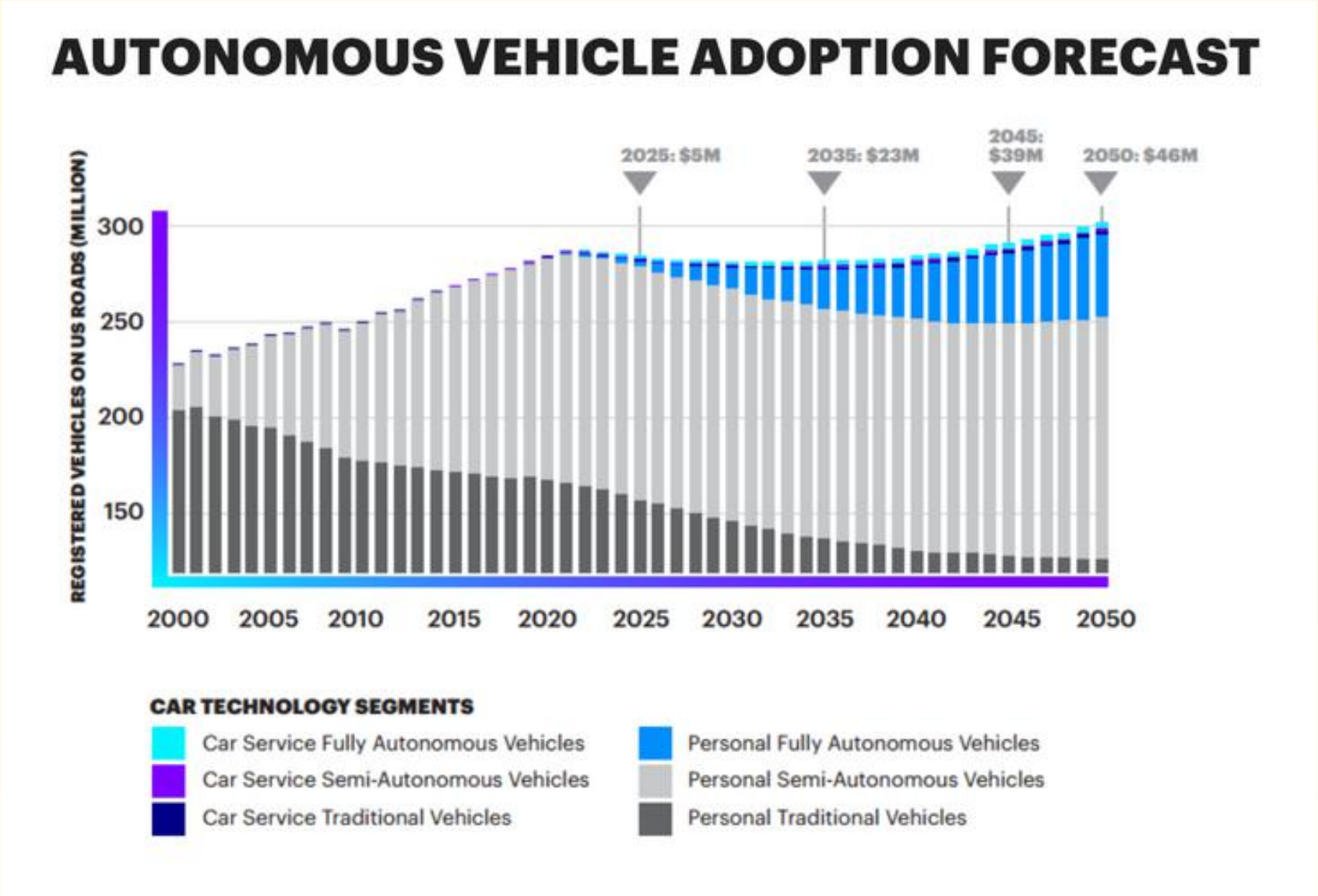
Resilient Computing – Imperative for Autonomous Systems

Jacob Abraham

The University of Texas at Austin

VLSI Design Conference, Bengaluru
January 5, 2020

Growth in Autonomous Vehicles



Source: Accenture

Constraints on Embedded Systems

Myriad of Intelligent systems

- Cost, power consumption constraints
- In critical applications, **Safety and Resiliency** are keys

Example: self-driving cars

- 100 Million lines of code for software, sensing and actuation
- 64 TOPS for cognition and control functions

Systems Must be Designed to Meet Many Different Requirements

Performance

- Real-time response
- Local computations with off-line training/learning

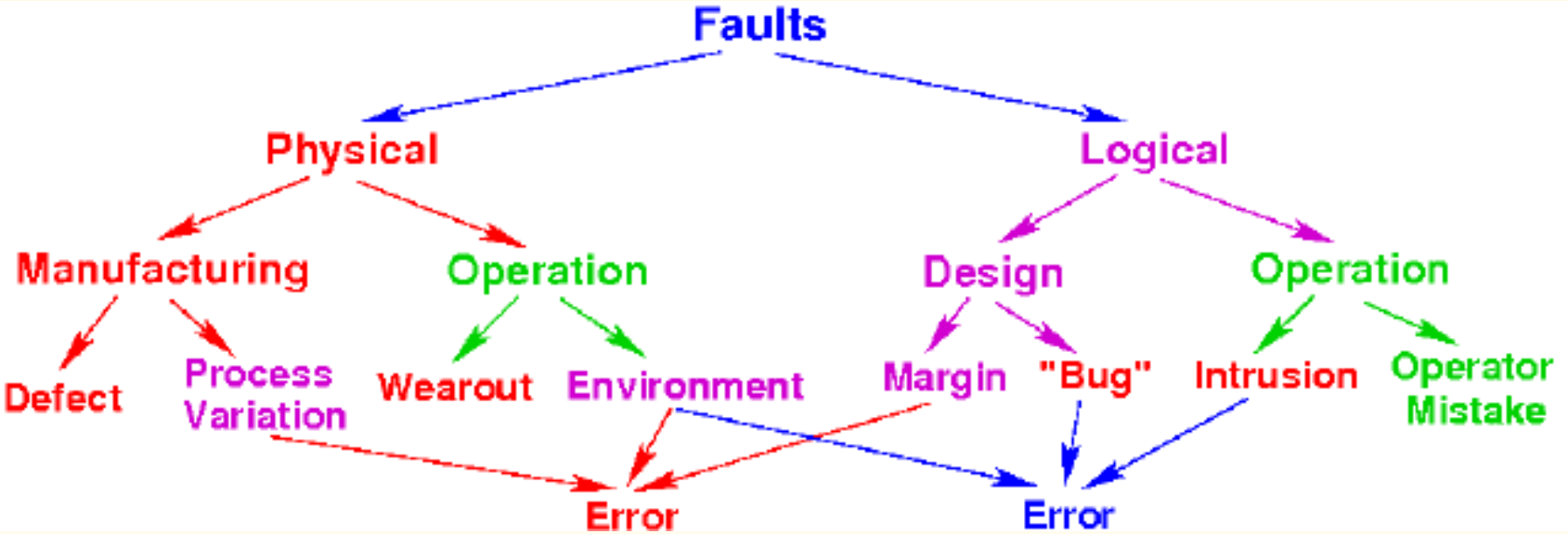
Safety and Reliability

- Deal with failures and wearout of subsystems
- Deal with undetected issues from the design and manufacturing process

Security

- Resilience to external attacks

Errors Produced by Faults or Interactions of Faults



Fundamental Requirements for Resilience

- **Redundancy**
- **Diversity**

"The most certain and effectual check upon errors which arise in the process of computation, is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if they make their computations by different methods"

Dionysius Lardner, "Babbage's calculating engine," *Edinburgh Review*, vol. 59, no. 120, pp. 263–327, 1834.

Reliability

View a system as providing a **service**

Faults $\Longrightarrow$ Errors $\Longrightarrow$ Failures

- Fault: an anomalous physical condition
- Error: an incorrect logic value as a consequence of the fault
- Failure: the condition where the system does not provide the expected service

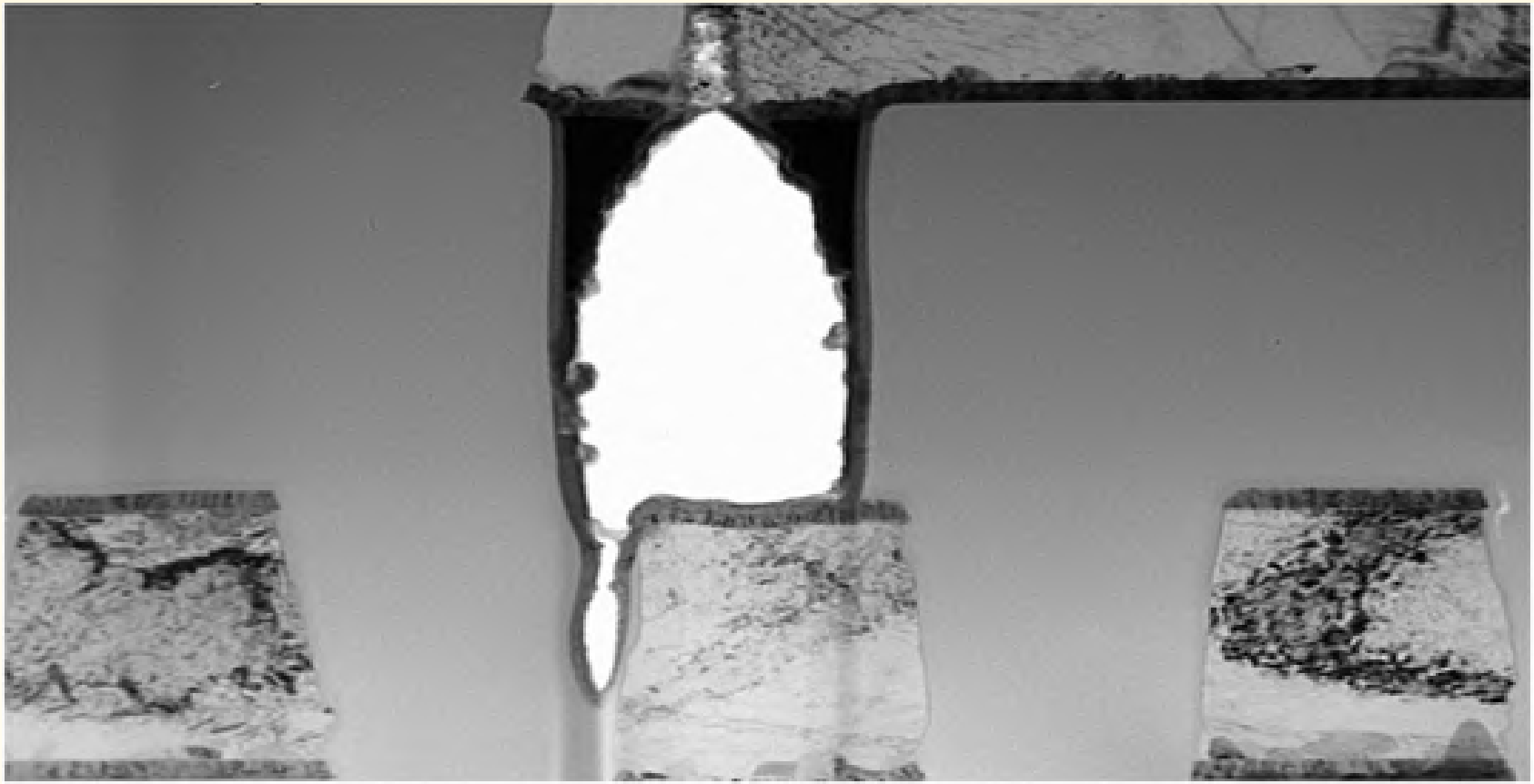
What Faults are Tolerated in Fault-Tolerant Computing?

- Module level – example in TMR
  - Any fault in a module
- Single "stuck-at" faults
  - At the gate or circuit level
- Single bit flip in storage elements
  - Due to cosmic (or other) radiation

The model is of secondary importance

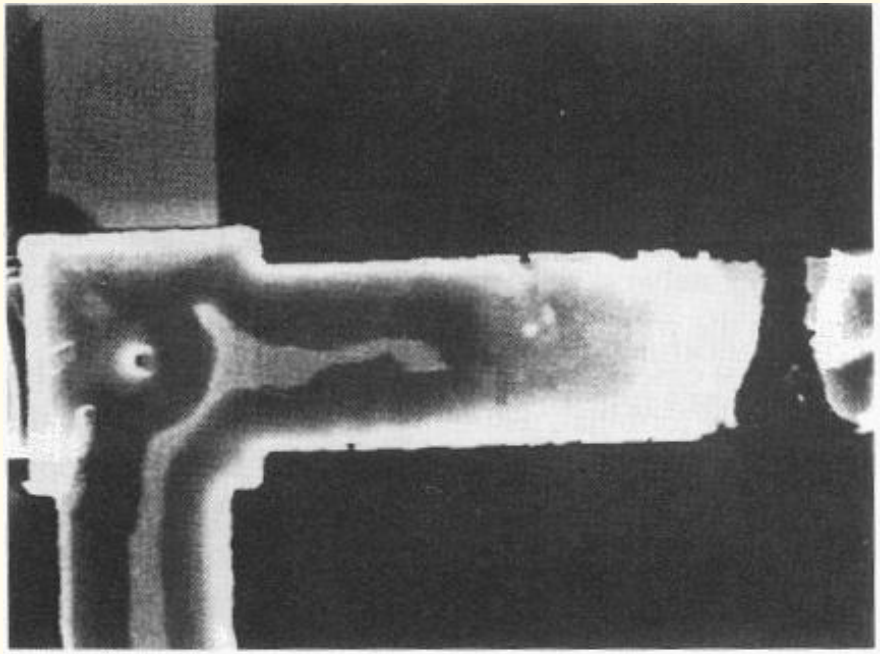
- What is important is whether errors are detected by the techniques developed for a given fault model

An Example of an IC Defect – a Resistive Via

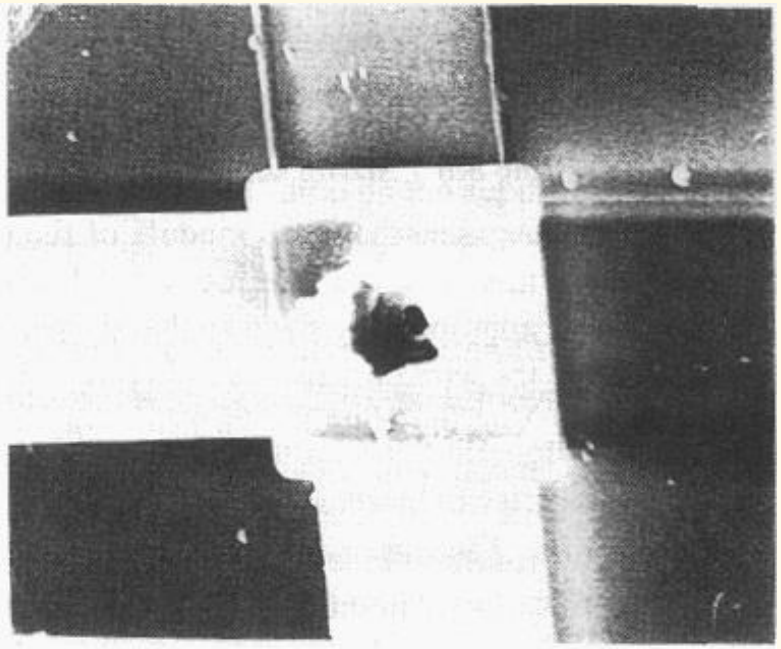


Source: Madge et al., D&T 2003

An Example of an IC Failure – Electromigration



Line-open failure



Open failure in contact plug

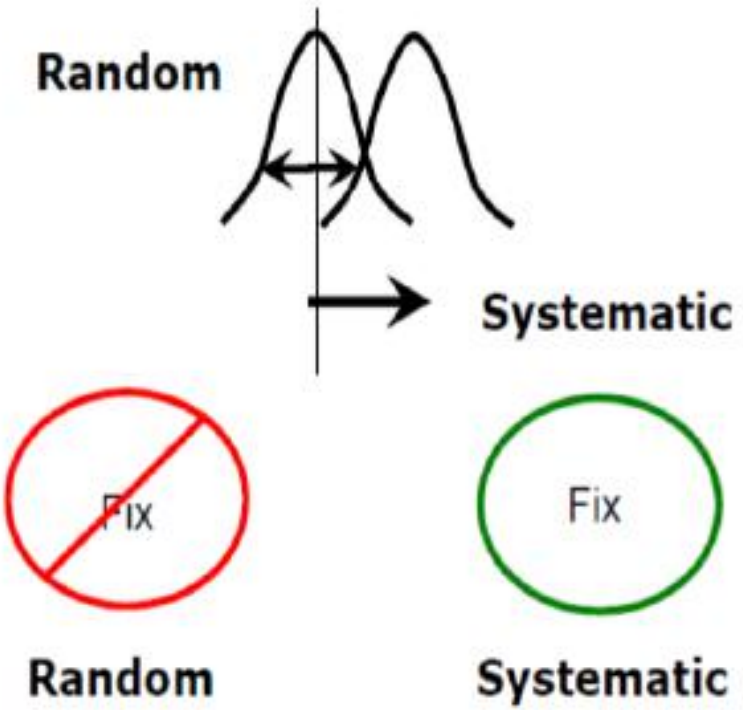
Source: N. Cheung and A. Tao, U.C. Berkeley

Sources of Variations in Integrated Circuits

Process Variations
- Dopant Fluctuation
- Oxide thickness
- Gate work function
- Line Edge Roughness
- Chemical-Mechanical Polish (CMP)
- Rapid-Thermal Anneal (RTA)
- Layout proximity effect

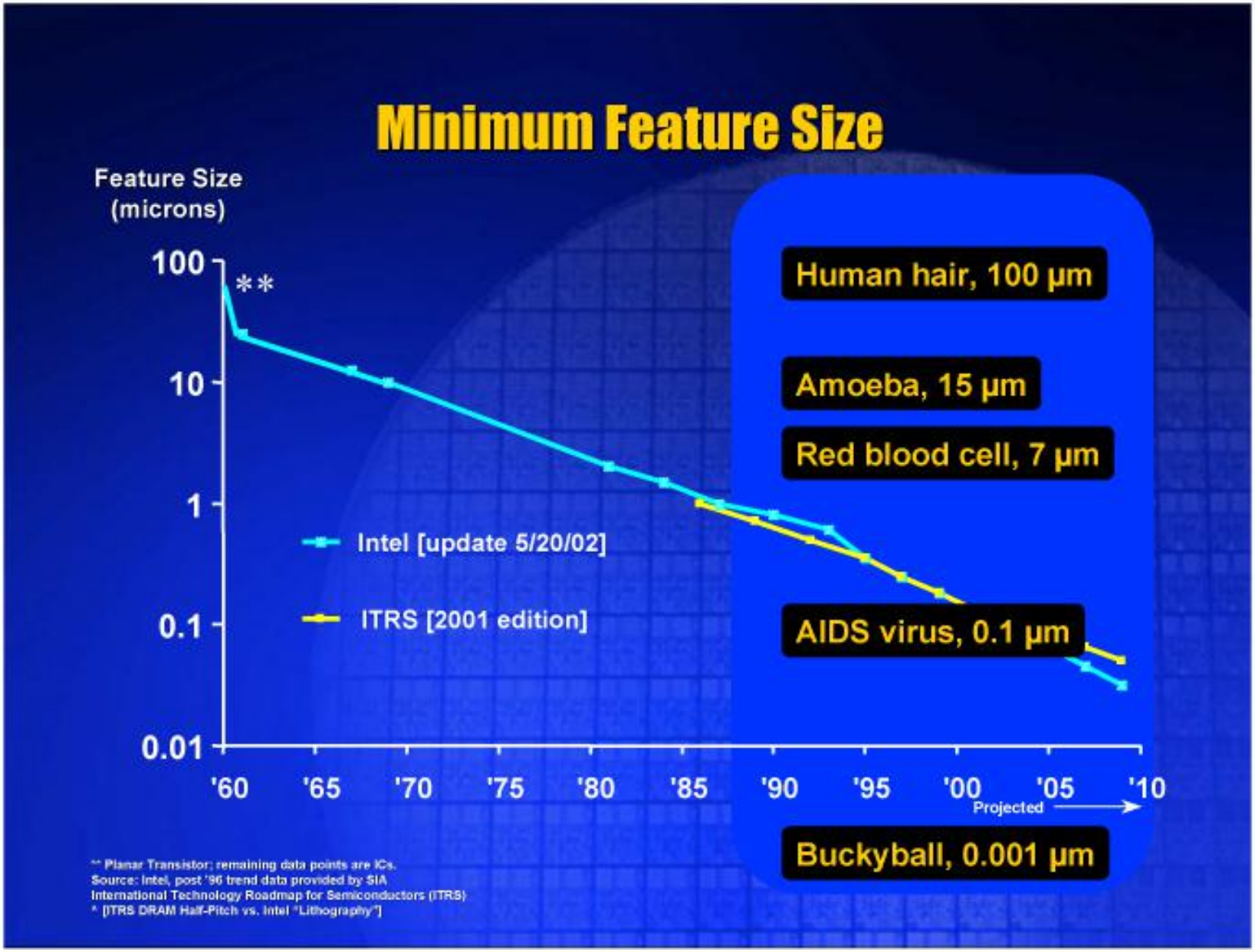
Environment Variations
- Temperature
- Voltage
- Package Stress, 3D Technologies.

Temporal Variation
- Transistor Aging (e.g. BTI)
- RTN induced Vmin fluctuation



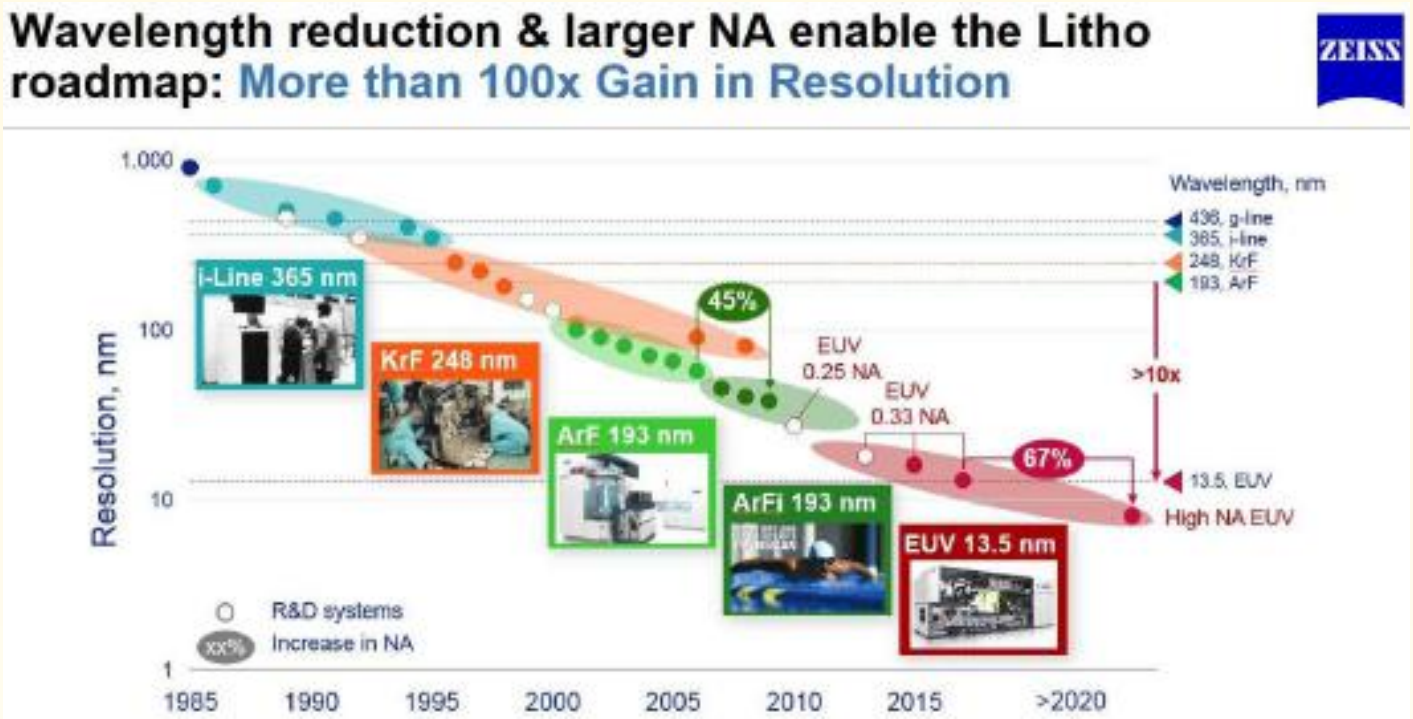
Source: J. Kulkarni

Causes of Variations



Features Smaller than Wavelengths

What is drawn is not what is printed on silicon

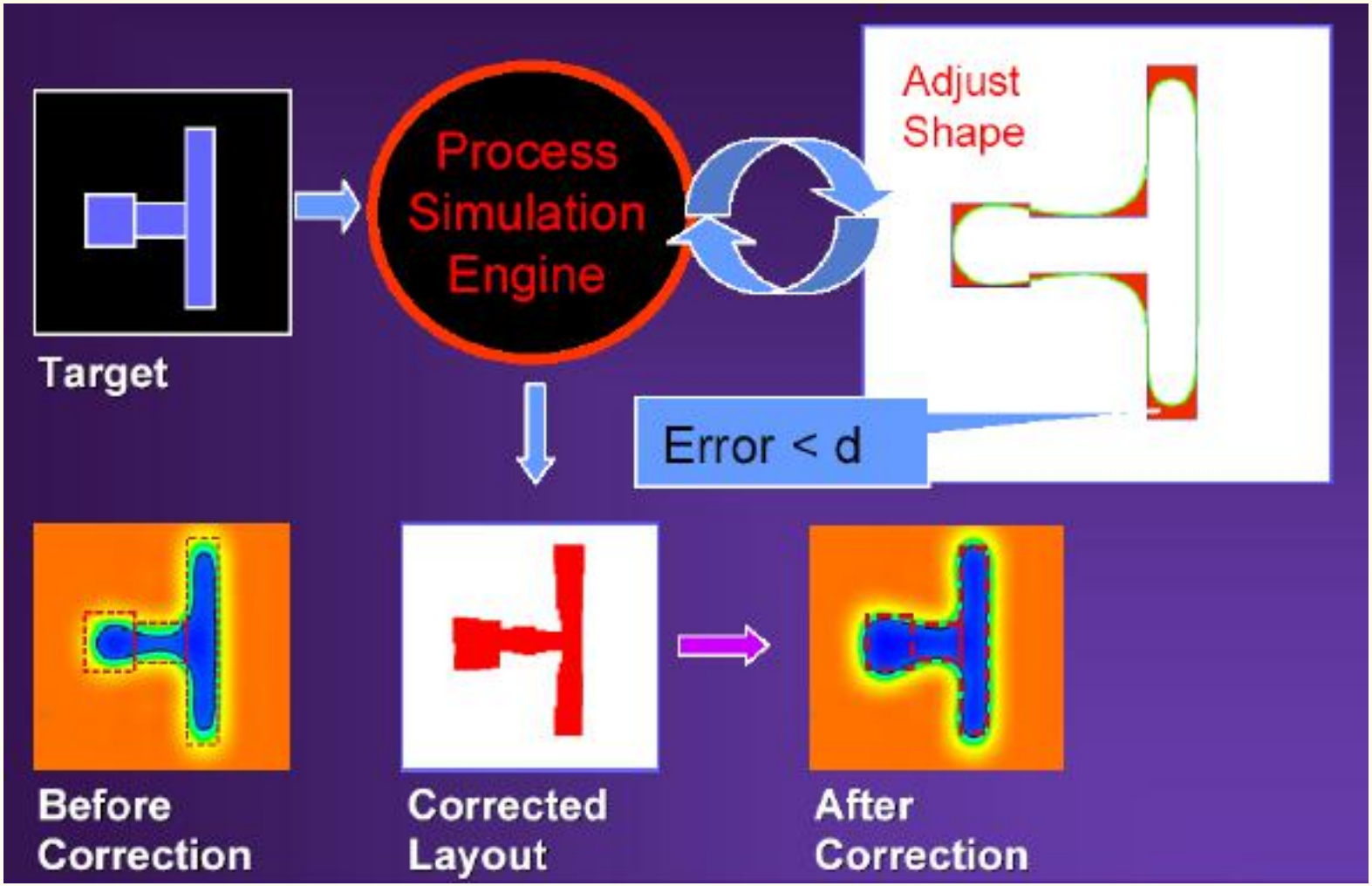


(NA: Numerical Aperture)

Source: Zeiss

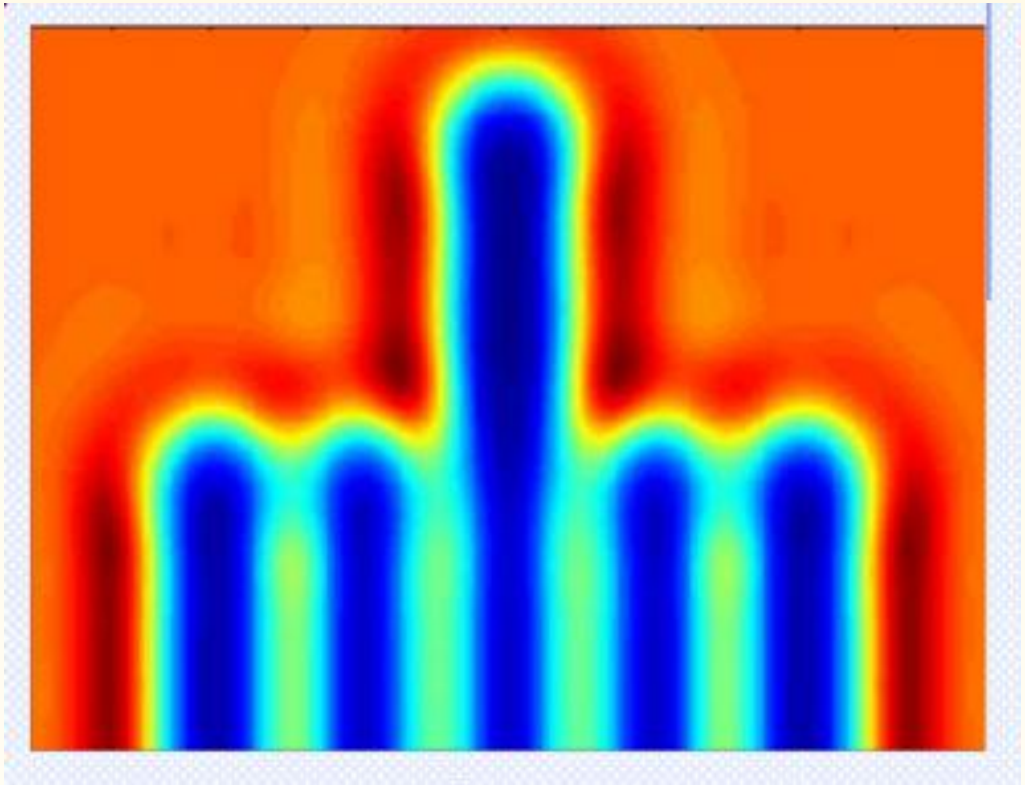
Optical Proximity Correction (OPC)

What you see is NOT what you get



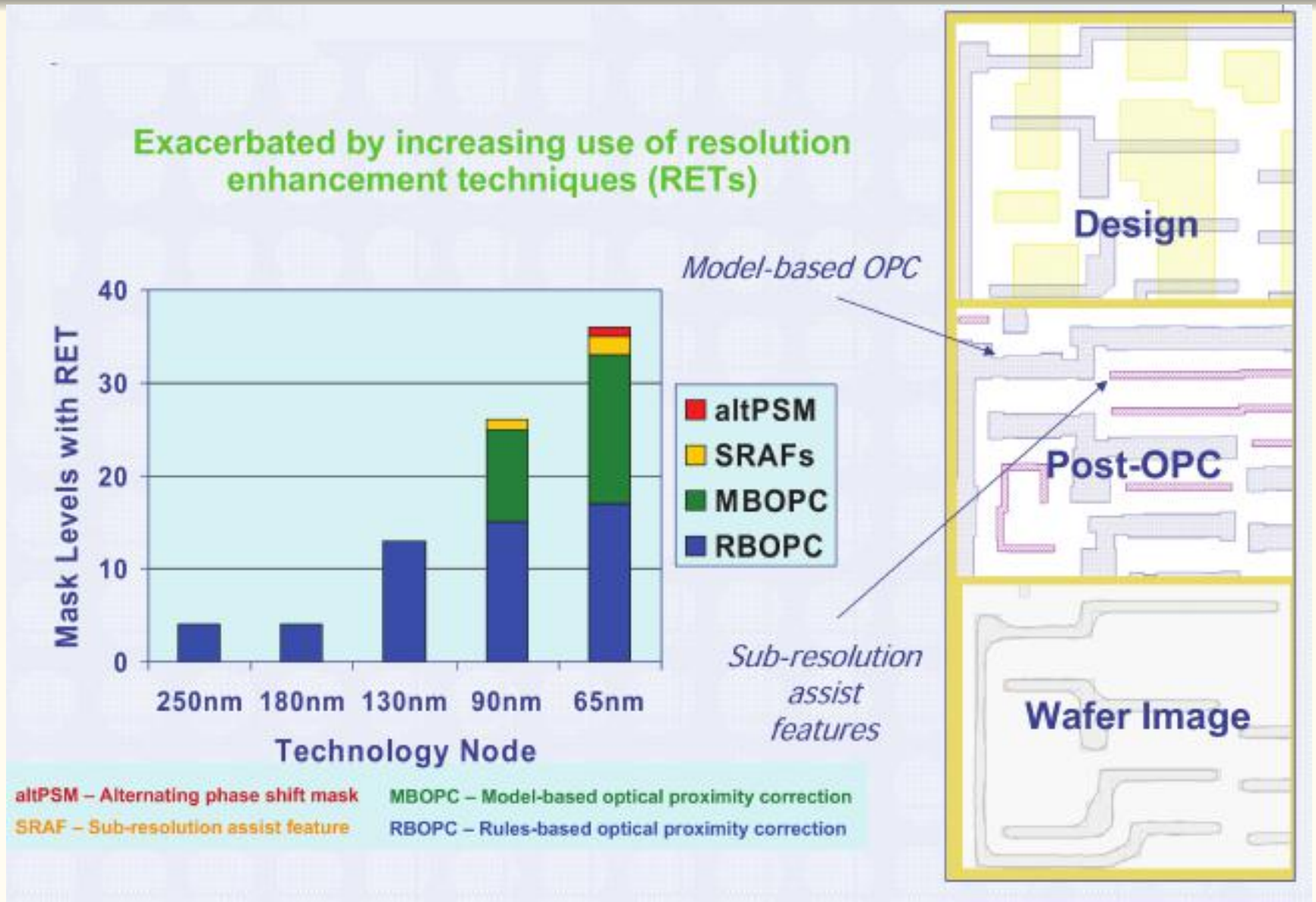
Imperfect Process Control

- Neighboring shapes interfere with the desired shape at some location: results in pattern sensitivity
- This is predominantly in the same plane
- There will be some interference from buried features for interconnect



Source: T. Brunner, ICP 2003

Increasing Mask Complexity

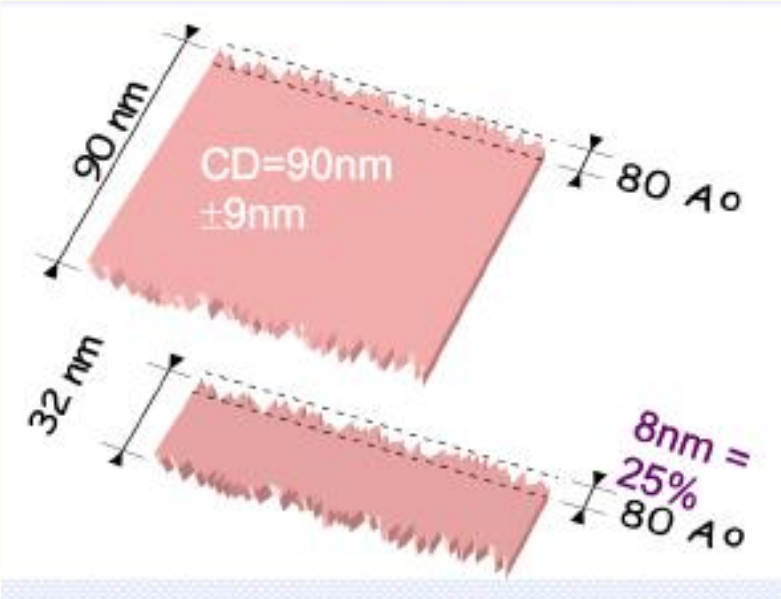


Source: K. Nowka, IBM

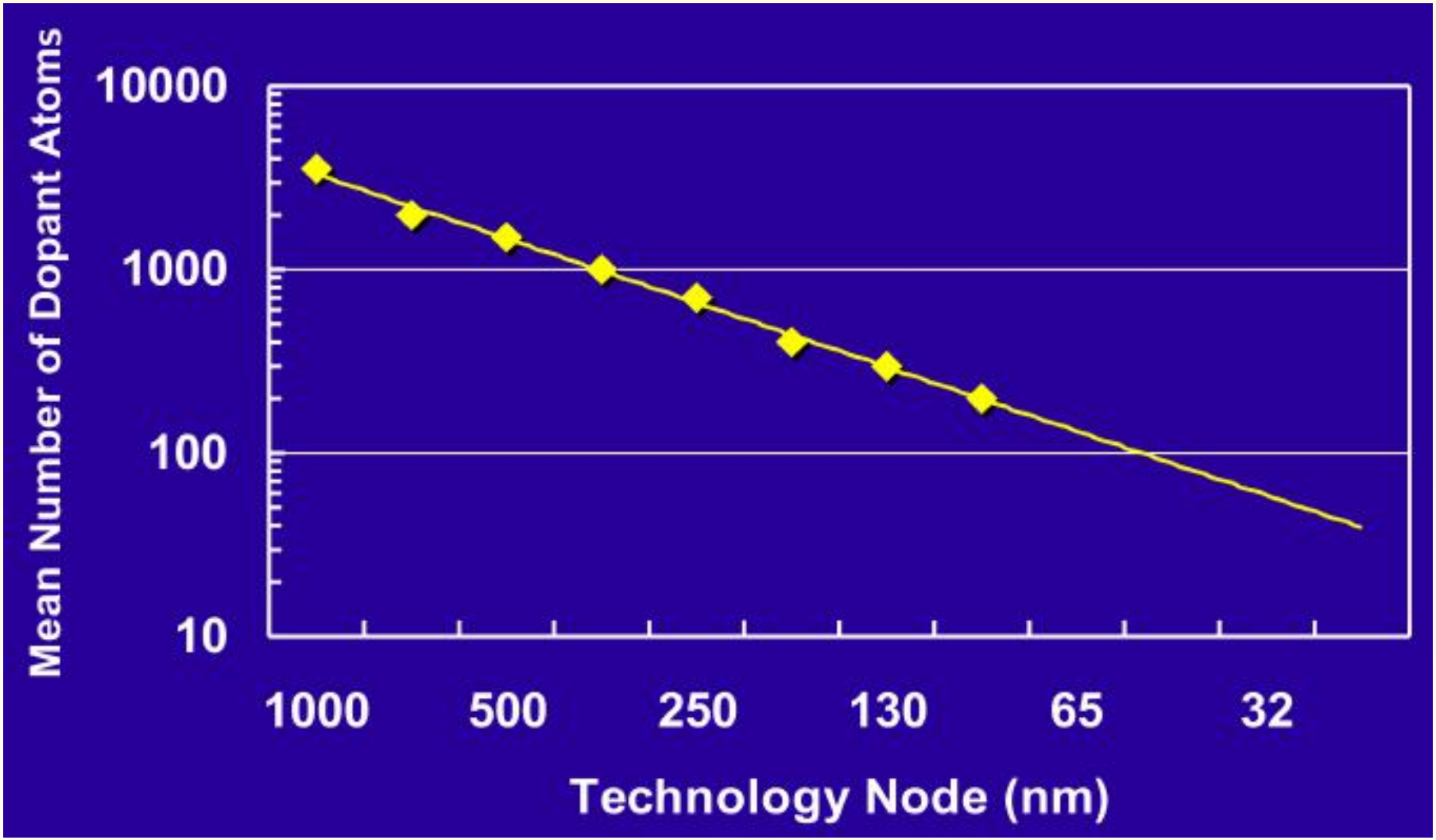
Line Edge Roughness

- In the lithography process, dose of photons will fluctuate due to finite quanta
  - Shot noise
- There will be fluctuations in the photon absorption positions
  - Due to nanoscale impurities in the resist composition

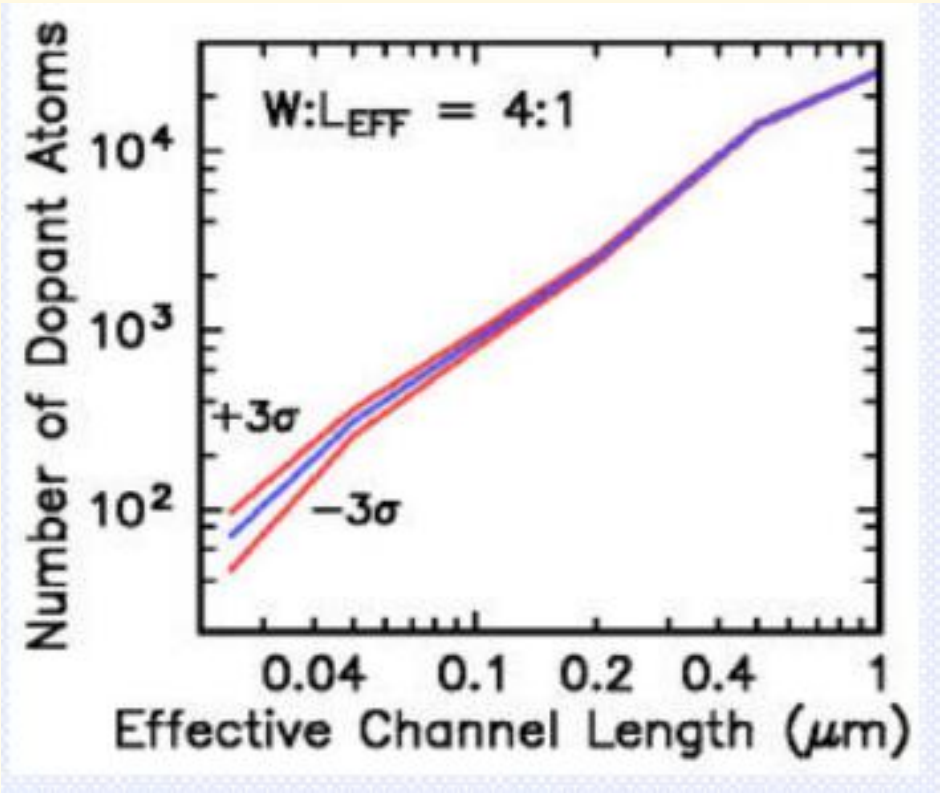
- Poly lines subject to increasing line edge roughness (LER)
  - Impact: circuit delay and leakage power

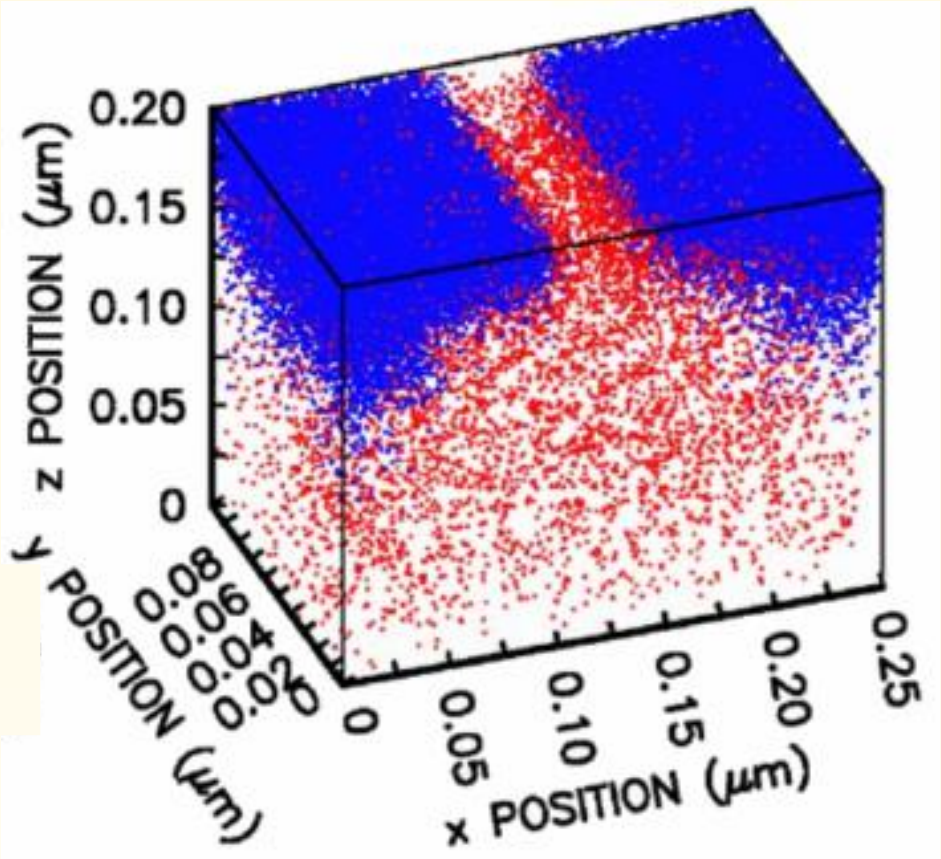


Random Dopant Fluctuations



Dopant Atoms in Channel





Source: D. Frank et al., VLSI Tech. 1999
D. Frank, H. Wong, IWCE, 2000

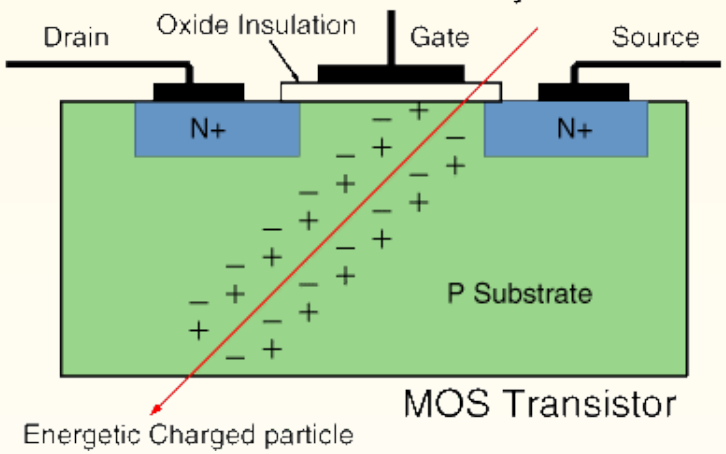
$>$ 200 mV V_t shift

Single-Event Upsets (Soft Errors)

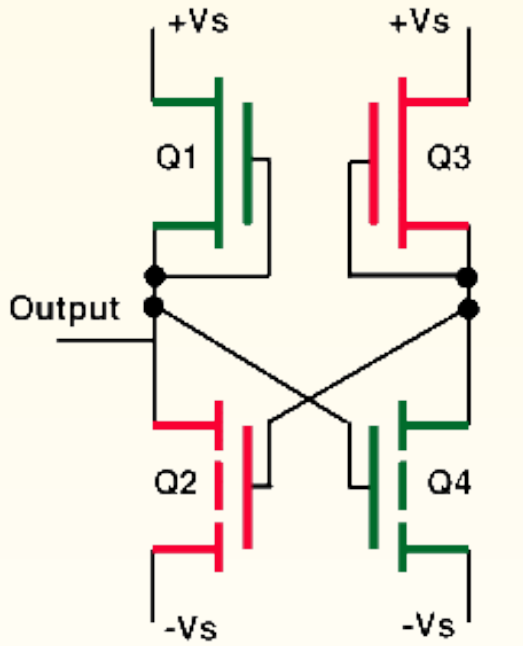
- High-energy particle produces electron-hole pairs in substrate; when collected at source and drain, will cause current pulse (Cosmic Radiation, etc.)
- A "bit-flip" can occur in the memory cell due to the charge generated by the particle – a "single-event upset"
- Seen in spacecraft electronics in the past, now in computers on the ground

Source: Aerospace Corporation

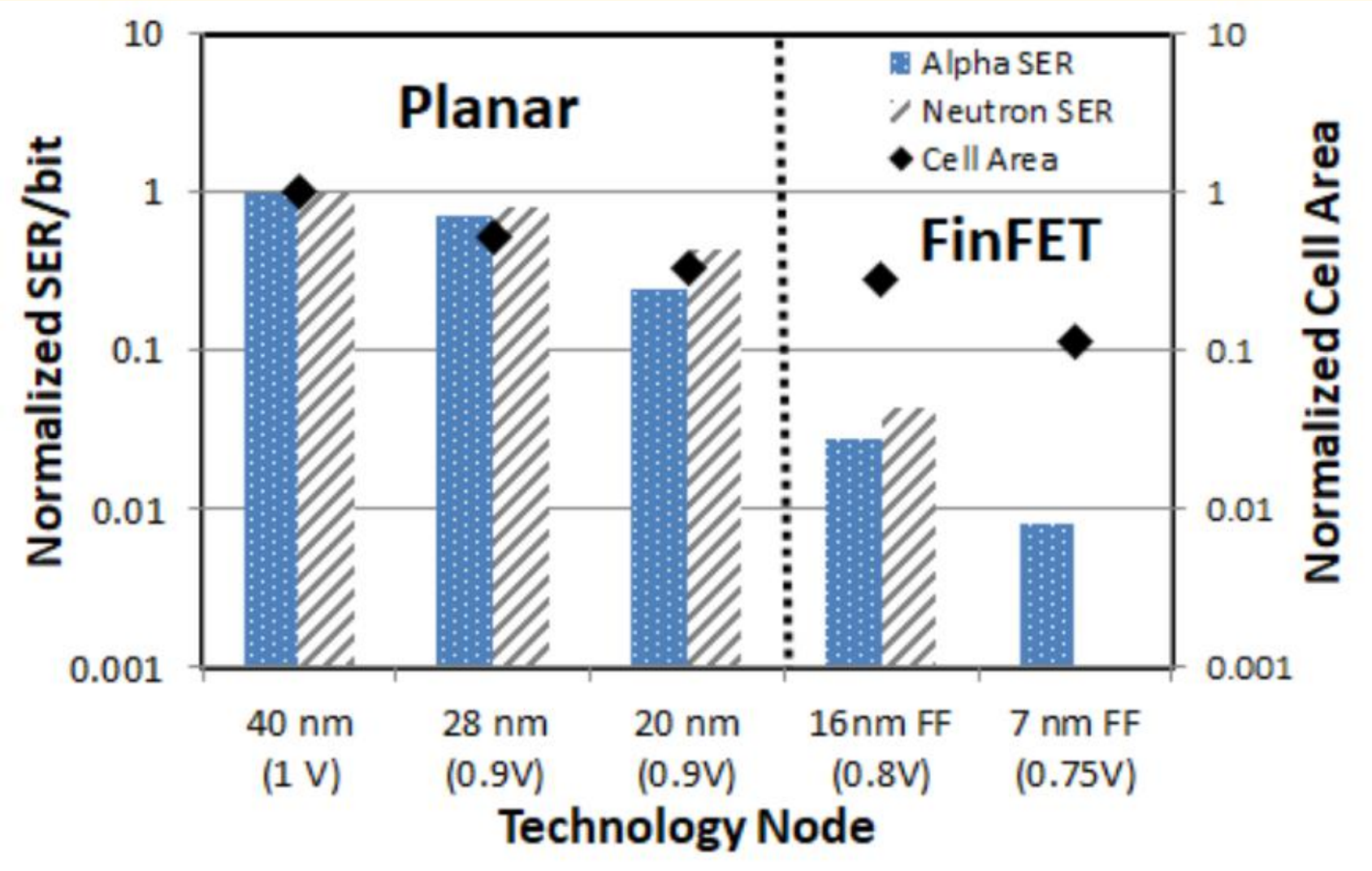
Interaction of a Cosmic Ray and Silicon



A simple memory element



Soft Error Rates



Source: Narasimhan et al., IRPS 2018

Are "Fault Tolerance" and "Resilience" the Same?

Fault Tolerance

- Errors (due to faults) detected and corrected, fault located, reconfiguration around faulty unit
- System designed to tolerate classes of faults
- User does not see anything wrong (except perhaps an additional delay)
- Service does not suffer any down time

Resilience

- User may see errors during the service, but the final results are correct
- System requires on-line error detection, but may use checkpoints, retry, etc., to achieve resilience
- Ability to deal with "unknown" faults
- Service may be down intermittently

Achieving Resilience

Start with fault-free hardware

- Testing after manufacturing
- On-line tests to detect wearout and degradation

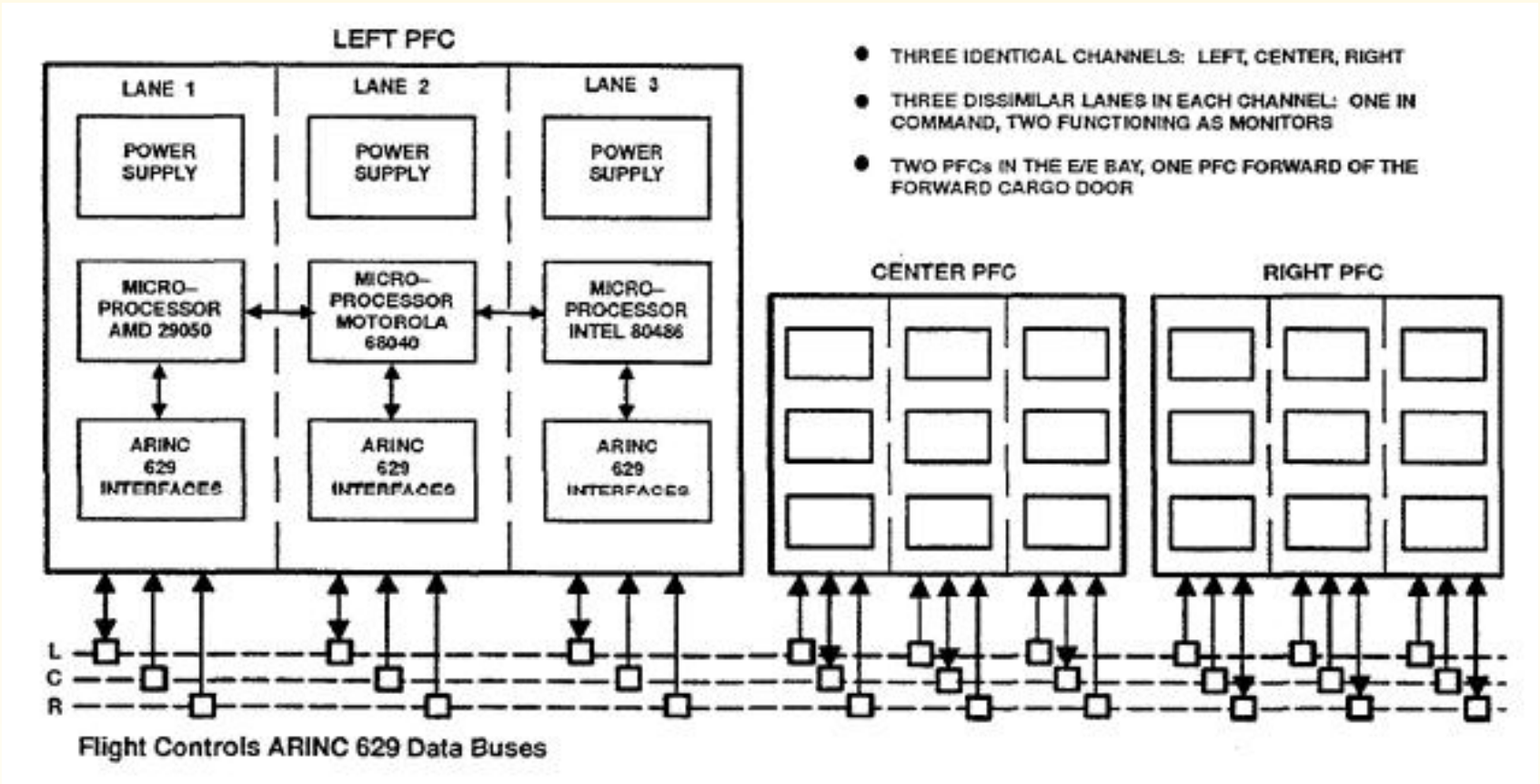
Detection is key

- Detect errors in results of computations
- Application-level results are, ultimately, what are important

Ensure correct results at the application level

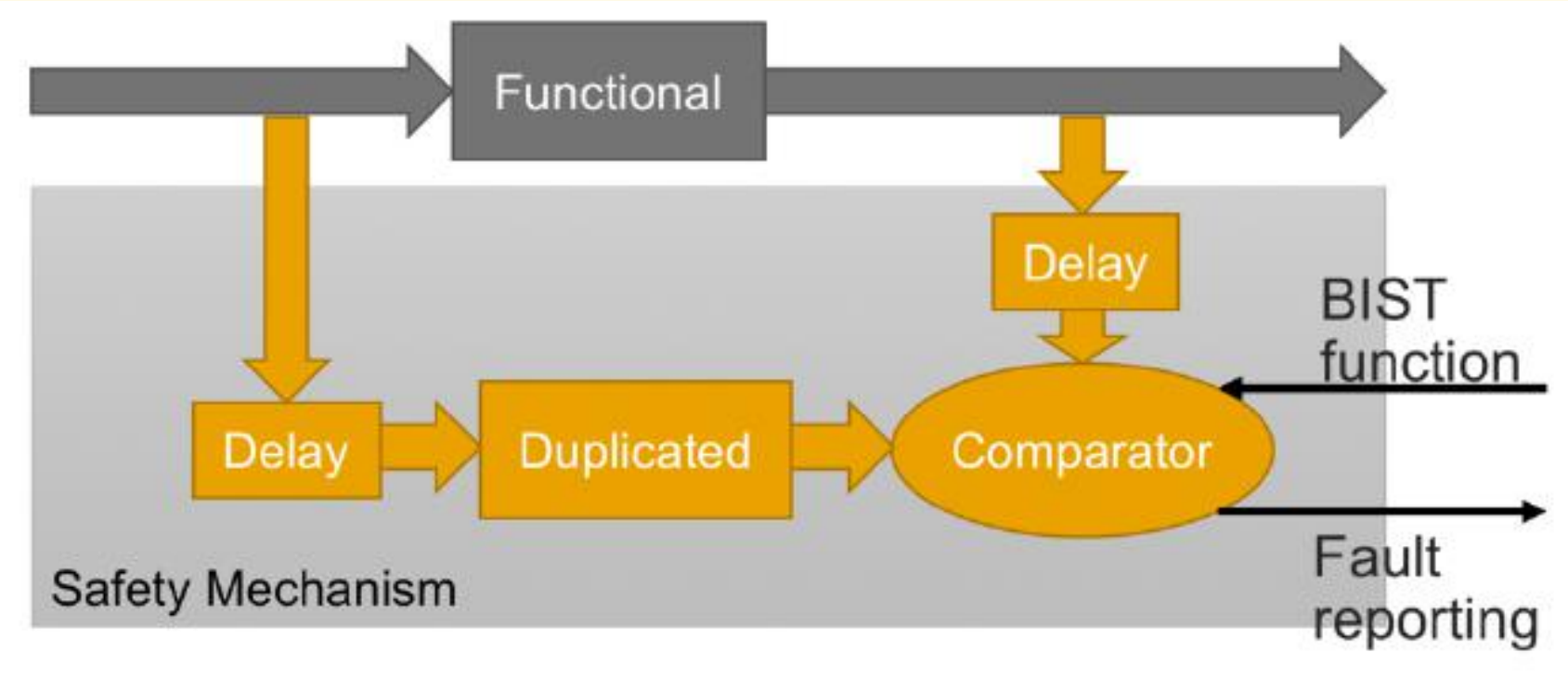
- Appropriate checks at different levels of the design
- High-level checks tend to have lower overheads (in general)

Example 1: Boeing 777 Primary Flight Computer



Example 2: ISO 26262, Functional Safety Standard Implementation

A much cheaper approach



Source: ArterisIP

Masking Redundancy Techniques

Redundant Components

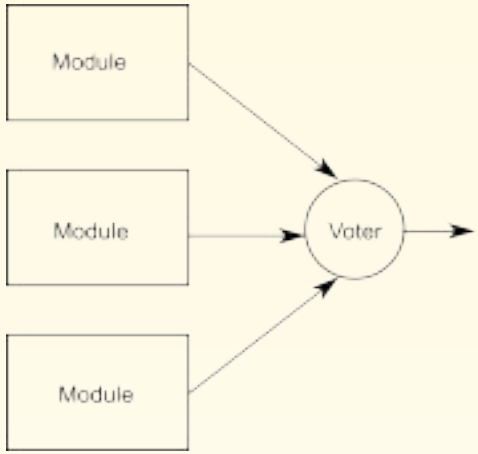
- Use multiple redundant components
- All components are active at a time
- When a fault occurs, the error produced by the faulty component is instantaneously masked

Considerations

- Systems based on masking redundancy will have high costs and power consumption
- Only a limited number of simultaneous component faults can be tolerated
- Masking can be applied at the module and logic/transistor levels

Voting Techniques

Triple-Modular Redundancy (based on von Neumann, 1956)

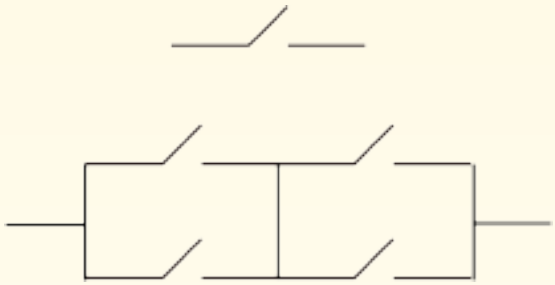


Modular Redundancy

- Technique can be generalized to NMR (N-modular redundancy)
- It is not straightforward to implement a practical voter which can synchronize the module outputs
- **Spare** modules can be used with NMR – replace faulty module with spare

Reliable Relay Networks – Possible Application to Emerging Technologies

Shannon, 1956

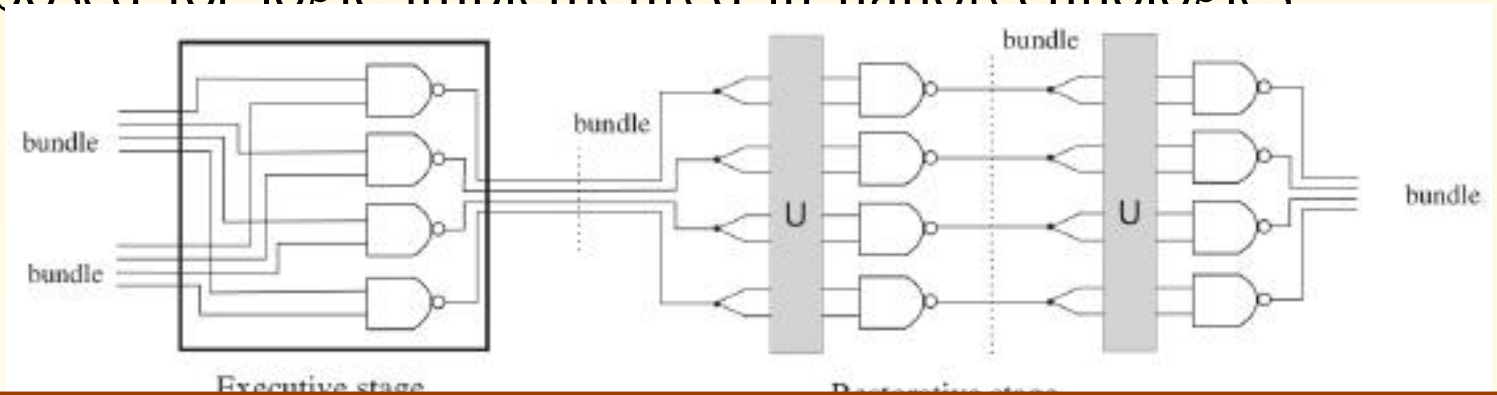


Can be applied to MOS transistors

- A single open or short of a transistor (relay) will be masked by the network
- Many multiple faults will also be tolerated

NAND Multiplexing – Application to Nanotechnology

Proposed in the 1956 von Neumann paper, now being proposed for logic implemented in nanotechnologies



Approach

- Similar to NMR, but instead of voting to decide the output, it is carried out in a *bundle*
- Two stages
  - Executive stage – performs operations
  - Restorative stage – reduces degradation caused by errors from the executive stage, acting as an "amplifier" of the output

Error-Detecting and Correcting Codes

One way of detecting (and correcting) errors in data transmission and storage, is to encode data, with a subset of the words being code words

Reasonable errors will change a code word to a non-code word, and the errors will be detectable

Errors which transform one code word into another will not be detectable

"Error Models" relate likely physical faults to the errors that they could cause

Distance between two code words is the number of distinct changes needed to change one code word into the other

Example: Parity codes

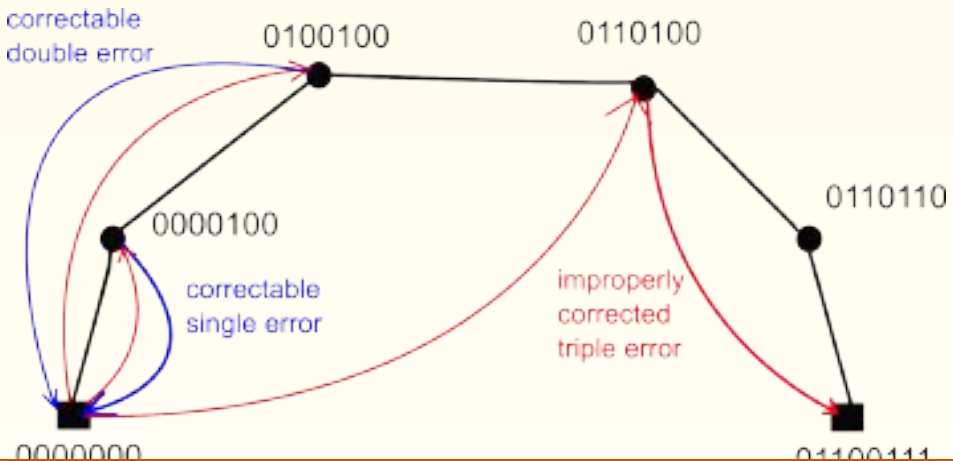
Distance Properties

The Hamming Weight of a vector, X, $W(X)$ is the number of non-zero components of X

The Hamming Distance between two vectors, X and Y, $d(X, Y)$, is the number of components in which they differ

The minimum distance of a code is the minimum of Hamming distances between all pairs of code words

To detect d-bit errors, need a code with distance $d + 1$, to correct d-bit errors, need a code with distance $2d + 1$.
Example:



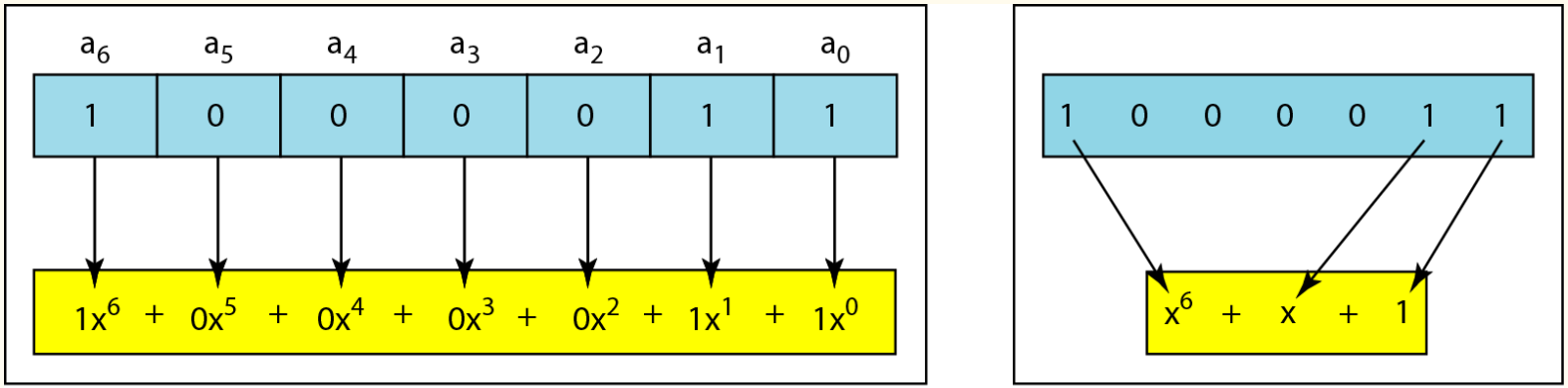
Cyclic Codes

A cyclic code is a parity check code where every cyclic shift of a code word is also a code word

Cyclic codes are conveniently described as polynomials

$$C(x) = C_{n-1}X^{n-1} + C_{n-2}X^{n-2} + \cdots + C_1X + C_0$$

Example:



a. Binary pattern and polynomial

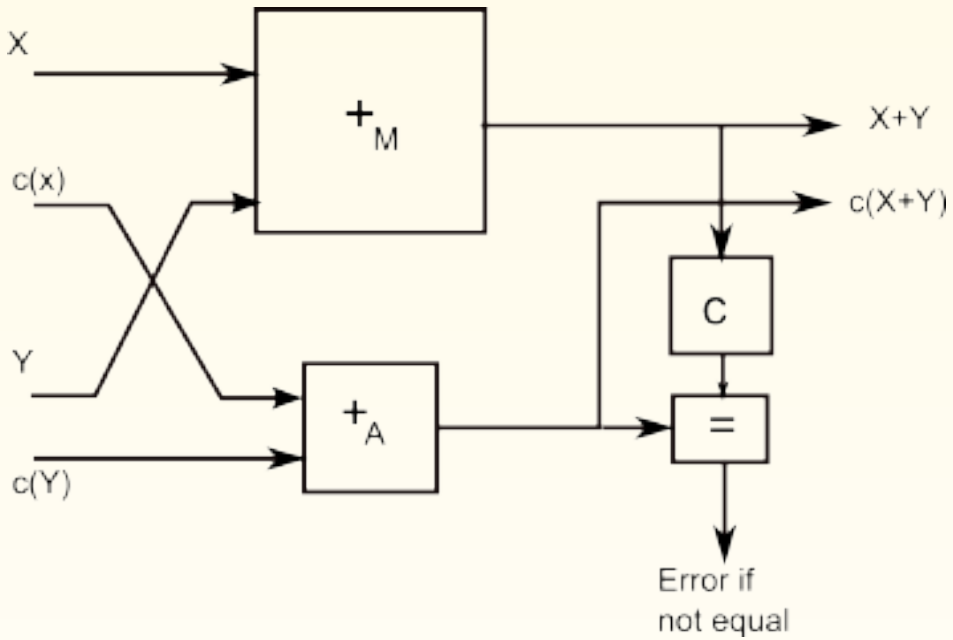
b. Short form

Arithmetic Codes

Parity check codes are useful for data transmission and storage, but are not preserved by arithmetic operations

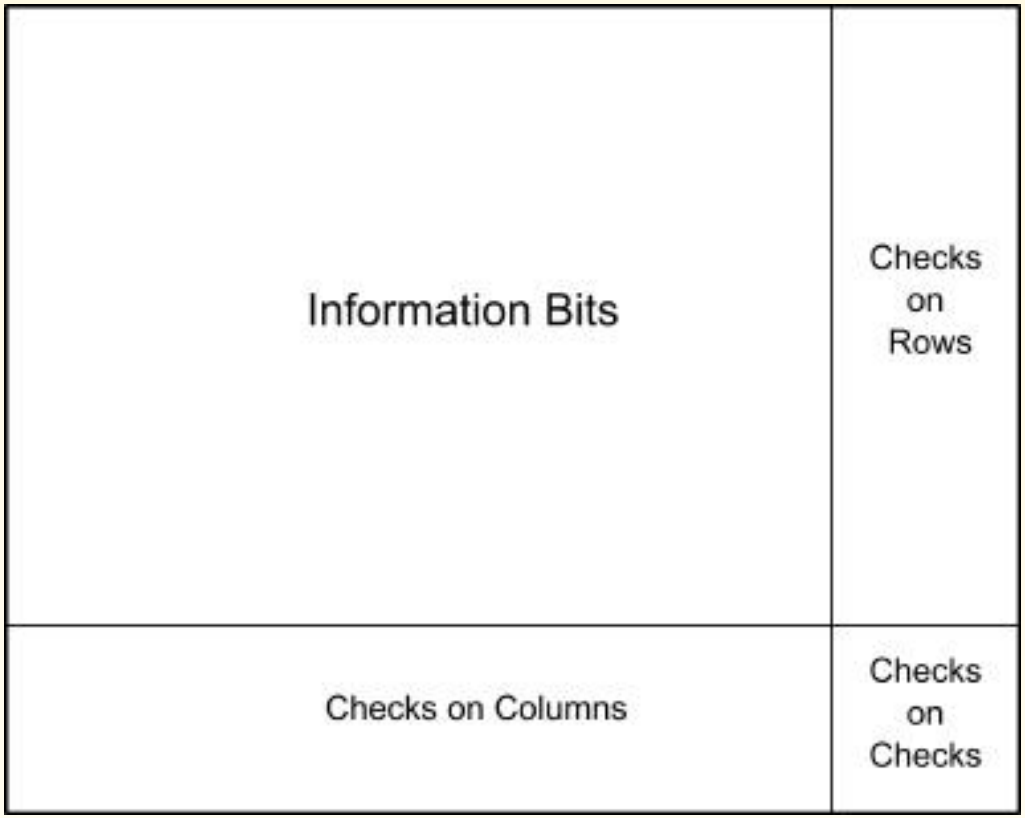
AN Codes (data multiplied by A (not a power of 2)) can be used to check arithmetic operations

Another code is the residue code: check, $c(x) = x \mod A$



Product Codes

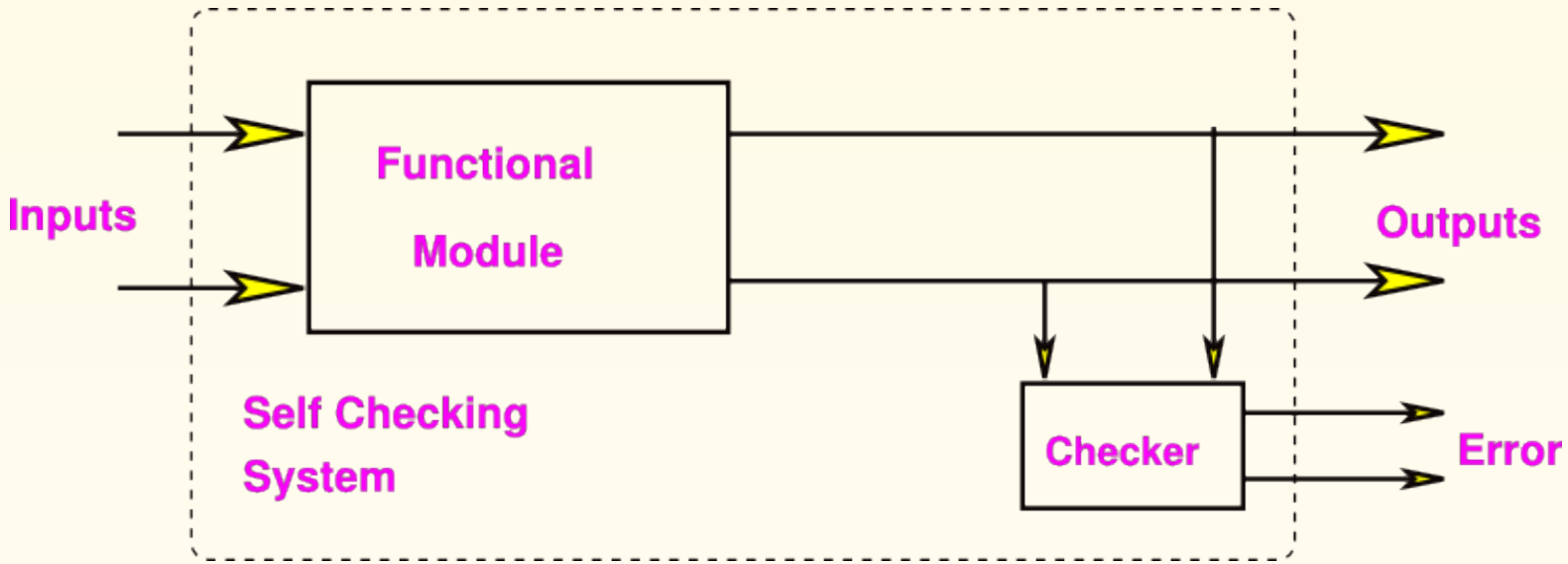
Two-dimensional code



Minimum distance is product of row and column distances
For simple parity on each, minimum distance is 4

Self-Checking Circuits

Self-Checking Circuits – encoded inputs and outputs, output checker



Error Detection by Duplicated Instructions (EDDI)

- Redundancy technique to deal with soft errors in logic
- Duplicates variables and data structures
- Uses different registers for duplicated instructions
- Interleaves duplicated instructions so that control flow errors are reduced
- EDDI is useful for detecting
  - Inter-block and intra-block control flow errors
  - Undesired data or code change in memory
  - Transient errors in functional units

Many Related Techniques

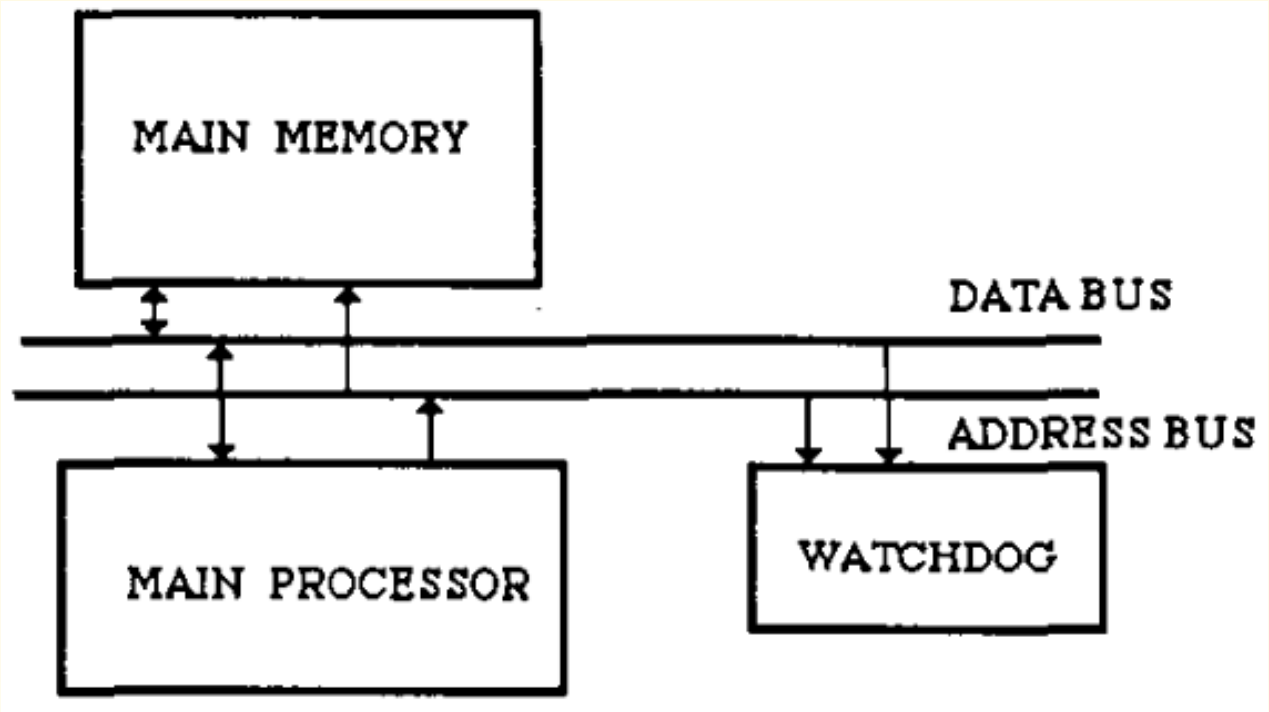
- CFCSS: control flow checking with software signatures
- SWIFT: software implemented fault tolerance
- ED^{4}I: error detection with diverse data and duplicated instructions (useful for Byzantine fault tolerance)

Exploiting Multithreading Capabilities of Modern Processors

- Redundant execution to tolerate errors – lower cost
- Replicate the threads – compare results before committing an instruction
- For arithmetic operations, can scale redundant computation (similar to RESO: recomputing with shifted operands)

Watchdog Processors (Liu, 1980)

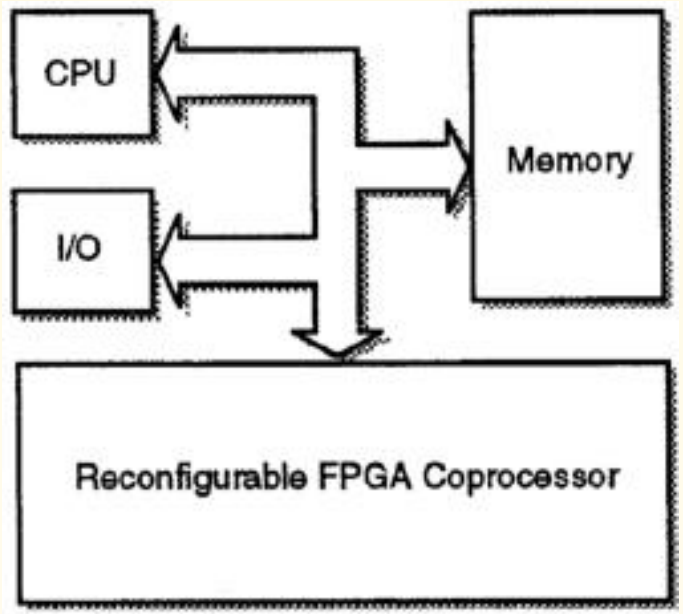
Extension of the idea of **watchdog timers**



- Watchdog has information of the process being checked
- Concurrently monitors the execution to detect errors

Implements both redundancy and diversity

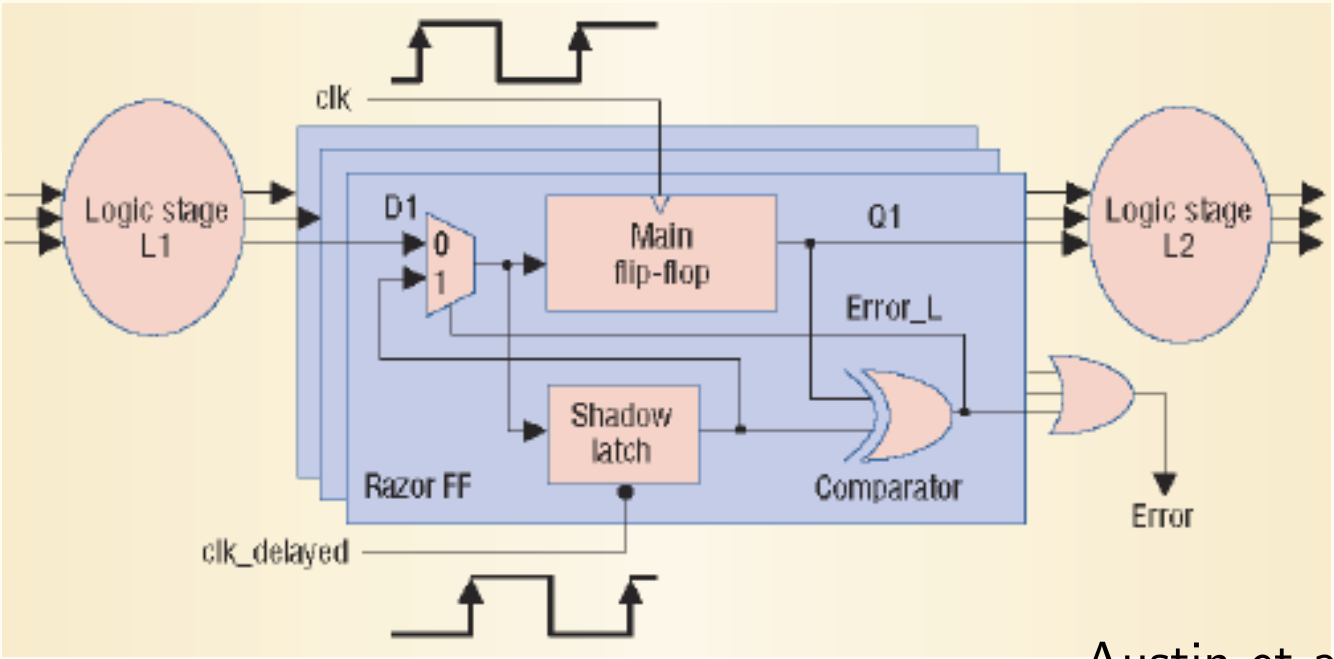
Using a Reconfigurable Coprocessor for Adaptive Reconfiguration (Saxena, 1998)



- Approach includes both redundancy and diversity
- New modules can be reconfigured within the coprocessor
- Non-failed parts of existing processors can be reused after reconfiguration

RAZOR

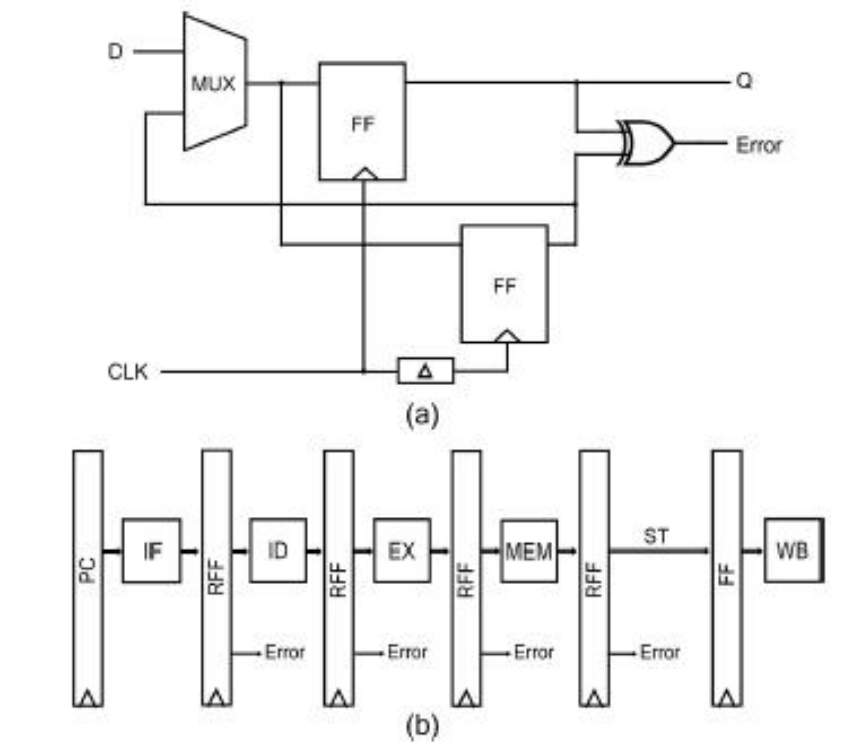
- Error-tolerant dynamic voltage scaling (DVS) technology which eliminates the need for the voltage margins required for "always correct" circuit operations design
- A different value in the shadow latch shows timing errors
- Pipeline state is recovered after timing-error detection
- Error detection is done at the circuit level
  - The design overhead is large if timing paths are well balanced in the design



Austin et al., 2003

Direct Monitoring of Critical Path

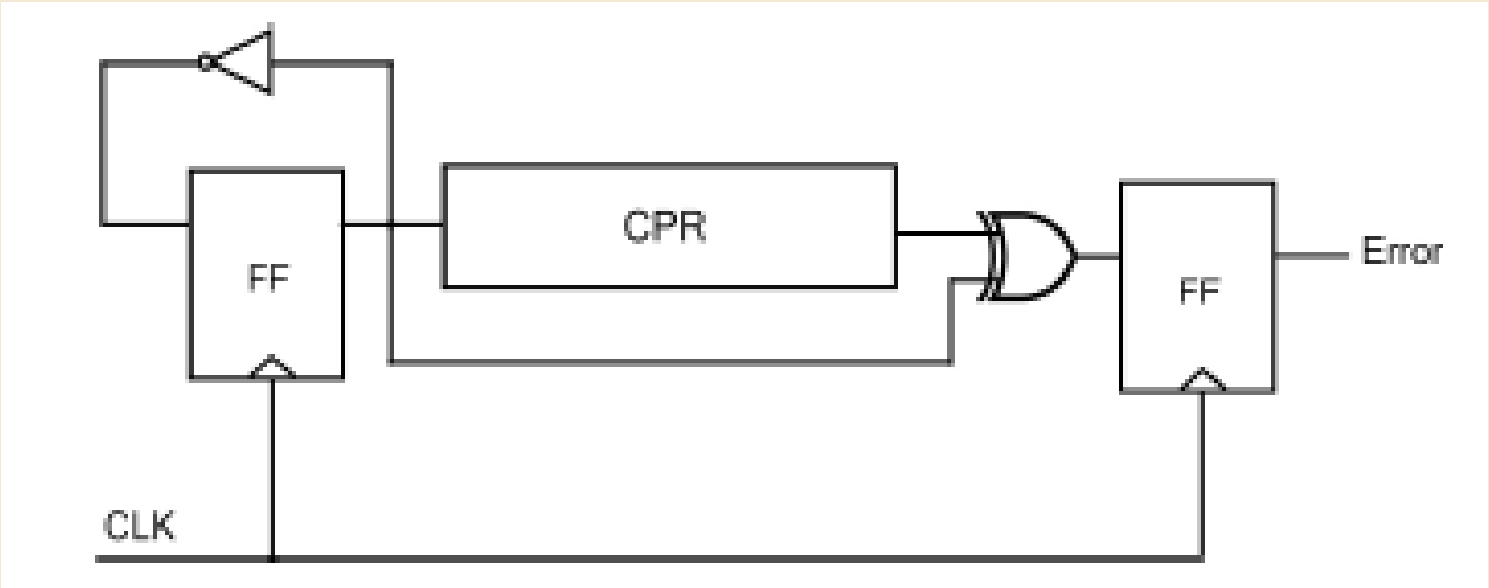
Razor Flip-Flop (a) and Architecture using it (b)



- Speculative operation requires an additional pipeline stage
- Design may not be suitable for designs that have many critical paths (increase in area and flip-flop power)

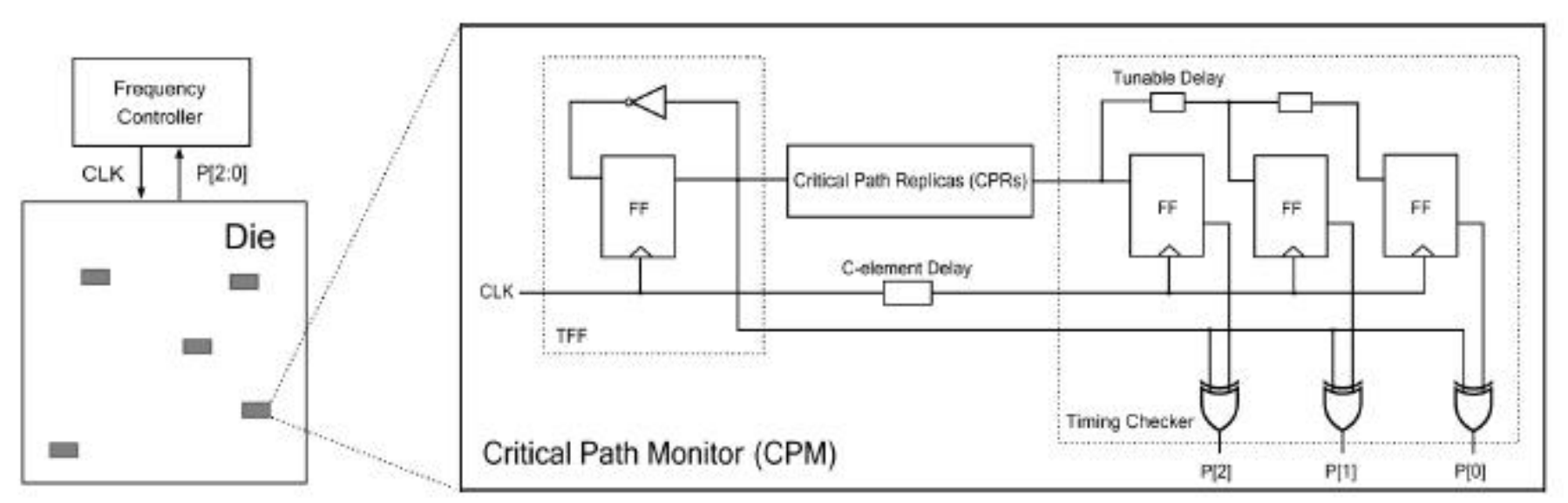
Indirect Critical Path Monitor

TEAtime approach



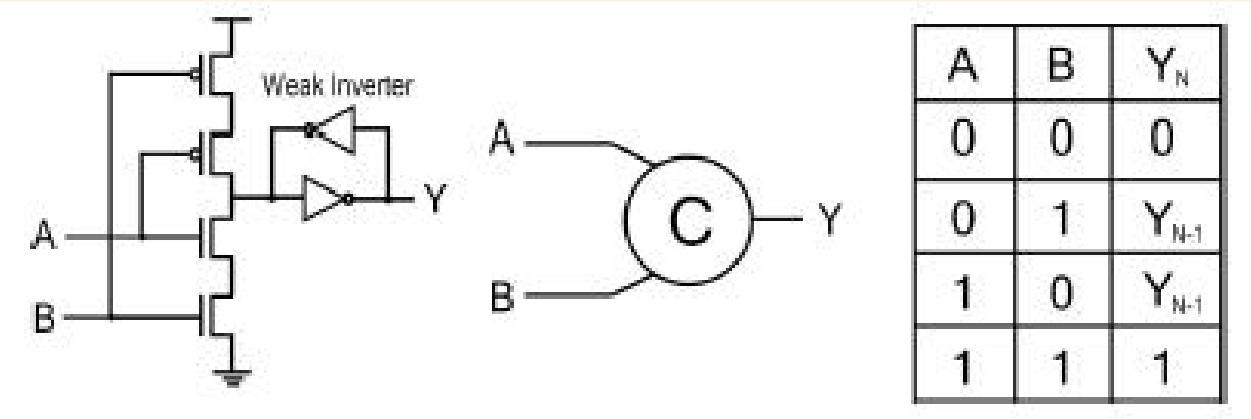
- Use of Critical Path Replicas (CPRs) to control voltage or frequency until one of them fail
- CPRs (1-bit version of potential critical paths) are located near potential critical paths to monitor them
- 1-bit detector may result in “oscillations”

Adaptive Frequency Control with Critical Path Monitor (Park, 2011)



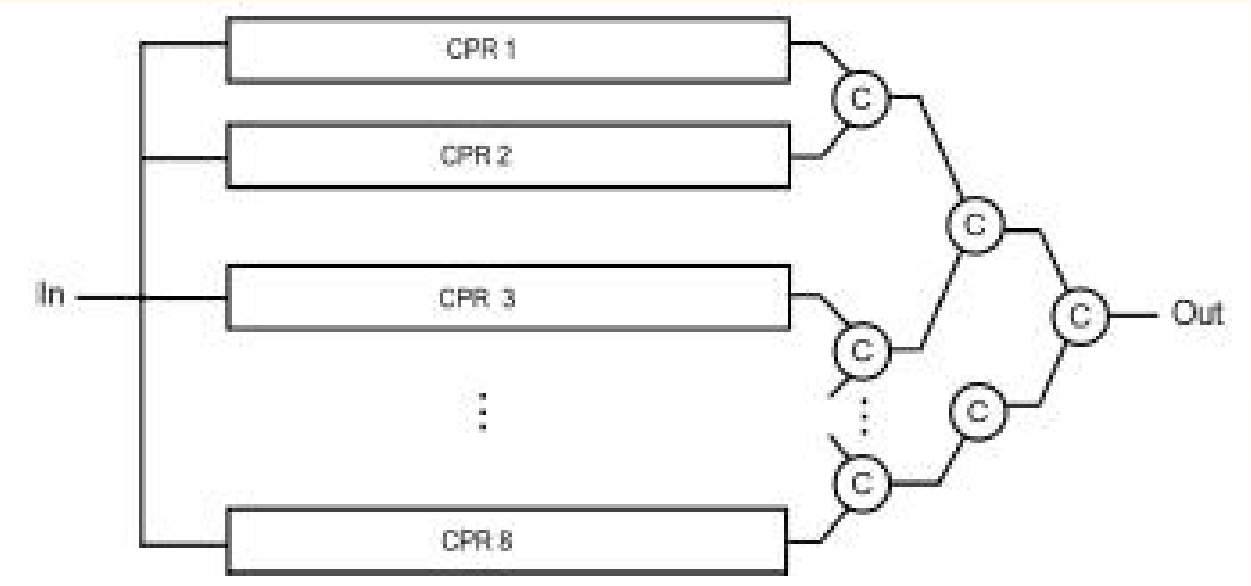
P[2:0]	Delay of CPRs	Frequency Control
{0,0,0}	Fast	↑
{0,0,1}	Appropriate	–
{0,1,1}	Slow (Safety Margin)	↓
{1,1,1}		

Use of C-elements to Combine CPRs



A	B	Y_N
0	0	0
0	1	Y_{N-1}
1	0	Y_{N-1}
1	1	1

C-element and logic function



Configuration of 8 CPRs

Application to Matrix Operations

Add an extra row and an extra column to encode the matrix

A is an $n \times m$ matrix and $e^T = [111 \ldots 1]$

Column Checksum Matrix,

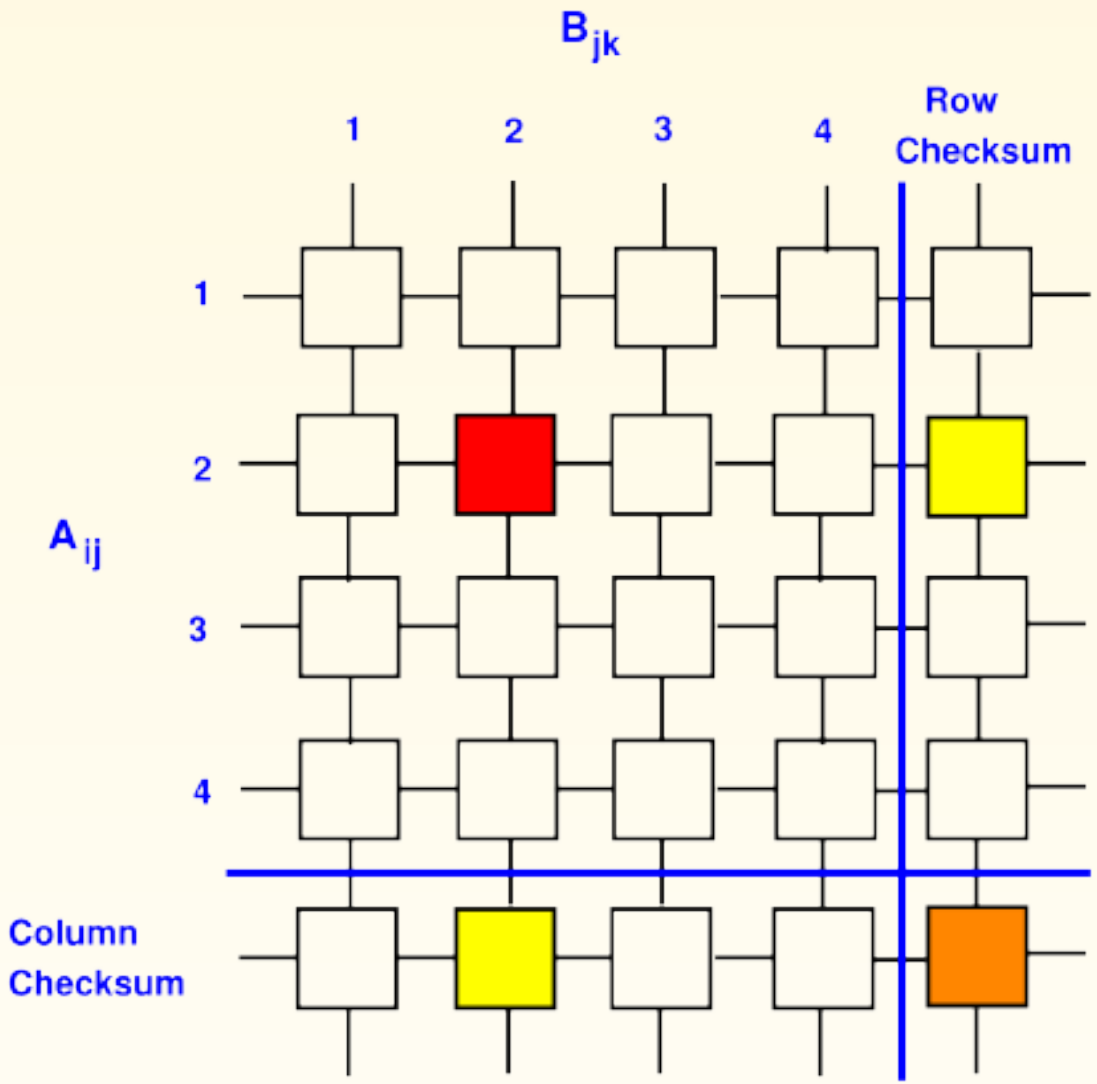
$$A_c = \left[\frac{A}{e^T A} \right]$$

Row Checksum Matrix, $A_r = [A \mid Ae]$

Full Checksum Matrix,

$$A_f = \left[\frac{A}{e^T A} \mid \frac{Ae}{e^T Ae} \right]$$

Illustration of Application to Matrix Operations



Properties of Checksum Matrices

The full checksum is a type of product code

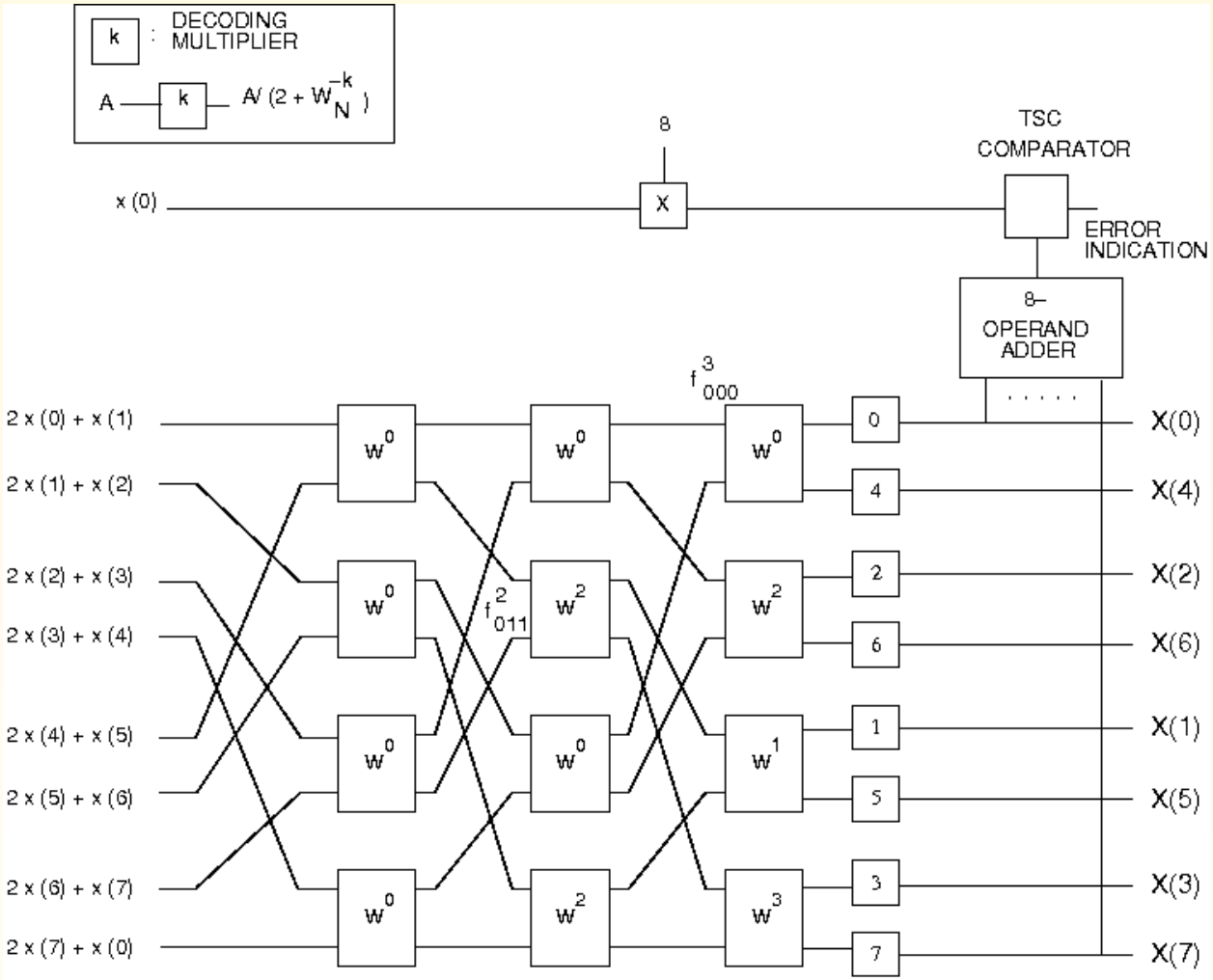
The Matrix Distance between two matrices is defined as the number of elements in which they differ

If two matrices are unequal, their full checksum matrices will have a distance of at least 4

Checksum Property is Preserved By

- Matrix multiplication
- Matrix addition
- Scalar product
- Matrix transpose
- LU-decomposition

Encoding Data to Detect Errors in FFT Networks



Reliable High-Performance Computing (HPC)

Issues with checkpointing

- Writing data to stable storage involves high cost
- Diskless checkpointing has been studied as a solution
- "Application–oblivious" checkpointing of message passing applications suffer from scaling issues
- Can incur considerable message passing performance penalties in some cases

Checkpointing-based applications do not scale

- As the number of processors increases, the overall reliability of the system decreases
- Then, checkpointing-restart is not a viable solution
- Scalable applications require recovery time to decrease as the number of processors increases

Source: Bosilca, Delmas, Dongarra and Langou, 2008.

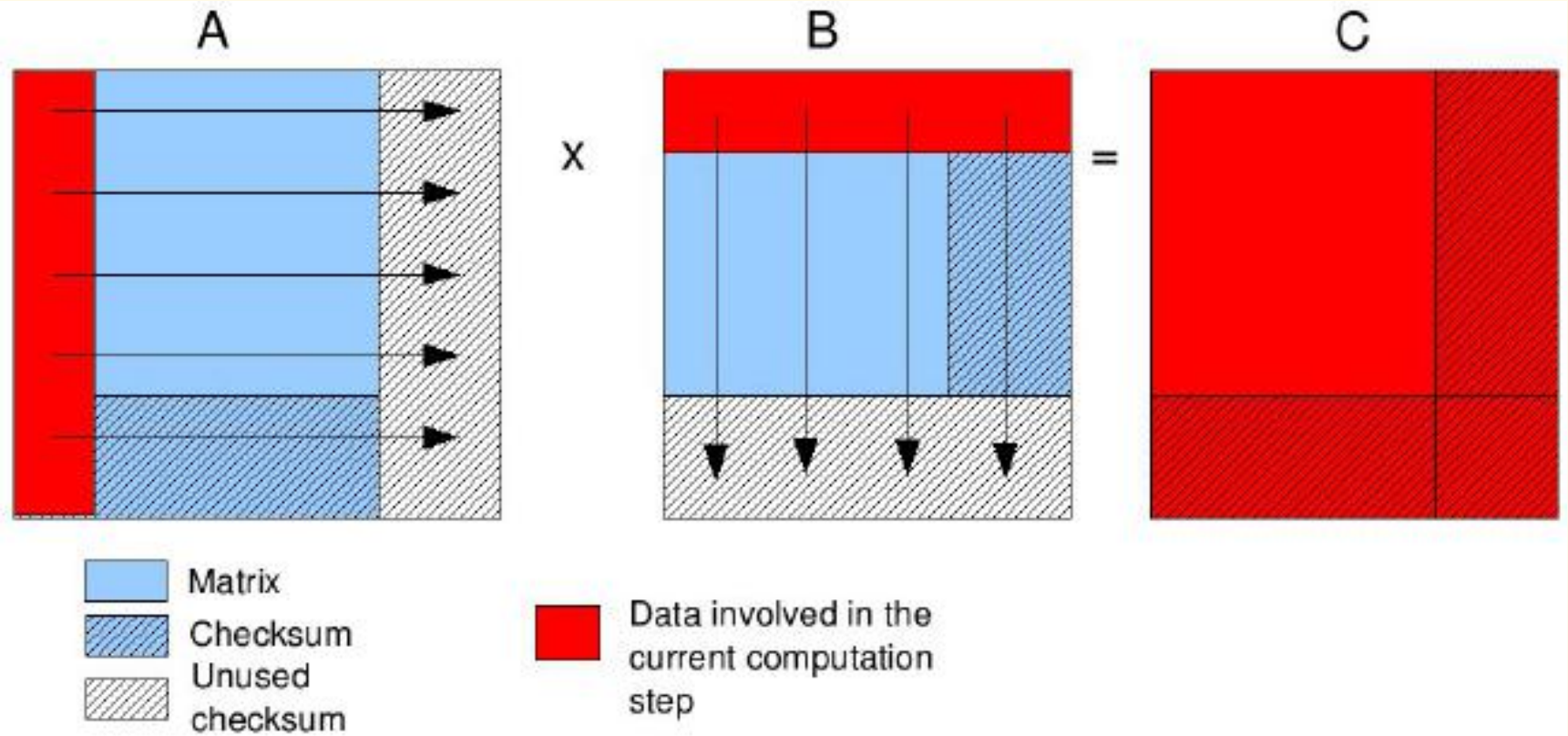
Use of ABFT in High-Performance Computing

- Algorithm-based fault tolerance has been extended to the parallel distributed context
- The classic algorithm corrects errors at the end of the matrix multiplication operation
  - Not ideal in HPC, where error correction should be done immediately after a failure
- Applied to matrix-matrix subroutine (PDGEMM)
  - Matrix-matrix multiplication is a kernel of fundamental importance to obtain efficient linear algebra subroutines
  - Application does not respond well to memory exclusion techniques, and so standard checkpointing techniques perform poorly
- Can encapsulate all the fault tolerance needed by the linear algebra subroutines in ScaLAPACK in a fault-tolerant *Basic Linear Algebra Subroutines* (BLAS)

Source: Bosilca, Delmas, Dongarra and Langou, 2008.

Illustration of Checksum Calculations

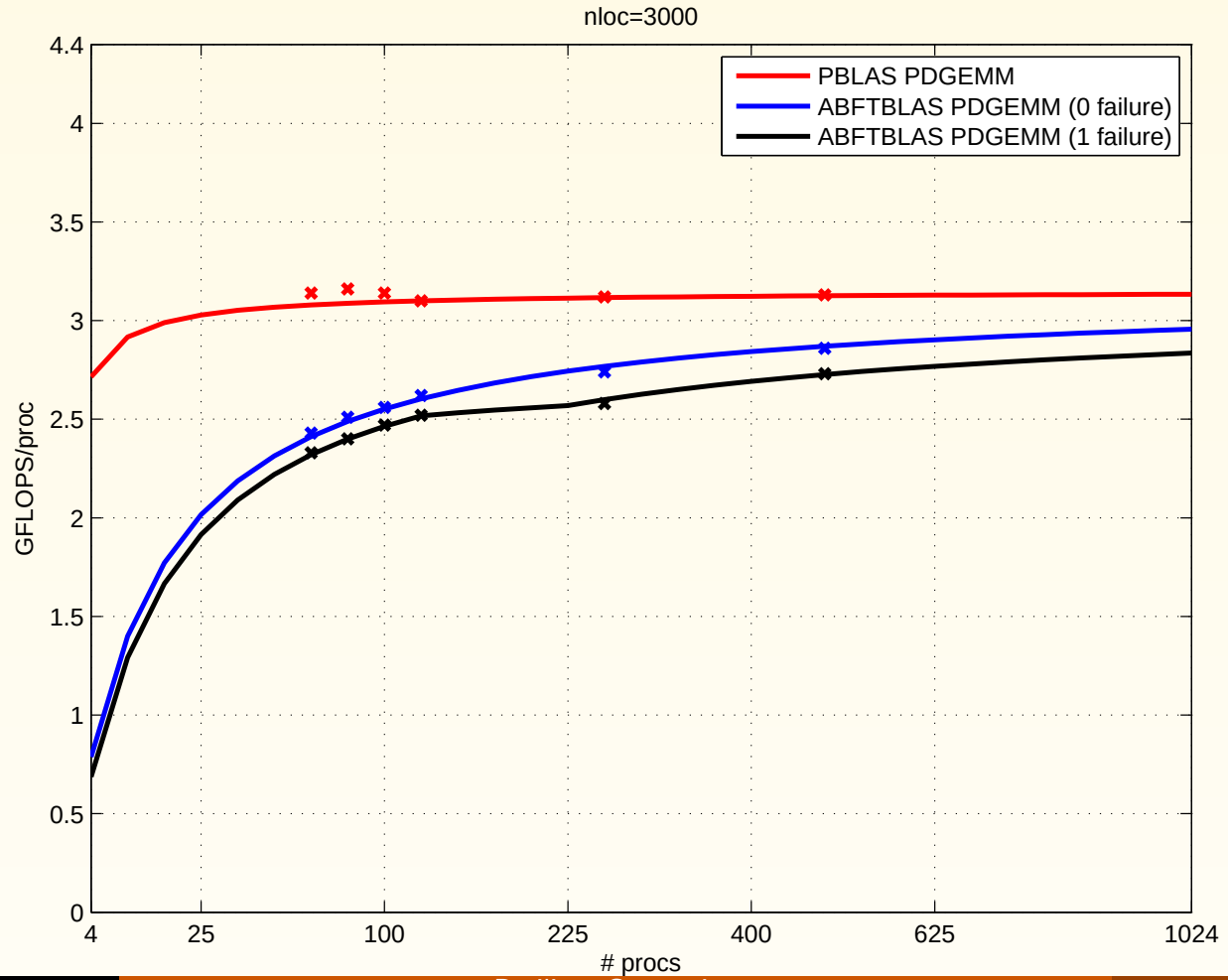
ABFT applied to DGEMM



Source: Bosilca, Delmas, Dongarra and Langou, 2008.

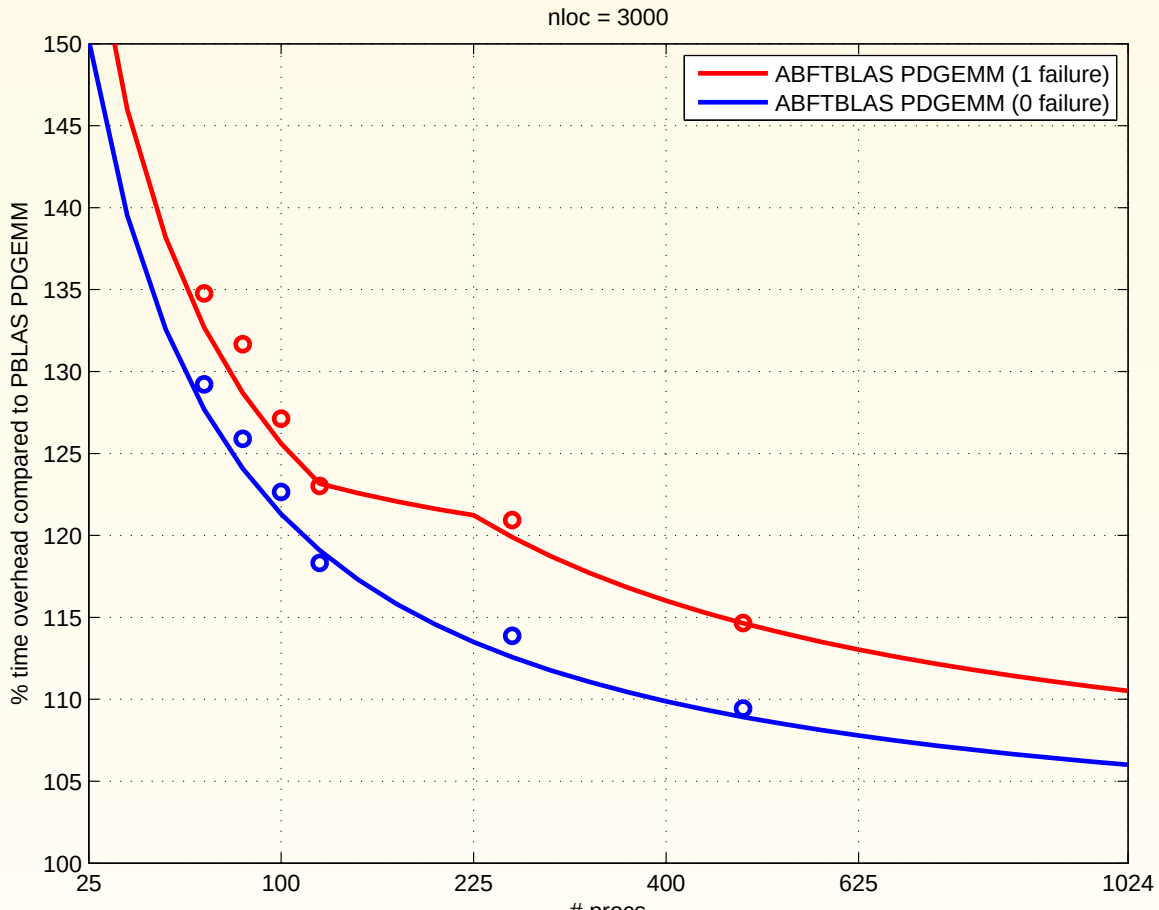
Performance Under Failure

Performance (GFLOPS/sec/proc) of PBLAS PDGEMM, ABFT BLAS PDGEMM (0 failure), and ABFT BLAS PDGEMM (1 failure)



Performance Overhead

Overhead of the fault tolerance with respect to the non-failure-resilient application PBLAS PDGEMM
The plain curves correspond to model while the circles correspond to experimental data



Control Flow Checking

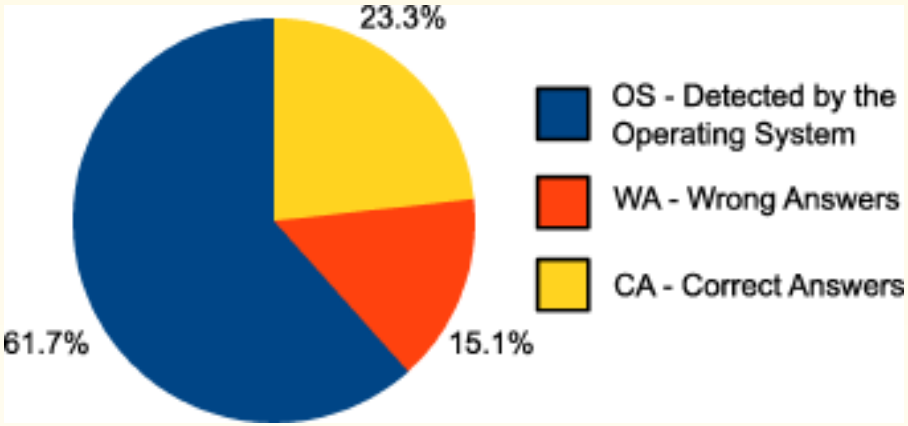
Technique that detects errors in computation due to failure in the underlying hardware by monitoring the execution sequence of a program and comparing that to allowable sequences defined by the system model
Program divided into loop-free intervals and code inserted for concurrent checking

Monitoring Embedded Signatures

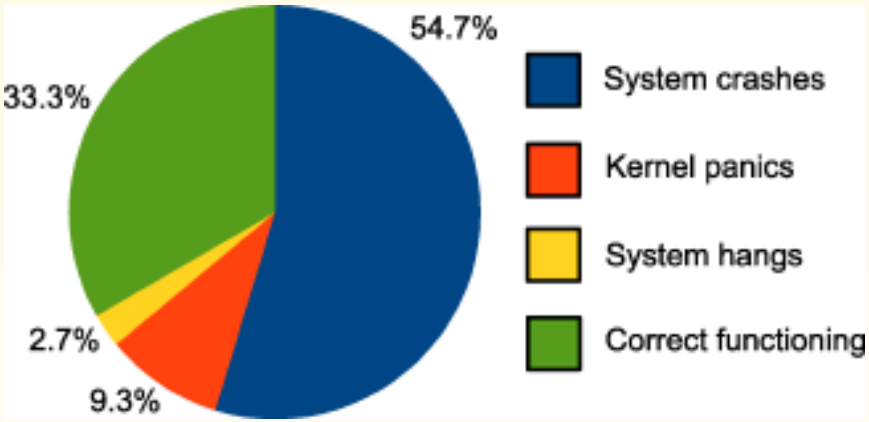
- Application program is partitioned, at assembly time, into a sequence of instructions (called segments) with one entry point and one exit point
- Signatures are generated (off-line) as a function of the sequence of the instructions within a segment
- A monitor (watchdog processor) uses pre-computed reference signatures to periodically check the operation

Effect of Control Flow Errors

On application software



On Linux OS



Detecting Control Flow Errors

Low-Cost Applications; example: Embedded, Mobile Systems

- Need to detect control-flow errors without high overhead in either
  - hardware or performance
- Supplement data checks

Control Flow Errors

- Execution of incorrect sequence of instructions
  - Can be caused by transient or permanent faults
  - 30% to 50% in RISC systems
- Detection methodologies
  - Hardware redundancy: high cost (not viable for low-cost, safety-critical applications
  - Software-based checking: high performance overhead at manageable cost levels

CEDA: Control-flow Error Detection through Assertions

Program represented as a control flow diagram

- Nodes: branch free instruction sequences
- Edges: represent legal execution sequences of nodes

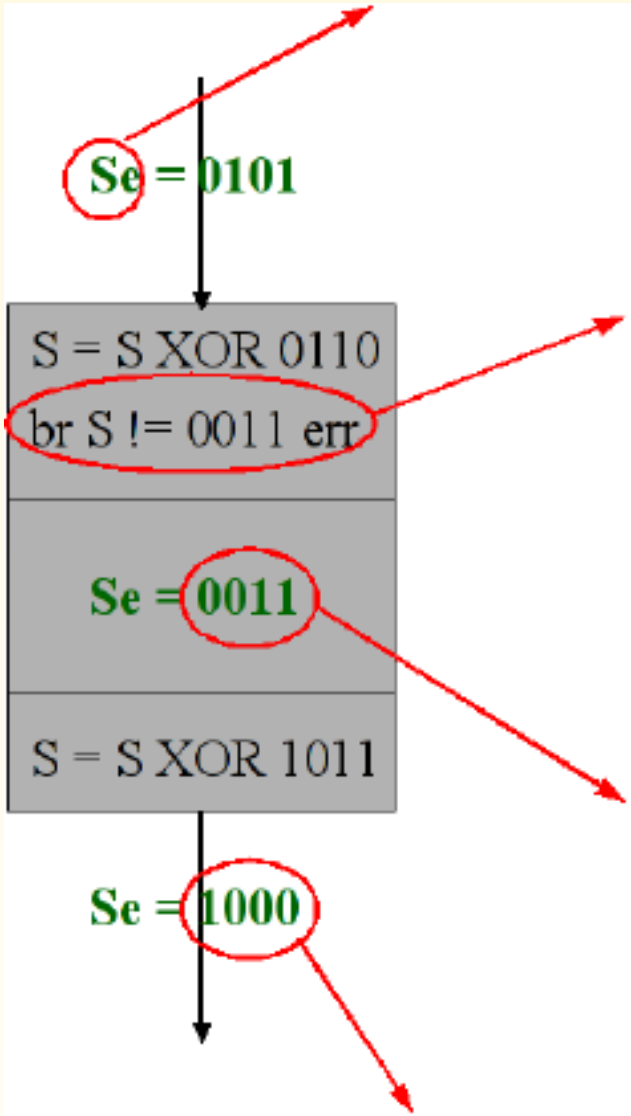
S, global runtime signature register

- Updated at the beginning and end of each node
- Each update either an XOR or an AND operation

Integration with GCC

- An additional pass
- Extra instructions automatically embedded within the program
  - Instructions for updating S
  - Instructions for checking the value of S

Overview of CEDA

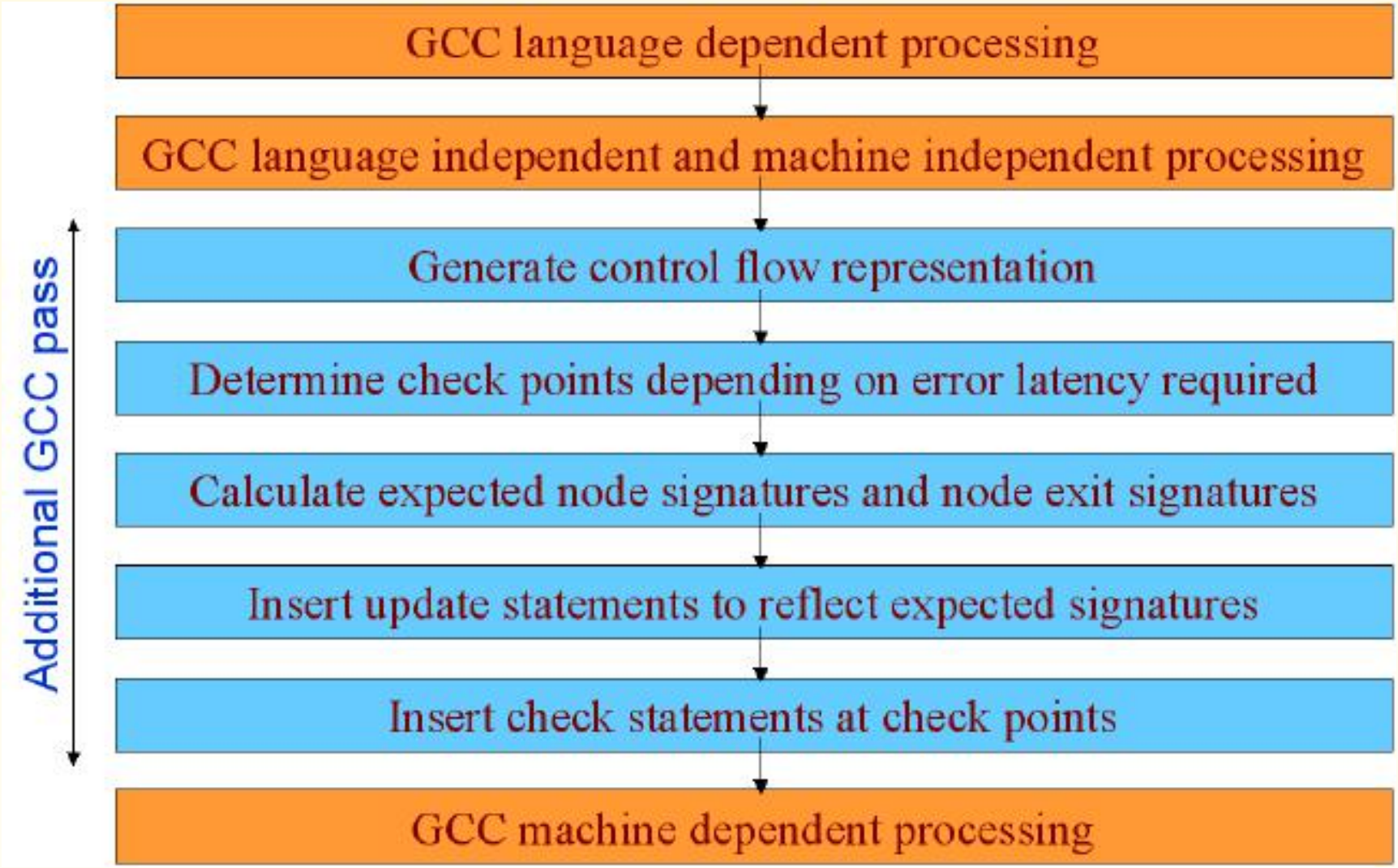


Se: expected value of S at each point in the program (calculated at compile time)

Check point: S is checked against its expected value (detects CFE if one occurred, not required inside every node)

Node signature: expected value of S inside a node

Integration with GCC

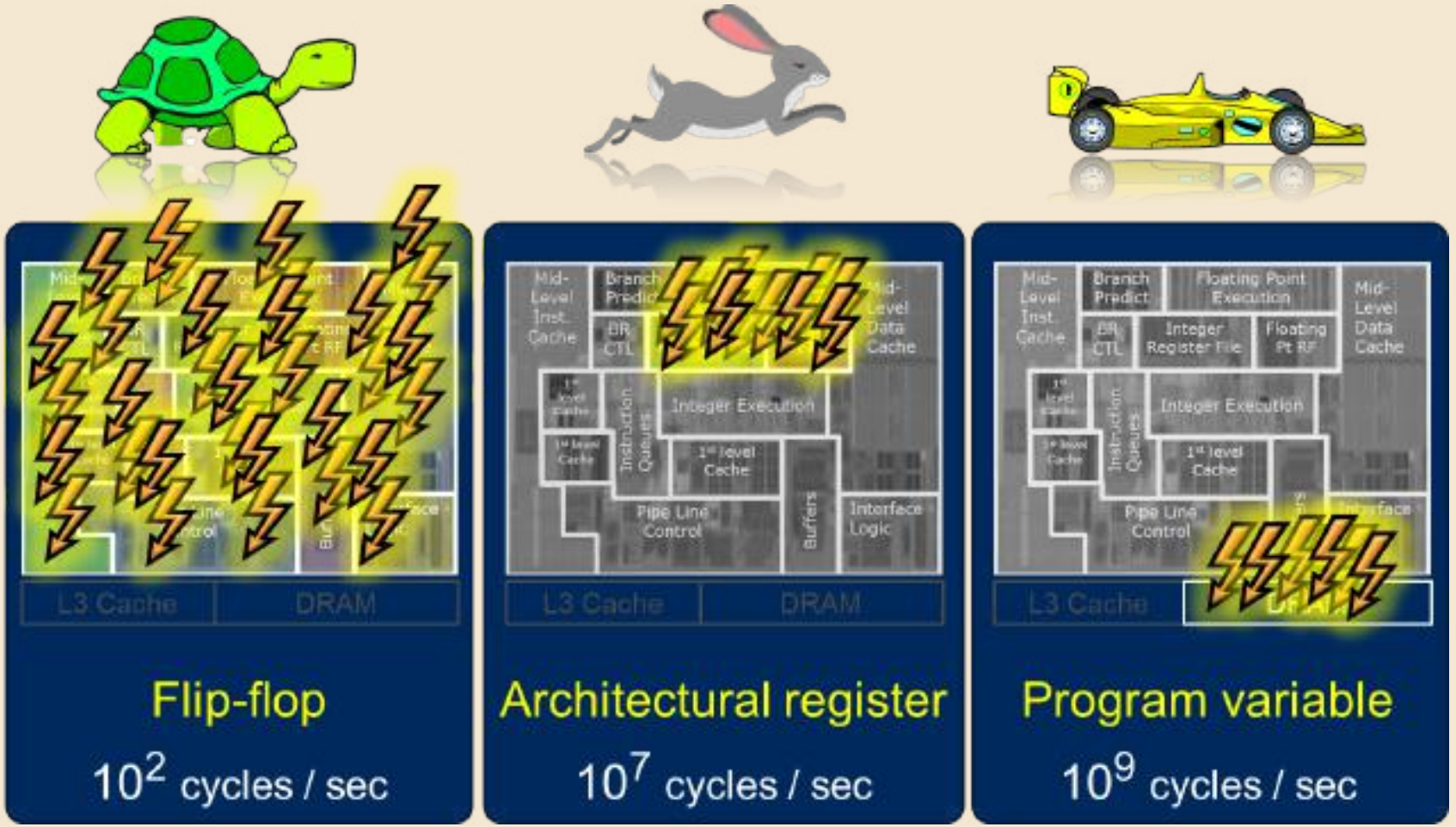


Hardware Support for Control Flow Checking

- Control flow checking can also detect errors caused by attacks
- Needs to be performance efficient
  - Necessary to be practically implementable
- Need to detect the errors early
  - Enables the processor to recover from the error
- Should be easily adaptable for legacy systems
  - Recompilation of software not possible for all applications

Error Injection Tradeoffs

Simulation speeds



Error Injection Study

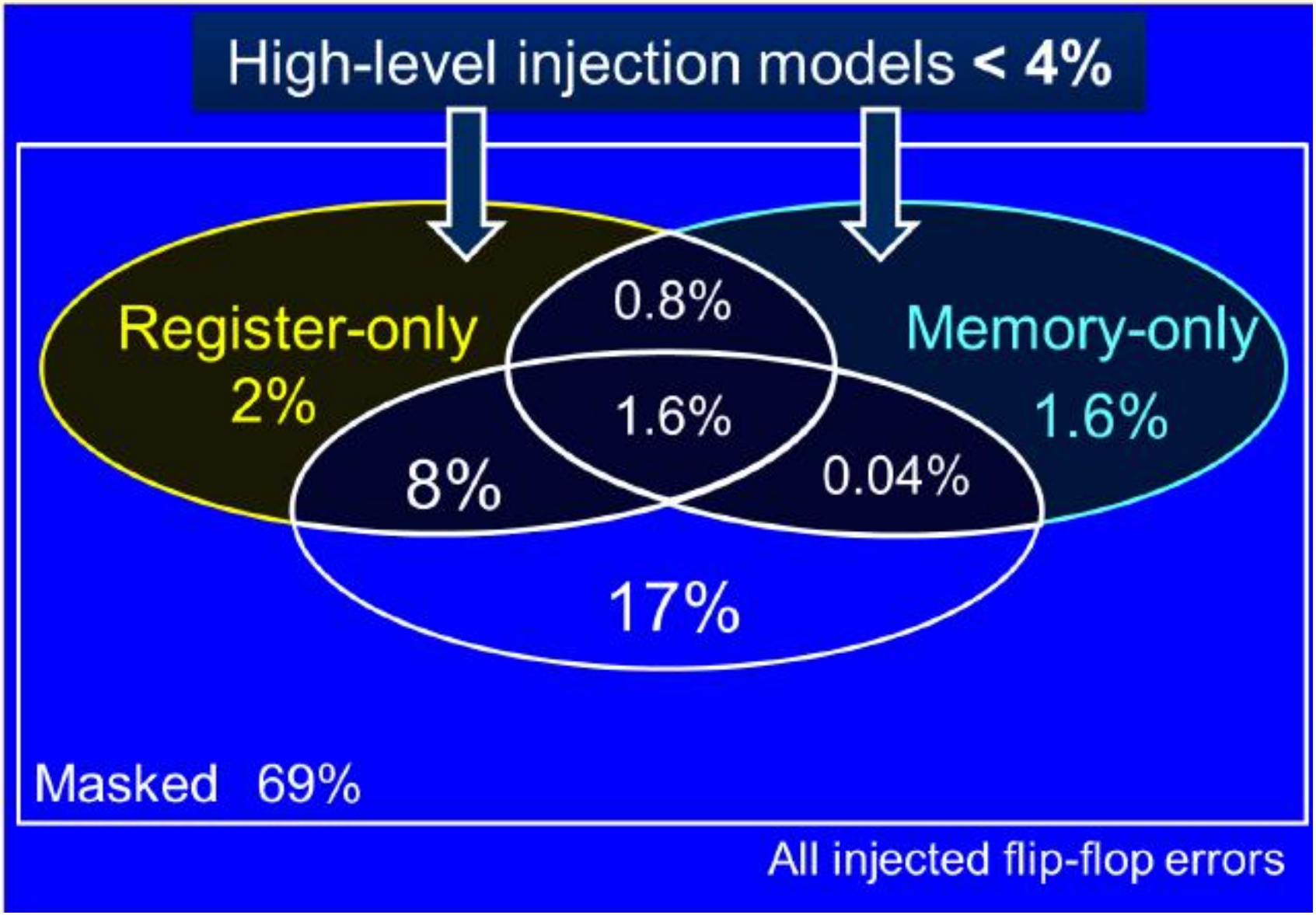
Joint research project, University of Texas at Austin and Stanford University

- Designs
  - LEON3 (in-order, single-issue)
  - ALPHA (out-of-order, superscalar)
- SPEC 2000 applications
- Bit flip in logic-level flip-flops
- 6 million error injection samples

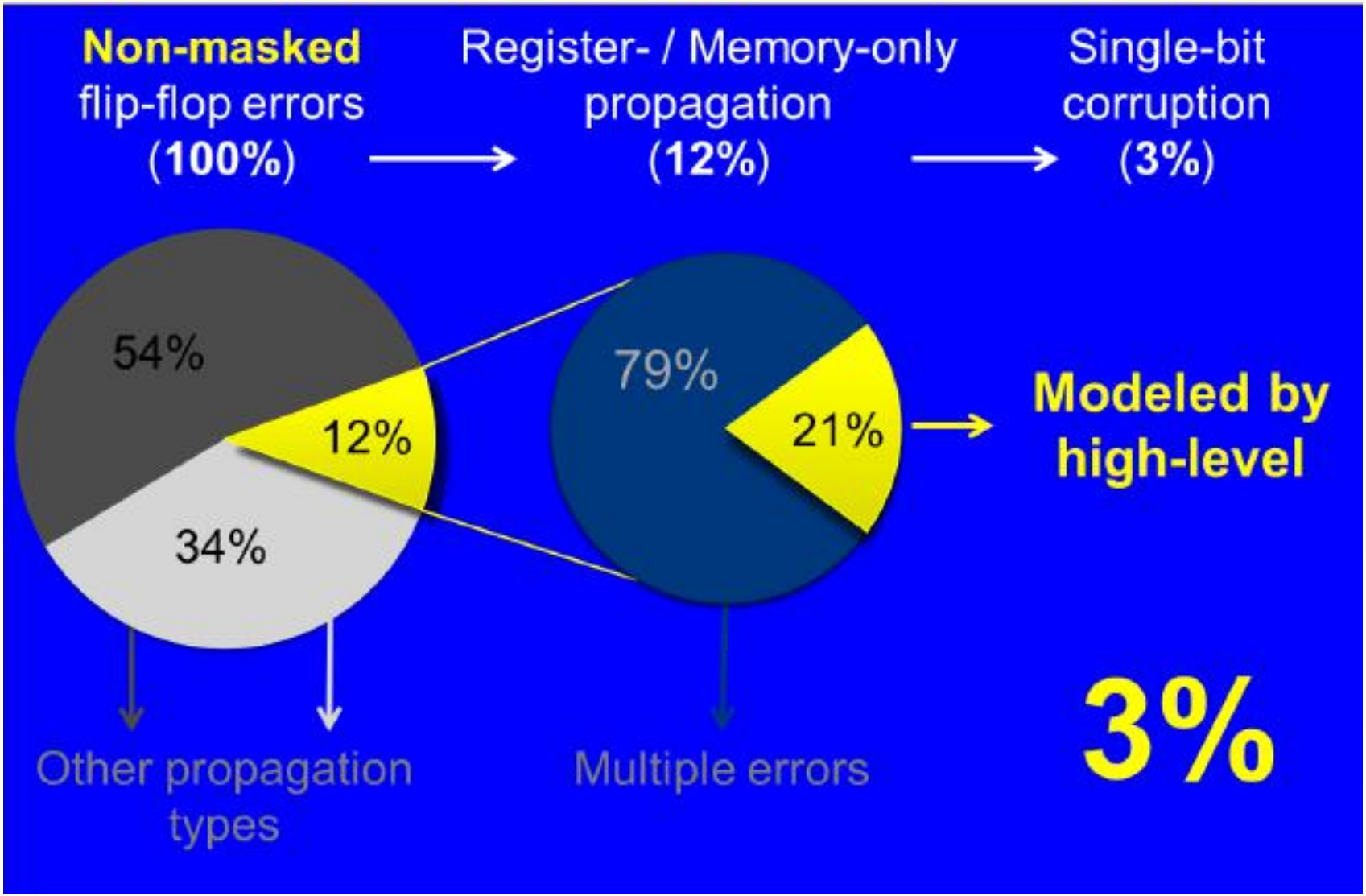
Outcomes

- Vanished
- **Output Mismatch**
- Unexpected Termination
- Hang

Tracking Flip-Flop Error Propagation



Limitations of High-Level Error Injection



Lessons Learned

High-Level Error Injections Inaccurate

- How inaccurate?
  - **Up to 45 X**
  - Neither optimistic or pessimistic
- Why inaccurate?
  - **Only 3% of flip-flop errors modeled**

Future directions

- Better high-level models
- Improved simulation techniques – hybrid, hierarchical
- Approximation of high-level error probabilities
- Beyond soft errors

FIESTA: Fault Injection for Embedded System Target Applications

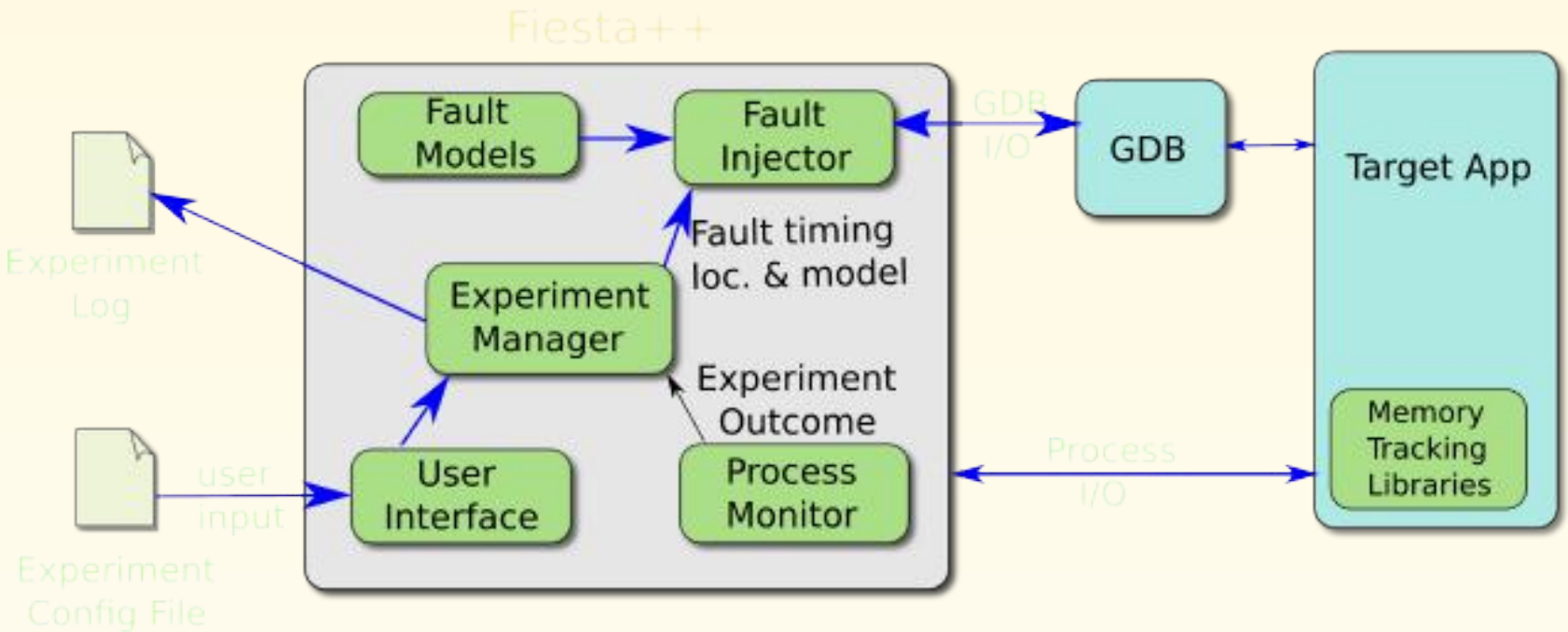
Evolution of FERRARI Tool

- **Emulating hardware faults using software**

Build fault injection on top of existing infrastructure

- Open architecture
- Support injection with "real-time" debuggers
- Initial system configuration
  - Host: Solaris 2.5.1, SUN 4
  - Target: VxWorks 5.3, MC68040
- Target embedded system applications (control computers)

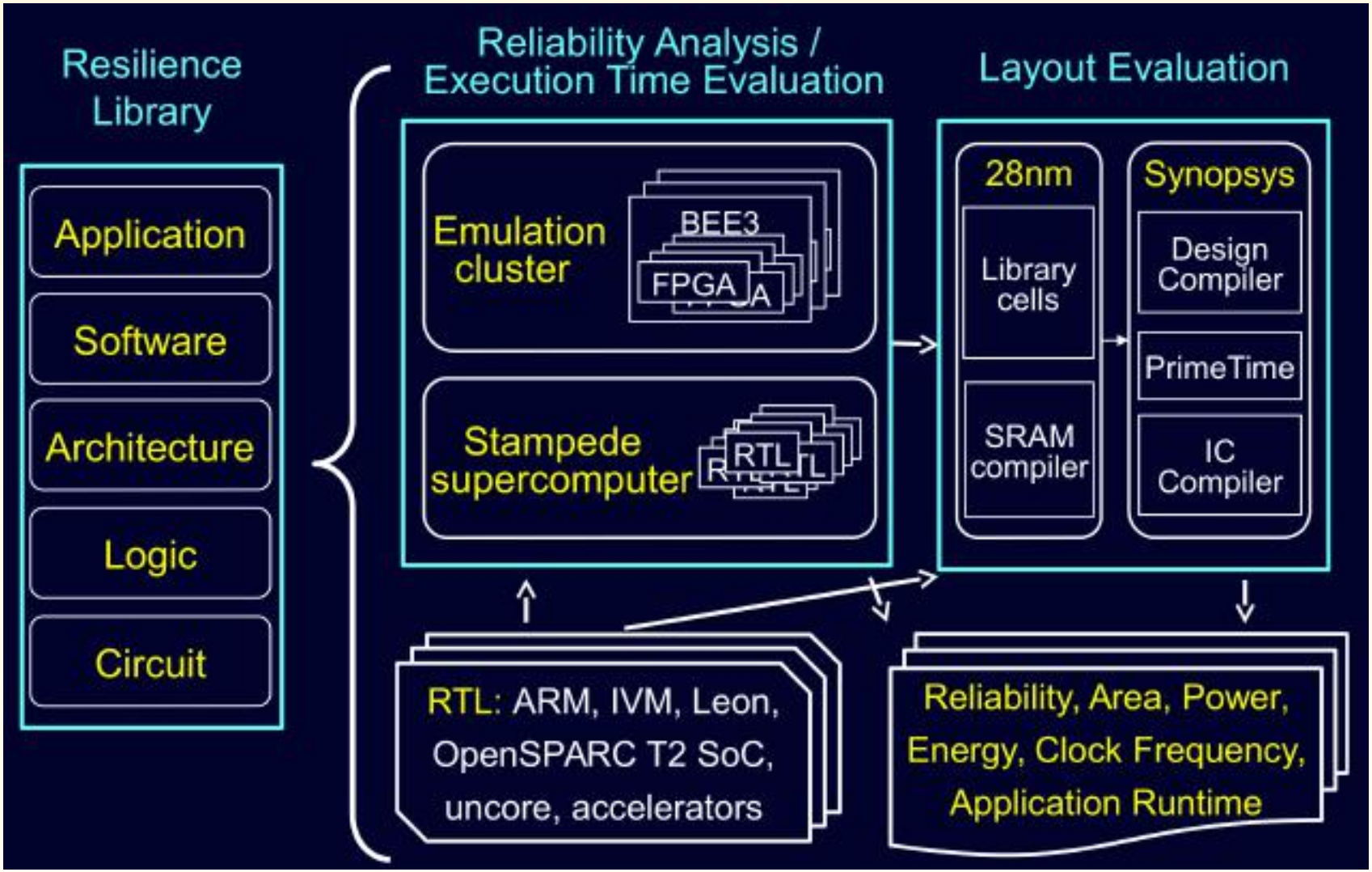
Fault/Error Injection Using FIESTA



Powerful tool

- Simple routines can be used to describe any desired fault or error
- Can even be used to model security attacks

CLEAR: Cross-Layer Exploration for Architecting Resilience



Extensive Study of an End-to-End Cross-Layer Approach to Resilience

Studied radiation-induced soft errors in flip-flops

- Single-Events – single-event upsets (SEUs) and single-event multiple upsets (SEMUs)
- Combinational logic soft errors not critical

Designs studied

- ARM, LEON3, Alpha, OpenSparc multi-core SoC, accelerators

Thorough flip-flop error injections

- FPGA clusters, Stampede supercomputer (522,000 cores)
- Full workloads (Spec, Parsec, Perfect, proprietary)
- Detailed physical design – wire routing, process/voltage/temperature corners

Representative Resilience Techniques

1. Algorithm Based Fault Tolerance (ABFT) Correction
2. ABFT Detection

3. Software Assertions
4. Control Flow Checking by Software Signatures (CFCSS)
5. Error Detection by Duplicated Instructions (EDDI)

6. Data Flow Checking (DFC)
7. Monitor Cores

8. Logic Parity

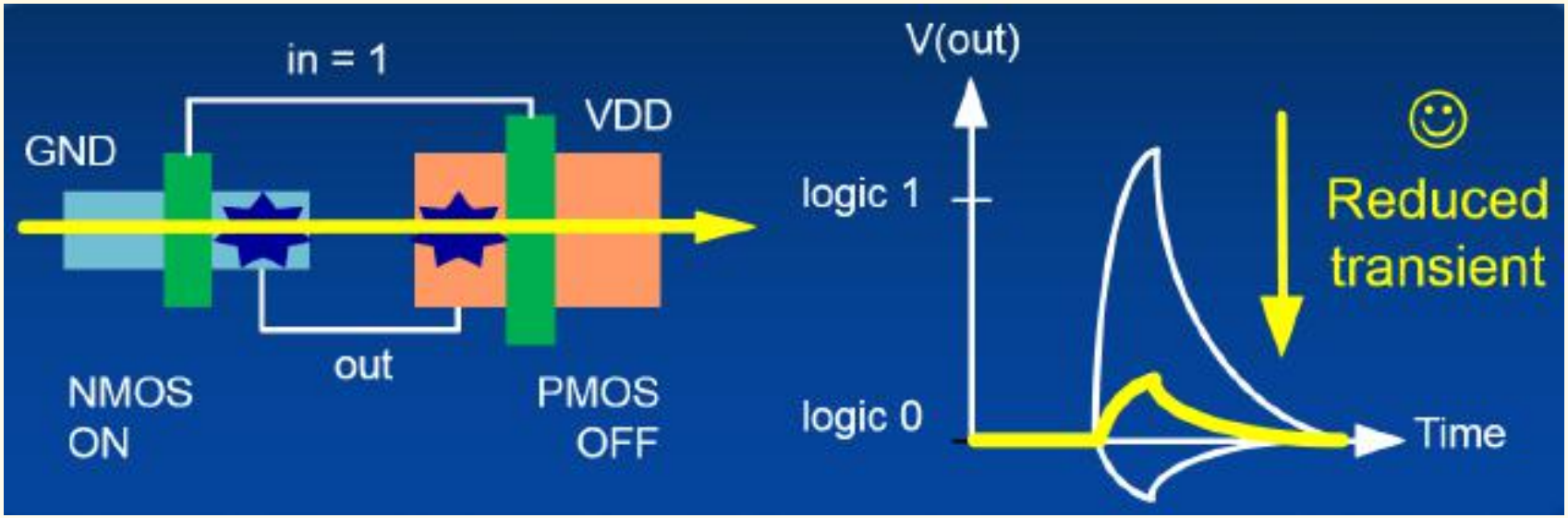
Micro-arch. Recovery
1. Flush
2. Reorder Buffer (RoB)
3. Instruction Replay (IR)
4. Extended IR (EIR)

9. Layout design through Error-Aware transistor Positioning (LEAP)
10. Error Detection Sequential (EDS)

37

Effective Circuit-Level Resilience

LEAP: Layout design through Error-Aware transistor positioning – Corrects SEUs and SEMUs



Technique evaluated through radiation-beam experiments

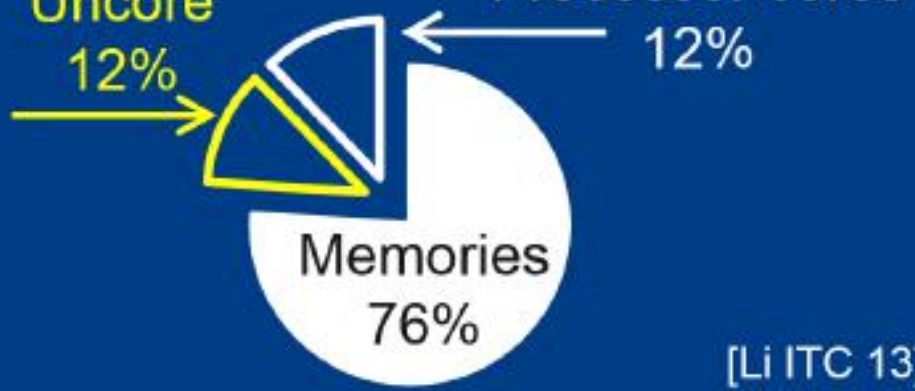
- 40nm, 28nm, 20nm, 14nm (Bulk and SOI)
- VDD: nominal, near-threshold

LEAP-DICE: 2×10^{-4} **lower soft error rate for 2X area and 1.8X power**

What About Uncore Components?

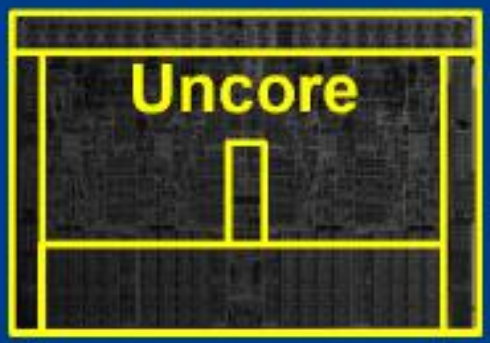
OpenSPARC T2 SoC

Area



[Li ITC 13]

Intel i7 quad-core SoC



Power

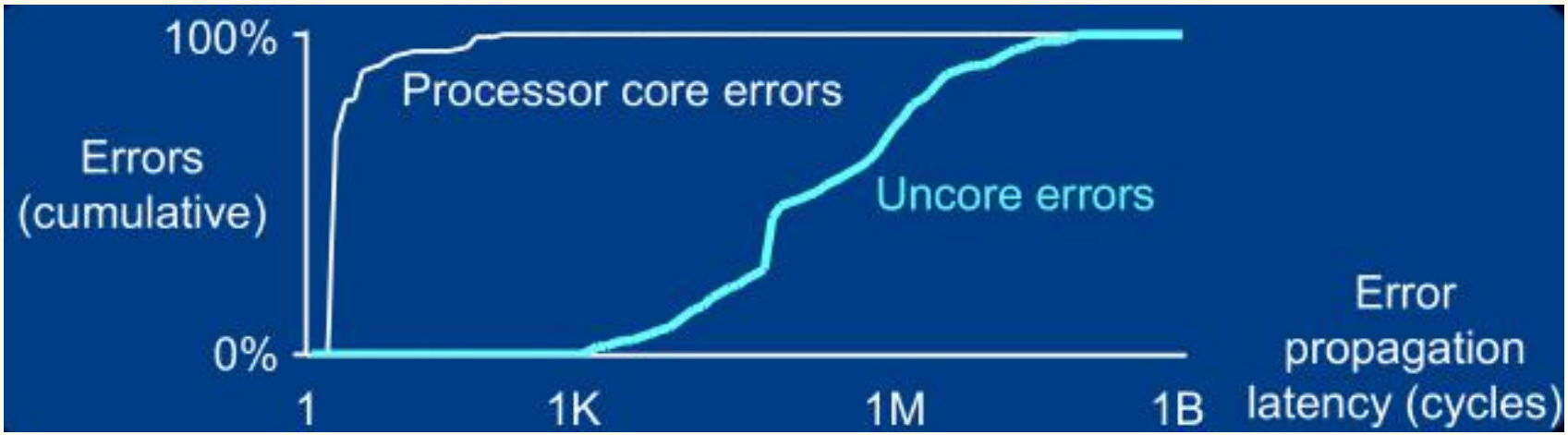
Processor cores 60.2%	Uncore 39.8%

[Gupta USENIX 12]

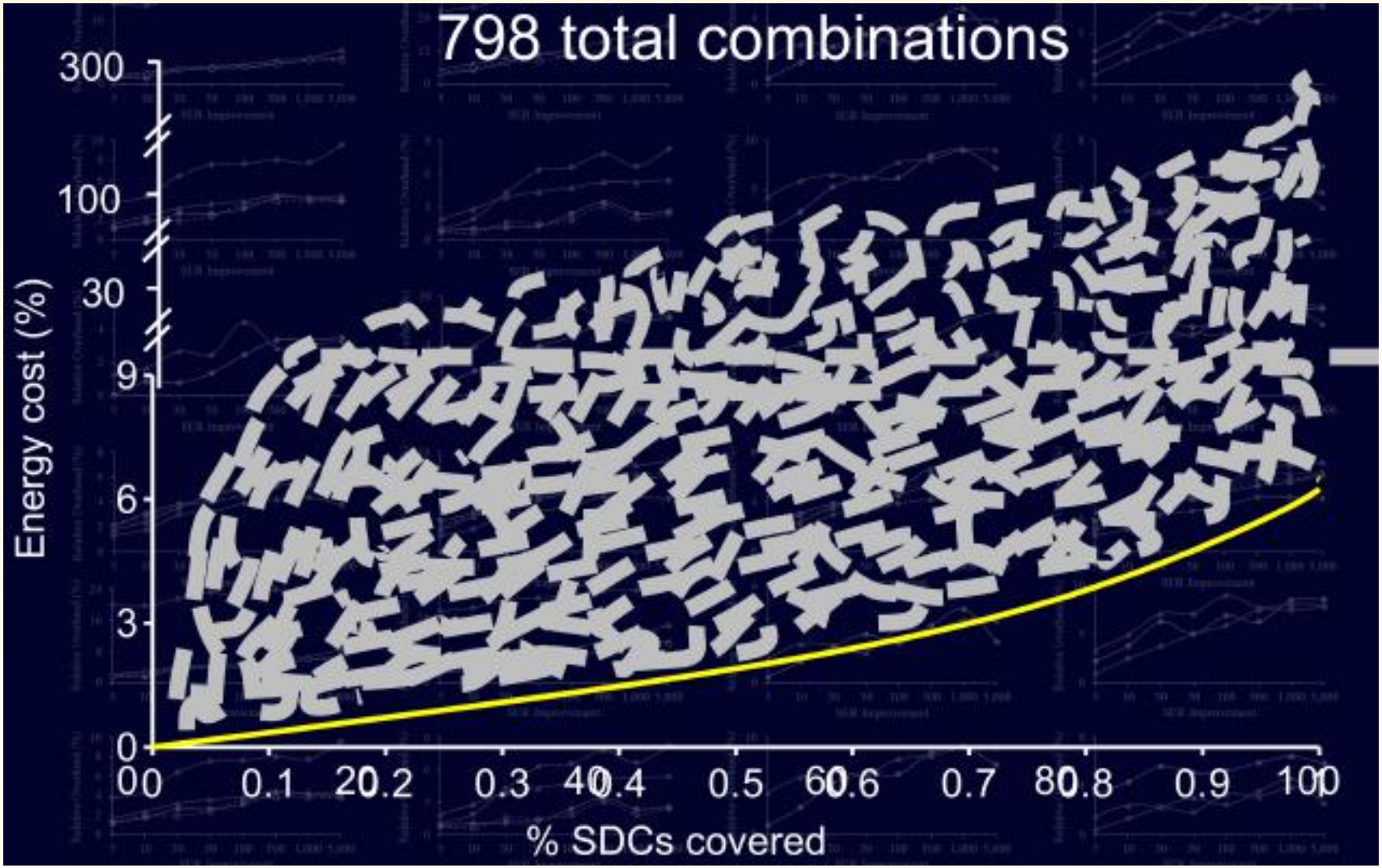
Uncore Soft Errors

New fast and accurate error injection technique: 20,000X speedup vs. RTL

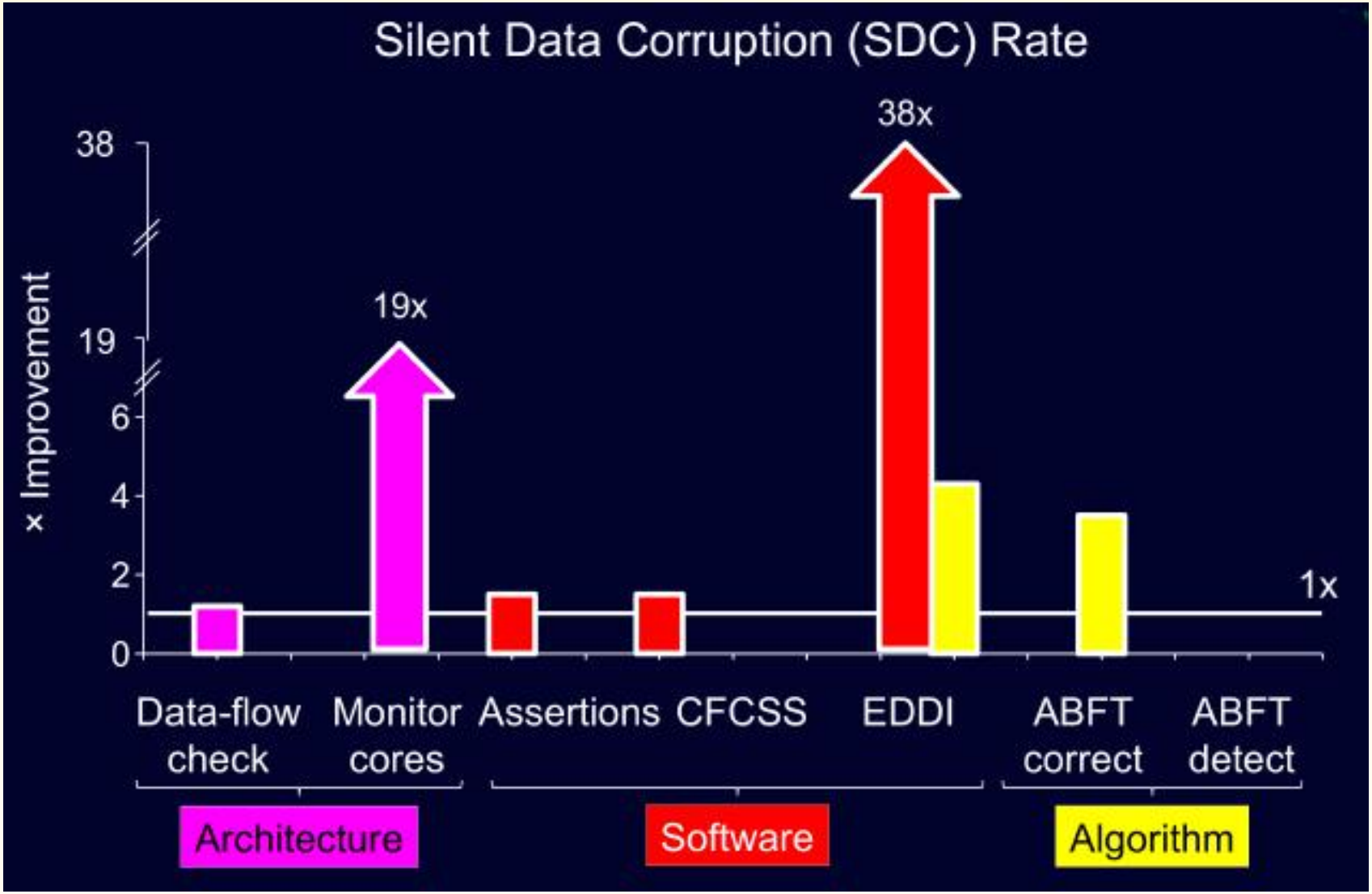
Reliability impact: uncore $\approx$ processor cores, but long error propagation latency



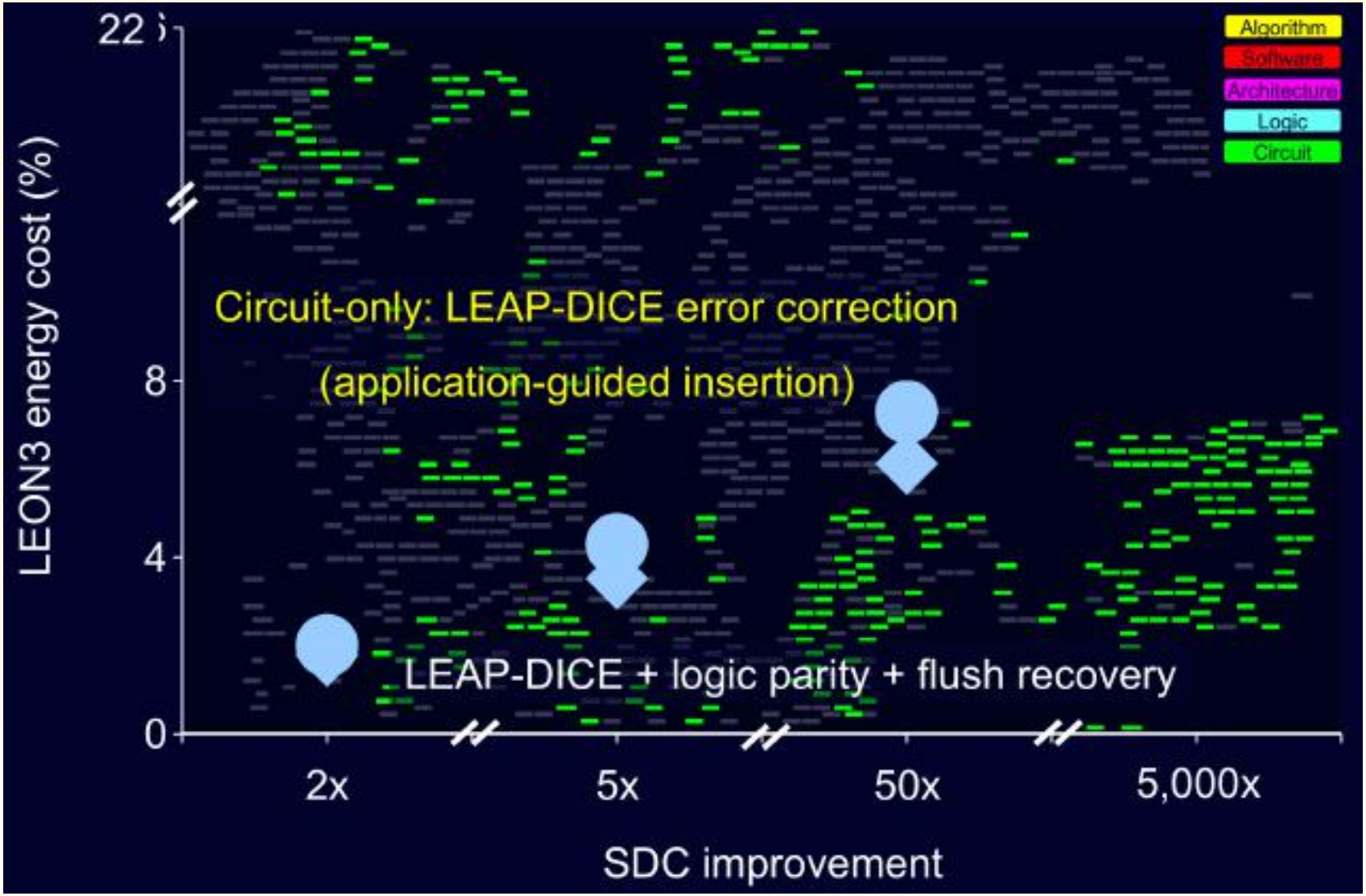
Lots of CLEAR Results



Are High-Level Techniques Sufficient?



Circuit Only (Application-Guided): Highly Effective



Other Issues Affecting Resilience

Test Escapes

- Can we guarantee extremely high quality in 2 Billion transistor chips?

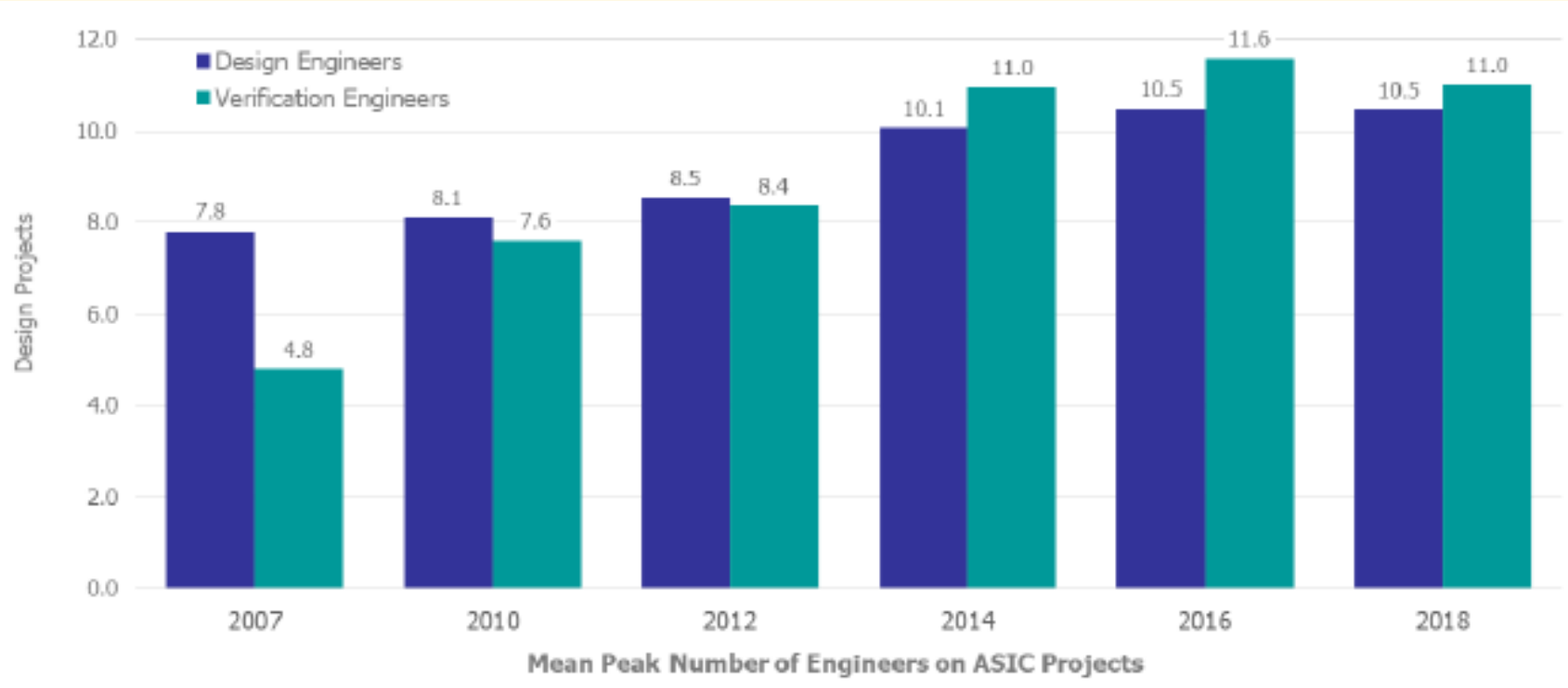
Logic bugs

- Verification is dominating the design cycle
- Unlikely that all design bugs are caught before deployment
- Diversity is necessary to deal with design bugs

Design margins

- Effects of real bugs are not easy to duplicate (in many cases, error latencies of many millions (or billions) of cycles)
- Gray: concepts of **Bohr bugs** (repeatable) versus **Heisenbugs** (not seen to be repeatable)

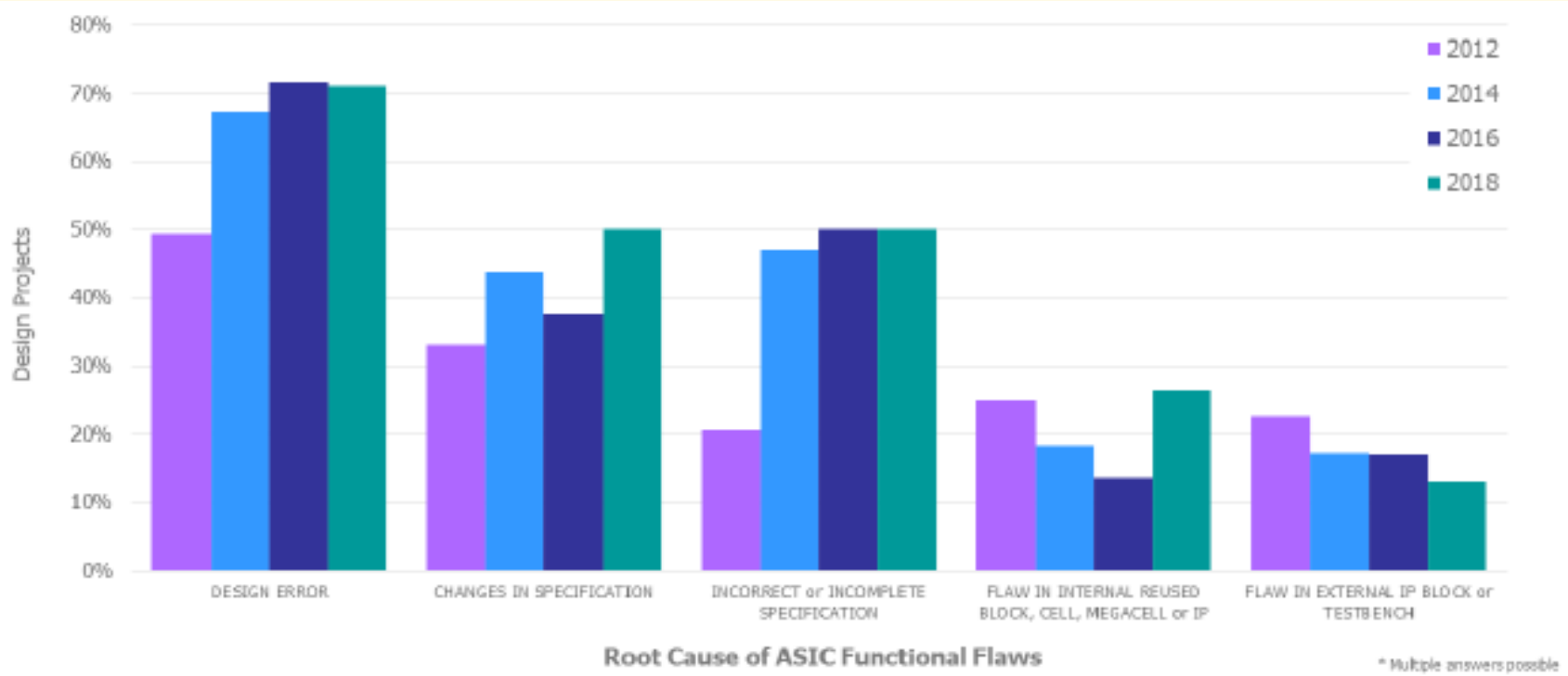
Mean Peak Number of Engineers on ASIC Projects



More verification engineers than designers!

Source: Wilson Research Group and Mentor, 2018 Functional Verification Study

Root Cause of Functional Flaws in ASICs



Source: Wilson Research Group and Mentor, 2018 Functional Verification Study

Dealing with Security – Very Different From Dealing with Physical Faults or Errors

Attacks are **Intentional**

- Faults and Errors related to design or physical causes are **systematic** **or** **random**
- Attacks are **deliberate**
  - Initiated by a clever adversary

Security Attacks

Hardware Trojans

- Malicious modification of designs
- Example of analog circuitry modifying a digital chip – extremely difficult to identify
- Design diversity may be a solution

External attacks

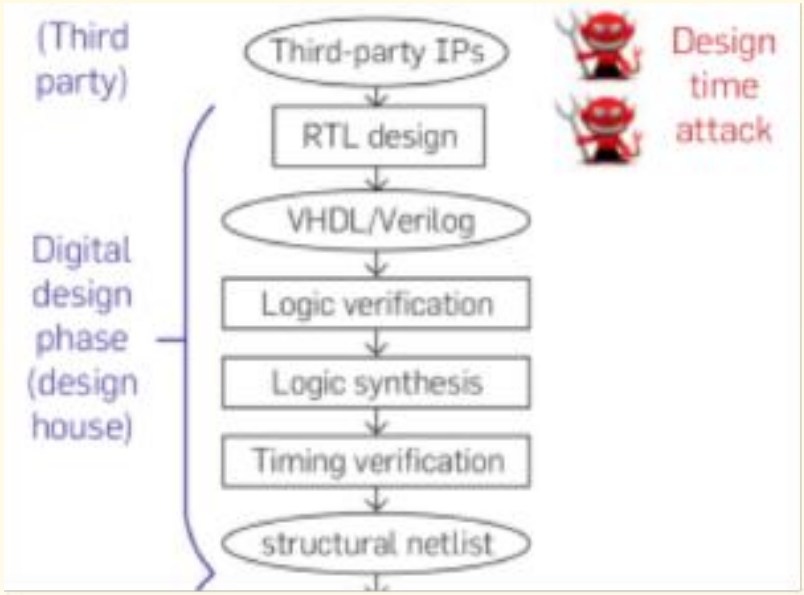
- Classic work (Abadi) suggested control flow checking to detect execution of undesired code
- Effects of attacks could include modification of data, execution sequences, denial of service, etc.
  - Require data checks in addition to control-flow checks
  - Need to detect DoS attacks during operation – example, shutting down GPS system (or spoofing GPS position)

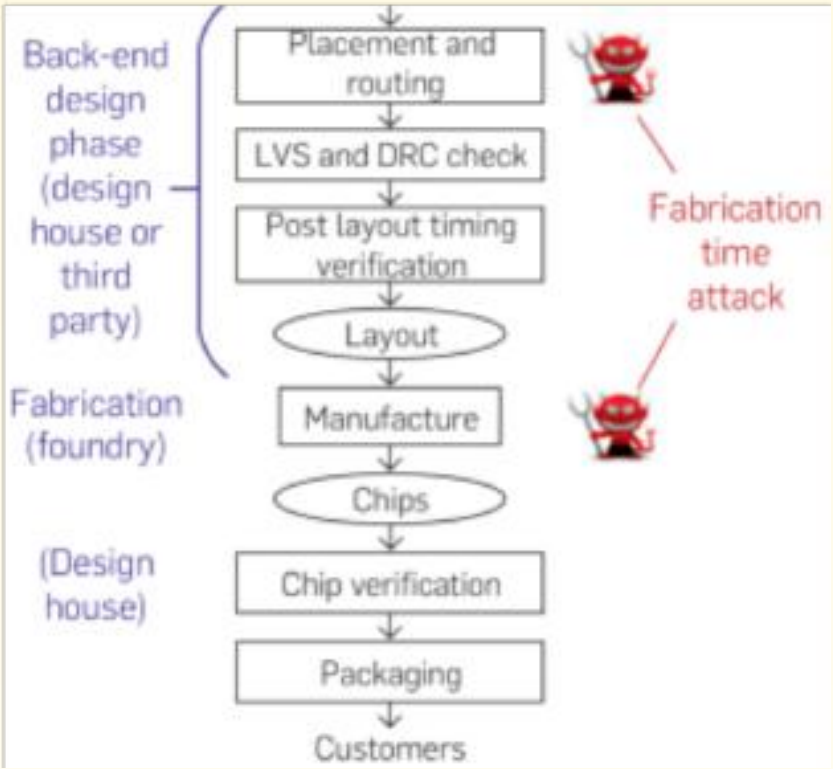
Example: Attacks on Automobiles

Attacks on CAN bus

- Allows the attacker to control the operation of an automobile over a WiFi network
- Attack was facilitated by the sharing of critical functions with automobile entertainment system

IC Design Process with Possible Attacks



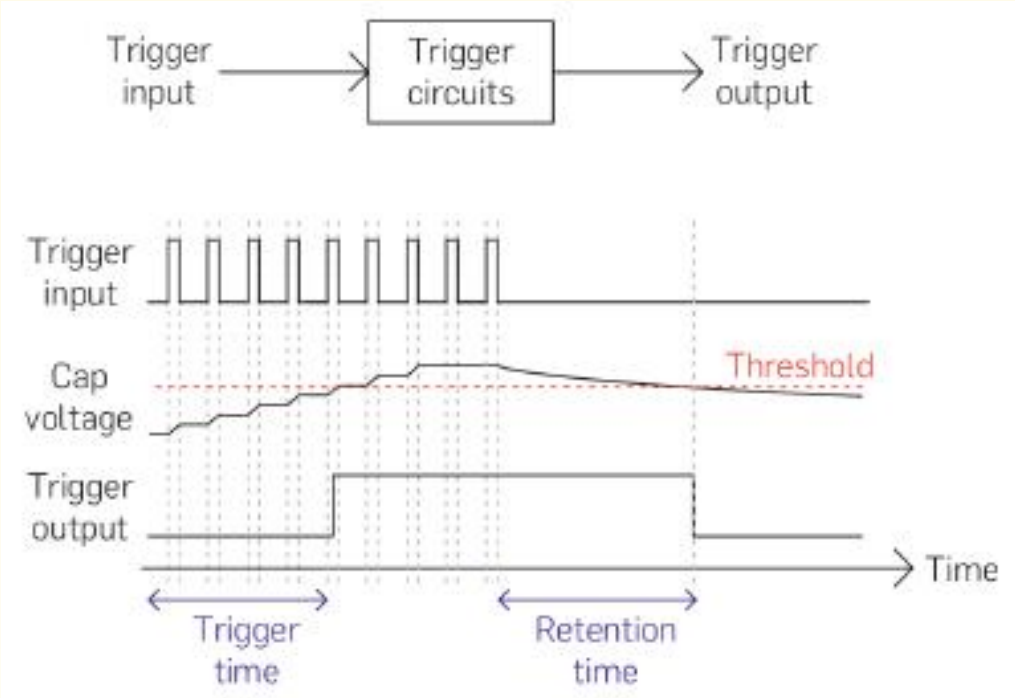


Source: Yang et. al, Communications of the ACM, September 2017.

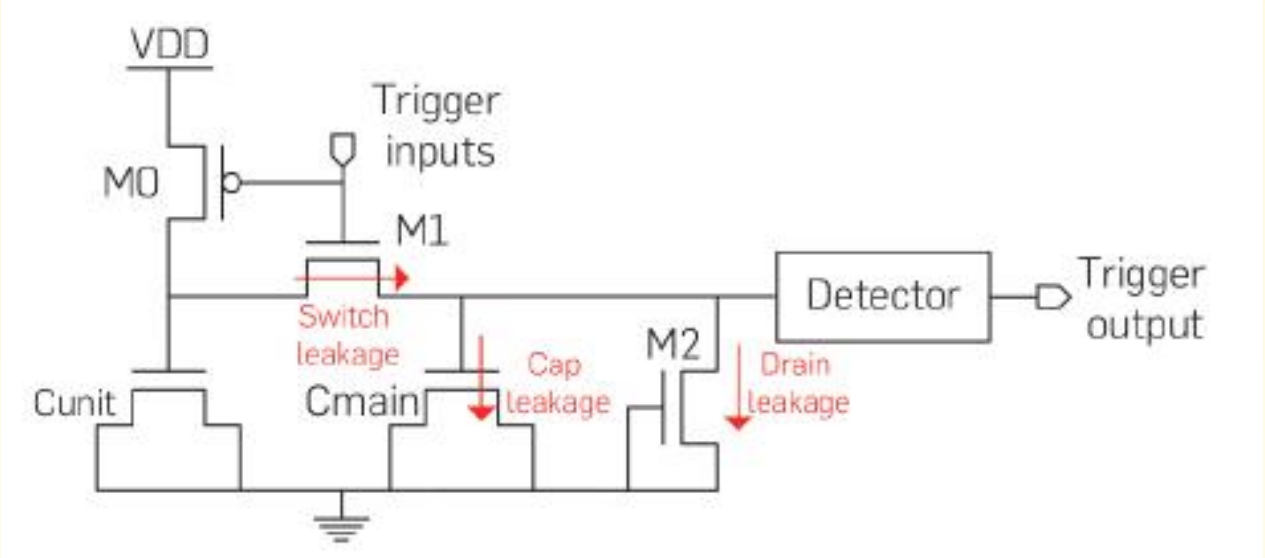
Trojans Could be Extremely Difficult to Detect

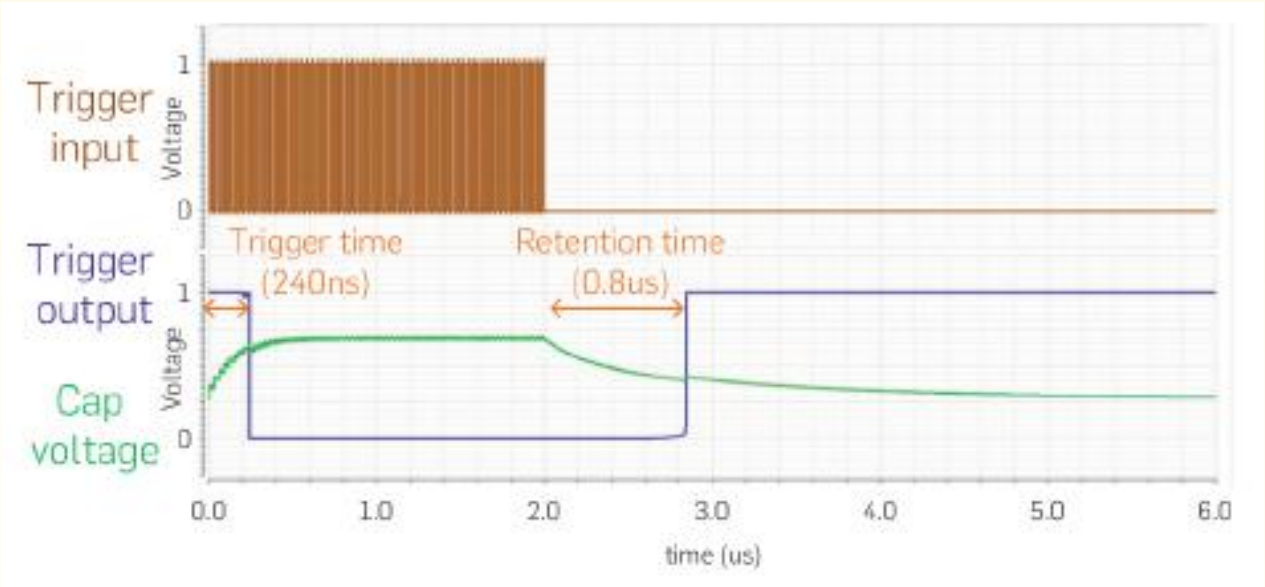
Analog Trojan in a Digital System

- Fabrication-time attack with trigger in the analog domain
- Based on charge accumulating on a capacitor from infrequent events inside the processor
- Very small area, and low impact on power and timing



Schematics and Simulation of Analog Trigger Circuit





Evaluation

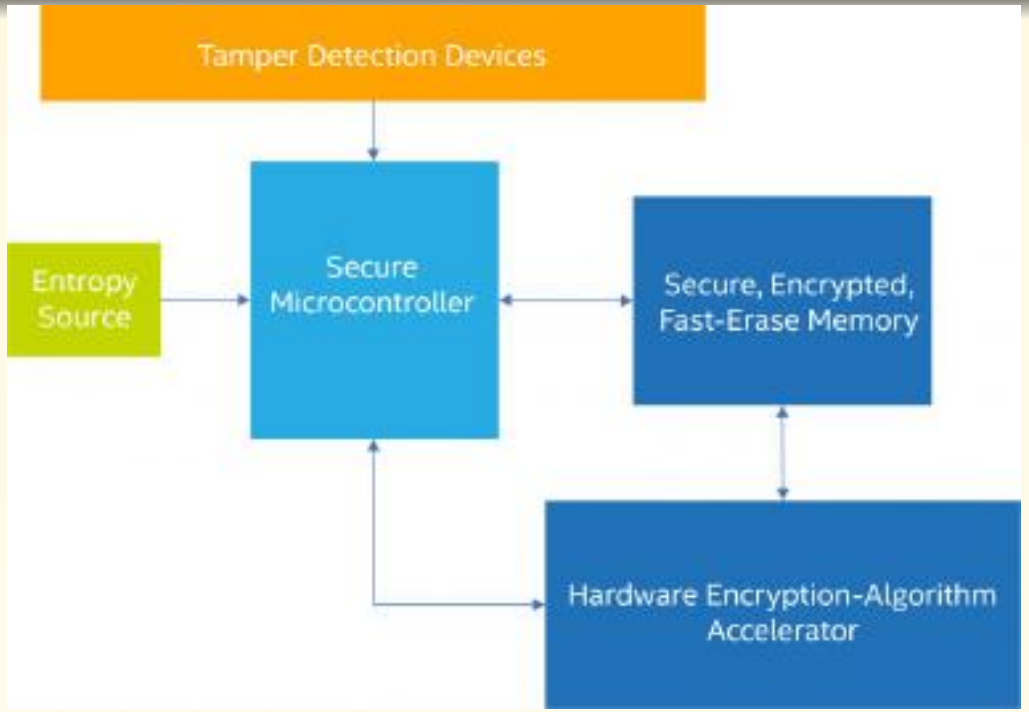
OR1200 Processor

- OR1200 core with 128 B instruction cache and an embedded 128 KB main program memory connected through a Wishbone bus
- Supervision Register controls MMU, flags, etc.
- SR[0] controls the privilege mode (0=user, 1=supervisor)
- Attack can escalate a user mode process to supervisor mode

Inserting Trigger into OR1200 Processor

- Even with 80% area utilization, it was easy to insert the attack
- Very low overhead for circuit
- Post-layout simulation shows the extra delay of victim wires is only 33% of the 4 ns clock period

Hardware-Based Security for Embedded Computing



Hardware Security Module

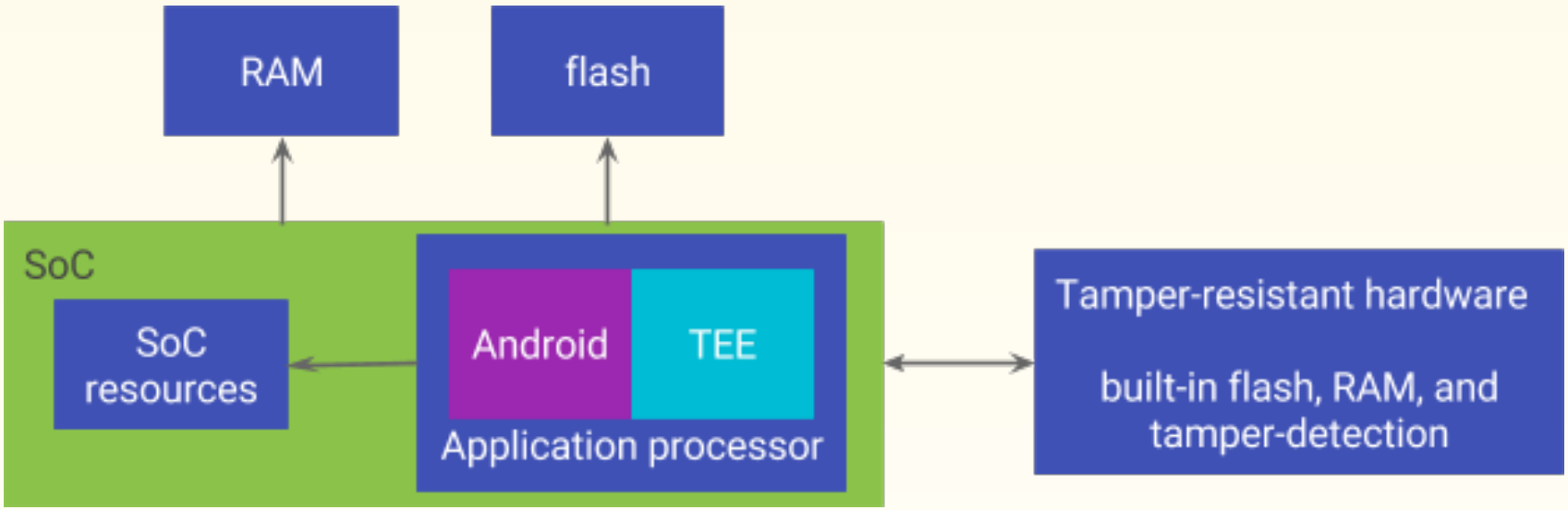
- Tamper resistant, with motion, capacitive, radiation, voltage and temperature sensors
- Secure microcontroller with secure memory for key storage

Source: Intel, "Embedded Computing Needs Hardware-Based Security,"

Android Pixel 2 Security Module

Tamper Resistant Hardware

- Discrete chip separate from the SoC, with its own flash, RAM
- Can control its own execution, and is robust against side channel information leakage attacks
- Loads its OS and software directly from internal ROM and flash, and controls all updates
- Resilient against fault injection and side channel attacks



Control Flow Deviation Detection for Application Level Security

Attacks subvert the control flow of the software

- Insert control-flow checks in the code (particularly useful for embedded software)
- Run-time signatures and checks can be inserted automatically during compile time

Implementation

- Signature update instructions inserted at the beginning and end of each function, as well as before and after the call instructions
- Illegal branches will result in signature mismatches

Proposed in 2005 (Abadi)

Example – Detection of Illegal Jump

Pre-computed signatures

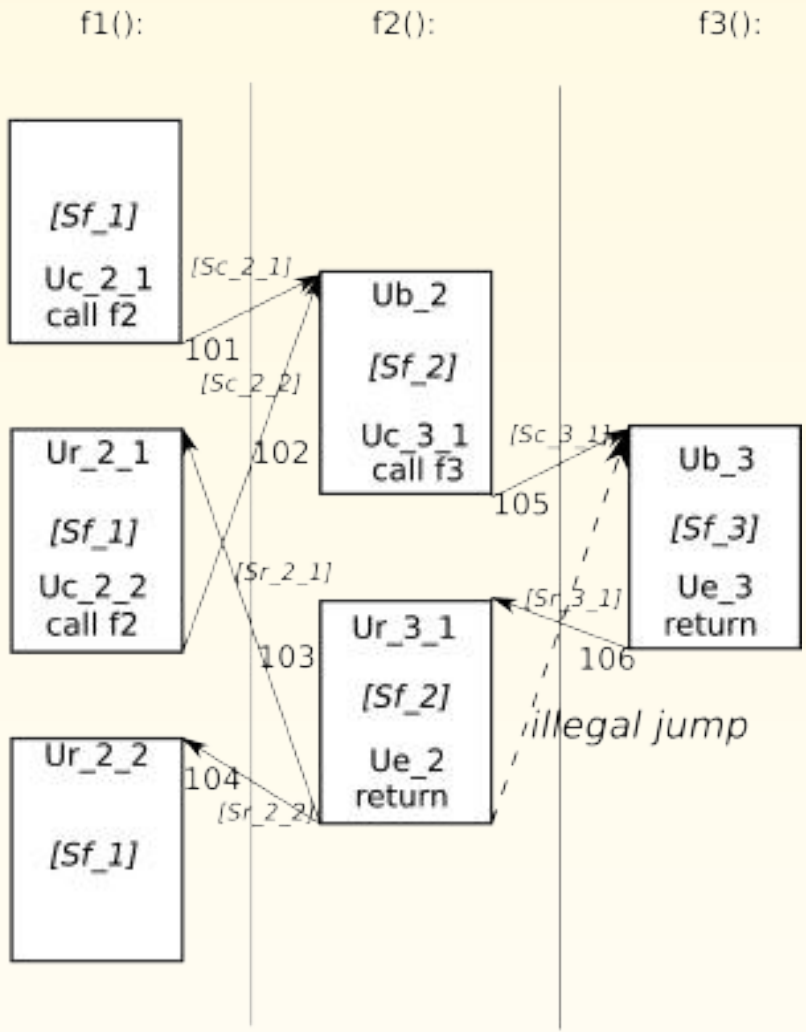
Sf_1 = 000
Sf_2 = 001
Sf_3 = 010
Sc_2_1 = 011
Sc_2_2 = 011
Sr_2_1 = 100
Sr_2_2 = 100
Sc_3_1 = 101
Sr_3_1 = 110

Update instructions

Ub_2: S = S XOR 010
Ue_2: S = S XOR 101
Ub_3: S = S XOR 111
Ue_3: S = S XOR 100
Uc_2_1: S = S XOR 011
Uc_2_2: S = S XOR 011
Ur_2_1: S = S XOR 100
Ur_2_2: S = S XOR 100
Uc_3_1: S = S XOR 100
Ur_3_1: S = S XOR 111

Detection of illegal jump

In case of illegal jump,
S inside f3 = Ub_3(Ue_2(Sf_2))
= 001 XOR 101 XOR 111
= 011
S inside F3 != Sf_3



Problems with Verifying Control Flow Integrity (CFI) in Software

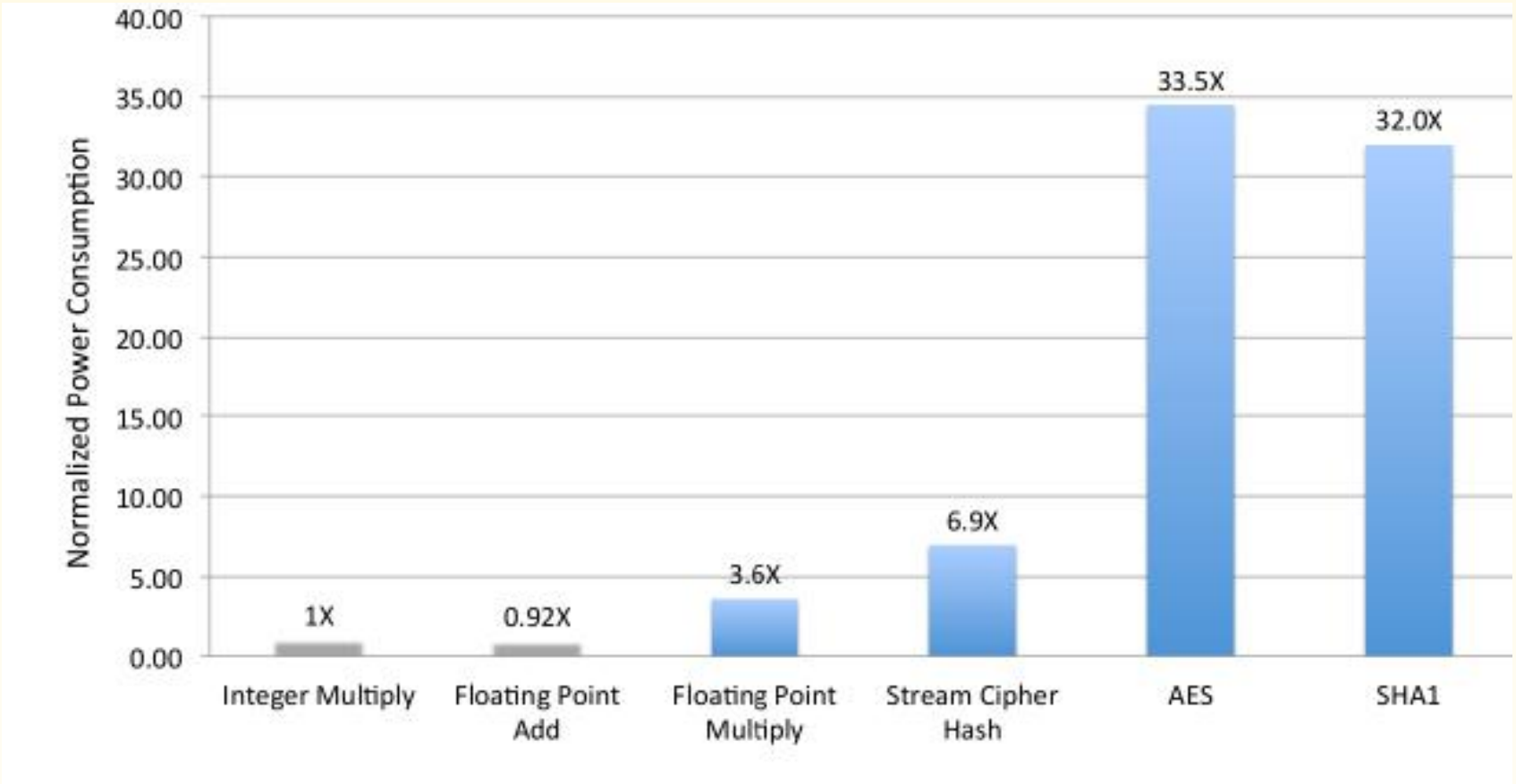
Security Holes

- Using fixed and unique labels
- Added instructions to check targets of register-based control flow rely on Data Execution Protection (W$\oplus$X mechanism)

Overhead

- Performance overhead could be 20–200%
- Power Consumption, particularly for cryptographic operations

Normalized Power Consumption of Cryptographic Hash Algorithms Compared with 16-bit Integer Multiplication



Approach for Control Flow Checking

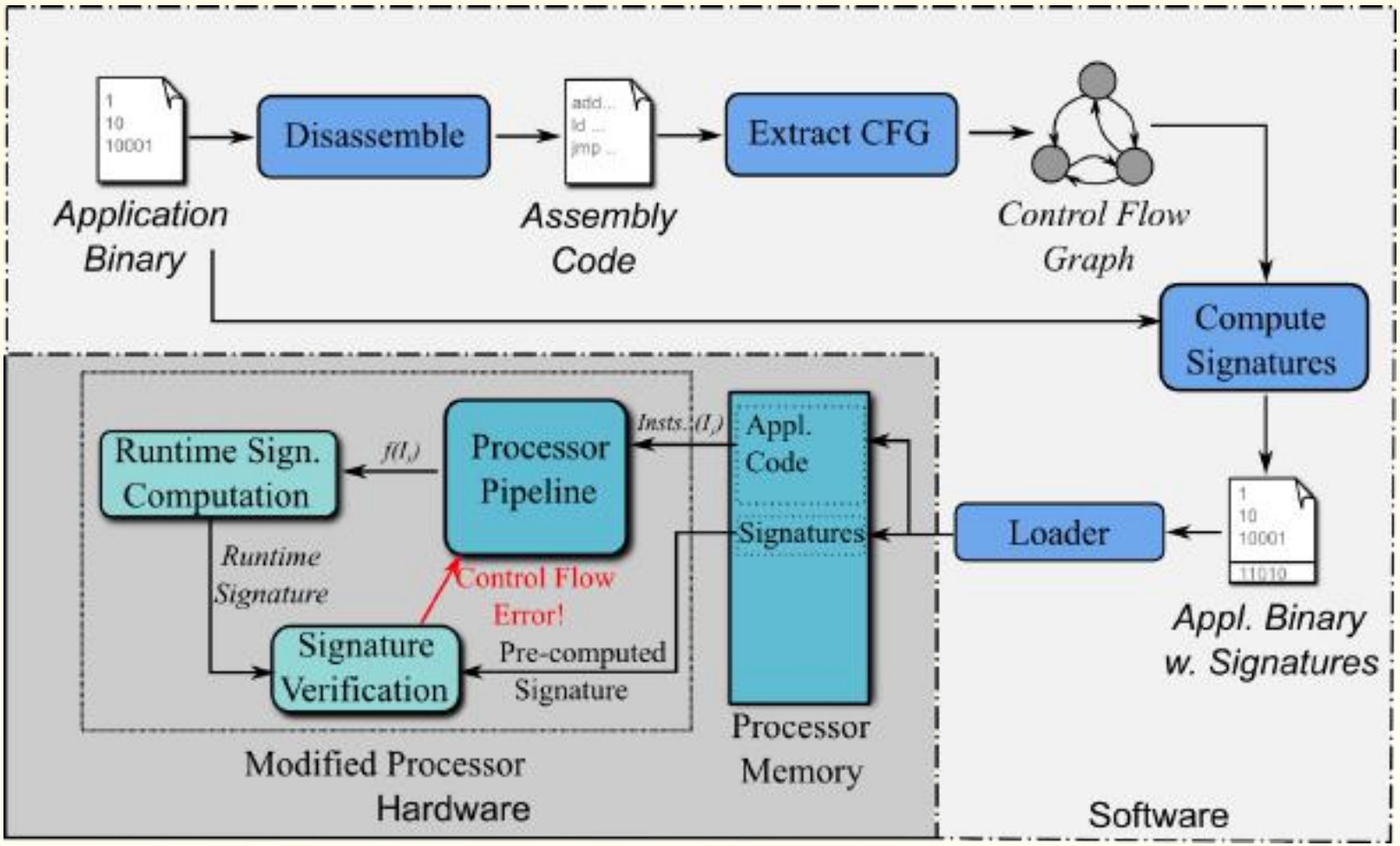
Threat Model

- Adversary can interact with applications through I/O streams, locally or remotely
  - Can exploit vulnerabilities through these streams
  - Adversary does not have physical access to the processor hardware
- This model is representative of the most common cases of security attacks
  - Attempts to break into system through Internet
  - Use of temporary input devices (such as USB drives)

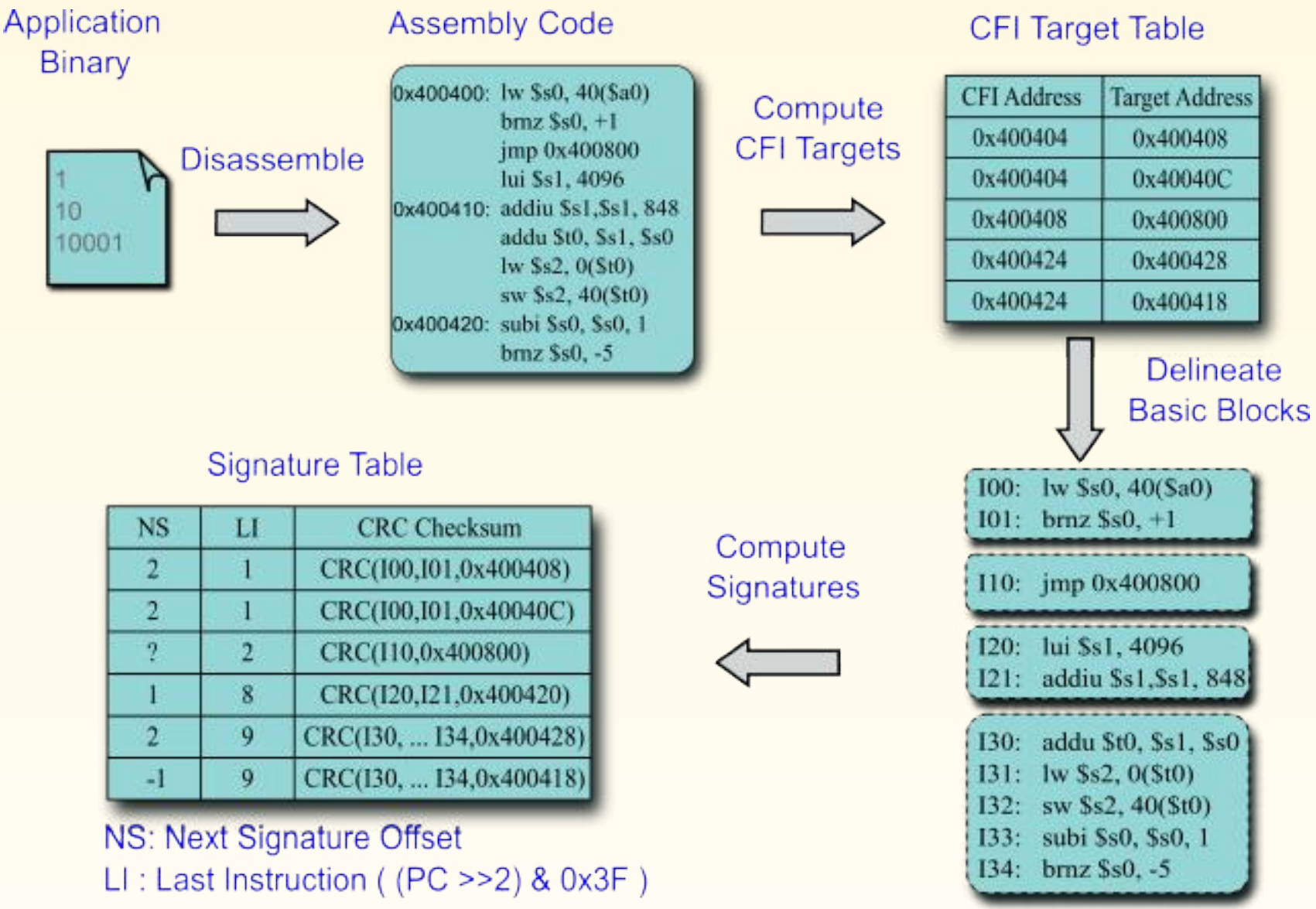
Mechanism

- Use low-power stream cipher based hash algorithms
- Implement checking in hardware

Framework for Hardware Control Flow Monitoring (*Chaudhari et al., 2012*)



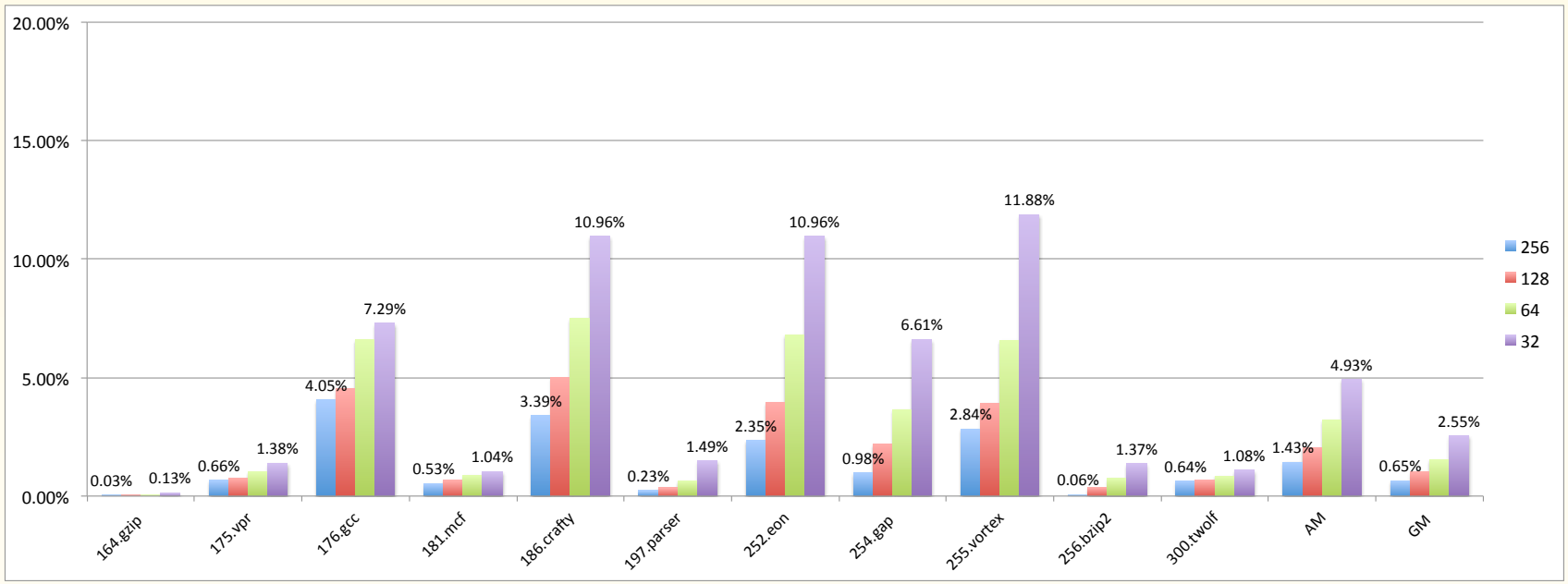
Signature Computation



Evaluation – Demonstrates Low Performance Overhead

SPEC 2000 benchmarks, GEM5 simulator for an out-of-order x86 processor issuing 4 instructions/cycle

32 KB Instruction and Data Cache, 256KB L2 Cache



Requirements for Comprehensive Checks on Computations

Ultimately, need to ensure correct **results** of computations

- How do we ensure that correct results are produced, and that they are produced at the correct time?
- Redundancy as well as diversity may be needed to achieve this

Where to place checks?

- Do we need control flow checks (if results can be checked)?
- What checks will be sufficient to deal with both failures and attacks?
- Application dependent?

Types of Attacks

- Control hijacking attacks
- Reverse engineering
- Malware
- Injected crafted packets or inputs
- Eavesdropping
- Brute-force search attacks
- Attacks during normal usage

Effects of Attacks

- Denial of service
- Code execution
- Integrity violation
- Information leakage
- Illegitimate access
- Financial loss
- Degraded level of protection
- Miscellaneous

Attack Injection

How do we evaluate proposed security techniques?

- Inject attacks into prototype systems
- Practically impossible to cover all circumstances

Comprehensive and general model for detecting attacks

- An incorrect address (instruction or data) will be sent to the memory
- The result of a computation will be incorrect
- There will be an abnormal delay in computing a result

These attacks are easy to inject into systems.
Any intrusion attack will be detected by a check for correct control flow, data or run time of a computation
Would checking for this broad class of effects results in high overhead?

Conclusions

- Resiliency is imperative for embedded systems
- Classical approaches to resiliency (based on fault tolerance) are not sufficient
- Resiliency techniques need to form the additional layer of protection to deal with test escapes, undetected design bugs and security attacks
- We have to ensure that none of these problems will result in incorrect results from the system
- Can we combine the checks for different problems to achieve viable resiliency at reasonable costs?