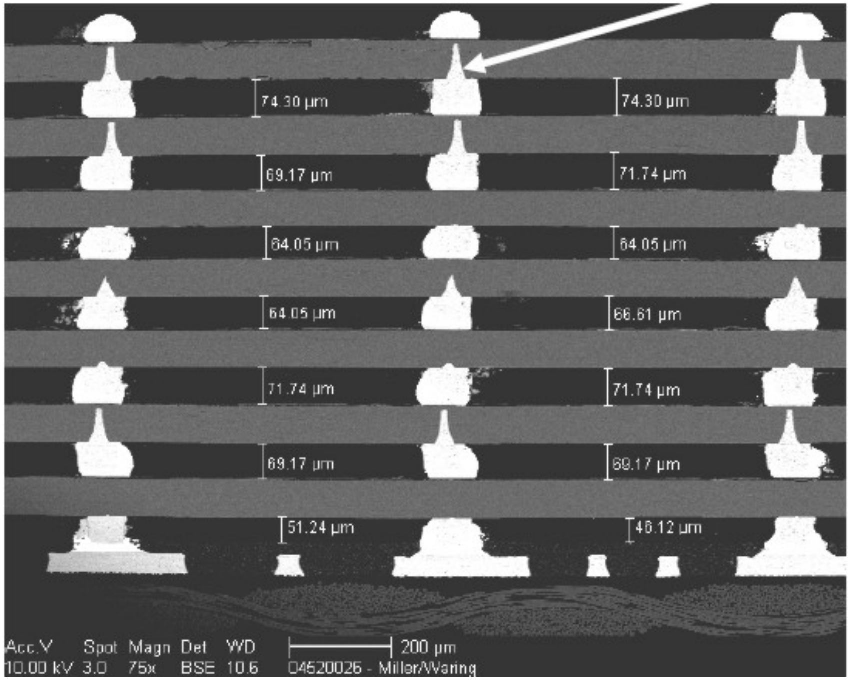
Hillock

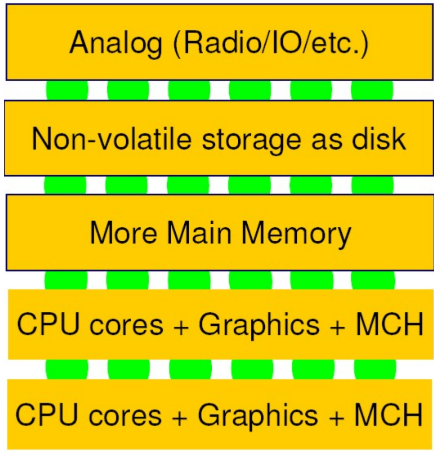
[W.D. Nix, 1992]

Temperature in 3D

- 3-D chips especially problematic

3-D structures




Analog (Radio/IO/etc.)

Non-volatile storage as disk

More Main Memory

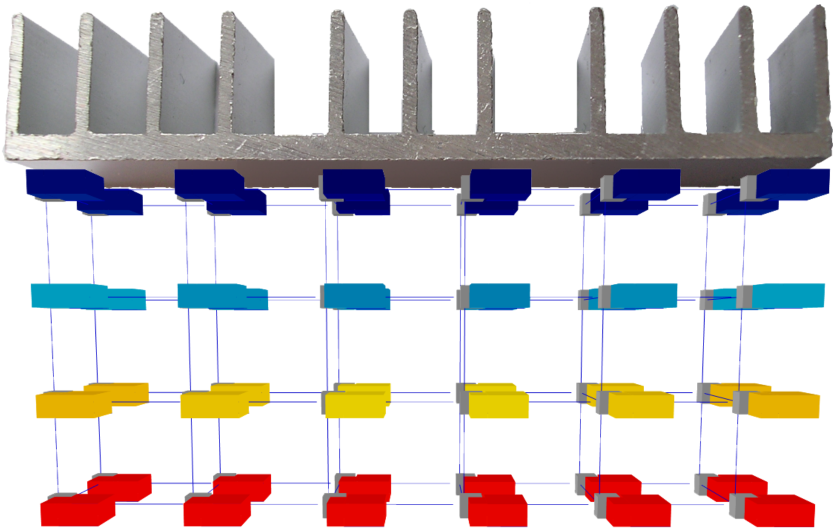
CPU cores + Graphics + MCH

CPU cores + Graphics + MCH

(Src: Y. Xie, PennState)

Temperature in 3D

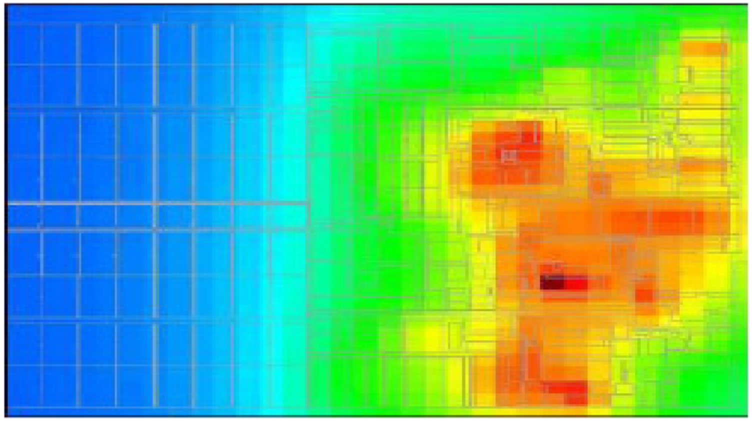
- Problem: vertical heat flow
  - Only one layer directly interfaces with the heat sink
  - Heat needs to dissipate through multiple layers



- The heat sink is located on top of the chip
- Hot cores distant to the heat sink dissipate their heat through other layers
- Silicon has a *low thermal conductivity!*
  - *150 W/(m*K) (Silicon)*
  - *401 W/(m*K) (Copper)*

Temperature: Gradients matter …

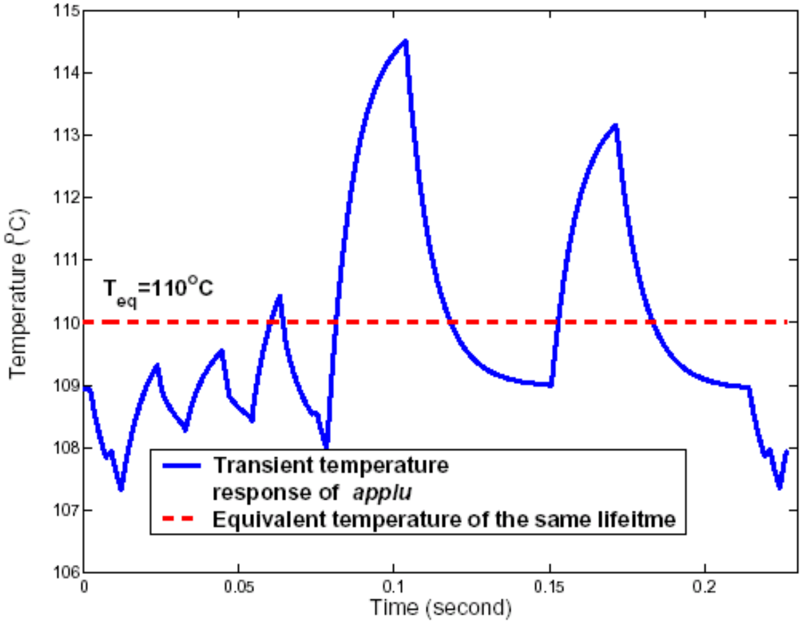
- *MTTF* also affected by thermal gradients



Spatial gradients

Simulated Thermal map Pentium M

[L.Finkelstein, Intel 2005]



Temporal gradients

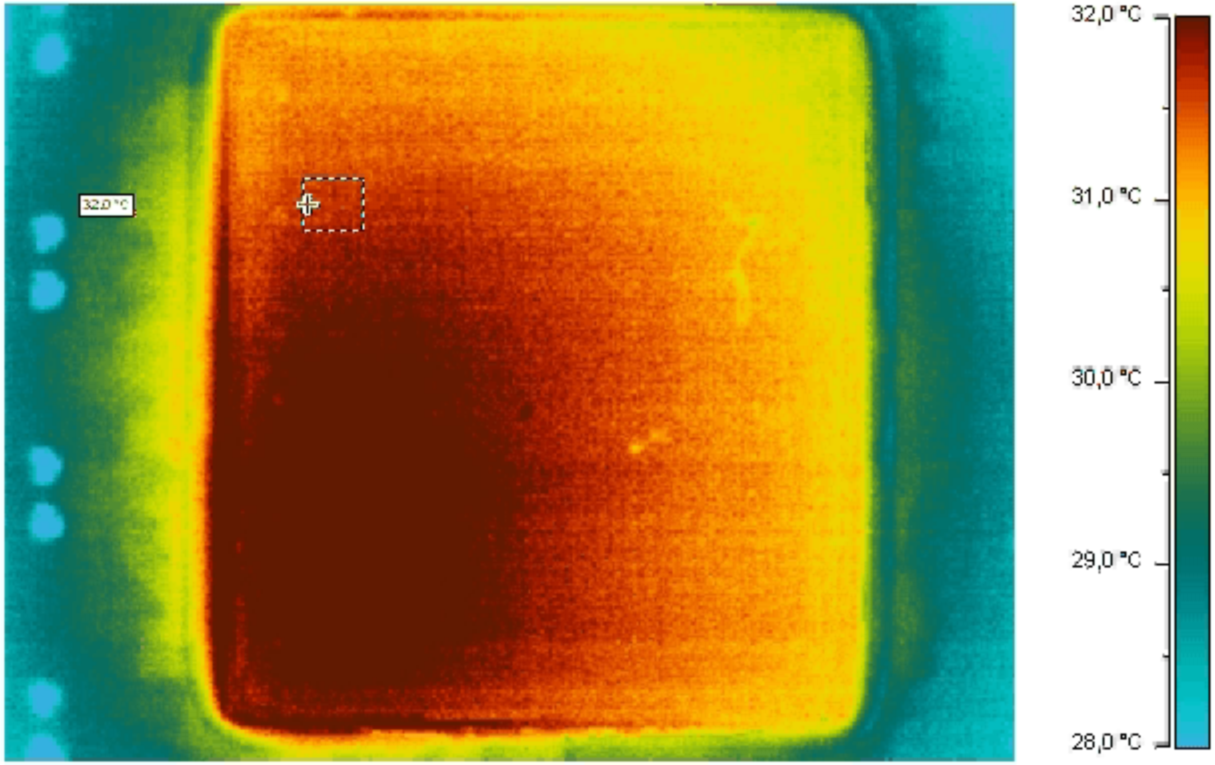
[K. Skadron, 2005]

- → Goal: balance temperatures

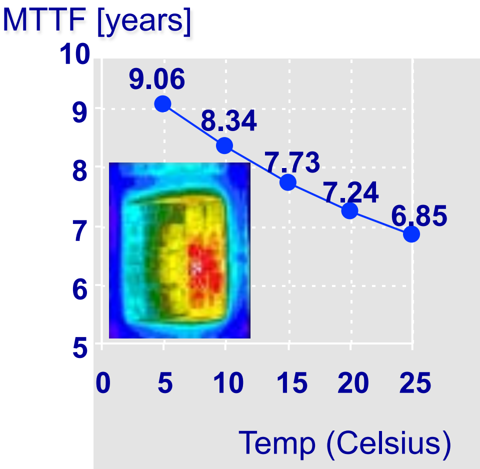
Temperature: Hot Spots

- Example showing localized computation

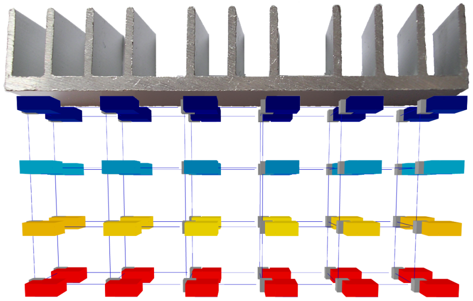
DIAS Infrared Systems



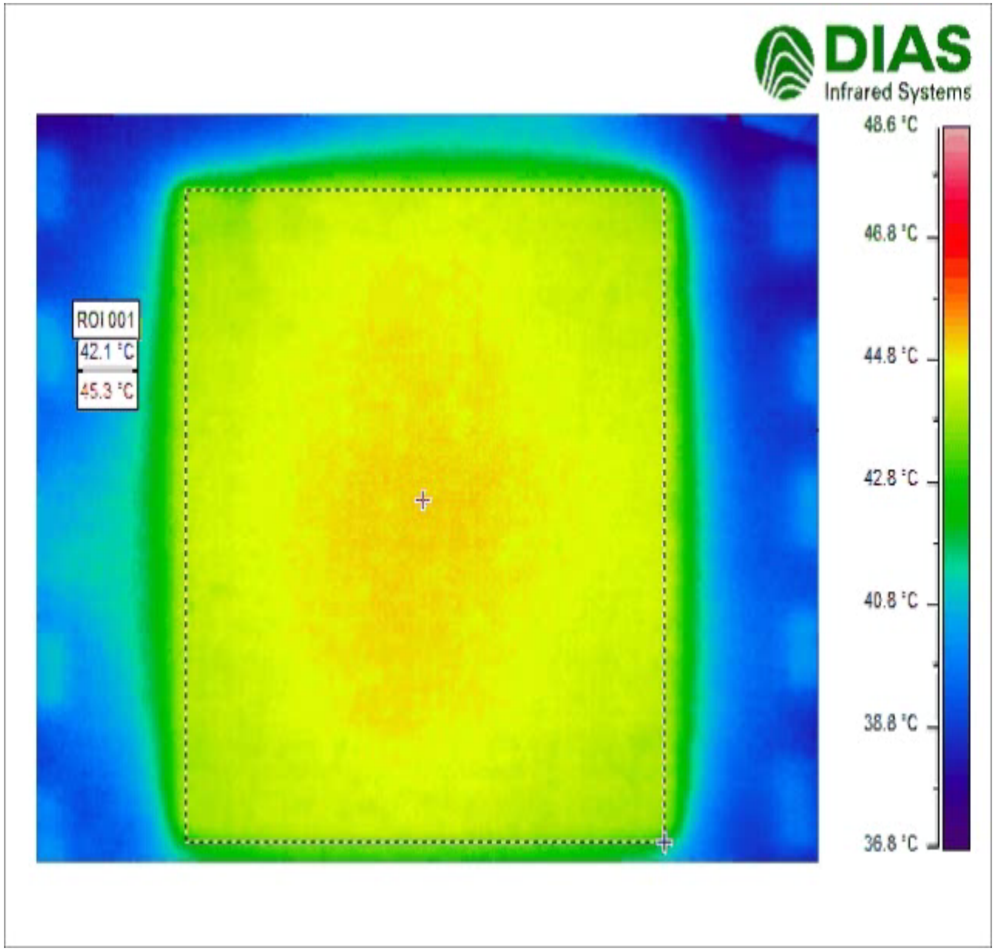
Src: Henkel, Ebi, Amrouch



K. Skadron et al., ICCAD 2004



Real Temperature Measurements



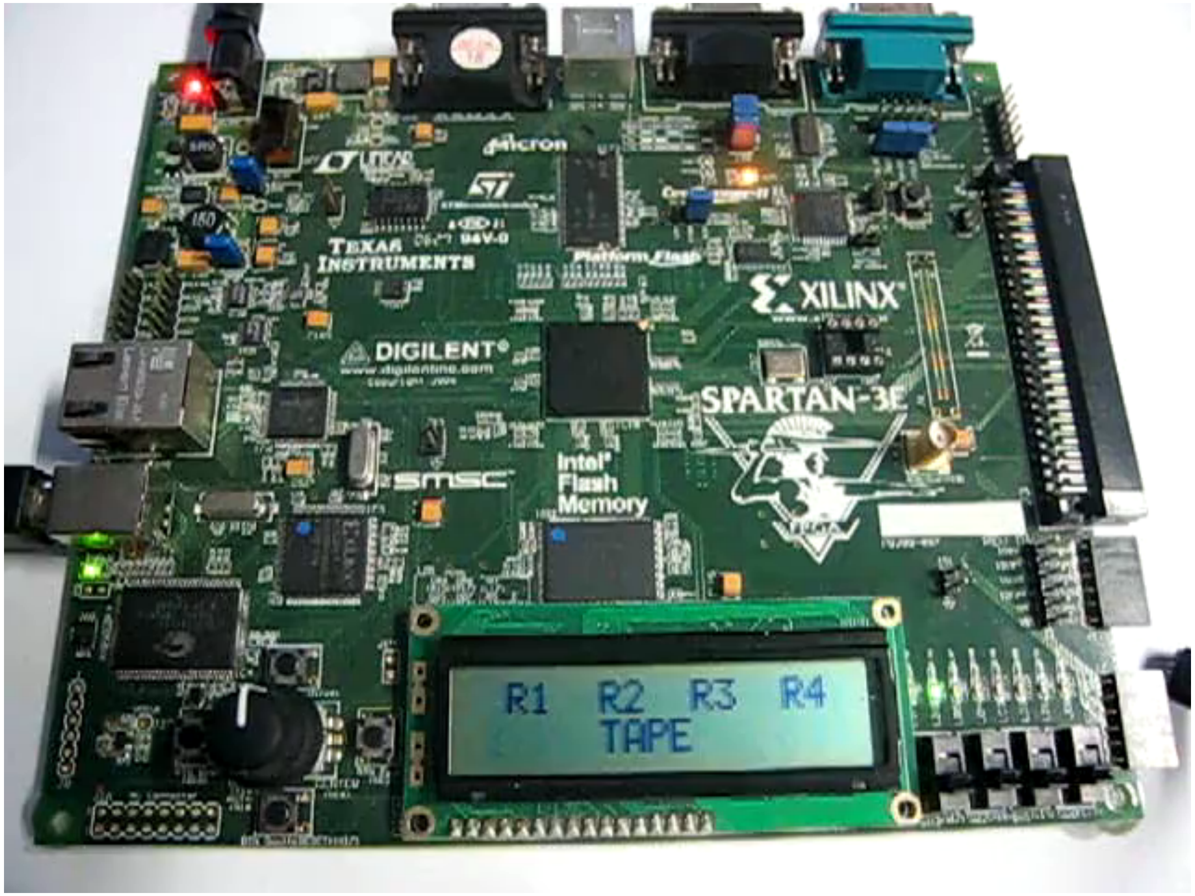
Temp max: 89.2 °C
Temp min: 60.2 °C
Thermal variation: 29°C
Spatial thermal gradient: ~1,93 °C/mm

Properties of the tested region

resources	
LUTs	2000
FFs	2000
DSPs	12
Bram	0
DCM	1
Frequency	600 MHz
Area	6% of chip

Effect of Temperature …

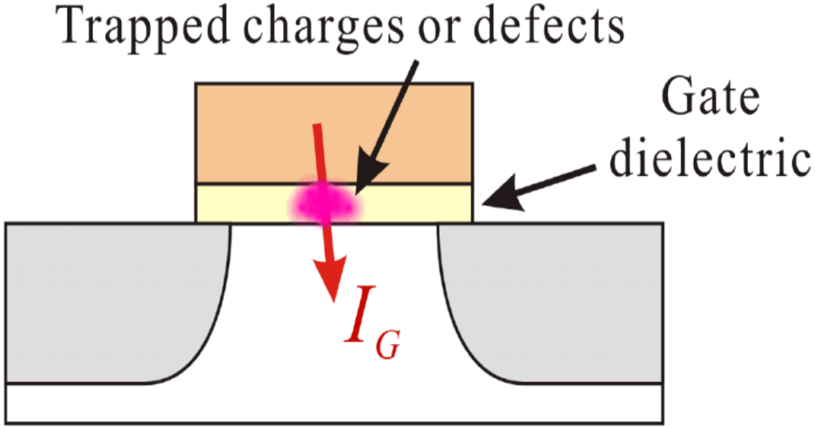
- Hardware prototype : Xilinx Spartan3e FPGA with 4 Picoblaze tiles; thermal sensors realized through ring oscillators



Src: Henkel, Ebi, Al Faruque

Aging: TDDB

- TDDB: Time Dependent Dielectric Breakdown
  - Created by:
    - Accumulation of trapped charges at dielectric
  - Effects:
    - Increase of power consumption
    - Slowing of switching speed
    - Or: may destroy transistor



(TDDB)

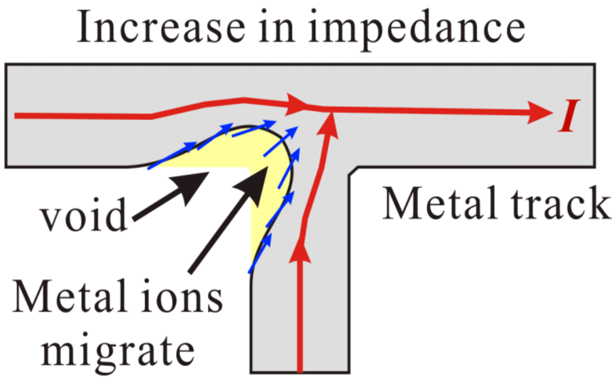
Aging: Electro-Migration

- Electro-migration: aging effect due to transport of mass in metal interconnects
- directly linked to temperature
  - Basic *Mean time to failure* modeled by Black's Equation:

$$MTTF = Aj^{-n}e^{\left(\frac{Q}{kT}\right)}$$

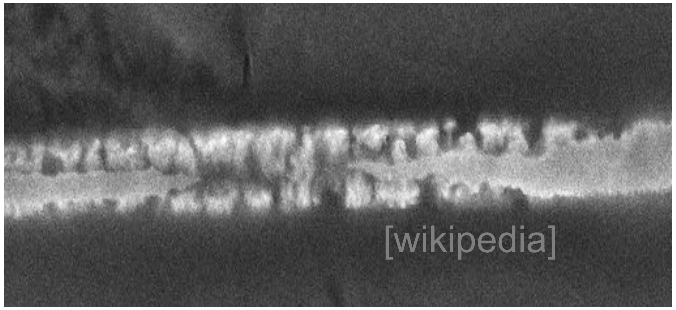
  - *MTTF* decreases exponentially with temperature

→ Goal: reduce peak temperatures



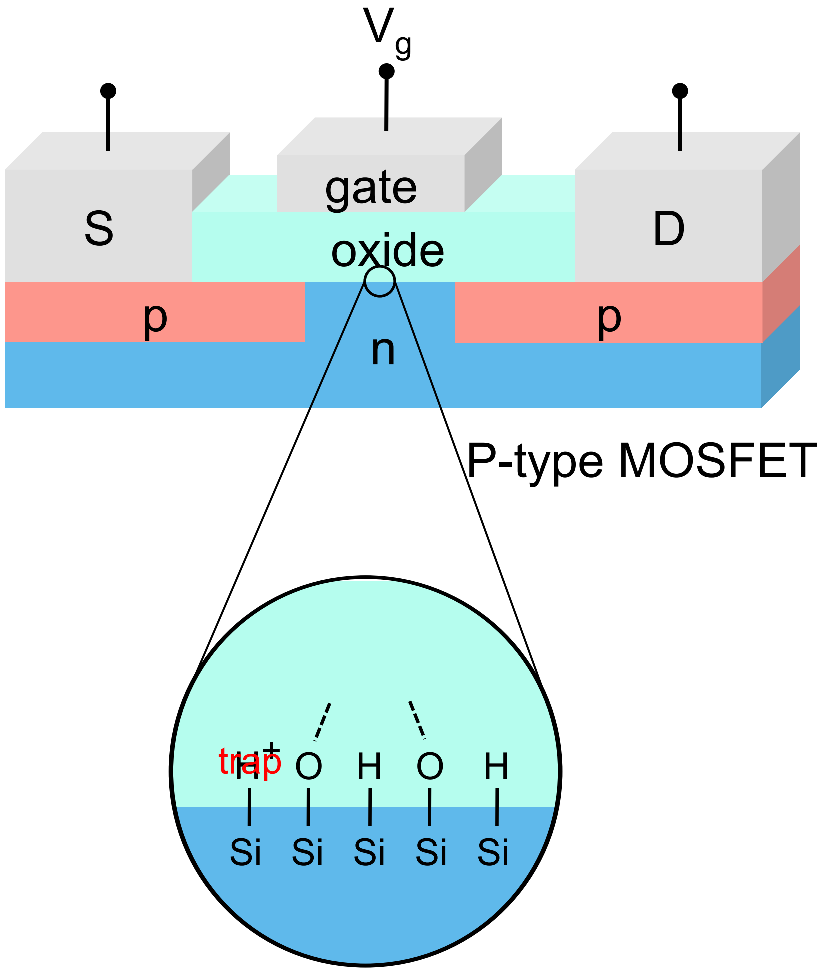
(Electro-migration)

Source: [Stott, 2010]



Aging: NBTI

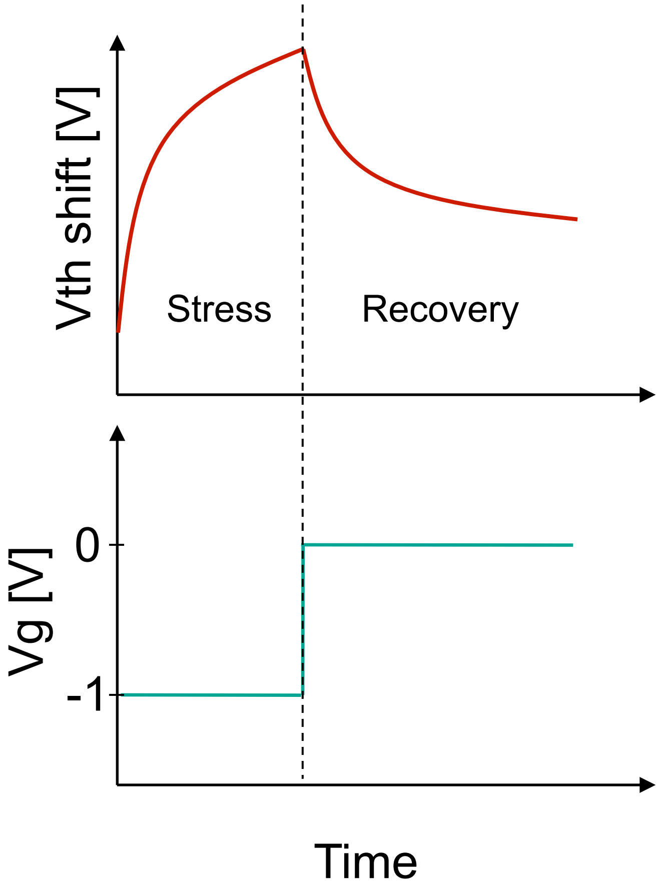
- Negative Bias Temperature Instability
  - Breakdown of Si-H bonds at the silicon-oxide interface due to voltage/thermal stress → causes interface traps
- Affects mostly P-MOSFETs because of negative gate bias
  - Effect in N-MOSFETS is negligible
- NBTI is not yet fully understood



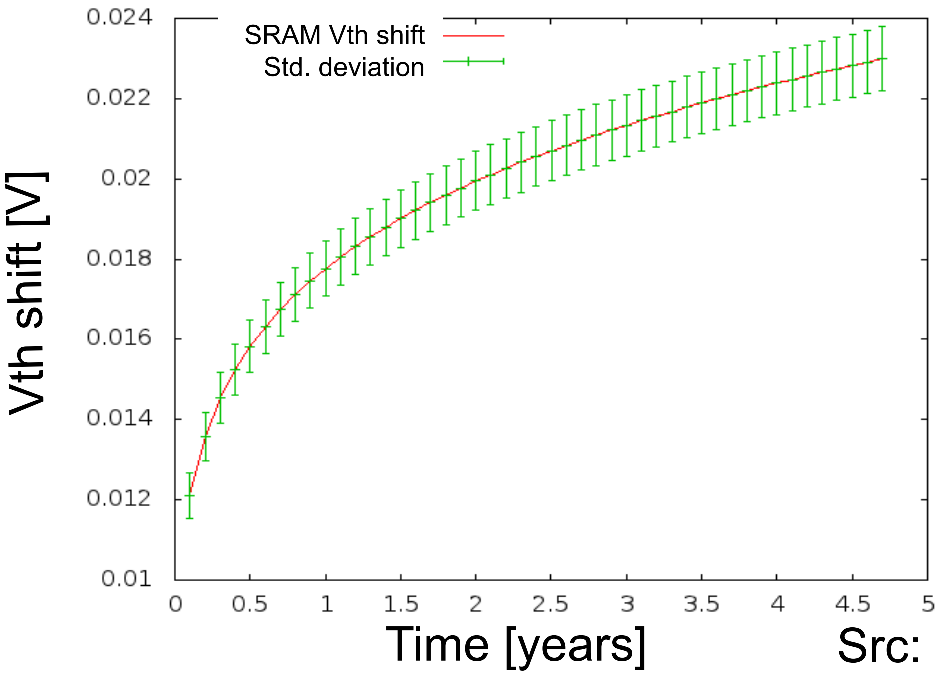
V_g < 0 → STRESS!

Aging: NBTI

- NBTI manifests itself as a shift in V_{th}
  - Causes increase in transistor delay
  - Delay faults are responsible for NBTI induced bit-flips and resulting circuit failure
- Recovery effect in periods of no stress
  - When voltage and temperature are low, V_{th} can shift back towards ist original value
  - Full recovery from a stress period only possible in infinite time
    → In practice overall V_{th} shift increases monotonously over longer periods, e.g. months/years

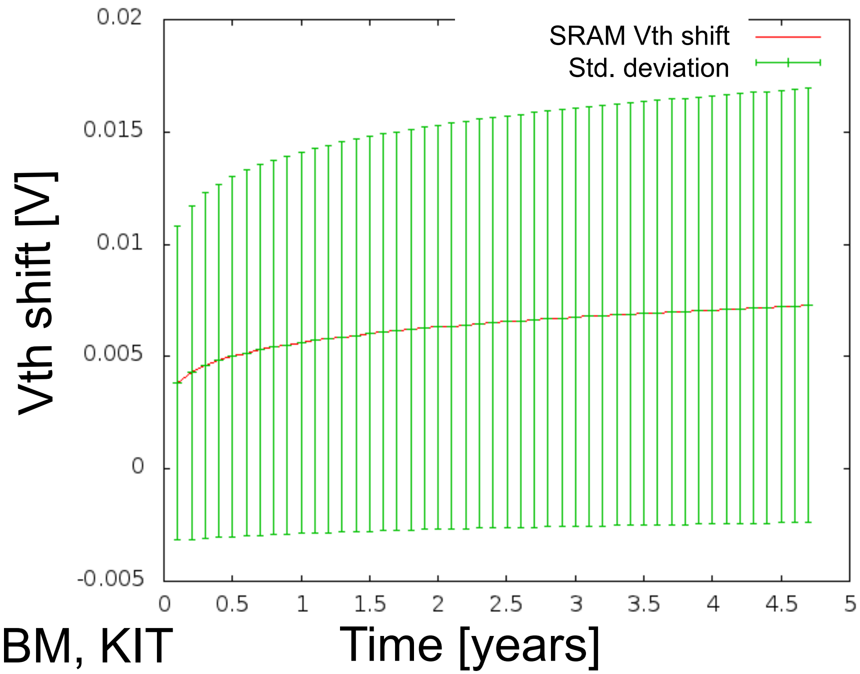


Aging: NBTI



Std deviation in 65nm SRAM P-MOSFETS

Src: IBM, KIT



Std deviation at 32nm

- Mean V_{th} shift mainly due to Temperature/Voltage
  - Small technology nodes have less V_{th} shift due to lower voltages
- However: Standard deviation of V_{th} shift mainly due to structure size
  - Small technology nodes and small P-MOSFETs (e.g. SRAM) show large deviations from the mean V_{th} shift → inceased reliability concern

Other Effects

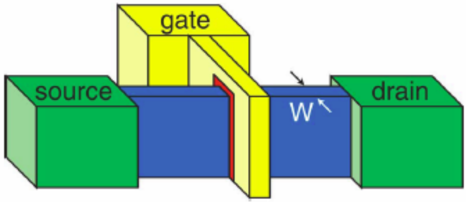
This was not a complete list …

Goal: How can we address the negative effects caused by the inherent unreliability observed at transistor and physical level when migrating to new technology nodes?

So, what are the solutions … ?

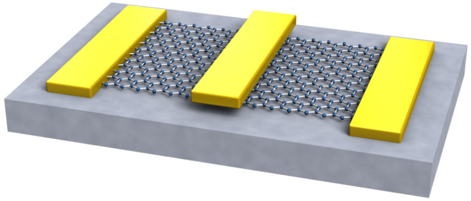
Solutions: Device Level

FinFET-Transistor

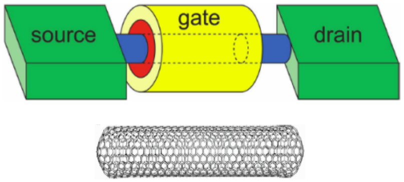


Idea: reduce channel thickness
But: reduced mobility

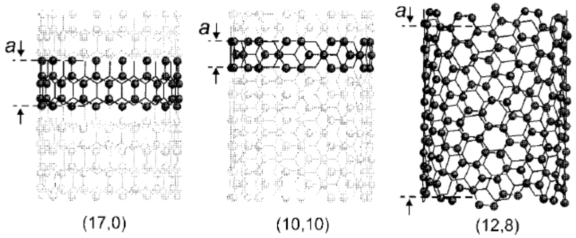
Graphene-Transistor



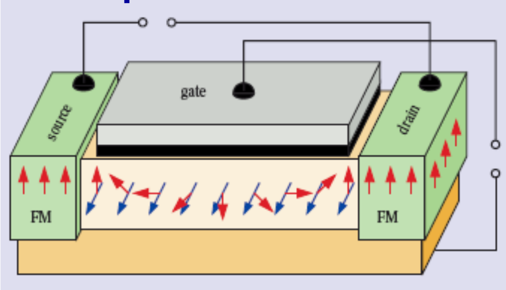
CNFET-Transistor



Idea: combine high mobility and thin channel width
But: problems in placement and structural growth

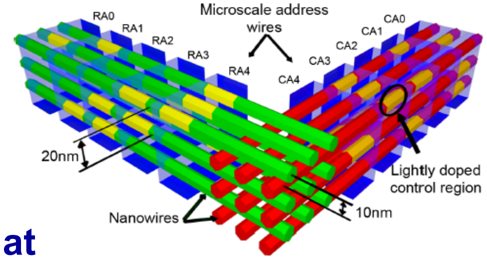


Spin-Transistor



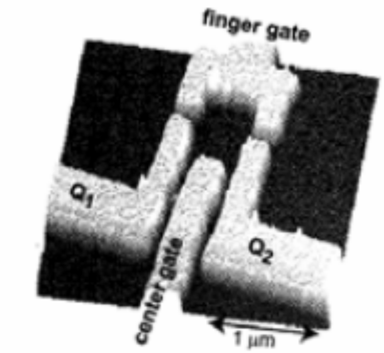
Injection of spin-polarized electrons at source V-gate affects spin trace electron current only when electron spin parallen to drain-spin
Idea:low power dissipation
But: hard to control => high error rates

NanoPLA block and 3D Interconnect

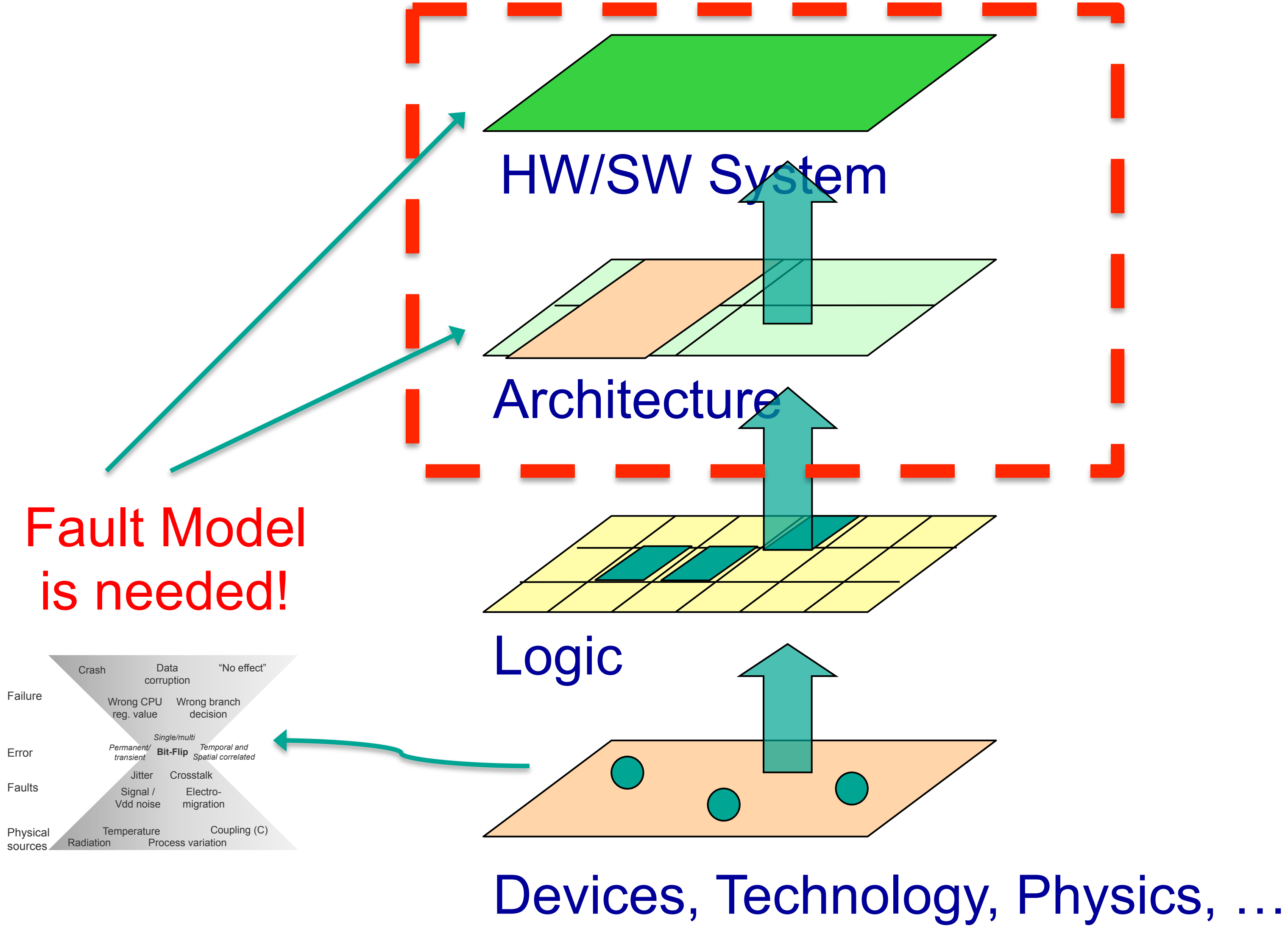


Source: DeHon

Single-Electron Transistor



Solutions: System Level



Fault Model is needed!

Solutions: System Level

Dependable Embedded Software

- Hardware-dependent software
  - Operating system and middleware
    - Management of observation strategies
    - Performing online tests
    - Perform adaptation
  - Scheduling and allocations schemes
  - Application software:
    - instruction-level
    - task-level
    - algorithm level

Solutions: System Level

Dependable Embedded Software

Dependable Hardware Architectures

- Hardware Architectures: various levels
  - Register-Transfer
  - Micro-Architecture
  - System-on-Chip
- Technology Abstractions provides physical properties
- Distinguish between:
  - Permanent and transient problems
  - Fabrication time and run-time (detect and fix)
- Possible means:
  - Masking of undependable components
  - Reconfiguration
    - static
    - dynamic

Solutions: System Level

Dependable Embedded Software

Dependable Hardware Architectures

Technology Abstraction

- This SPP does not deal with technology!
- Means and architectures should be as technology independent as possible
- Technology abstraction should:
  - Characterize technology
  - Provide technology parameters
  - Model undependability
  - …

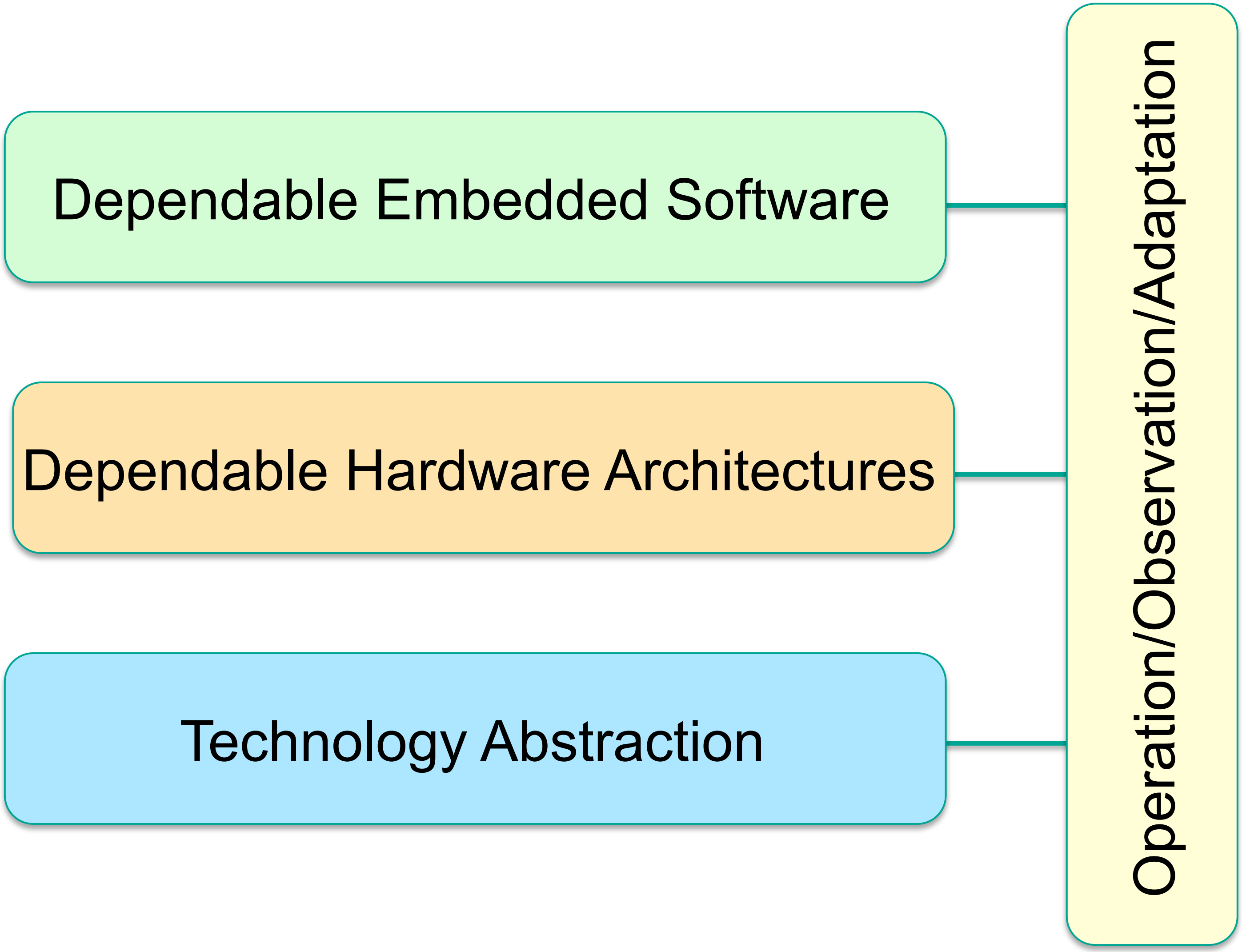
Solutions: System Level

Dependable Embedded Software

Dependable Hardware Architectures

Technology Abstraction

Solutions: System Level



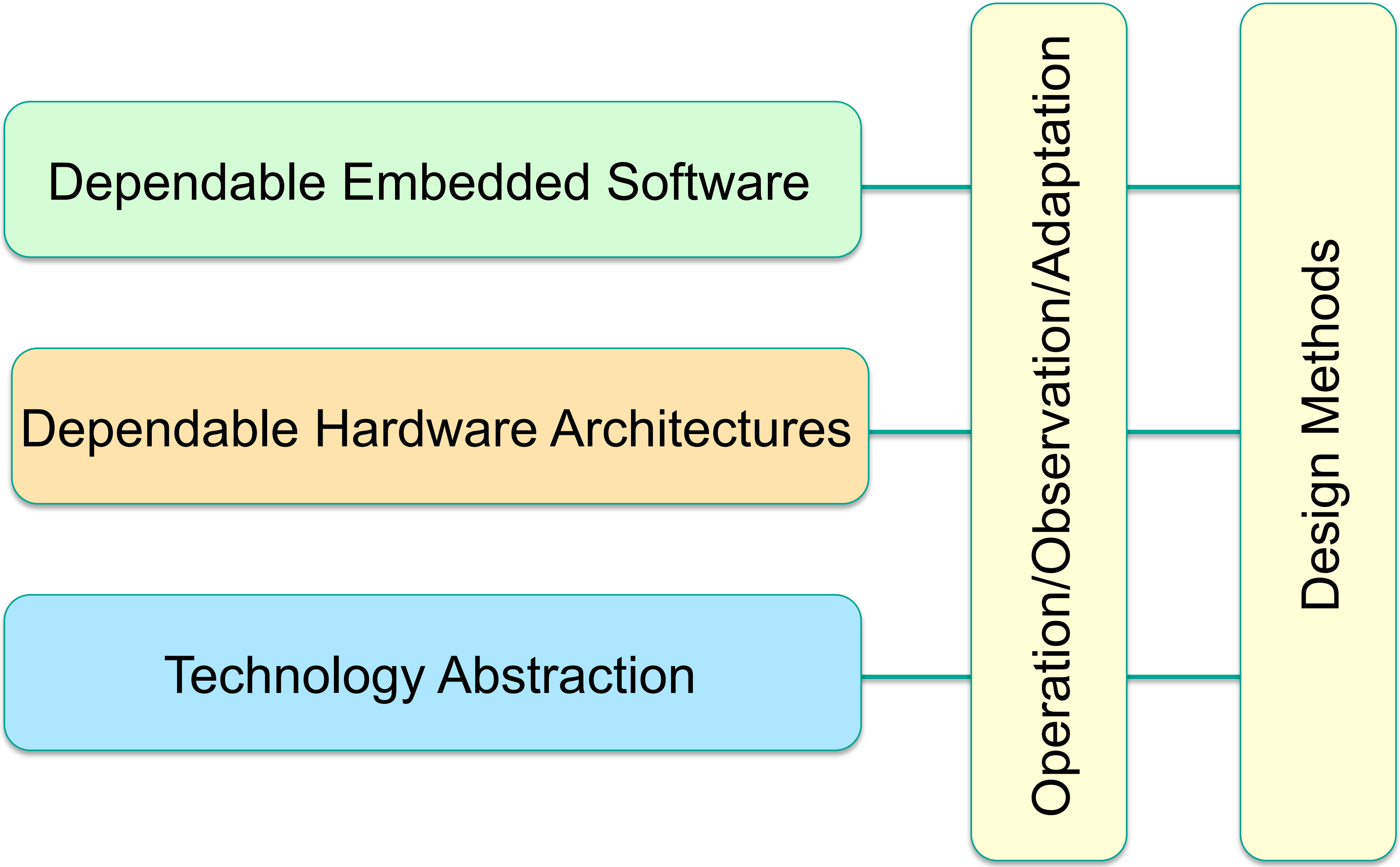
Dependable Embedded Software

Dependable Hardware Architectures

Technology Abstraction

Operation/Observation/Adaptation

Solutions: System Level



Dependable Embedded Software

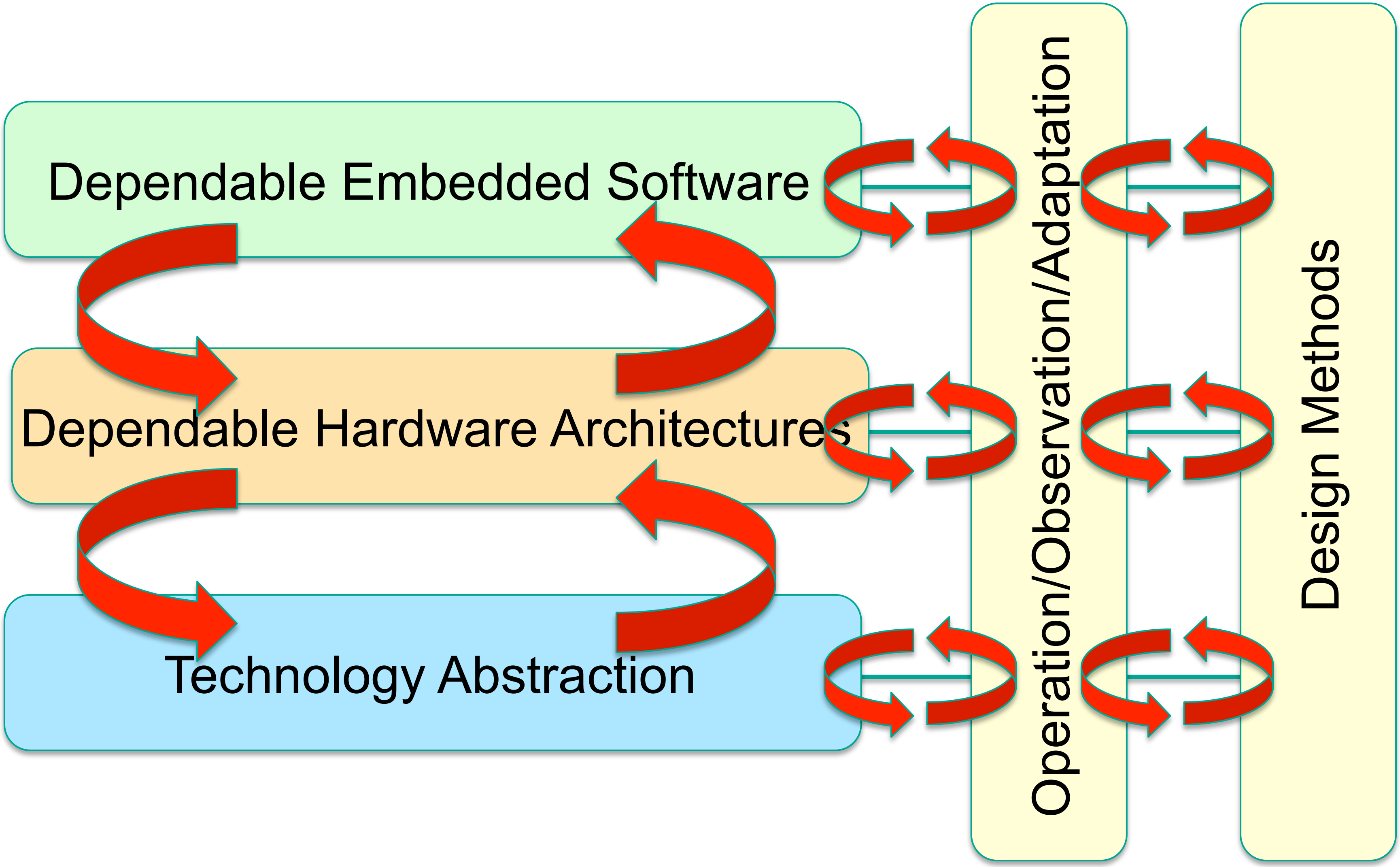
Dependable Hardware Architectures

Technology Abstraction

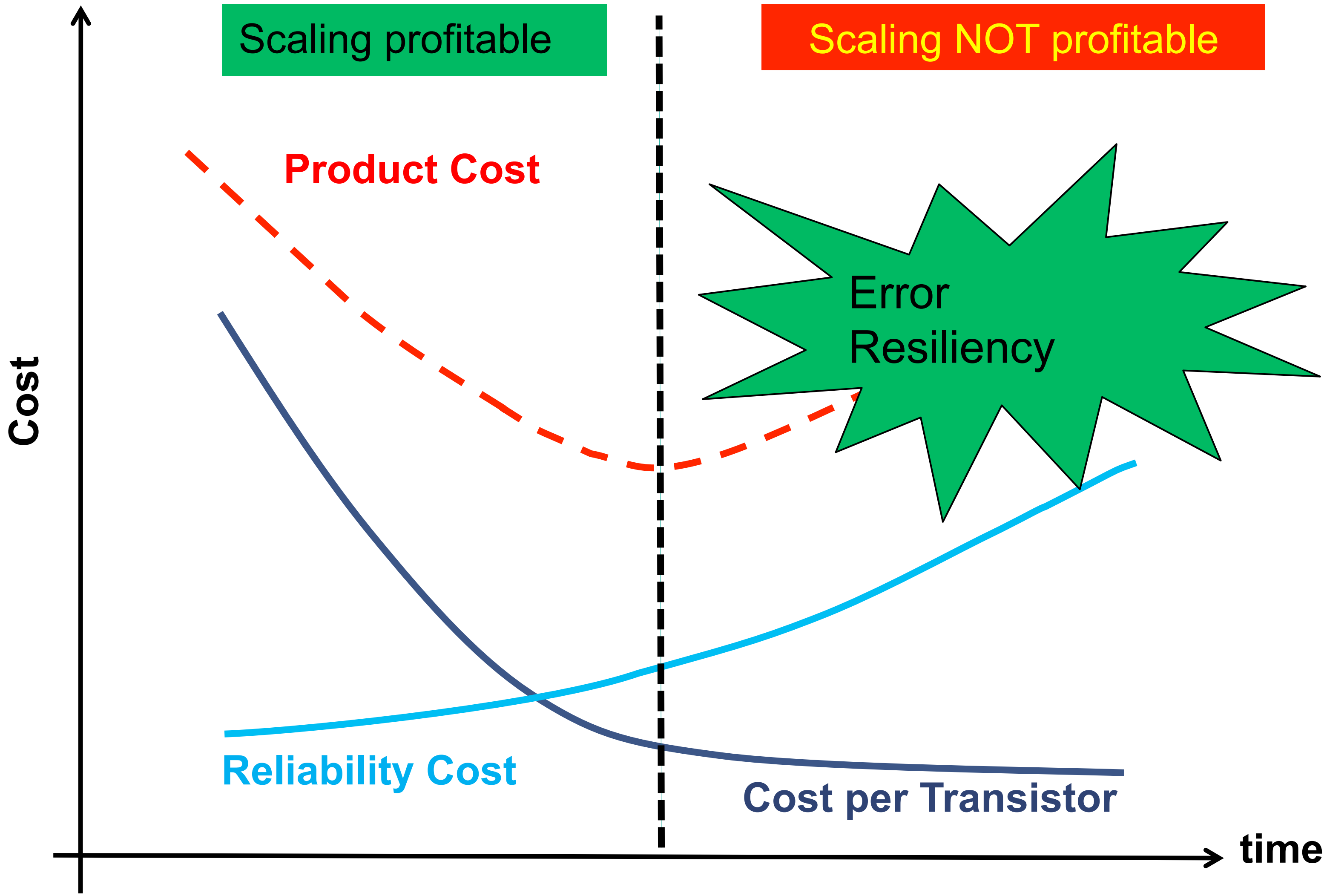
Operation/Observation/Adaptation

Design Methods

Solutions: System Level


Dependable Embedded Software
Dependable Hardware Architectures
Technology Abstraction
Operation/Observation/Adaptation
Design Methods

Goal

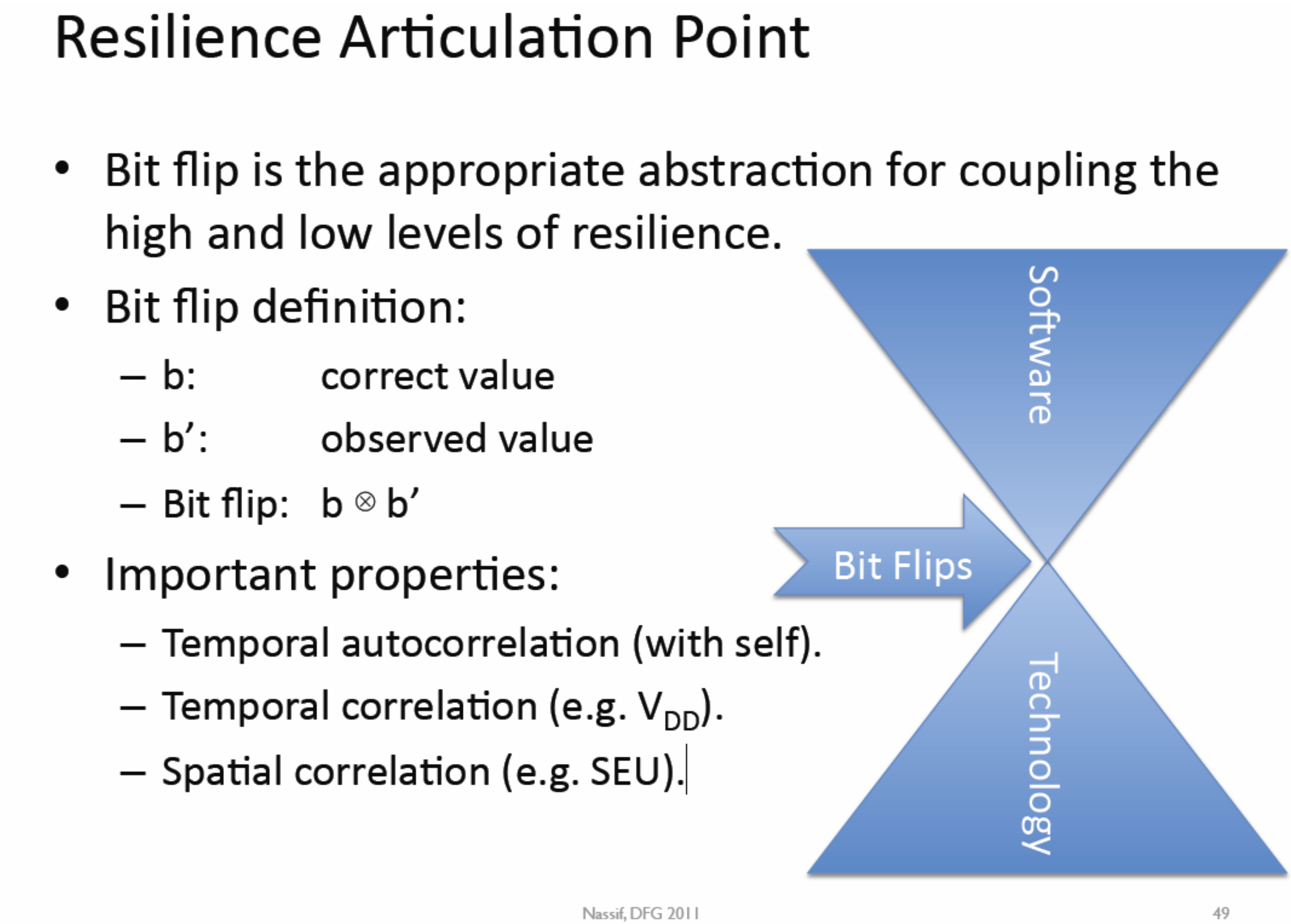

Scaling profitable
Scaling NOT profitable
Product Cost
Error Resiliency
Cost
Reliability Cost
Cost per Transistor
time

Can we address these problems at Software Level ?

```
me("org.hsqldb.
 = "jdbc:hsqldb:
 = DriverManager
n.setAutoCommit(false
 = connection.create
pdate="UPDATE employees
nt.executeUpdate(update);
on.commit();
Exception e) {
rollback();
ace();
```

(Src: paragoninnovations)

… leads to the questions: How does an Error articulate ?



Resilience Articulation Point

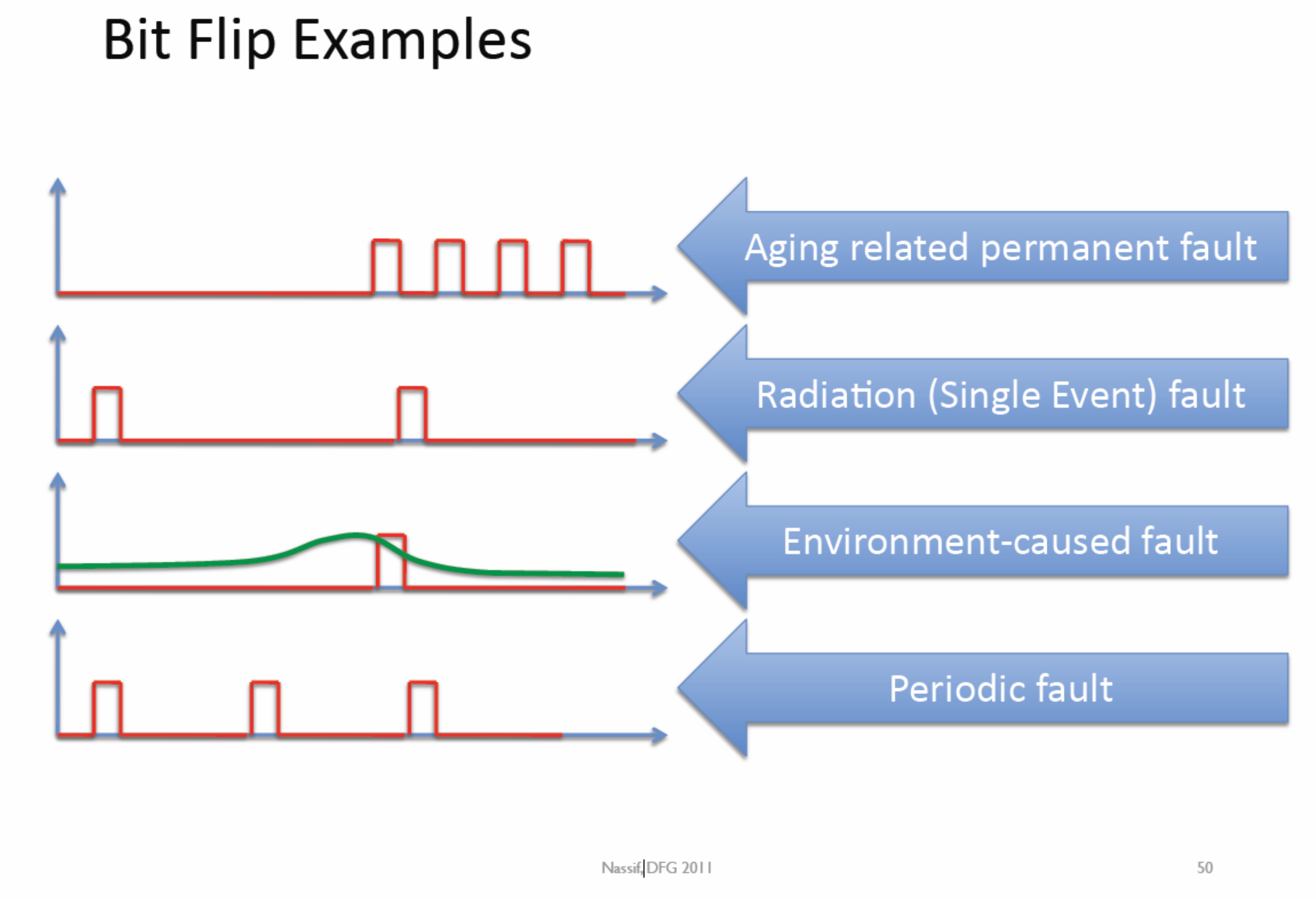
- Bit flip is the appropriate abstraction for coupling the high and low levels of resilience.
- Bit flip definition:
  - b: correct value
  - b’: observed value
  - Bit flip: $b \otimes b'$
- Important properties:
  - Temporal autocorrelation (with self).
  - Temporal correlation (e.g. V_{DD}).
  - Spatial correlation (e.g. SEU).

Nassif, DFG 2011

49

src: Sani Nassif, talk @ SPP 1500 Colloquium in Stuttgart 2011

How does an Error articulate ?



Bit Flip Examples

Aging related permanent fault

Radiation (Single Event) fault

Environment-caused fault

Periodic fault

Nassif|DFG 2011

50

src: Sani Nassif, talk @ SPP 1500 Colloquium in Stuttgart 2011

Improving Reliability at Software Level

Software Techniques

- Redundant instructions, Comparison & Control flow checking instructions, EDDI

Compiler-directed techniques may help increase reliability

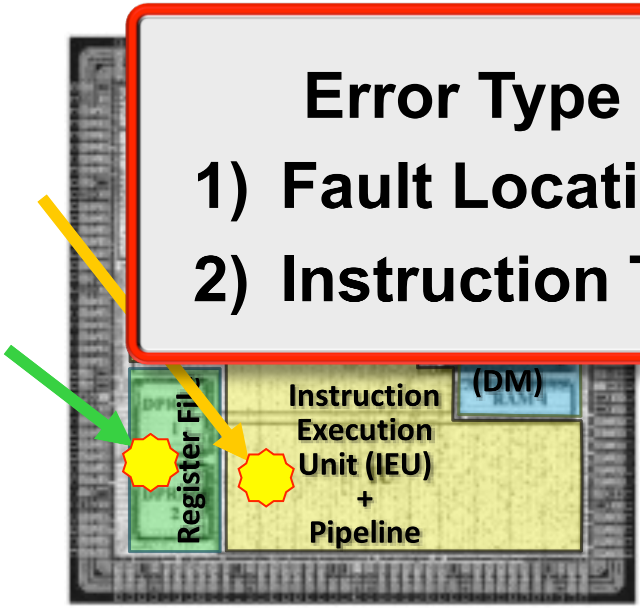
→ Often insufficient reliability estimation and improvement : 2%-9%
→ do not consider the vulnerability of overall processor resources used by different instructions

Not all Soft Errors are of same Criticality

- Soft Error propagation into the Software Layer
- Different impact dependent upon affected component

Error Type at Software Layer depends on

1) **Fault Location**
2) **Instruction Type using different components**



[Photo: Gaisler @ IEEE DSN'02]

File

nd

lue

```
0x194: add   g1, g2, g1
0x198: ld    [g1+(0xc00)], g1
0x19c: xor   g4, g1, g1
0x1a0: xor   i5, g1, g1
```

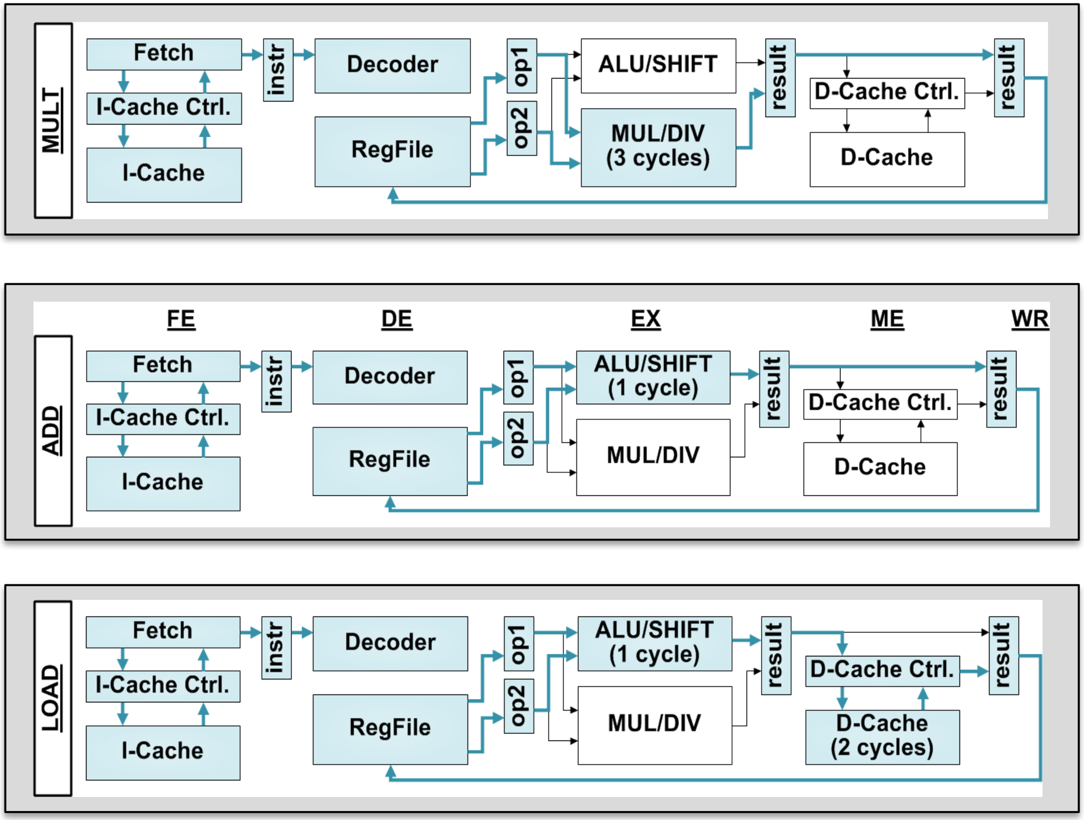
Strike on Instruction Decoder

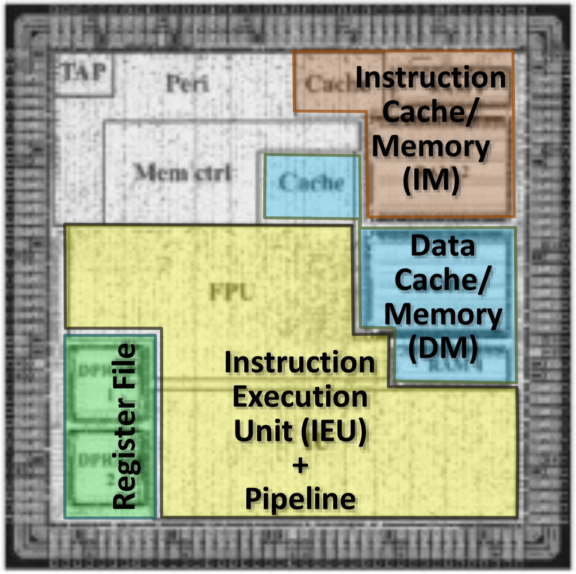
- corrupted opcode
- crash

→ **Not Tolerable?**

Spatial and Temporal Vulnerability

- **Spatial vulnerability**: probability of a fault depending upon the area of specific processor resources used by the instructions
- **Temporal vulnerability**: probability of a fault depending upon the vulnerable periods of an instruction in a certain pipeline stage





[Photo: Gaisler @ IEEE DSN'02]

Reliability Model: Instruction Vulnerability Index

- Individual vulnerability of the instruction *'i'* at component *'c'*

$$IVI_{ic} = \frac{VulnerablePeriod_{ic} * Bits_{Vulnerable-c}}{\sum_{c \in Proc} TotalBits_c}$$

IVI$_{ic}$→Individual vulnerability of instruction i at component **C**

Bits$_{Vulnerable-c}$ →Vulnerable-bits of component *'c'* out of *TotalBits*$_c$

TotalBits$_c$ → architecturally-defined size

- Accumulated vulnerability of instruction during its complete execution in pipeline stages

$$IVI_i = \frac{\sum_{c \in Proc} IVI_{ic} * A_c * P_{fault}(c)}{\sum_{c \in Proc} A_c}$$

i→Instruction

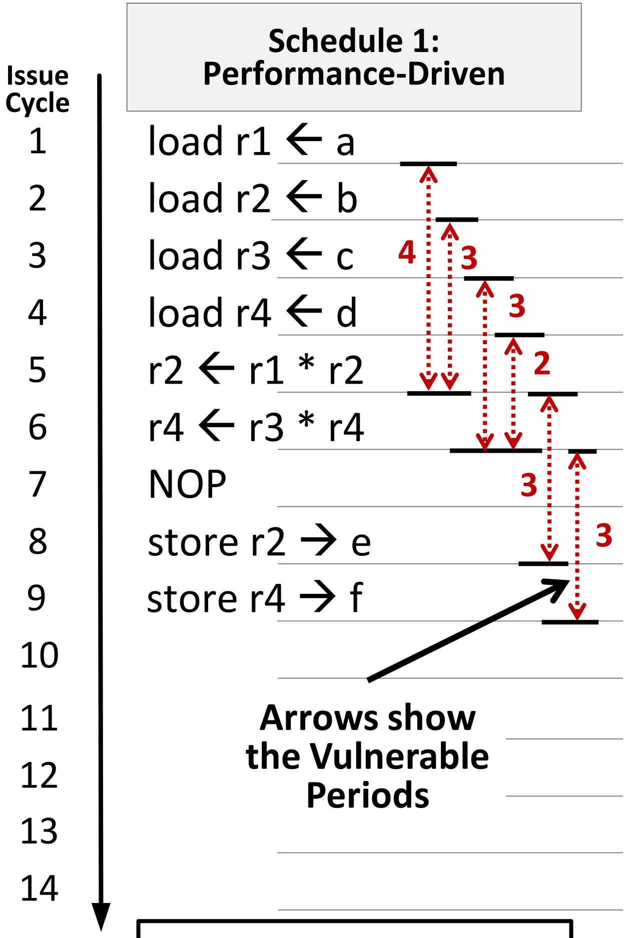
Proc→Processor components

C→Particular processor component

A$_c$→ Area of the component

P$_{fault}$ **(C)**→ Probability of a fault observed at the output of component 'c'

Analyzing Reliability Impact of Instruction Scheduling



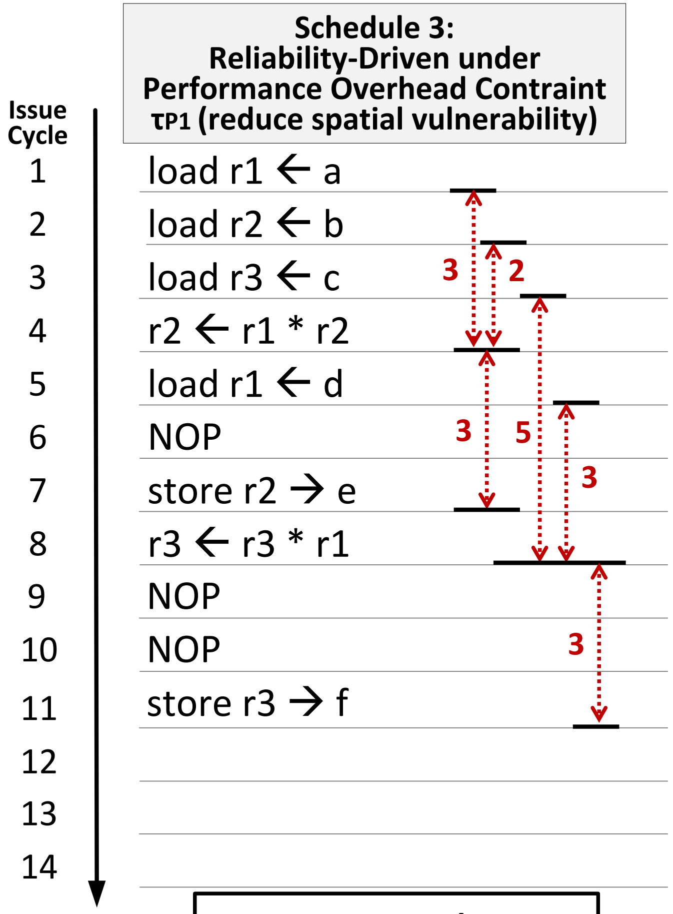
Schedule 1: Performance-Driven

Issue Cycle	
1	load r1 ← a
2	load r2 ← b
3	load r3 ← c
4	load r4 ← d
5	r2 ← r1 * r2
6	r4 ← r3 * r4
7	NOP
8	store r2 → e
9	store r4 → f
10	
11	
12	
13	
14	

Arrows show the Vulnerable Periods

#Reg = 4, #Cycles=9
Vulnerable Periods=18
FVI_{Reg}=6.3%

Analyzing Reliability Impact of Instruction Scheduling



Schedule 3:
Reliability-Driven under
Performance Overhead Contraint
τP1 (reduce spatial vulnerability)

Issue Cycle	
1	load r1 ← a
2	load r2 ← b
3	load r3 ← c
4	r2 ← r1 * r2
5	load r1 ← d
6	NOP
7	store r2 → e
8	r3 ← r3 * r1
9	NOP
10	NOP
11	store r3 → f
12	
13	
14	

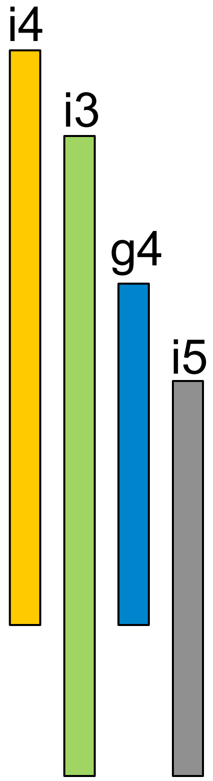
#Reg = 3, #Cycles=11
Vulnerable Periods=19
FVI_{Reg}=5.6%

Example: SAD

Schedule 1

```
128:   ld      [o3+g0],g2
12c:   ld      [o4+g0],g3
130:   and     g2,(0xff),i4
134:   and     g2,o5,g1
138:   sra     g1,(0x8),i3
13c:   and     g2,o7,g1
...
14c:   and     g3,(0xff),g4
150:   and     g3,o5,g1
154:   sra     g1,(0x8),i5
...
164:   subcc   i4,g4,g2
168:   bpos    0x174
...
170:   sub     g4,i4,g2
174:   subcc   i3,i5,g4
178:   bneg,a  0x180
17c:   sub     i5,i3,g4
...
```

Bars denote register vulnerable periods

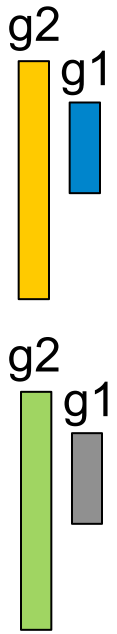


Registers used: 27

Vulnerable period: 45,642

Schedule 2

```
108:   ldub    [o0+g0],g2
10c:   ldub    [o1+g0],g1
110:   sub     g1,g2,g3
114:   subcc   g2,g1,g2
118:   bpos,a  0x120
11c:   mov     g2,g3
120:   add     g3,g4,g4
124:   ldub    [o0+(0x1)],g2
128:   ldub    [o1+(0x1)],g1
12c:   sub     g1,g2,g3
130:   subcc   g2,g1,g2
134:   bpos,a  0x13c
138:   mov     g2,g3
...
```



Bars denote register vulnerable periods

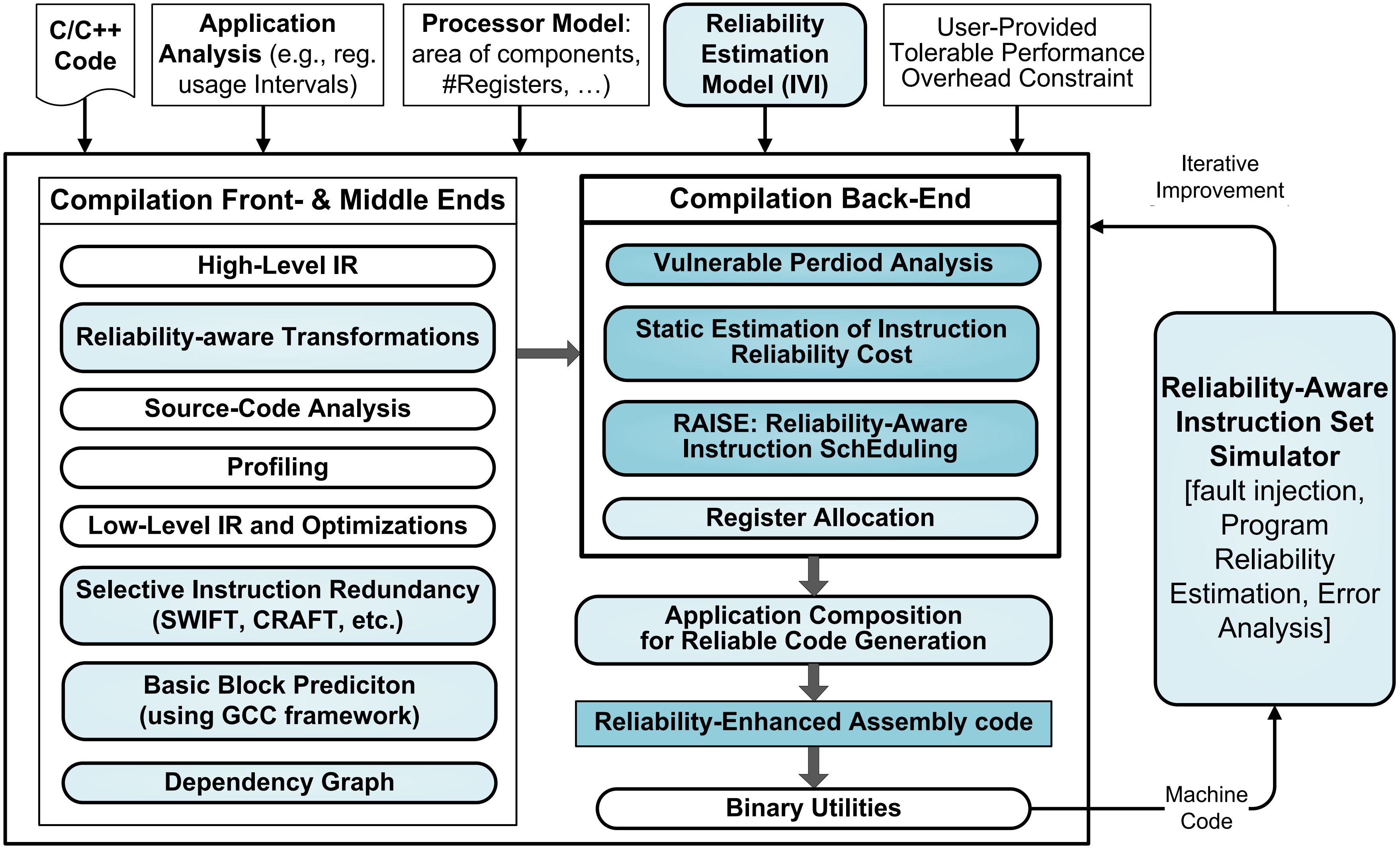
Registers used: 17

Vulnerable period: 31,923

SAD: Analysis

	Schedule 1	Schedule 2	
Performance [cycles]	2124	2241	↗
Registers used [number]	27	17	↘
Vulnerable period [cycles]	45,642	31,923	↘
IVI_{REG}	0.0817	0.0542	↘
IVI_{CC}	0.1205	0.4569	↗
IVI_{ALU}	0.7505	0.4212	↘
IVI_{ALL}	0.1584	0.1367	↘

Compiler Infrastructure for Reliabe Software



C/C++ Code

Application Analysis (e.g., reg. usage Intervals)

Processor Model: area of components, #Registers, …)

Reliability Estimation Model (IVI)

User-Provided Tolerable Performance Overhead Constraint

Compilation Front- & Middle Ends

- High-Level IR
- Reliability-aware Transformations
- Source-Code Analysis
- Profiling
- Low-Level IR and Optimizations
- Selective Instruction Redundancy (SWIFT, CRAFT, etc.)
- Basic Block Prediciton (using GCC framework)
- Dependency Graph

Compilation Back-End

- Vulnerable Perdiod Analysis
- Static Estimation of Instruction Reliability Cost
- RAISE: Reliability-Aware Instruction SchEduling
- Register Allocation

Application Composition for Reliable Code Generation

Reliability-Enhanced Assembly code

Binary Utilities

Machine Code

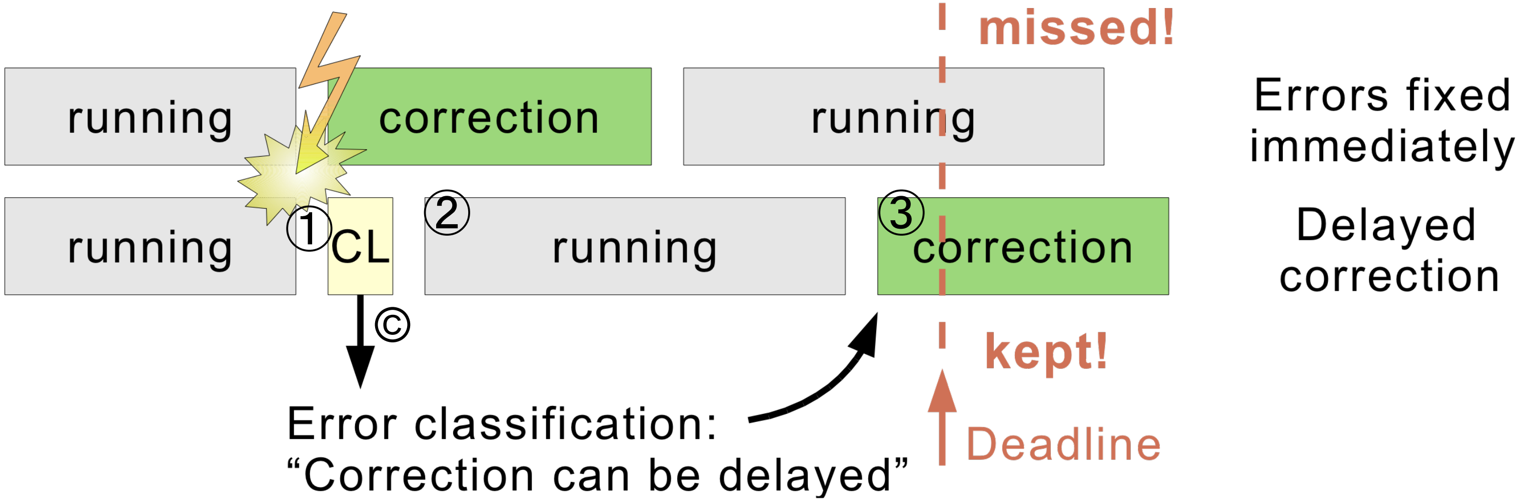
Reliability-Aware Instruction Set Simulator [fault injection, Program Reliability Estimation, Error Analysis]

Iterative Improvement

Software Level

- ... other approaches ...

Application and OS Level



① An error is signaled,

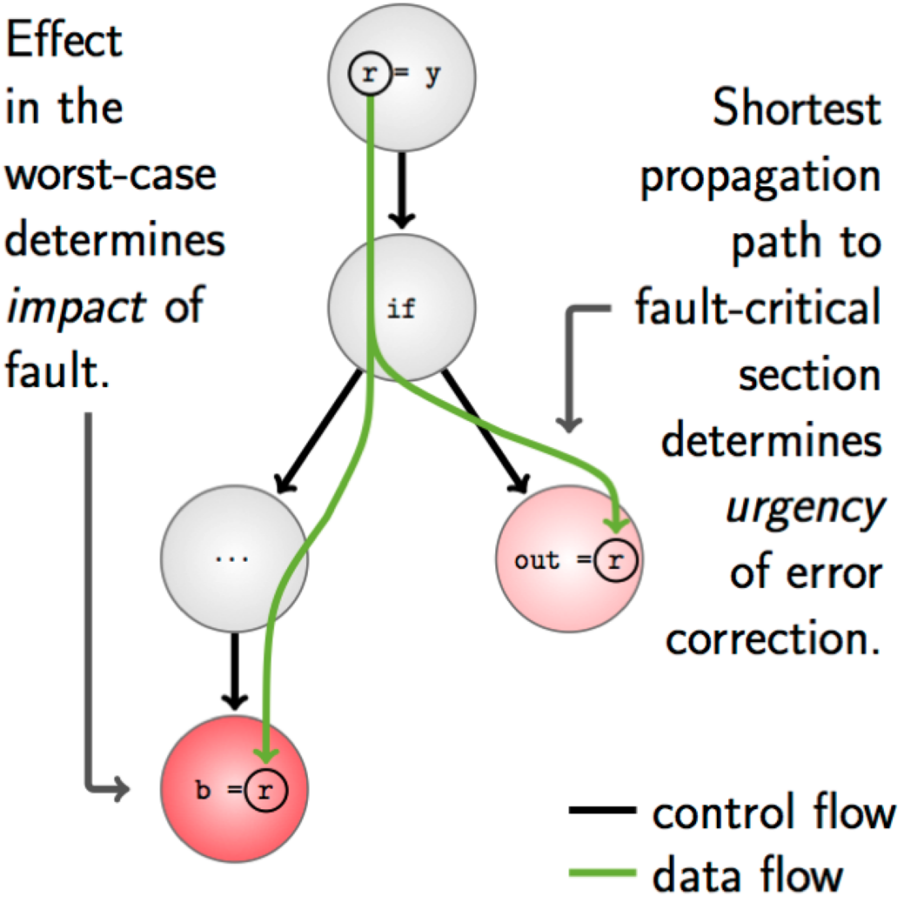
© Error detection executed in a short amount of time, classification decides *if*, *when* and *how* to handle the error,

② Normal system execution continues,

③ If required, error correction takes place *after* timing-critical tasks have finished but *before* error has fatal consequences.

(Source: Marwedel/Engel)

Application analysis provides information on error propagation



Effect in the worst-case determines *impact* of fault.

Shortest propagation path to fault-critical section determines *urgency* of error correction.

▬ control flow

🟩 data flow

(Source: Marwedel/Engel)

- Values assigned to *reliable* variables must also be reliable
- *Unreliable* variables can tolerate errors
- Constraints:
  - Pointers/array indices must be reliable
  - Loop Conditions must be reliable
  - Reliability of if-conditions depends on statements inside body

Conclusion

- Each new technology nodes introduces new dependability problems or makes existing ones worse
- Natural way to fix the problem: technology and device level
- However: there are opportunities at HW architecture and Software …
- Software Level:
  - Software can’t erase the problem of unreliable hardware
  - BUT: it can contribute and relieve the problem
  - Reliability increase basically comes for free (probably some performance overhead
- Conclusion: Technology-induced reliability problems should be addressed at ALL Abstraction Levels!

Thank you for Attention!

Literature

“Reliable software for unreliable hardware: Embedded code generation aiming at reliability” **BEST PAPER AWARD**

In Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2011
Proceedings of the 9th International Conference on
9-14 Oct. 2011, Rehman, Shafique, M. ; Kriebel, F. ; Henkel, J.
Page(s): 237 - 246

-> Paper attached

Reliable Software for Unreliable Hardware: Embedded Code Generation Aiming at Reliability

Semeen Rehman, Muhammad Shafique, Florian Kriebel, Jörg Henkel
Karlsruhe Institute of Technology (KIT), Chair for Embedded Systems, Karlsruhe, Germany
{semeen.rehman, florian.kriebel} @ student.kit.edu; {muhammad.shafique, henkel} @ kit.edu

ABSTRACT

A compilation technique for reliability-aware software transformations is presented. An instruction-level reliability estimation technique quantifies the effects of hardware-level faults at the instruction-level while considering spatial and temporal vulnerabilities. It bridges the gap between hardware – where faults occur according to our fault model – and software (the abstraction level where we aim to increase reliability). For a given tolerable performance overhead, an optimization algorithm compiles an application software with respect to a tradeoff between performance and reliability. Compared to performance-optimized compilation, our method incurs 60%-80% lower application failures, averaged over various fault injection scenarios and fault rates.

Categories and Subject Descriptors

D.3.4 [**Processors**]: *Code Generation, Compilers;* B.8 [**Performance and Reliability**]: *Reliability, Testing, and Fault-Tolerance*

General Terms: Algorithms, Design, Reliability, Performance

Keywords: Reliability, dependability, reliability estimation, instruction vulnerability estimation, reliable software, code generation, embedded systems, technology scaling, reliability-aware software transformations

1. INTRODUCTION AND RELATED WORK

Shrinking feature sizes as a result of technology scaling have led to an increased hardware susceptibility to soft errors (transient faults due to voltage scaling or high energy particles from cosmic rays or packing materials strike on the underlying transistors) [1][2]. Soft errors may cause spurious bit flips in the underlying hardware that may then propagate through the software layer and finally jeopardize software correctness. Extensive reliability-increasing research has been conducted at hardware-level [3][4][5]. Hardware-level soft-error mitigation methods typically incur significant area, performance, and power overhead. Software-level reliability techniques [6]-[13] have evolved to provide further improved system reliability and may be used in addition to hardware techniques.

State-of-the-art approaches on instruction scheduling aim at improving the reliability of register file (by reducing the vulnerable intervals of different register values) or instruction queue (by reducing the residency cycles of vulnerable bits in the instruction queue of super-scalar processors) [17][18][30]. State-of-the-art techniques based on instruction redundancy (EDDI [11], SWIFT [10], CRAFT [14]) provide software reliability by embedding redundant instructions, comparison instructions, and control flow checking instructions. As a result, these techniques incur a significant performance overhead. In order to provide enhanced control flow protection, CRAFT [14] and IVF-based [19] techniques duplicate the *critical instructions*, i.e. instructions that have a relatively high probability to lead to a software failure/crash in case of a soft error, for instance load, store, jump, branches, calls, etc. Therefore, these techniques incur additional >40% performance loss, increased register pressure (i.e. more register usage), and excessive memory overhead (because of instruction and data redundancy) [14]. Furthermore, an increased number of critical instruction executions may lead to excessive rollbacks during recovery because of an increased probability of software failures and fault propagation to/from memory, when a fault occurs in the hardware of the memory pipeline stage [10][14][20].

Besides excessive performance overhead, one of the primary issues of instruction redundancy and scheduling techniques ([10][11][14], [17][18][30]) is that they treat all instructions in the same way. Their software-level reliability estimation models (RVF: Register Vulnerability Factor[1] [18] or PVF: Program Vulnerability Factor[2] [8][9]) do not distinguish between different types of errors in the software caused by the hardware-level faults during the execution of different instructions that use diverse processor components in different pipeline stages (see discussion in Section 2.1 and 5.1). Moreover, RVF [18] and PVF [8][9] are computed without considering the processor architecture. **As a result, software-level reliability techniques of this kind are not very efficient**. Furthermore, state-of-the-art instruction redundancy and scheduling techniques do not consider other compiler stages (like front-/middle-end optimizations) and their impact on the software (data types and structures, etc.) for improving its reliability with reduced performance overhead.

A reduced performance overhead or, alternatively, improved reliability may be achieved by employing reliability-aware software transformations (before the instruction-redundancy and scheduling), which reduce the number of critical instruction executions and modify the instruction profile to increase the software reliability. *To employ such reliability-aware transformations, the gap between the hardware and software needs to be bridged by quantifying the effect of hardware-level faults at the instruction level for software-level reliability estimation, while considering the knowledge of the processor architecture and layout.* Moreover, it is important to understand which instructions lead to which type of error in the application software. The type of error is dependent upon the processor component in which the fault occurs.

1.1 Problem Statement

Traditionally, software transformations have been studied from the perspective of performance or energy optimization [15][21]. The **goal of this work** is to increase the reliability of fault-susceptible hardware/software systems by means of reliability-aware software

CODES+ISSS'11, October 9–14, 2011, Taipei, Taiwan.

[1] RVF considers the register live period as a measure for the reliability.

[2] PVF relates the software reliability to the bits for Architecturally Correct Execution (ACE) in different programmer-visible architectural components (Register File, ALU, etc.), but hides the physical components (e.g., there are 256 physical registers, but 32 are visible to the programmer).

transformations and reliability-guided compiler techniques, which consider the *spatial vulnerability* (different processor components occupy different chip area) and *temporal vulnerability* (different instructions have different execution latencies, instruction dependencies, and vulnerable intervals of the operand values).

In order to perform reliability-aware transformations at source-code level, *an instruction-level reliability estimation technique* is required that quantifies the effect of hardware-level faults at the instruction level to effectively bridge the gap between hardware and software, i.e. the software level techniques consider the knowledge of the underlying hardware and how these faults are manifested and propagate through the software layer.

1.2 Our Novel Contributions and Basic Idea

1) We propose an **Instruction Vulnerability Index (IVI)** for software-level reliability estimation. It jointly considers the effect of faults in different processor components (spatial) during the execution of different instructions (temporal), types of errors, critical instructions, and ACE analysis (i.e. the bits for Architecturally Correct Execution). Based on IVI, the *Function Vulnerability Index* (FVI) and *Application Vulnerability Index* (AVI) are computed for a given function and application software, respectively (see Section 2).

2) Exploiting the knowledge of IVI and FVI, the following **two reliability-aware software transformations** are proposed that transform the code of a given function to aim for higher reliability.

- *FVI-Guided Data Type Optimization* employs different data types for a given data structure, and affects the amount of data to be loaded from and/or stored into the memory along with instructions using this data (see Section 3.1).
- *FVI-Guided Loop Unrolling* determines an 'appropriate' unrolling factor with minimum FVI (see Section 3.2).

3) **Application Composition and Reliable Code Generation:** For a user-provided tolerable performance overhead constraint, an application composition algorithm selects and combines various transformation functions for reliable code generation (see Section 4).

Fig. 1 shows our novel contributions (dark orange boxes) in a reliability-aware compiler.

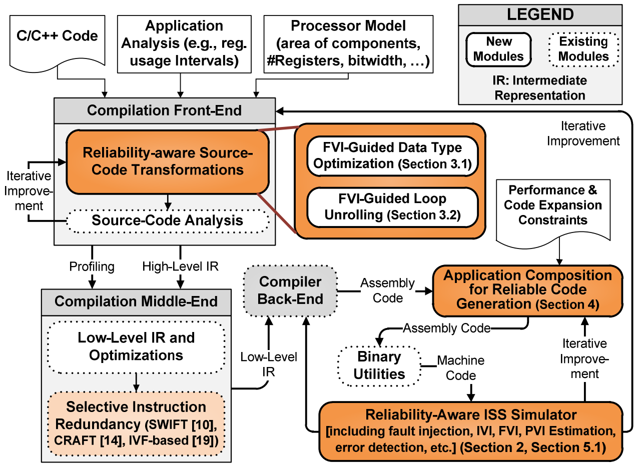

C/C++ Code
Application Analysis (e.g., reg. usage Intervals)
Processor Model (area of components, #Registers, bitwidth, …)
LEGEND
New Modules
Existing Modules
IR: Intermediate Representation
Compilation Front-End
Reliability-aware Source-Code Transformations
FVI-Guided Data Type Optimization (Section 3.1)
FVI-Guided Loop Unrolling (Section 3.2)
Iterative Improvement
Performance & Code Expansion Constraints
Iterative Improvement
Source-Code Analysis
Profiling
High-Level IR
Compilation Middle-End
Low-Level IR and Optimizations
Selective Instruction Redundancy (SWIFT [10], CRAFT [14], IVF-based [19])
Low-Level IR
Compiler Back-End
Assembly Code
Application Composition for Reliable Code Generation (Section 4)
Assembly Code
Binary Utilities
Machine Code
Iterative Improvement
Reliability-Aware ISS Simulator [including fault injection, IVI, FVI, PVI Estimation, error detection, etc.] (Section 2, Section 5.1)

Fig. 1 Reliability-aware compiler flow and our novel contribution (in dark orange filled boxes)

This is the first reliability-aware compiler that employs instruction-level vulnerability quantification (considering spatial and temporal vulnerabilities) and performs reliability-guided software transformations to generate more reliable software code under given set of constraints.

The approach is orthogonal to hardware-level techniques, i.e. traditional hardware techniques may be applied in conjunction with our approach. We believe that each abstraction layer of a system should be involved and contribute its particular advantages to design highly-reliable hardware/software systems.

2. FAULT MODEL AND SOFTWARE-LEVEL RELIABILITY ESTIMATION

We consider single bit-flip transient faults at a given fault rate. It is assumed that faults are evenly distributed/hit throughout the processor area. Accordingly, faults are injected in various processor components (in different pipeline stages) according to the components' area during the execution of different instructions. Their effect on the software level is studied as distinct types of manifested errors (Fig. 2). Since ECC- and parity-protected caches is a well-established practice in various research and industrial projects (IBM [28], AMD [29], [27]), in this work, we consider ECC-protected instruction and data memories. However, the register file is not ECC protected because of high area and power overhead under frequent usage scenarios [10][11][14][18][19], thus vulnerable to transient faults. Note, our proposed model and solution are applicable to both protected and unprotected register files.

As discussed earlier, the motivation of this work is the observation that faults lead to distinct errors (see Fig. 2) during the execution of different instructions and different times and different contexts of execution. Now, we present an analysis to corroborate this motivation and to devise a software-level reliability estimation metric "*Instruction Vulnerability Index* (IVI)" to quantify the vulnerability of different instructions.

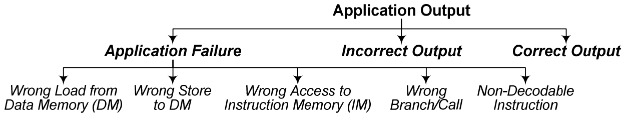


Fig. 2 Different types of manifested errors

2.1 Analyzing the effect of faults during the execution

Fig. 3 illustrates the distribution of different errors for Motion-Compensated Interpolation Filter ('MC-FIR') and Discrete Cosine Transform ('DCT') on an embedded processor subjected to a fault rate of 50 faults/10MCycles (see discussion on fault rates and experimental setup in Section 5). The key observations are as follows:

a). Failures during the instruction-fetch (i.e. *wrong access to instruction memory (IM)*) and instruction-decode stages (i.e. *non-decodable instructions*) occur with the same probability for all instructions, as all instructions use instruction fetch unit and instruction decoder. For instance, if a bit flips in the opcode field of an instruction word, this may lead to a non-decodable instruction.

b). An application/software failure ('abort', 'exception', etc.) may occur due to a wrong branch/call, load/store from/to a wrong location of data memory (DM), or wrong access to the IM, as a result of bit flips in the operands containing the address. This type of failures is typically not tolerable. In contrast, bit flips in the operands of arithmetic instructions (except address generation) may lead to an *incorrect output* error that might be in a user-tolerable range (e.g., faulty pixel distribution in videos) or bare no impact due to control flow.

- Since the probability of failures in the instruction-fetch and instruction-decode stages is the same for all instructions, the key difference of processor components' usage occur in other pipeline stages, like execute, memory, and write-back stages. Therefore, considering the severity of an error as a result of faults in the pipeline stages (other than instruction-fetch and instruction-decode), we categorize load, store, jumps, branches, calls, and address generation instructions as **critical instructions**, while all other (mainly arithmetic and logical) instructions are denoted as **non-critical instructions**. *For a given fault rate, the probability of failures is directly proportional to the number of critical instruction executions*.

c). Fig. 3 shows that in case of 'DCT', *the failures for wrong store to DM* are dominant compared to that in 'MC-FIR', due to more *store* instruction executions. The *failures for wrong load from the DM* happen primarily due to: (i) the bit flips in the operand containing the address (during the memory pipeline stage), or (ii) the bit flips in the Address Generation Unit (AGU) during the address computation (in the execute pipeline stage).

- Different processor components (instruction decoder, ALU, multiplier, AGU, memory controller, etc.) in different pipeline stages exhibit distinct area. Considering that faults are evenly distributed/hit over the surface area, the probability of fault in a processor component is directly proportional to its area. This is denoted as **spatial vulnerability**, which is the probability of a fault during the execution of an instruction w.r.t. to the area of various processor components it uses.
- As discussed above, error types vary depending upon the instruction type and the pipeline stage in which they occur. Therefore, in order to quantify the reliability at the instruction level, *spatial vulnerability* of different instructions needs to be considered.

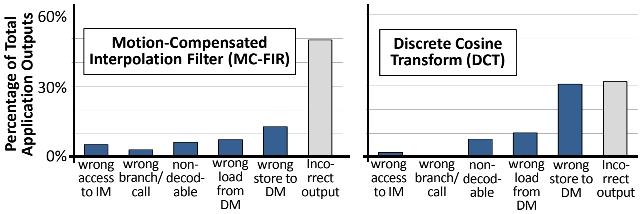


Fig. 3 Analyzing the distribution of different error types

d). *incorrect output* errors are mostly due to bit flips (in ALU, multiplier, etc.) occurring during the execution of arithmetic and logical instructions in the *execute pipeline* stage. Even in this case, the probability of an error in a multiplier is higher compared to an ALU due to its increased spatial vulnerability.

e). Different instructions spend varying amount of time (in terms of cycles) in different pipeline stages. For instance, multiply requires 3 cycles, while an add instruction requires 1 cycle (using adder in the execute stage). This is denoted as **temporal vulnerability**, which is the probability of a fault during the execution of an instruction w.r.t. to the time it spends in various processor components. Furthermore, this longer execution time of instructions results in longer intervals between the variable usages (stored in the register file), which results in an *increased temporal vulnerability* of the register file. Usage of more variables results in an *increased spatial vulnerability*.

f). It is important to consider that not all bits of operand variables are vulnerable for the correct software execution *due to interlayer masking* from microarchitecture-state to the ISA-visible state [8][9] (i.e. a fault in the hardware does not lead to an erroneous application/software output or user-visible error, e.g., due to control flow or due to subsequent instructions). A bit of the correct software execution is deemed *necessary* for Architecturally Correct Execution (ACE-bit). All other bits are unACE-bits. To demonstrate, let us consider the following example.

R0 = R1 & R2;   *R1 = 32'bit x;*   *R2 = 0x0000FFFF;*

A fault may occur in *R1* or *R2*, but not in both. There, a fault in the upper 16 bits of *R1* will not affect the value of *R0*. However, a fault in *R2* will affect the value of *R0*. Therefore, bits 0-15 of *R1* are ACE-bits, while bits 16-31 of *R1* are unACE-bits. In contrast to this, all 32 bits of *R2* are ACE-bits. Therefore, ACE analysis is important for devising the *Instruction Vulnerability Index (IVI)*.

Summarizing, in order to precisely quantify the hardware-level faults at the software level, the *Instruction Vulnerability Index (IVI)* needs to jointly consider the spatial and temporal vulnerability, critical and non-critical instructions, and the ACE analysis.

2.2 Instruction Vulnerability Index

We define the *Instruction Vulnerability Index (IVI)* of an instruction *'i'* as its accumulated vulnerability during its execution in pipeline stages using diverse processor components (*Proc*) while considering their respective area (in terms of vulnerable gates); see Eq. 1.

$$IVI_i = \frac{\sum_{c \in Proc} IVI_{ic} * A_c * P_{fault}(c)}{\sum_{c \in Proc} A_c} \qquad (1)$$

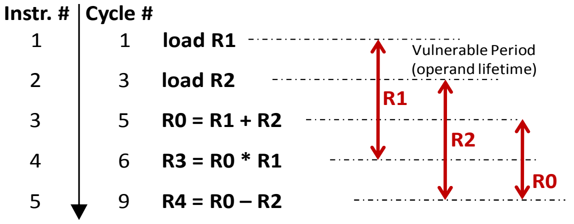
where *'c'* is a particular processor component and A_c is its area in gate equivalents (obtained after synthesis and place & route results). $P_{fault}(c)$ is the probability of a fault observed at the output of the component *'c'*. It is employed to incorporate the *logical masking* effects, i.e. a transient fault in a combinational circuit is not latched by a memory element, as the fault is blocked from affecting the output due to a subsequent gate whose output is only determined by its other inputs [1]. P_{fault}*(register file)* is 100%, while in case of a combinational circuit it depends upon its microarchitecture [1]. IVI_{ic} is the individual vulnerability of the instruction *'i'* at component *'c'* and it is defined as the product of $Bits_{ACE\text{-}c}$ (ACE-bits of a component *'c'* of an architecturally-defined size $TotalBits_c$ in bits) and their vulnerable period (Eq. 2).

$$IVI_{ic} = \frac{VulnerablePeriod_{ic} * Bits_{ACE-c}}{\sum_{c \in Proc} TotalBits_c} \qquad (2)$$

In case of the register file, IVI_{ic} depends upon the number of operands, and Eq. 2 can be modified accordingly (Eq. 3) to obtain IVI_{iReg}.

$$IVI_{iReg} = \frac{\sum_{op \in operands} VulnerablePeriod_{op} * Bits_{ACE-op}}{\sum_{c \in Proc} TotalBits_c} \qquad (3)$$

Fig. 4 demonstrates a case, where two operands have different vulnerable periods, i.e. lifetime of the operand variables in terms of cycles. For an instruction, the vulnerable periods of its operands depend upon the latency of previously executed instructions and instruction dependencies. For example, for the 5th instruction at cycle#9, the vulnerable periods for *R0* and *R2* are 4 and 6 cycles, respectively.



Instr. #	Cycle #	
1	1	load R1
2	3	load R2
3	5	R0 = R1 + R2
4	6	R3 = R0 * R1
5	9	R4 = R0 – R2

VulnerablePeriod = (Cycle of Current Usage – Last Write Cycle)
for i= 5; $VulnerablePeriod_{R0}$ = 4 Cycles; $VulnerablePeriod_{R2}$ = 6 Cycles

Fig. 4 Vulnerable periods of operands

The *vulnerable period* denotes the *temporal vulnerability* and $Bits_{ACE\text{-}c}$ and A_r denote the *spatial vulnerability*. $Bits_{ACE\text{-}c}$ are obtained by performing comprehensive software-level ACE analysis. ACE analysis captures the vulnerable portion of architectural components (without considering fault injection and the underlying microarchitecture) by exploiting the read/write dependencies w.r.t. the register file and instruction dependencies. The bits written in a certain cycle but not read are denoted as UnACE-bits.

Our ACE analysis is similar to the one employed by PVF [8][9]. However, PVF does not distinguish between different types of errors that appear as a result of faulty ACE-bits. Therefore, our IVI metric incorporates the ACE analysis in conjunction with the knowledge of critical/non-critical instructions and varying probabilities of failures and incorrect output in order to quantify the vulnerability index of a function.

2.3 Function and Application Vulnerability Index

As discussed in Section 2.1, faults occurring during the execution of critical instructions typically lead to application failures, which are more severe compared to incorrect output (from the user perspective). Therefore, the *Function Vulnerability Index* (*FVI*) is computed as the weighted average of FVI for critical Instructions (FVI_{CI}) and non-critical Instructions (FVI_{nCI}).

$$\begin{aligned} FVI_{Failures} &= \sum\nolimits_{i \in CI} IVI_i * P_{Failures}(FaultRate) \\ FVI_{IncorrOP} &= \sum\nolimits_{i \in nCI} IVI_i * P_{IncorrOP-nCI}(FaultRate) \\ &\quad + \sum\nolimits_{i \in CI} IVI_i * P_{IncorrOP-CI}(FaultRate) \end{aligned} \tag{4}$$

$$FVI = \frac{\alpha * FVI_{Failures} + \beta * FVI_{IncorrOP}}{\sum I_F} \tag{5}$$

$\sum I_F$ is the number of instructions in a function *'F'*. *'CI'* and *'nCI'* are the critical and non-critical instructions, respectively. $P_{Failures}()$, $P_{IncorrOP-CI}()$, and $P_{IncorrOP-nCI}()$, are the probabilities for failure, incorrect output due to critical instructions, incorrect output due to non-critical instructions, respectively, at a certain fault rate. These probabilities can be obtained by employing fault-injection techniques. Various fault injection techniques are available to inject the faults on different level such as RT-level, ISS-level [24] and software-level [22]. Fault injection analysis at RT-level ([2][23][25]) requires significant development time and a long experimental duration (9.7 simulated instructions per second [25]). For a complete application (like a complete H.264 video encoder with several million instructions), it will require weeks. Alternatively, software-level techniques [22][24] also exist for fault injection analysis. However, these software-level techniques do not consider the knowledge of processor layout with architecture-specific details (area of different components, number and bit-sizes of physical registers, etc.) for fault distribution and analysis. For example, the SymPLFIED [22] approach enumerates transient faults in registers, memory, and computation block of hardware without considering the processor architecture and layout in their machine model, therefore, lacks accuracy and may lead to over- or under-estimation or may even not cover certain fault scenarios (see Section 5.5 for accuracy comparison).

The reliability of a complete application software is quantified using the *Application Vulnerability Index* (AVI); see Eq. 6.

$$AVI = \frac{\sum\limits_{f \in F} \sum\limits_{i=0}^{numExec_f} (T_{fi} * (\alpha * FVI_{Failures} + \beta * FVI_{IncorrOP}))}{\sum\limits_{f \in F} \sum\limits_{i=0}^{numExec_f} T_{fi}} \tag{6}$$

where $numExec_f$ is the number of iterations of the function *'f'* and T_{fi} is the execution latency (in cycles) in its i^{th} iteration.

Our proposed reliability-aware software transformations reduce the FVI and AVI for a given application software by:

- lowering the $P_{Failures}()$, $P_{IncorrOP-CI}()$, and $P_{IncorrOP-nCI}()$ probabilities, achieved by reducing the number of critical instruction executions.
- lowering the IVI_{ic}, achieved by modifying the instruction profile that leads to a different usage pattern of the processor components by means of executing alternative instructions.

3. RELIABILITY-AWARE SOFTWARE TRANSFORMATIONS

The following two reliability-aware software (source-level) transformations are proposed.

1) FVI-Guided Data Type Optimization
2) FVI-Guided Loop Unrolling

3.1 FVI-Guided Data Type Optimization

Data type optimization is a method to transform the data types with smaller bit widths (like 8-bit unsigned char) into the data types with larger bit widths (16-bit short or 32-bit unsigned int) for a given data structure in order to reduce the number of critical instruction executions, while minimizing the FVI.

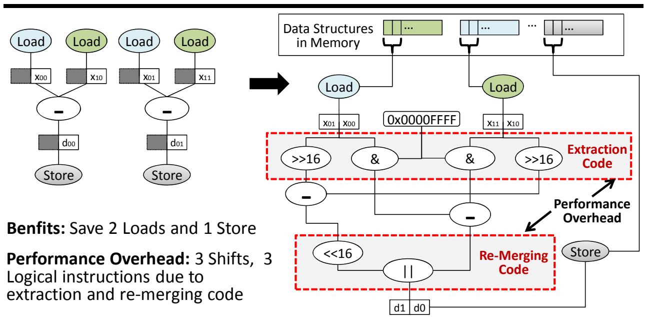
It affects the amount of data to be loaded from and/or stored to the memory. The input/output and the internal data structures become distinct with different data types that impact the instructions executed, thus resulting in a different instruction histogram compared to the original function[3].

Original Code

```
short int ref[16];
short int cand[16];
…
r00=ref[0];
r01=ref[1];
r10=cand[0];
r11=cand[1];
…
d0=r00-r10;
d1=r01-r11;
…
…
```

Transformed Code

```
int ref[4];
int cand[4];
…
r0=ref[0]; r1=cand[0];
r00=r0&0x0000FFFF;
r01=r0>>16;
r10=r1&0x0000FFFF;
r11=r1>>16;
…
d0=r00-r10;
d1=r01-r11;
…
```



Benfits: Save 2 Loads and 1 Store

Performance Overhead: 3 Shifts, 3 Logical instructions due to extraction and re-merging code

Fig. 5 (a) Example code showing data type optimization transformation, (b) Corresponding data flow graphs

Fig. 5 shows an example with original and transformed codes along with their data flow graphs. The original code executes 2x more load/store instructions due to 16-bit data loading into one 32-bit variable (stored in the register file) at a time; see left-side graph in Fig. 5b. In contrast, in the transformed code, two 16-bit data values are loaded into a 32-bit variable in a packed format; right-side graph in Fig. 5b illustrates two loads from two different arrays. The reduced number of executions of load/store instructions results in a lower probability for failures, as discussed in Section 2. Moreover, when using instruction redundancy for fault detection to achieve higher reliability, the incurred performance penalty is lower after deploying the '*data type optimization*' transformation.

[3] In case data types of the input and output parameters are changed, a modification in the function interface is required.

However, this transformation comes with certain side-effects, as shown by the additional *extraction and merging code* in Fig. 5b (highlighted by dashed boxes), which is required to unpack and repack the data values when using a 32-bit RISC processor. Since after unpacking the data, variables and instructions are still in 32-bit format, the overflow of signed values is avoided. Additional instructions for packing and unpacking of data incur a performance penalty in addition to a relatively higher IVI for ALU. Therefore, this overhead needs to be amortized by the IVI reduction due to reduced number of executions of load and store instructions. Load instructions may incur stalls due to cache misses. Therefore, a reduced number of load instruction executions may even amortize the performance overhead of additional extraction and merging code. Still the merging algorithm considers a tolerable performance overhead. Note, in case of VLIW architectures, this transformation may even be better due to the availability of SIMD instructions.

When using data types with even smaller bit-widths, like unsigned char (8-bit, typical in image and video processing applications), the critical instruction executions can be further reduced to lower the probability of failures. However, it might incur a significant performance overhead and code size expansion due to the excessive extraction and merging code for packing/unpacking of data. That is why we propose an algorithm that performs data type optimization for load/store instructions under the constraint of a given tolerable performance overhead ($P\tau$).

```
1.  Input: $\mathcal{G}(V, E)$, $P_\tau$, $FVI_{Orig}$, $P_{Orig}$, DataType
2.  Output: Transformed Function fd        // with reduced $FVI_{failuers}$
3.  BEGIN
4.  $\mathcal{A} \leftarrow$ getAllArrays($\mathcal{G}$);
5.  For all $a \in \mathcal{A}$
6.    list<V> $\mathcal{L} \leftarrow$ getLoads($a$, $\mathcal{G}$);
7.    If (DType= INT) Then
8.      continue;
9.    $FVI_{Best} \leftarrow FVI_{Orig}$;
10.   While $\mathcal{L}$ != Ø {
11.     $\mathcal{G}' \leftarrow \mathcal{G}$
12.     $(\ell_1,\ell_2) \leftarrow$ GetCurrent&NextLoads($\mathcal{L}$) ;
13.     $\ell \leftarrow$ Merge($\ell_1,\ell_2$);
14.     $\mathcal{G}'$.Remove($\ell_1,\ell_2$);   $\mathcal{G}'$.Insert($\ell$);   $\mathcal{G}'$.InsertExtractionCode( );
15.     (FVI, P, Spill) $\leftarrow$ Evaluate($\mathcal{G}'$); // Compile and Execute,
                              and estimate FVI, performance, and check for spilling
16.     If ((P/$P_{Orig}$ – 1)> $P_\tau$) Then             break;
17.     If ((FVI < $FVI_{Best}$) & (!Spill)) Then
18.       $FVI_{Best} \leftarrow$ FVI;      $\mathcal{L}$.Remove($\ell_1,\ell_2$);
19.       $\mathcal{G}$.Remove($\ell_1,\ell_2$); $\mathcal{G}$.Insert($\ell$);
20.       $\mathcal{G}$.InsertExtractionCode( );
21.     End If
22.   End While
23. End For
24. fd $\leftarrow \mathcal{G}$;
25. return fd;
26. END
```

Fig. 6 Algorithm for FVI-guided data type optimization

Fig. 6 presents the pseudo-code for data type optimizations targeting load merging (for store instructions, the procedure is similar).

- **Input**: Graph $\mathcal{G}$ *(V, E)* of the function $\mathcal{F}$, $P\tau$ as the tolerable performance overhead, Data Type, FVI and performance of the original code (FVI_{Orig}, P_{Orig}),
- **Output**: Transformed function *fd* with merged loads and extraction code as a result of the data type optimization

First, all arrays $\mathcal{A}$ are extracted from the graph $\mathcal{G}$ (line 4). Then, for each array $a \in \mathcal{A}$, a list $\mathcal{L}$ of all *load* vertices is obtained from the input graph (line 6). If the data type is integer (32-bit), no merging is performed for array a (line 7). Otherwise, the algorithm iterates until all load vertices are evaluated (lines 10-22). First a temporary copy $\mathcal{G}'$ of the graph $\mathcal{G}$ is created (line 11). Then, two consecutive load vertices are extracted from the load list and merged (line 12, 13). These load vertices are removed from the temporary graph $\mathcal{G}'$ and the merged load vertex is inserted along with the extraction code (line 14). Afterwards, the temporary graph $\mathcal{G}'$ is compiled and simulated to estimate the performance and reliability (FVI) in line 15. In case the performance loss is greater than the tolerable performance, the algorithm returns the currently best Graph (line 16, 24). Otherwise, the FVI is compared to the currently best FVI (line 17). In case of a better solution, the vertices under evaluation are removed from the original graph $\mathcal{G}$ and the merged load vertex is inserted along with the extraction code (lines 18-20).

This algorithm only merges two load vertices in each iteration. Therefore, when optimizing from 8-bit to 32-bit data types, it is invoked two times.

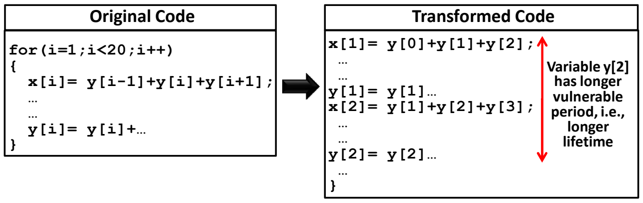
3.2 FVI-Guided Loop Unrolling

Loop unrolling is a method to expand/unroll source code loops by determining an appropriate unrolling factor (among several unrolling options) such that the Function Vulnerability Index (FVI) is minimized, while reducing the number of critical instruction executions.

The *unrolling factor* is defined as the number of loop body replications after unrolling. Loop unrolling techniques have been extensively explored for improving the performance and power consumption while considering side effects like increased software code size, instruction cache overflow, register spilling, etc. [15][16]. However, loop unrolling has not yet been well explored from the reliability perspective.

On one hand loop unrolling has an impact on the reduced number of critical instruction executions such as load/store and branches. On the other hand it may result in an increased *FVI* due to,

a) increased variable lifetime via engaging the same register for storing this variable for a longer time. The variables of the unrolled code are typically kept for a relatively longer time inside the registers until the relevant instructions are executed. Therefore, it increases the *temporal vulnerability* of variables stored in the register file. This effect can be seen in the example of Fig. 7 where the vulnerable period of variable y[2] has been significantly increased. In contrast, the original code reloads y[2], thus reducing the vulnerable period.
b) increased *spatial vulnerability* as more registers are required for storing live variables; Fig. 7 shows that more y[i] data values are alive, thus requiring more registers.


Original Code
Transformed Code
Variable y[2] has longer vulnerable period, i.e., longer lifetime

```
for(i=1;i<20;i++)
{
  x[i]= y[i-1]+y[i]+y[i+1];
  …
  …
  y[i]= y[i]+…
}
```

```
x[1]= y[0]+y[1]+y[2];
 …
 …
y[1]= y[1]…
x[2]= y[1]+y[2]+y[3];
 …
 …
y[2]= y[2]…
 …
}
```

Fig. 7 Example code shows the increased temporal vulnerability of variables as a consequence of loop unrolling

*The challenge in this case is to determine an appropriate **unrolling factor** which is guided by the Function Vulnerability Index (FVI) to jointly optimize for reliability and performance, while considering the relative increase in the code size.*

We cope with these reliability-related concerns of loop unrolling by means of our *FVI-guided Loop Unroller* that determines – for each given loop ℓ of a function 𝓕 – an appropriate unrolling factor by minimizing the:

a) *Function Vulnerability Index (FVI),* considering utilization of different processor components by different instructions, and

b) Performance loss compared to the maximum achievable performance when using a performance-based unrolling,

while avoiding the spilling and incurring a relatively small increase in code size (i.e. number of assembler instructions). Our *FVI-guided Loop Unroller* discards the unrolling factors that cause register spilling[4] (as a consequence of excessive loop unrolling), as it may incur additional critical instructions such as store and then load (thus, an increased probability for failures, as discussed in Section 2) due to the spill code. The goal is to maximize the following *profit* function (Eq. 7).

$$Profit = \frac{\gamma \times (FVI / FVI_{Orig}) - (P / P_{Orig})}{\mu \times (C / C_{Orig})} \qquad (7)$$

(FVI_{Orig}, P_{Orig}, C_{Orig}) and (FVI, P, and C) denote the FVI, performance, and code size (number of assembler instructions) of the original code (i.e. performance-optimized) and the transformed code, respectively. The parameter μ activates or deactivates the normalization effect due to code expansion. In case the instruction cache is protected by ECC or parity ([2][18][27]), μ is set to be C_{Orig}/C, otherwise, it is set to be *one* (i.e. the case of unprotected instruction cache). The optional parameter γ scales up the importance of reliability.

3.2.1 Algorithm of our FVI-Guided Loop Unroller

The proposed *FVI-guided Loop Unroller* (Fig. 8) requires the *loop iteration counts*. This is *known* for fixed-sized and input-invariant loops and *unknown* for variable-sized loops where the loop iterations depend on a variable's value that may change at run time depending upon the input data[5]. For each loop of a given function, the maximum unrolling factor (*maxUnrollFactor*) is then determined as the Greatest Common Divisor of all the corresponding loop iterations due to profiling for varying input data.

A set of *maxUnrollFactors* for all loops of a function 𝓕 is then forwarded as an input to our *FVI-guided Loop Unroller*. Further input parameters are the FVI, performance, and code size of the original function 𝓕 (FVI_{Orig}, P_{Orig}, and C_{Orig}), i.e. with performance-optimized compilation (line 1). Similar to various state-of-the-art loop unrolling approaches (like [15]), this information is computed at the assembly level and made accessible at the source code level through back-annotation. The output is the transformed function *fd* with loop unrolling applied by an FVI-minimizing unrolling factor.

Fig. 8 shows the pseudo-code of the implemented *FVI-guided Loop Unroller*. First, all loops are extracted from 𝓕 and stored in a list 𝓛 (line 4). Afterwards, all loops of the function 𝓕 are processed and an appropriate unrolling factor is determined (lines 5-21). For each loop ℓ, the corresponding maximum unrolling factor *maxUF* is extracted from the input set *maxUnrollFactors* (line 6). Then, for each loop ℓ, the proposed algorithm computes the *profit* (Eq. 7, line 15)[6] for all possible unrolling factors from 1 to the corresponding *maxUnrollFactors* (line 8). A copy of the loop ℓ is created as $ℓ_{temp}$, which is then evaluated for loop unrolling without affecting the original code (line 9). After unrolling the loop $ℓ_{temp}$ (line 10), the function is compiled and simulated to compute the *FVI*, performance *P*, code size *C* (line 11). For FVI estimation, our reliability-aware compiler has access to the architectural features (number and bit-width of registers, ALU, Instruction Word with operands and opcodes, etc.) and their usage profiles/lifetime from simulation (see Section 5.1 for the simulation details). Furthermore, the spilling condition is also checked.

```
1.  Input: Function 𝓕, Set of maxUnrollFactors, $FVI_{Orig}$, $P_{Orig}$, $C_{Orig}$, $\gamma$, μ
2.  Output: Transformed Function fd     // with FVI-guided loop unrolling
3.  BEGIN
4.  list<Loop> 𝓛 ← getLoops(𝓕);
5.  For all ℓ ∈ 𝓛{    // determine FVI-guided unrolling factor for each loop
6.    maxUF = getFactor(ℓ, maxUnrollFactors);
7.    $unRollProfit_{Best}$ ← maxINT;           $uF_{Best}$ ← 1;
8.    For $uF_i$ = 1 to maxUF{
9.      $ℓ_{temp}$ ← ℓ;                    // create a temporary copy of the loop
10.     $𝓕_{uFi}$ ← Unroll(𝓕, $ℓ_{temp}$, $uF_i$);   // Unroll by factor $uF_i$
11.     (FVI, P, C, Spill) ← Evaluate($𝓕_{uFi}$); // Compile and Execute, and
        estimate FVI, performance, code size, and check for spilling
12.     Benefit_FVI ← FVI / $FVI_{Orig}$;   // FVI Improvement
13.     Loss_P ← P / $P_{Orig}$;            // Performance Loss
14.     Loss_C ← C / $C_{Orig}$;            // Code Size Increase
15.     unRollProfit = computeProfit(Benefit_FVI, Loss_P, Loss_C, $\gamma$, μ);
        // Eq. 7
16.     If ((unRollProfit > $unRollProfit_{Best}$) & (!Spill)) Then
17.       $unRollProfit_{Best}$ ← unRollProfit;     $uF_{Best}$ ← $uF_i$;
18.     End If
19.   End For
20.   setBestUnrollFactor(ℓ, $uF_{Best}$);
21. End For
22. // generate the transformed function using the best unroll factors
23. For all ℓ ∈ 𝓛{
24.   $UF_{Best}$ = getBestUnrollFactor(ℓ);
25.   fd ← unRoll (fd, ℓ, $UF_{Best}$);
26. End For
27. return fd;
28. END
```

Fig. 8 Algorithm for FVI-guided loop unrolling

The FVI reduction, performance loss, and code expansion are computed and forwarded as an input for the *profit* calculation (lines 12-14). In case the *profit* of unrolling the loop ℓ (*unRollProfit*) is more than the current best *profit* ($unRollProfit_{Best}$) and in case there is no spilling, the current unrolling factor is set as the best unrolling factor (uF_{Best}) and the best *profit* ($unRollProfit_{Best}$) is updated accordingly. This ensures that – for a given loop ℓ – only that unrolling factor will be selected that provides maximal FVI reduction while incurring relatively small performance loss and code-size increase. As discussed earlier, our approach discards the spilling cases as the spill code increases the number of critical instruction executions which lead to a higher susceptibility of the application software towards failures. After all the unrolling factors (till *maxUF*) are evaluated, the *FVI-guided best unrolling factor* for the loop ℓ is set and the next loop is evaluated.

Once the *FVI-guided best unrolling factors* for all loops are determined, the transformed function *fd* is generated by unrolling all of its loops by their calculated best unrolling factors (lines 22-26). In case of nested loops, the transformed function *fd* is processing iteratively to generate another transformed function *fd2*.

Functions with different software transformations applied exhibit different performance and reliability values (*FVI*, $FVI_{Failure}$,

[4] Note, in embedded processors, the number of physical registers is typically much less compared to high-end microprocessors.

[5] Identifying fixed loops and variable loops is out of the scope of this paper; see [15] for further details on such a static loop analysis.

[6] As discussed in Section 2, we consider protected cache and memory.

$FVI_{IncorrOP}$; see Section 2) due to their distinct instruction profiles that use the underlying processor components in diverse ways These transformed functions are then forwarded to an application composition algorithm that selects and combines (links) various transformed functions.

4. APPLICATION COMPOSITION AND RELIABLE CODE GENERATION

In the following, we explain the algorithm for composing an application binary using the above-discussed transformations.

Input: an application software is composed of *n* kernel functions, where each function undergoes different reliability-aware transformations (Section 3) to obtain a set of transformed functions F_i.

$$A = \{F_1, F_2, ..., F_n\}, \qquad F_i = \{fd_1, fd_2, ..., fd_m\}$$

Each fd_{ij} transformed function represents a tuple with certain properties $\{n_{ij}, FVI_{ij}, \{FVI_k\}_{ij}, P_{ij}\}$, where n_{ij}, FVI_{ij}, P_{ij} are the numbers of assembly instructions, Function Vulnerability Index (FVI, see Section 2), and latency (cycles) of the fd_{ij} transformed function, respectively. $\{FVI_k\}_{ij}$ is the set of separate FVIs for failures and incorrect output (i.e. $FVI_{Failure}$, $FVI_{IncorrOP}$) at a given fault rate.

Output: *C* as a set of chosen transformed functions of an application software

Constraint: user-given tolerable performance overhead (Pτ) and tolerable code expansion (Cτ);

$$\left(\sum_{i\in C} P_{fd_i} + \sum_{i\in C} P_{fd_i \rightarrow fd_{i+1}}\right) / P_{Max} - 1 \leq P\tau \qquad (8)$$

$$\left(\sum_{i\in C} C_{fd_i} + \sum_{i\in C} C_{fd_i \rightarrow fd_{i+1}}\right) / C_{Orig} - 1 \leq C\tau \qquad (9)$$

P_{Max} is the execution time of the application software with performance-optimized compilation, $\sum_{i\in C} P_{fd_i}$ and $\sum_{i\in C} C_{fd_i}$ are the total execution time and total code size (in number of assembly instructions) of all chosen transformed functions, respectively. The terms $\sum_{i\in C} P_{fd_i \rightarrow fd_{i+1}}$ and $\sum_{i\in C} C_{fd_i \rightarrow fd_{i+1}}$ denote the execution time and size of the sequential code between fd_i and fd_{i+1}, i.e. between two consecutive transformed functions. We only consider the kernel functions and not the concatenated calling functions. For each kernel function, only one transformed function (with a certain reliability-aware software transformation) is chosen in a valid solution. The goal of the algorithm is to determine a valid solution that meets the constraints of Eqs. 8 and 9, while minimizing the following optimization goal of Eq. 10.

Optimization Goal: minimize the AVI (Eq. 6); see Eq. 10.

$$min\left(\frac{\sum\limits_{i\in C}\sum\limits_{j=0}^{numExec_{fdi}} (T_{fdij} *(\alpha * fd_i.FVI_{Failures} + \beta * fd_i.FVI_{IncorrOP}))}{\sum\limits_{i\in C}\sum\limits_{j=0}^{numExec_{fdi}} T_{fdij}}\right) \qquad (10)$$

Selection Algorithm: It is a compile-time selection problem that can be solved optimally using a branch and bound algorithm or sub-optimally using a heuristic. Since the search space in our case is small (e.g., less than 20 kernel functions for a complex H.264 video encoder application), we employ a branch and bound algorithm to traverse the overall design space.

5. EVALUATION AND RESULTS

5.1 Experimental Setup

Fig. 9 shows our reliability analysis and estimation, and simulation framework based on an Instruction Set Simulator (ISS); see experimental setup in Table I. A fault rate (in #faults/10MCycles) is computed from the neutron flux (determined using the geographical location and altitude where the device will be used [26]) and fault probability, processor layout, and the processor frequency. We used 10, 50, and 100 faults/10MCycles, which conforms to the test conditions opted by prominent related work [12][32] and as such eases comparison.

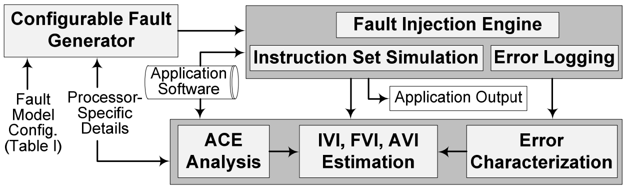


Fig. 9 ISS-based reliability analysis and estimation framework with integrated processor-aware fault injection

A fault generator that generates different fault scenarios considering *fault models*, *fault rates*, and faults in different processor components (e.g., register file, PC, Instruction Word (IW), ALU, Multiplier, etc.) is employed. The number of injected faults per component is determined by the component area (in gate equivalents, or percentage of total processor area, which is obtained after RTL synthesis and place & route results) in order to incorporate spatial vulnerability. For example, more faults are injected in a multiplier/divider compared to the ALU according to the areas these components occupy on the chip.

Table I: Different parameters for fault generator

Parameter	Description	Proper-ties/Values
Distribution	Distribution models for fault generation	random
Bit Flips	Min/Max number of bits flipped	1/1, 1/2, …
Fault Probability	Probability that strike becomes a fault [1]	10%-100%
Fault Location	List of target processor components	Register file, PC, IW, etc.
Processor Layout/Area	Size of the complete target device	in gate equivalents or mm² [27]
Component Area	Area of different processor components given as percentage of processor area	0%-100%
Place and Altitude	City and altitude at which the device is used to determine the flux rate (N_{Flux})	Oslo, 1- 20km
Frequency	Operating frequency of the processor	50, 100 MHz

The faults in various processor components are modeled at the ISS level considering the spatial and temporal vulnerabilities. Fig. 10 demonstrates the spatial and temporal vulnerabilities of Add, Multiply, and Load instructions in different pipeline stages, using an abstract 5-stage integer unit pipeline of the Leon 2 processor, where the used components are denoted as (light blue) filled boxes. It is noteworthy that Add and Multiply instructions are not vulnerable in the memory stage. In contrast, a load instruction is vulnerable in the memory stage, too. The vulnerability of load/store instructions in the execute stage is primarily due to the address calculation. Fig. 10 illustrates that the spatial vulnerability of the load instruction is higher compared to the add instruction due to the usage of more processor components. Furthermore, the probability of injecting a fault during the multiply instruction is higher compared to that in ALU due to the higher temporal vulnerability of a multiplier (due to longer execution).

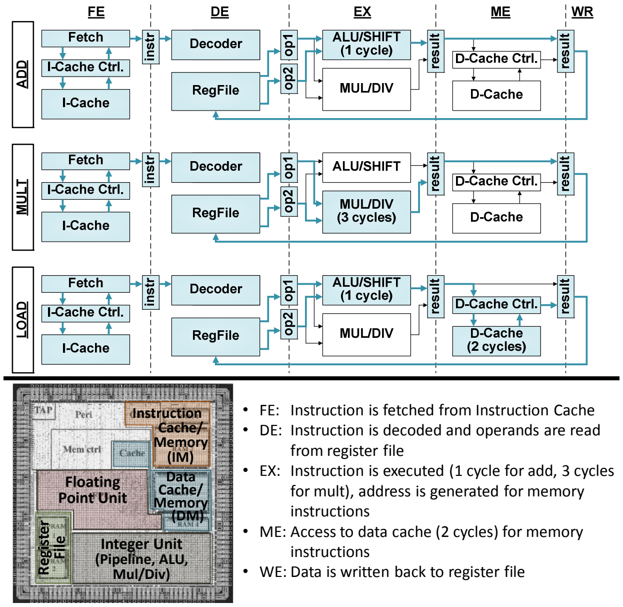
Considering this notion of spatial and temporal vulnerability, the faults and their impacts in different processor components are modeled at the ISS level (see detailed modeling procedure in Table II). For example, a fault in the instruction decoder or in the instruction word is modeled as corrupting one/multiple fields of the instruction word in the ISS that results in a wrong opcode or

Processor Components	Area (Leon 2) [27]	Fault Symptom	Modeling Procedure	Fault Impact
Instruction Fetch & Decode + Instruction Word (IW), (Sparcv8, 5-stage pipeline)	Pipeline and Integer Unit (0.86 mm²)	one/multiple fields of an instruction word are corrupted	opcode field(s) is corrupted	wrong instruction is executed, instruction format is changed, instruction is not decodable
			source/destination register field(s) is corrupted	data is fetched from/written to wrong register(s)
			immediate value is corrupted	wrong input value for calculation
Program Counter (PC), Next Program Counter (NPC)	Floating Point Unit (0.86 mm²)	wrong instruction(s) are executed	PC is corrupted	single instruction is fetched from the wrong location/no access to the designated region
			NPC is corrupted	multiple instructions are fetched from the wrong location/no access to the designated region
Integer Execution Unit (IEU) and Floating Point Unit (FPU)		result of the Execution units are corrupted	sources (input values) of the Execution Units are corrupted	wrong result because of incorrect source register content/wrong computation
			destination (output value) of the Execution Units is corrupted	wrong result because of incorrect destination register content/wrong computation
Register File (windowed, 264x32 bit)	0.19 mm²	data in the register file is corrupted	register in current window is corrupted	wrong content is fetched if window does not move, corrupting source operands
			register not in current window is corrupted	wrong content is fetched when window is moved, corrupting source operands
Instruction Memory (IM) + Data Memory (DM), 16 Kbyte	2.59 mm²	data in the caches is corrupted	corrupted data	load instruction fetches incorrect content
			corrupted instruction	same impact as fault in IW
Others (peripheral units, …)	0.45 mm²	not simulated		

Table II: Modeling faults in different processor components at the ISS-level; as an example for the case of Leon 2

wrong operand. Furthermore, a fault in the data bus or in the cache/memory controller has the same effect from the software perspective as it manifests as a wrongly loaded value. In the register file, the fault is retained until it is overwritten.

Following the modeling procedure, the fault injection engine then injects the faults during the execution of the application software. Note, if a fault is injected into the multiplier while an add instruction is being executed, it will have no effect on the application software output. A target processor component for fault injection is *randomly* selected.



- FE: Instruction is fetched from Instruction Cache
- DE: Instruction is decoded and operands are read from register file
- EX: Instruction is executed (1 cycle for add, 3 cycles for mult), address is generated for memory instructions
- ME: Access to data cache (2 cycles) for memory instructions
- WE: Data is written back to register file

Fig. 10 Spatial and temporal vulnerabilities: different instructions using diverse processor components (with distinct area, see layout of Leon 2 [27]) in pipeline stages

In order to provide a fair comparison, the original code is used with all basic compiler optimizations. The tolerable performance overhead (Pτ) and tolerable code expansion (Cτ) are given as 5%. For evaluation, we have benchmarked an entire H.264 video encoder [31] as it exhibits various compute-intensive functional blocks (e.g., 'SAD', 'SATD', 'DCT', 'HT', 'MC-FIR', 'IPRED') with diverse computational properties, thus providing a representative challenge.

5.2 Discussion on Transformation Results

Fig. 11 compares the error distribution of a performance-optimized and our reliability optimized functions for three different fault rates. Significant improvements are observed for 'DCT', 'HT', 'MC-FIR', 'IPRED', and 'SATD' because of considerable reductions in the number of critical instruction executions and vulnerable periods (>20x). This effect is also visible in Fig. 12 in terms of up to 92% (on average 64%) reduction in FVI. For 'SAD', the benefit of 50% reduced critical instruction executions is balanced by 9% increased arithmetic instructions, that lead to an increased number of failures due to *'non-decodable'* instructions as a result of an increased vulnerability of instructions in the *instruction-decode* stage.

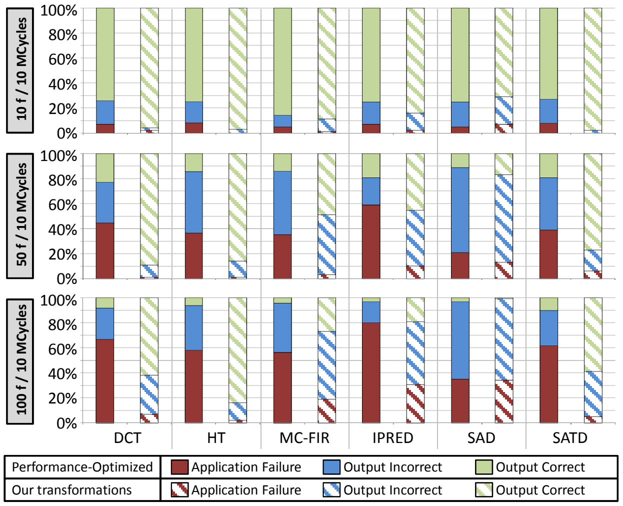


Fig. 11 Comparing the error distribution of performance-optimized and our reliability-optimized functions

The detailed error distribution in Fig. 13 shows that the major reduction in application failures for 'SATD' comes from the reduced number of *'wrong load from DM'* and *'wrong store to DM'* failures. The *'Incorrect Output'* cases are reduced due to reduced vulnerability of an ALU and reduction in the vulnerable periods.

Fig. 11 shows that our transformations provide on average 63.8%, 81.9%, and 67.8% reduced application failures for 10, 50, 100 faults/10MCycles, respectively. Furthermore, our transformations provide on average 43.7%, 16.2%, and -38.6% reduced (negative value denotes an increase) incorrect output for 10, 50, 100 faults/10MCycles, respectively. Note, the percentage of incorrect output is increasing when using our transformations at higher fault rates. This is due to the fact that our transformations prioritize reducing the application failures that are not tolerable. To further increase the reliability, instruction redundancy techniques may be deployed. In the following, we will demonstrate the benefit of our transformations when employed in conjunction with state-of-the-art instruction redundancy techniques, i.e. EDDI [11], SWIFT [10], CRAFT [14].

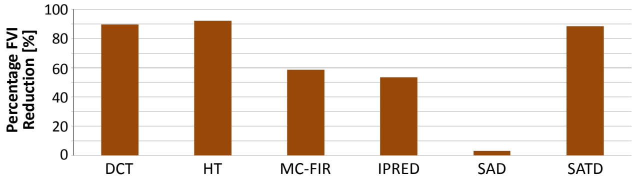


Fig. 12 FVI reductions of our reliability-optimized functions

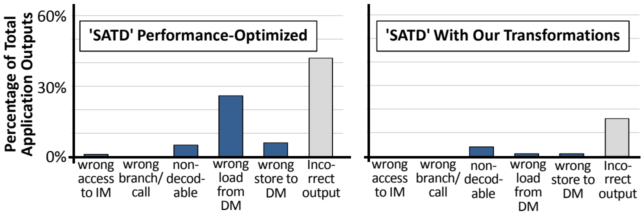


Fig. 13 Comparing the distribution of different error types for performance-optimized and our reliability-optimized SATD

5.3 Applying our Transformations with Instruction-Redundancy Techniques

State-of-the-art instruction-redundancy schemes protect critical instructions and control flow (like load, store, jump/branches, etc.) using several additional compare and branch instructions in the original code. Our proposed transformations aim at reducing the number of critical instruction executions, thus reducing the overhead of additional instructions for protection.

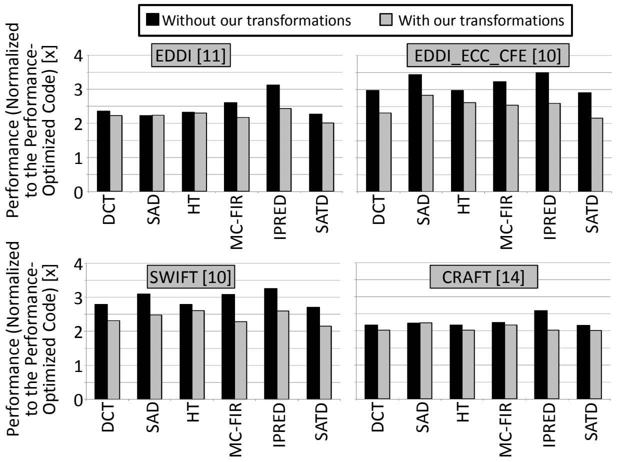


Fig. 14 Comparing instruction-redundancy techniques with and without our proposed software transformations

Fig. 14 shows the performance of instruction redundancy techniques *with* and *without* our transformations normalized to the performance-optimized code. It demonstrates that employing our transformations in conjunction with state-of-the-art instruction-redundancy based techniques (EDDI [11], SWIFT [10], CRAFT [14]) provides up to 39.5% (average 21.8%) lower performance penalty compared to when not applying our transformations. One of the primary reasons for this reduced penalty is the reduction in branches and store instructions that are used as synchronization points for inserting the check instructions in SWIFT [10] and CRAFT [14]. In application softwares with extensive load/store instructions (like DCT, IPRED, SATD, and MC-FIR), instruction redundancy techniques *with* our transformations incur significantly reduced performance overhead compared to instruction redundancy *without* our transformations. Note, this paper does not introduce any new instruction redundancy technique; rather we demonstrate how our transformations may help in reducing the performance penalty of existing instruction redundancy techniques (like SWIFT [10], CRAFT [14]).

5.4 Discussion on Loop Unrolling Factors

Fig. 15 presents the detailed evaluation of our *FVI-guided Loop Unroller* showing different unrolling possibilities and the FVI-aware selected unrolling factor for the 'SATD' function. Fig. 15 shows that the IVI for register file in case of the unrolling factor 8 (FD_{LU8}) is increased by 45% (5.63% → 8.20%) compared to IVI of the unrolling factor 4 (FD_{LU4}), which is mainly due to the usage of 11 more registers (31 vs. 20). Due to the complete unrolling, there are no loop test (jump/branch) instructions in FD_{LU8}. However, the sequence of consecutive arithmetic instructions results in an increased IVI for the ALU. Moreover, since almost all the instructions are arithmetic instructions, ALU is vulnerable all the time.

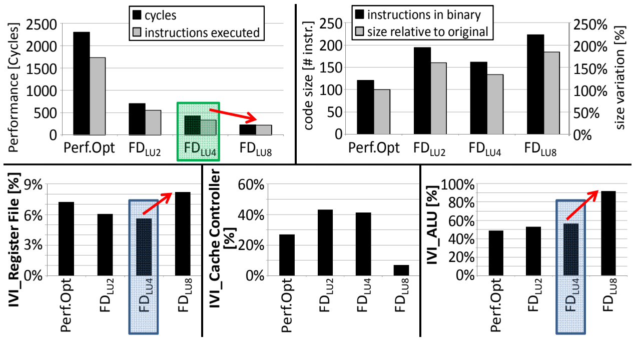


Fig. 15 Effect of loop unrolling on reliability and performance

Although FD_{LU8} improves the performance from 427 cycles to 231 compared to FD_{LU4}, there is a significant increase in the IVI for the ALU and register file components (exhibit higher spatial vulnerability compared to the cache controller), which leads to a relatively lower reliability. Therefore, our *FVI-guided Loop Unroller* chooses 4 as reliability-wise better unrolling factor the 'SATD' function, which provides 45% IVI reduction at the cost of 1.8x performance loss.

5.5 Evaluation of Simulation-/Compilation-Times and Reliability Estimation Accuracy

The reliability analysis and estimation was performed using a 24-core (2.4 GHz) Opteron processor 8431 with 64 GB memory. The average runtime of our reliability analysis and simulation is $72x10^3$ SIPS (simulated instructions per second) with extensive error logging (40MB/MCycles). The simulation runtime of a state-of-the-art software-level reliability analysis technique SymPLFIED [22] is 15.2 SIPS. It shows that our methodology provides significant simulation runtime improvement (>4K times) compared to SymPLFIED [22], which is mainly due to the extensive model computation in SymPLFIED.

Fig. 16 illustrates the comparison of **our reliability estimation accuracy with SymPLFIED [22]**. In case of SymPLFIED, the number of application software *crashes due to wrong access to Instruction Memory* increases significantly. This is due to an increased number of faults in the PC. The main reason is the ignorance of processor layout with several architecture-specific features in SymPLFIED's machine model. Therefore, the percentage fault in PC increases from 0.1% to 7.1%, which leads to an average 27% over-estimation of application failures (for 100f/10MCycles). The comparison in Fig. 16 demonstrates that when using the SymPLFIED technique, the probabilities for failure and incorrect output are over-estimated, which lead to an inaccurate FVI estimation (see Section 2). *It thereby demonstrates the improved accuracy of our reliability analysis*.

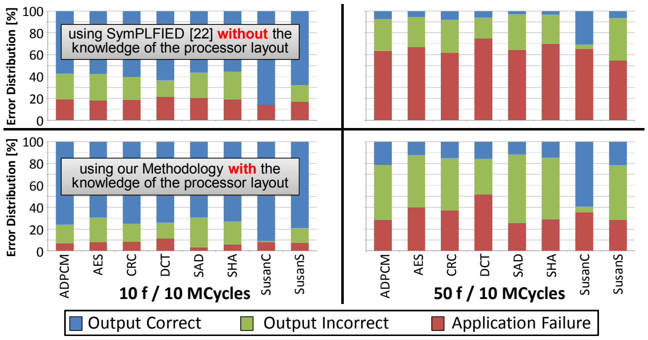


Fig. 16 Detailed error characterization in different applications using our methodology and SymPLFIED [22]

Compared to the performance-optimized compilation, our methodology for reliability-aware compilation suffers from a significant compilation-time overhead due to reliability analysis and estimation (which is still >4K times faster compared to state-of-the-art) and iterative evaluations. It instigates the need to explore intelligent reliability-aware Back-Annotation techniques.

6. CONCLUSION

A novel compilation technique for reliability-aware software transformations is proposed along with an instruction-level vulnerability estimation method. Compared to performance-optimized compilation, our new compilation technique incurs 60%-80% lower application failures, averaged over various fault injection scenarios and fault rates, while reducing an application's vulnerability by avg. 64% compared to performance-optimized compilation.

Our work demonstrates that software-level techniques can significantly contribute towards reliable hardware/software systems. We believe that both software and hardware abstraction layers of a system should be involved and contribute its particular advantages towards highly-reliable hardware/software systems.

7. REFERENCES

[1] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," IEEE TDMR, vol. 5, no. 3, pp. 305-316, 2005.

[2] P. Giacinto et al., "An experimental Study of Soft Error in Microprocessors", MICRO, pp. 30-39, 2005.

[3] R. Vadlamani et al.,"Multicore soft error rate stabilization using adaptive dual modular redundancy", DATE, pp. 27-32, 2010.

[4] D. Ernst et al., "Razor: circuit-level correction of timing errors for low-power operation," IEEE MICRO, vol. 24, no. 3, pp. 10-20, 2004.

[5] S. S. Mukherjee, et al., "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor", MICRO, pp. 29-40, 2003.

[6] R. Venkatasubramanianw et al., "Low cost on-line fault detection using control flow assertions". IEEE IOLTS, pp.137–143, 2003.

[7] P. P. Shirvani et al., "Software implemented EDAC protection against SEUs". IEEE Transactions on Reliability, vol. 49, pp. 273–284, 2000.

[8] V. Sridharan, "Introducing Abstraction to Vulnerability Analysis", Ph.D. Thesis, March 2010.

[9] V. Sridharan et al., "Eliminating Micro-architectural Dependency from Architectural Vulnerability", HPCA, pp. 117-128, 2009.

[10] G. A. Reis et al., "SWIFT: Software Implemented Fault Tolerance", IEEE CGO, pp. 243-254, 2005.

[11] N. Oh et al., "Error detection by duplicated instructions in super-scalar processors", IEEE Transaction on Reliability, vol. 51, no. 1, pp. 63-75, 2002.

[12] J. Hu et al., "In-Register Duplication: Exploiting Narrow-Width Value for Improving Register File Reliability," DSN, pp. 281-290, 2006.

[13] J. S. Hu et al., "Compiler-Directed Instruction Duplication for Soft Error Detection," DATE, vol.2, pp. 1056-1057, 2005

[14] G. A. Reis et al., "Software controlled fault tolerance," ACM TACO, vol. 2, pp. 366-396, 2005.

[15] P. Lokuciejewski et al., "Combining Worst-Case Timing Models, Loop Unrolling, and Static Loop Analysis for WCET Minimization," ECRTS, pp. 35-44, 2009.

[16] V. Sarkar, "Optimized Unrolling of Nested Loops", International Journal on Parallel Programing, 29(5):545–581, 2001.

[17] J. Lee et al., "Compiler approach for reducing soft errors in register file", IEEE LCTES, pp. 41-49, 2009.

[18] J. Yan et al., "Compiler guided register reliability improvement against soft errors," IEEE EMSOFT, pp. 203-209, 2005.

[19] D. Borodin et al., "Protected Redundancy Overhead Reduction Using Instruction Vulnerability Factor," IEEE CF, pp. 319-326, 2010.

[20] U. Schiffel et al., "Software-Implemented Hardware Error Detection: Costs and Gains," IEEE DEPEND, pp. 51-57, 2010.

[21] C. Lee et al., "Compiler optimization on instruction scheduling for low power," IEEE ISSS, pp. 55-60, 2000.

[22] K. Pattabiraman et al., "SymPLFIED: Symbolic program-level fault injection and error detection framework", DSN, pp. 472-481, 2008.

[23] H. Ziade et al., "A Survey on Fault Injection Techniques", IAJIT, vol. 1, no. 2, pp. 171-186, 2004.

[24] R. Velazco et al., "Injecting Bit Flip Faults by Means of a purely Software Approach: a Case Studied", IEEE DFT, pp. 108-116, 2002.

[25] M. Rebaudengo, M. S. Reorda, M. Violante, "Analysis of SEU effects in a pipelined processor", IEEE IOLTW, pp.112-116, 2002.

[26] Flux calculator: www.seutest.com/cgi-bin/FluxCalculator.cgi.

[27] J. Gaisler, "A portable and fault-tolerant microprocessor based on the SPARC v8 architecture", DSN, pp. 409-415, 2002.

[28] IBM® XIV® Storage System cache: http://publib.boulder.ibm.com/infocenter/ibmxiv/r2/index.jsp.

[29] AMD Phenom^{TM} II Processor Product Data Sheet 2010.

[30] X. Fu, W. Zhang, T. Li, J. Fortes, "Optimizing Issue Queue Reliability to Soft Errors on Simultaneous Multithreaded Architectures", International Conference on Parallel Processing, pp. 190-197, 2008.

[31] H.264 Codec: http://iphome.hhi.de/suehring/tml/index.htm

[32] L. Lin et al., "Soft error and energy consumption interactions: a data cache perspective", ISLPED, pp. 132-137, 2004.