A Case for the Runtime Validation of Hardware

Sharad Malik

Princeton University

IBM Verification Conference, Haifa

13 November, 2005

Research supported by the Microelectronics Advanced Research Consortium (MARCO) through the Gigascale Systems Research Center (GSRC)

Outline

- The Verification Gap/Crisis
- Different Failure Modes and Runtime Validation
- Microarchitectural Solutions using Runtime Validation
- Runtime Property Checking
- A General RTL Methodology
- Summary and Conclusions

Rate of Increase of Design Complexity

- Moore's Law: Growth rate of transistors/chip is exponential
- Corollary 1: Growth rate of state bits/chip is exponential
- Corollary 2: Growth rate of state space (proxy for complexity) is doubly exponential

But…

- Corollary 3: Growth rate of compute power is exponential

Thus…

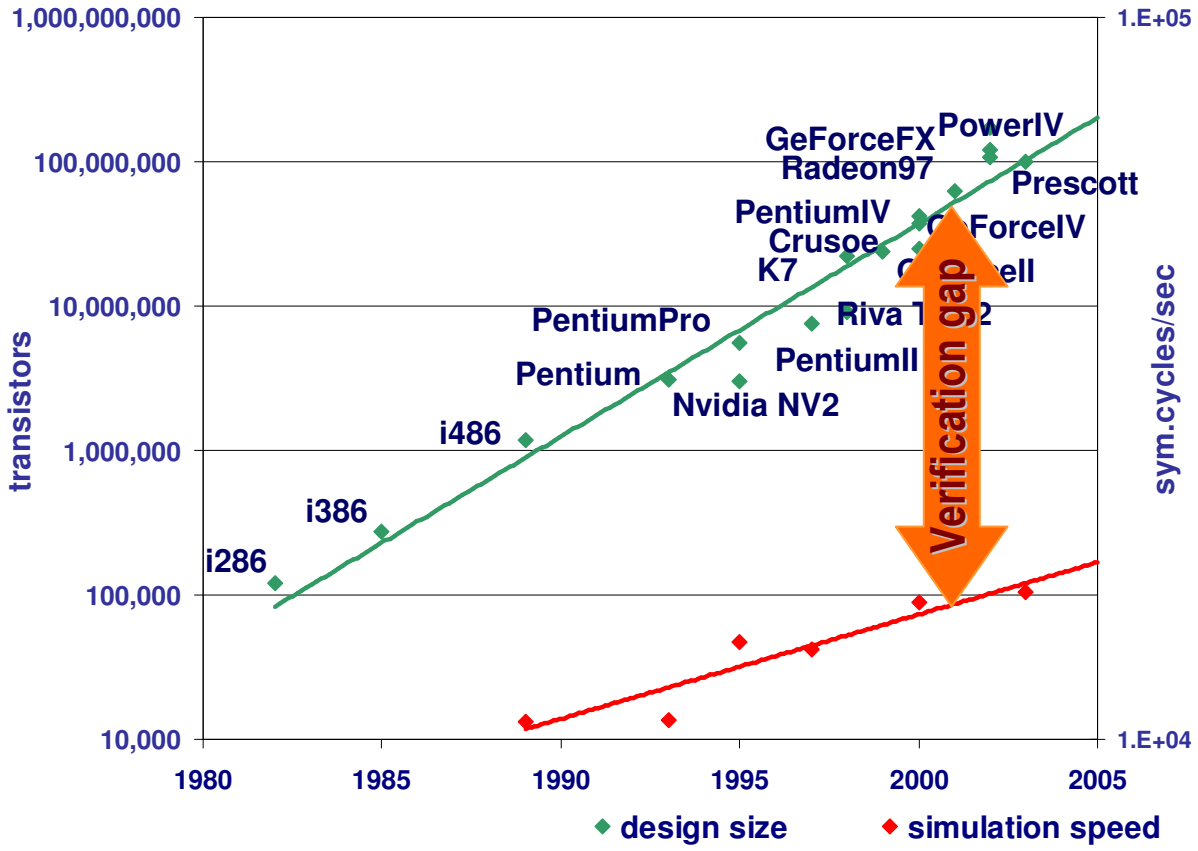
- Growth rate of complexity is exponential relative to our ability to deal with it

But what about…

- The fact that not all state bits contribute equally to complexity (e.g. memory)
- The fact that design reuse enables use of pre-verified IP

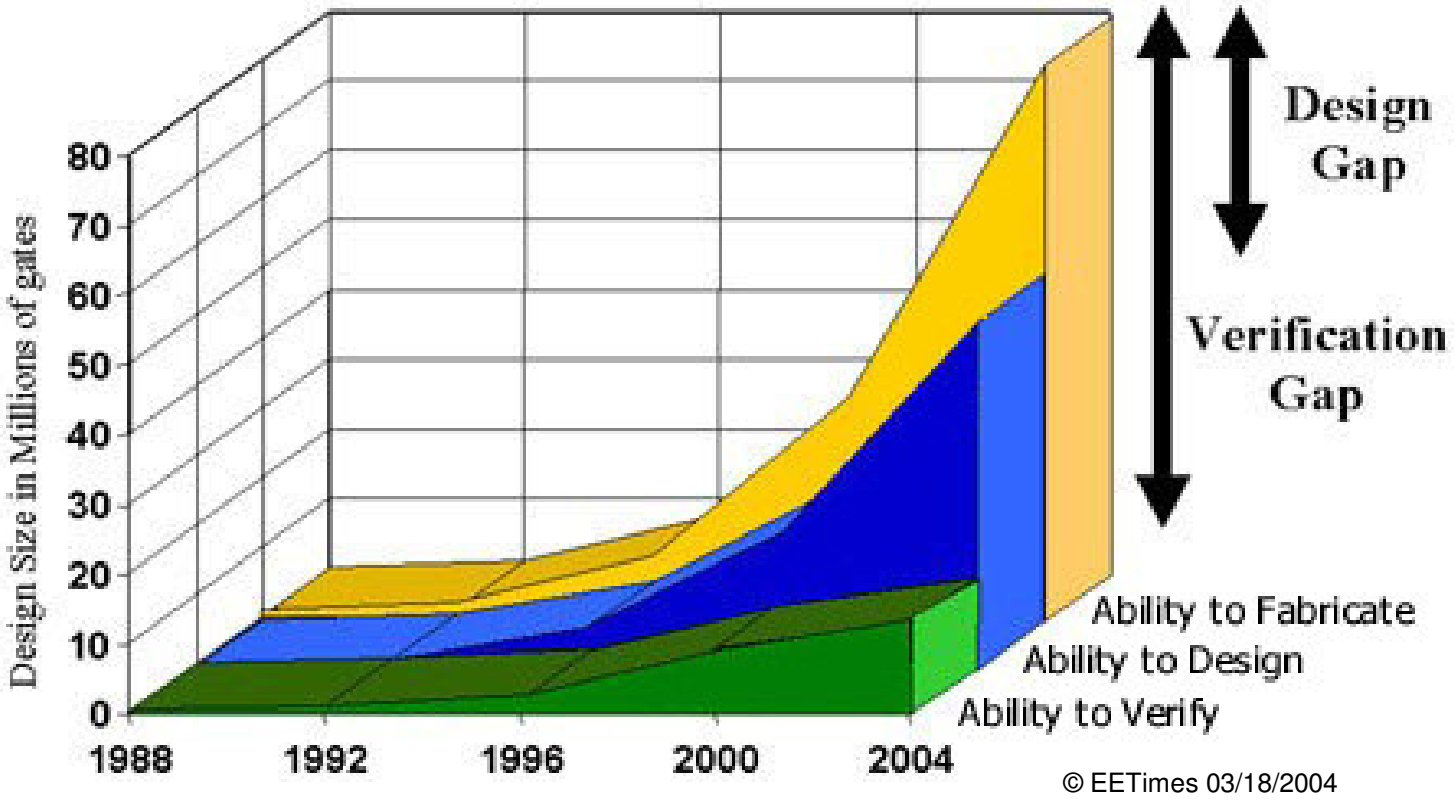
So maybe its not that bad…

The Verification Gap



Source: Valeria Bertacco, Univ. of Michigan

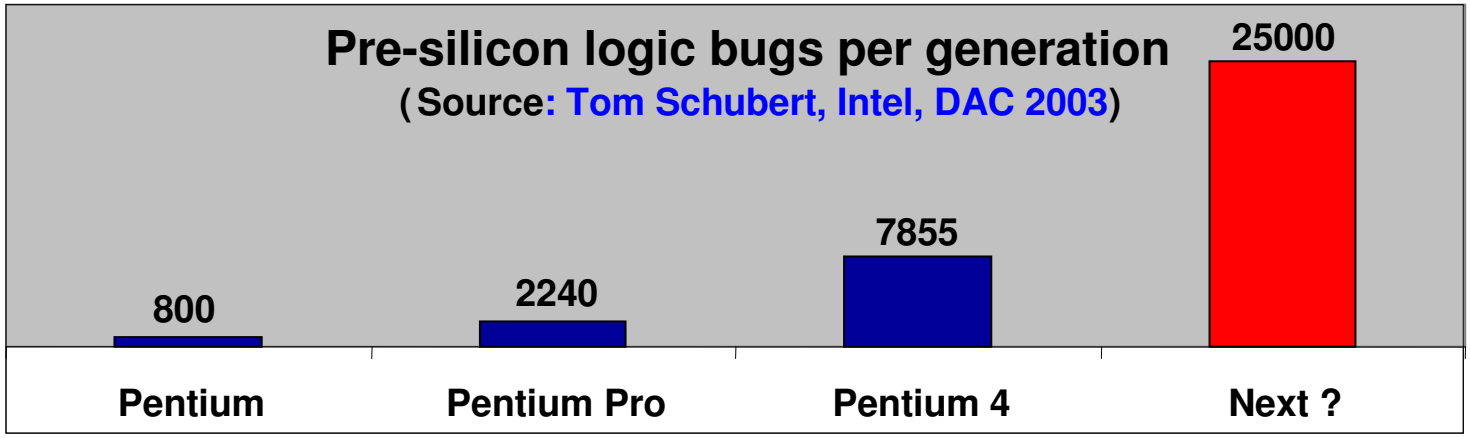
The Verification Gap



Orginal source : SIA

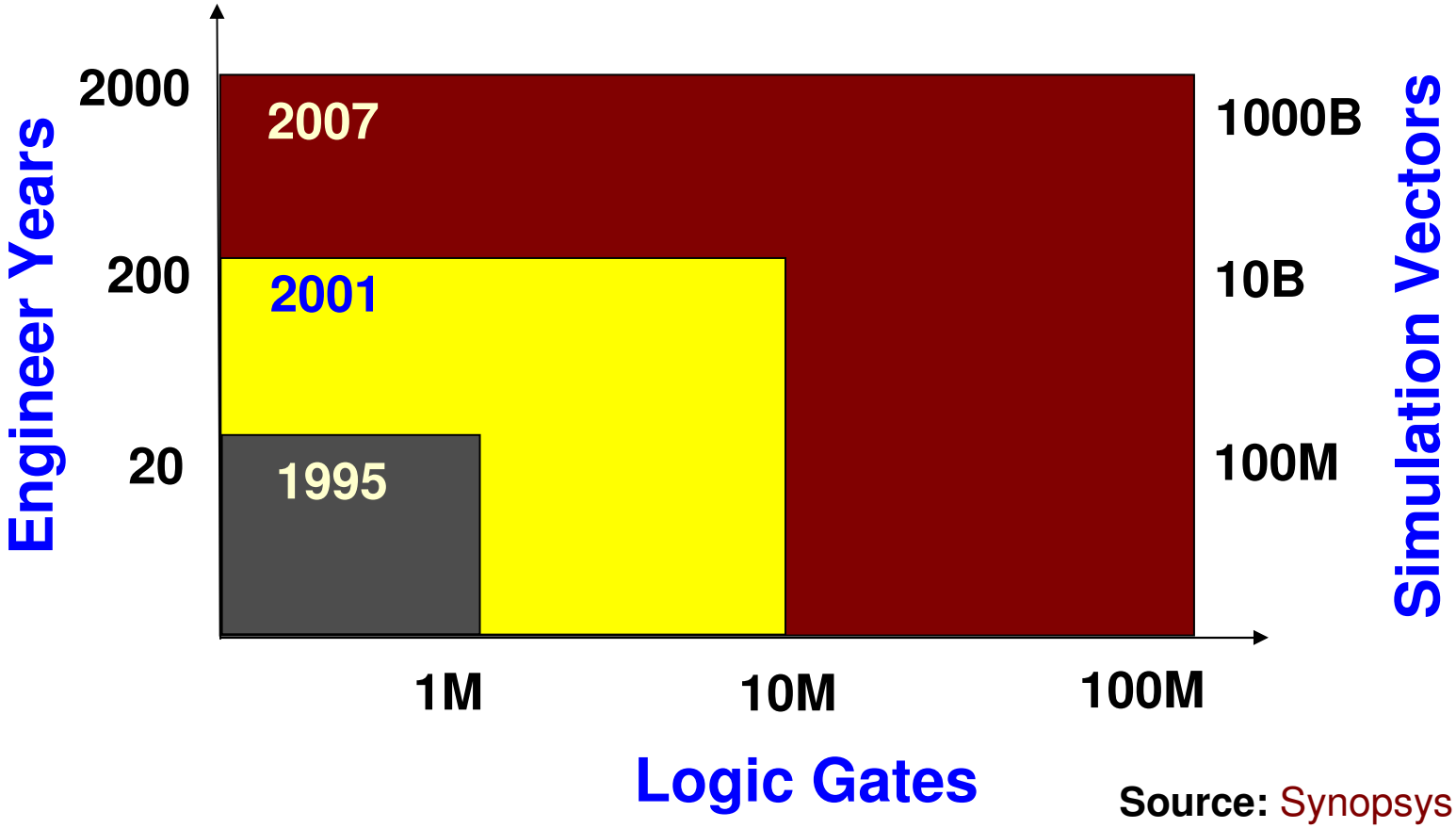
Data from Microprocessor Verification

- Functional validation is a major bottleneck
  - Deeply pipelined complex microarchitectures



- Logic bugs increase at 3-4 times/generation
  - Bugs increase is exponential over time

High Cost of Verification

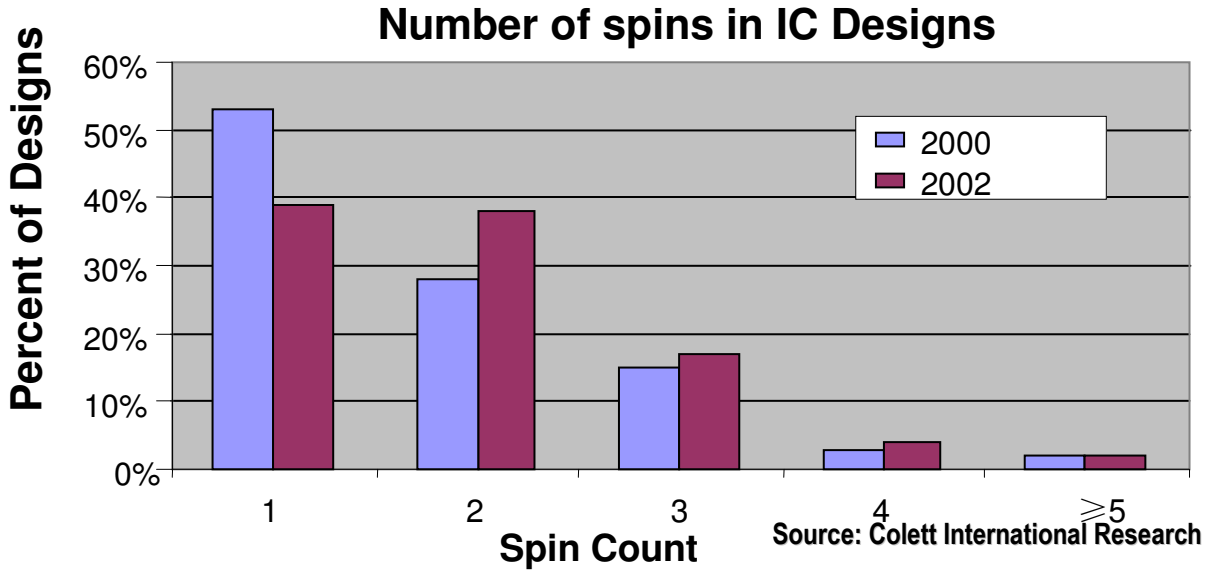


Source: Synopsys

Data from SoC Designs

Source: G. Spirakis, keynote address at DATE 2004

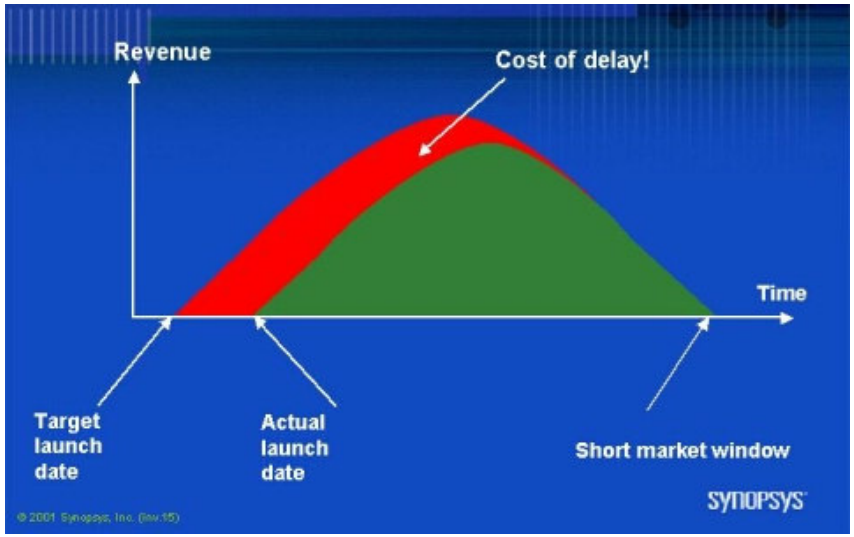
Yet Increasing Number of Bug Escapes…



71% of SoC re-spins are due to logic bugs

- High cost of re-spins
  - Increasing mask costs ($5M and increasing)
- Cost of recalls even higher
  - Pentium FDIV Bug Recall – several million $

Time to Market Costs



- Delays in re-spins ⇒ Lost time to market
  - Strong pressure for "first time correct" silicon
  - Chips verified to death

Runtime Validation

- Increasingly need to reconcile ourselves to the fact that hardware like software will be shipped with bugs
- Runtime verification (through error detection and recovery) offers a potentially scalable solution
  - Provide robustness in the face of inevitable bug escapes
- Significantly reduce verification costs
  - Verify chips “to life” rather than “to death”

Outline

- The Verification Gap/Crisis
- Different Failure Modes and Runtime Validation
- Microarchitectural Solutions using Runtime Validation
- Runtime Property Checking
- A General RTL Methodology
- Summary and Conclusions

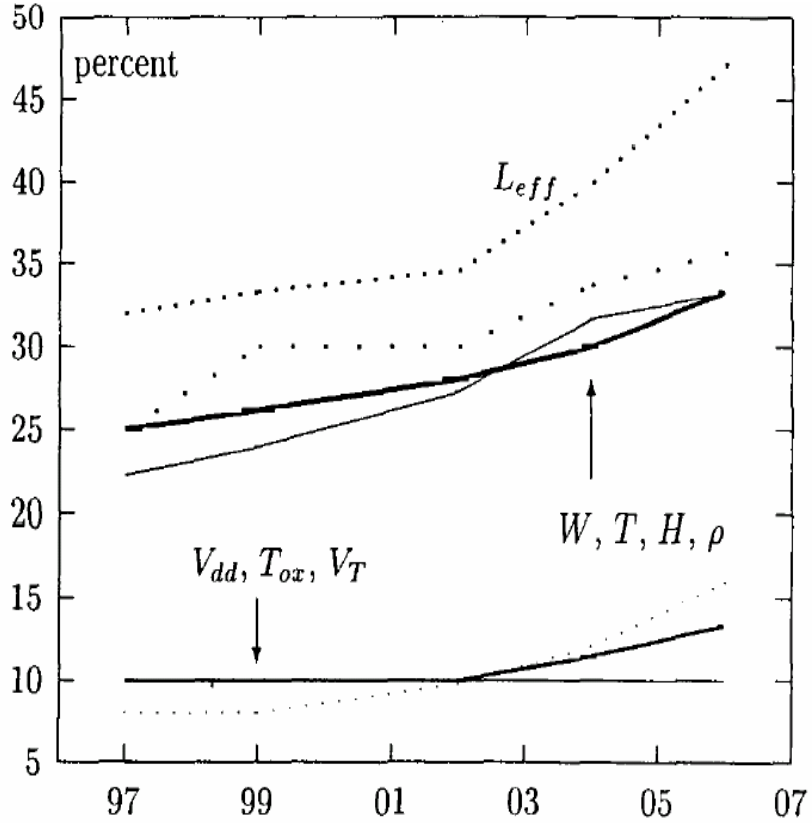
Sources of Errors

- Process/Operating Condition Variations, Aging
  - Device failure
  - Timing Failure
- Soft Errors
- Aggressive Deployment
  - Low Vdd
  - Overclocking
  - Push the design to the edge so as to result in device/timing failures
- Manufacturing Defects
- Design Complexity

Operational failures

Functional failures

Increasing Process Variations



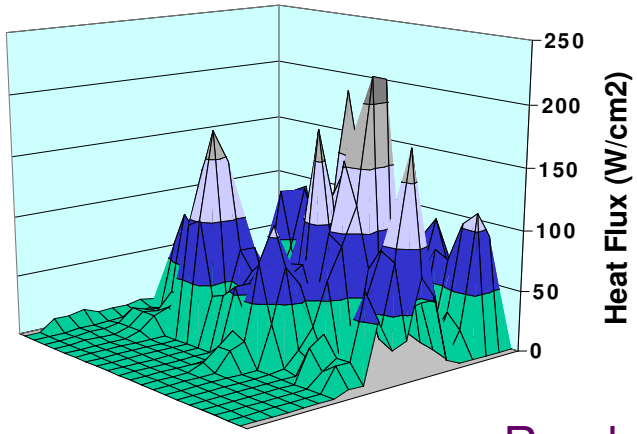
Percentage of total variation accounted for by within-die variation(device and interconnect)

Original Source: Sani Nassif IBM

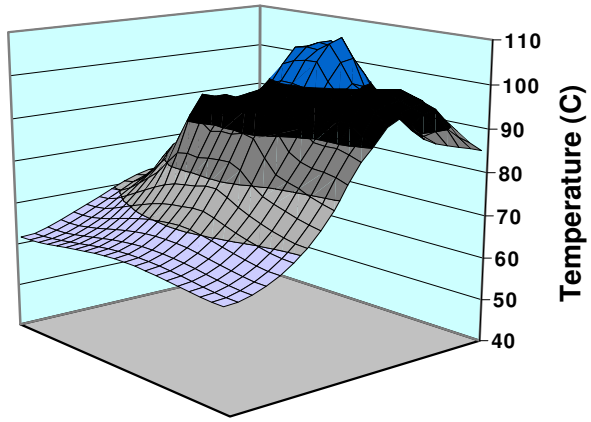
- **Increase in variation of process parameters over generations**
- **Worst-case design getting more expensive**
- **"Better than worst-case" design must be error tolerant**

Other Variations

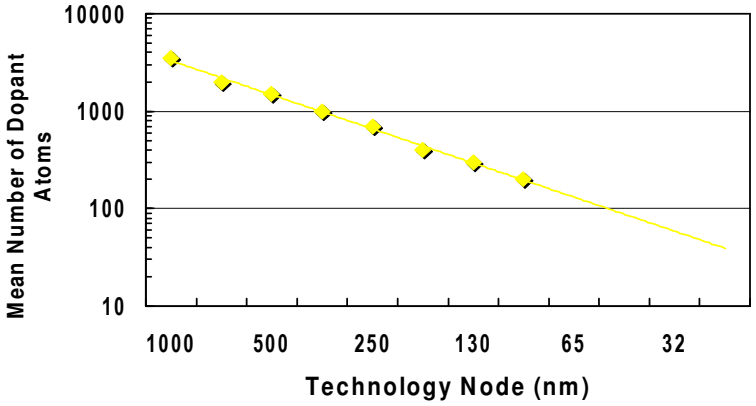
Heat Flux (W/cm^2)
Results in Vcc variation



Temperature Variation (℃)
Results in Hot spots

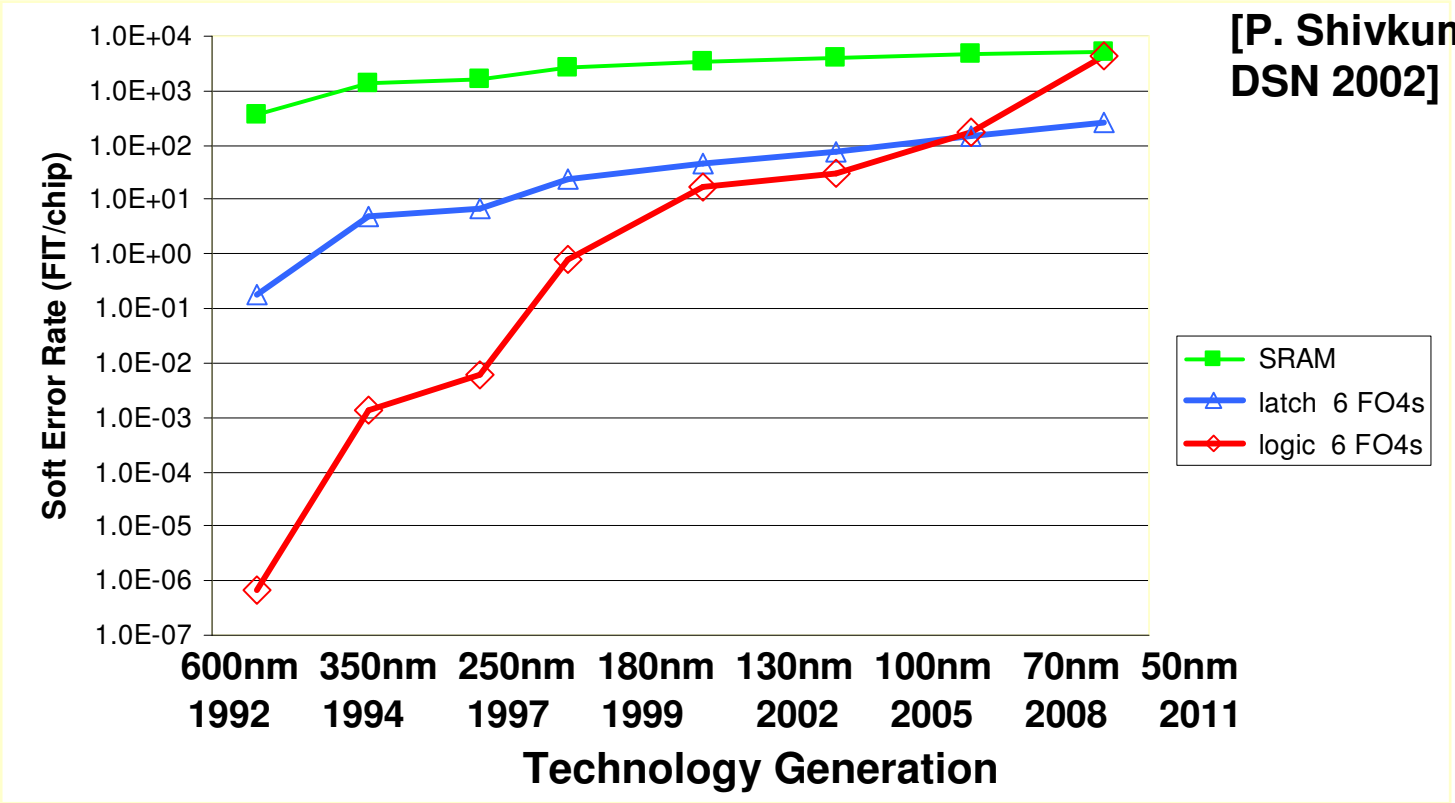


Random Dopant Fluctuations



Source: Shekhar Borkar, Intel

Soft-Error Trends



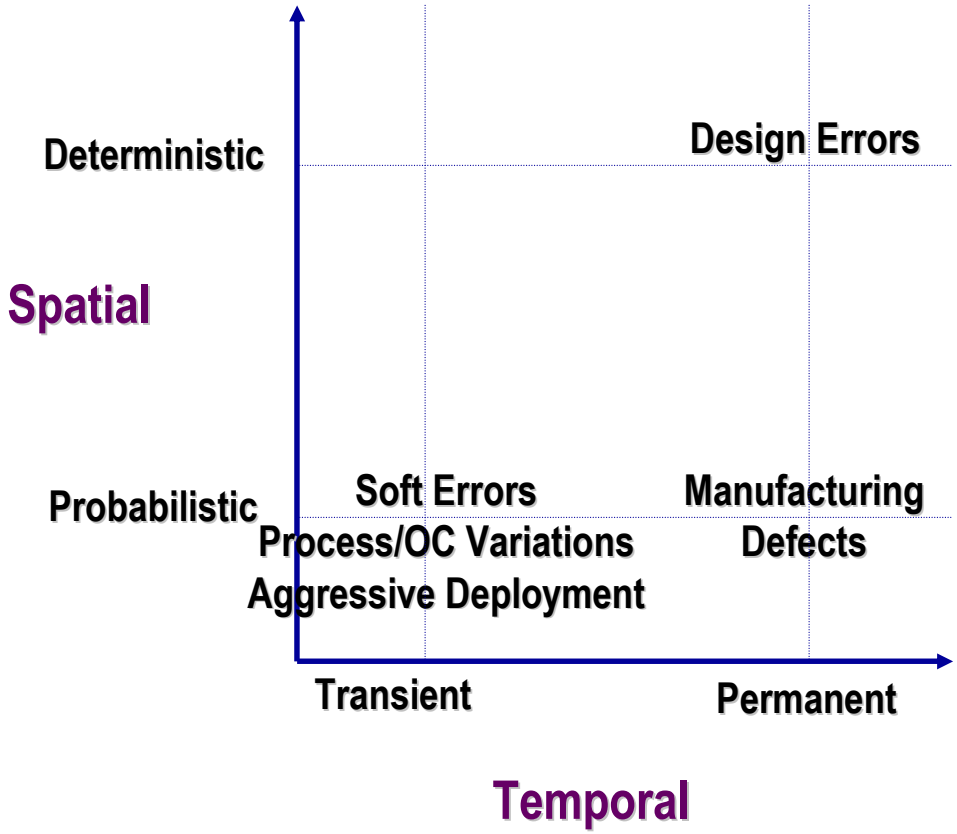
- SER per chip of logic circuits
  - Nine orders of magnitude increase from 600 nm to 50 nm
  - Dominant source of soft errors after 50 nm

Classification of Error Modes

- Two axes of classifications
  - Temporal (same instance, different invocations)
  - Spatial (different instances, possibly on the same chip)

Runtime error detection and correction applicable to all – however error mode determines which class of techniques can be applied:

- Replication works for probabilistic errors, but not deterministic errors.
- Repeating computation works for transient errors but not permanent ones.



Design errors dominate along both axes

Runtime Validation: Quick Analysis

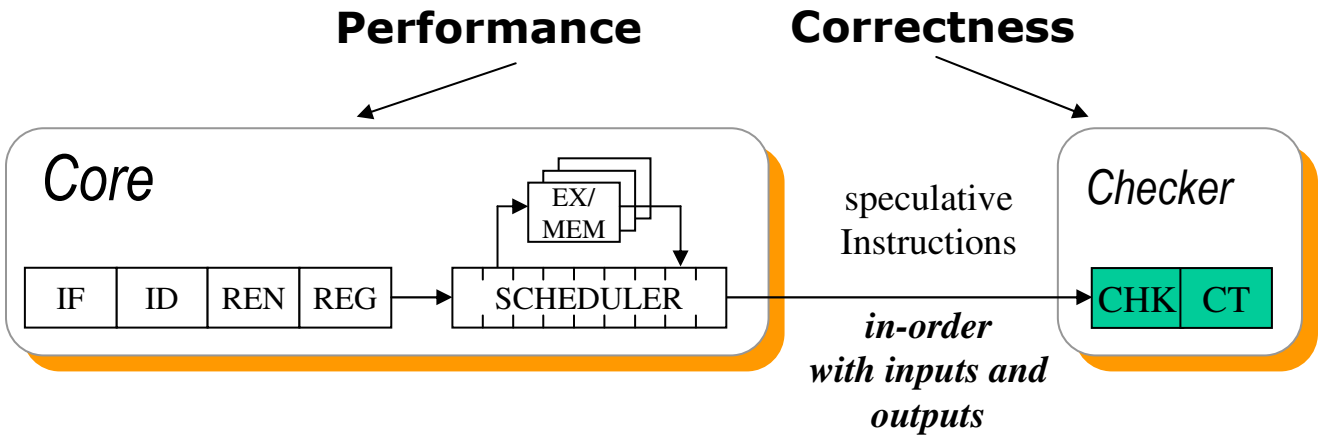
- Pros
  - Do not have to deal with all possible behavior, only the actual behavior
    - No state explosion problem
  - Checking circuits only need to recognize the symptoms when the bug is exercised, not its cause, e.g.
    - Check the results of division using multiplication
    - Check if a bus is deadlocked
  - Recovery does not need to fix the bug, rather only ensure forward progress
- Cons
  - Recovery likely to be hard
    - Need a trusted alternate computation path
      - Tradeoff performance for simplicity
        - Simpler slower divider
        - Simpler slower protocol
  - Additional complexity
    - Who checks the checker?
  - Performance, power, area overhead

Outline

- The Verification Gap/Crisis
- Different Failure Modes and Runtime Validation
- Microarchitectural Solutions using Runtime Validation
- Runtime Property Checking
- A General RTL Methodology
- Summary and Conclusions

Instruction Set Processor Validation

- Perform online checking to detect/rectify faults [Austin, Micro 1999]



[Source: Todd Austin, Univ. of Michigan]

- Key observations
  - Correctness: checker implements the same abstract functional interface
    - *Simple* checker design enables high-quality functional verification
  - Performance: leverage the core processor pre-execution to streamline checker
    - Cache pre-fetching and branch/value prediction
  - Robustness: focus on reliability of the checker
    - *Simpler, slower* checker can be constructed with large time margin, large transistors

Validating Simultaneous Multithreading

- **Motivation**
  - **High practical application**
    - **General processing trend**
  - **High potential for general lessons**
    - **Race conditions**
    - **Synchronization**
    - **Access to shared data**
    - **…**
- **So what do we need to do beyond checking each thread with a checker processor…**

Kaiyu Chen and Sharad Malik, "Runtime Validation of Multithreaded Processors," Technical Report, Dept. of Electrical Engineering, Princeton University, May 2005. Available by email from kchen@princeton.edu

Memory Consistency Between Threads

Shared Data

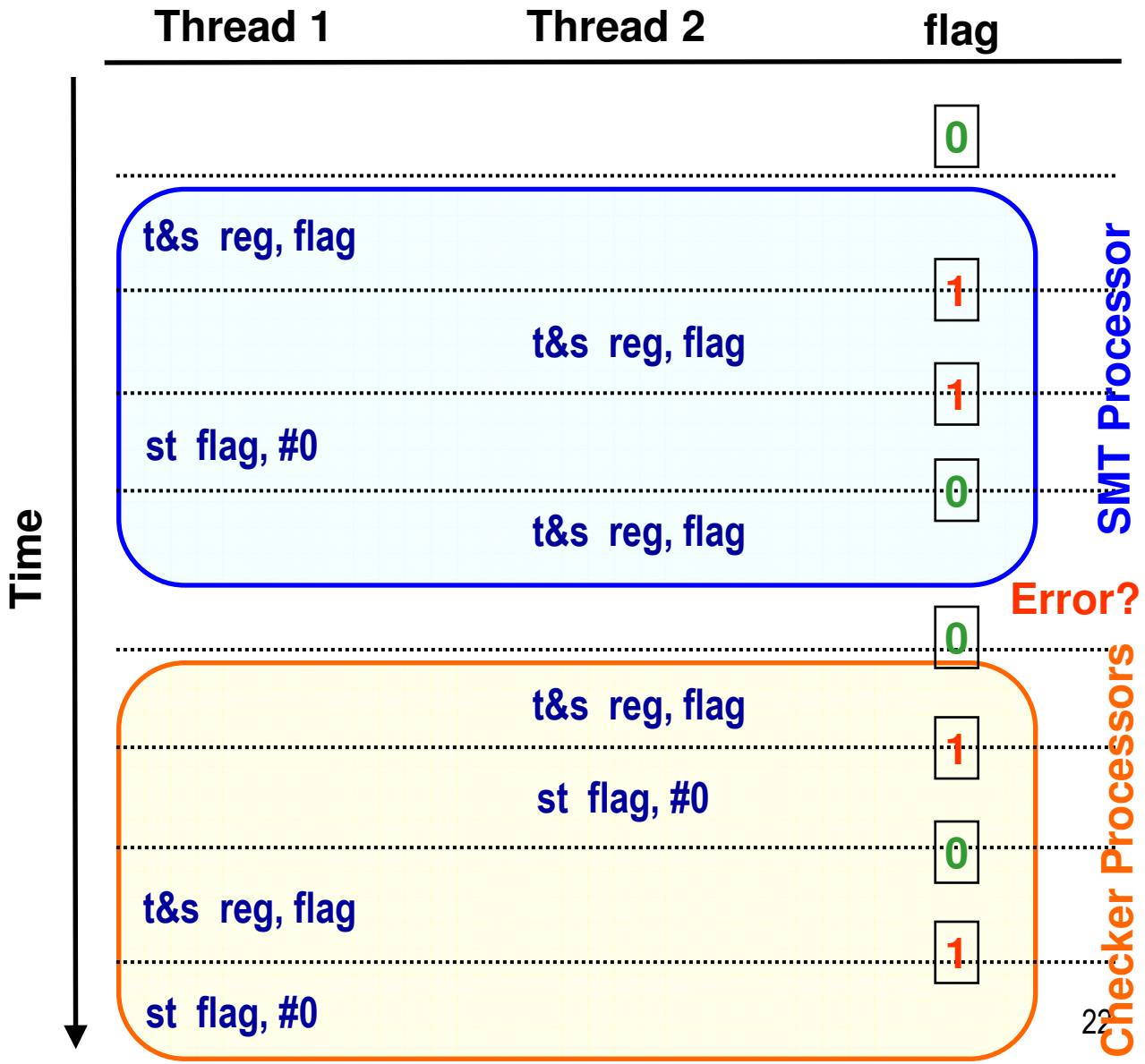
flag is initialized to 0

Thread - 1

```
lock:   t&s  reg, flag
        bnz lock
        Critical Section
unlock: st flag, #0
```

Thread - 2

```
lock:   t&s  reg, flag
        bnz lock
        Critical Section
unlock: st flag, #0
```

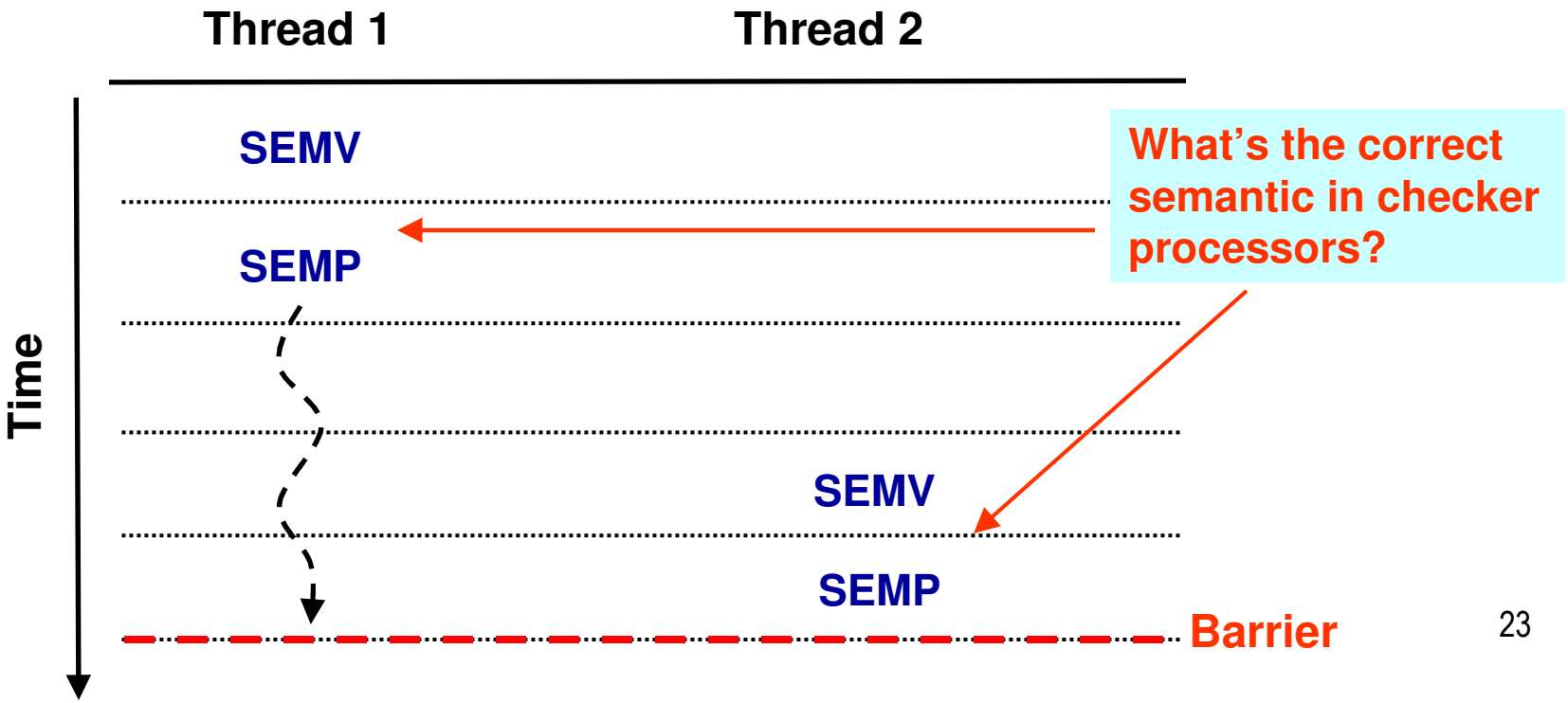


Correctness of Synchronization Instructions

Two semaphores sem1, sem2 are initialized to 0

```
Thread 1:
...
 SEMV(sem1);
 SEMP(sem2);
 …
```

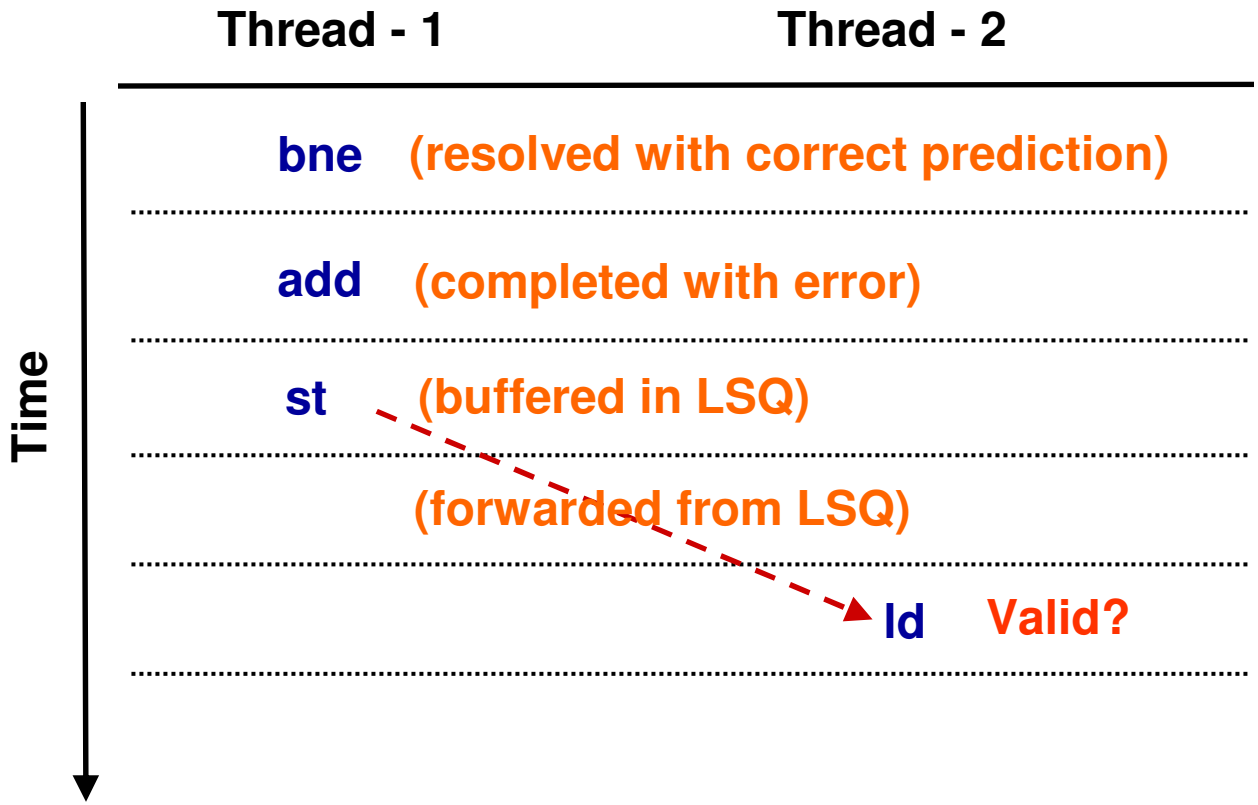
```
Thread 2:
...
 SEMV(sem2);
 SEMP(sem1);
 …
```



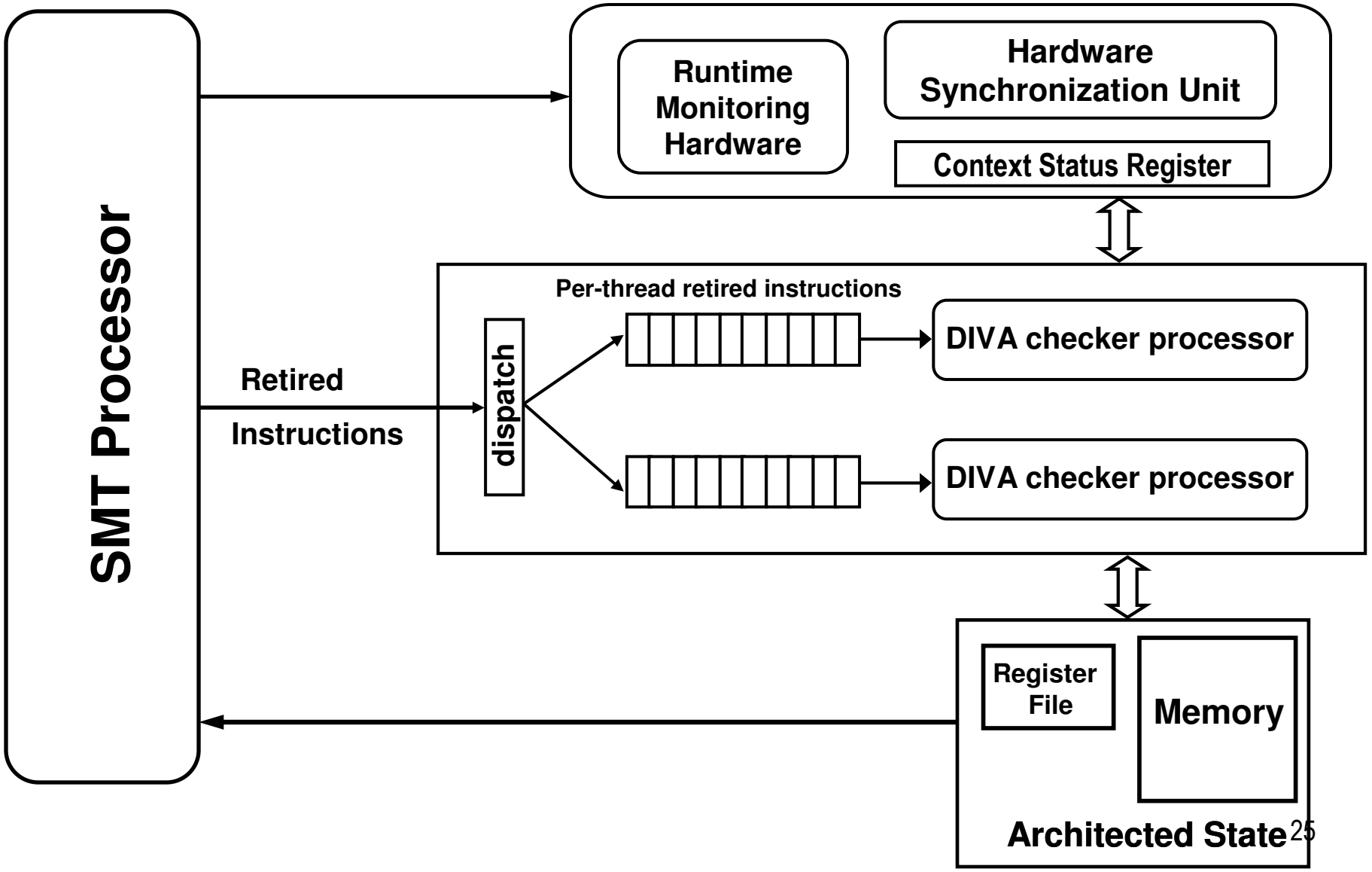
Forwarding of Erroneous Results

```
Thread 1:
...
 if (flag == 0)
    i = i + 1;
    data = i;
 …
```

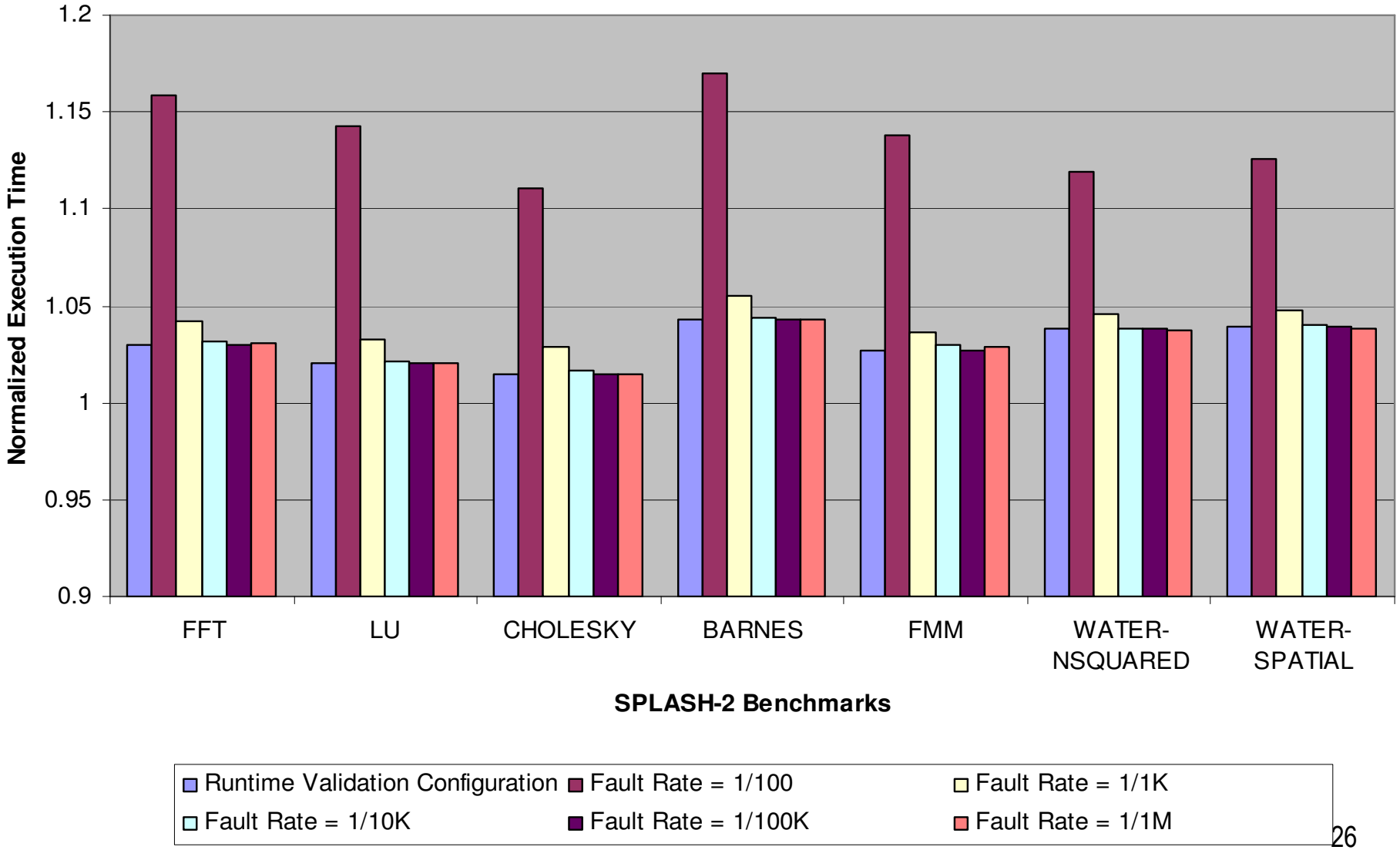
```
Thread 2:
...
 key = data;
 result = foo(key);
 …
```



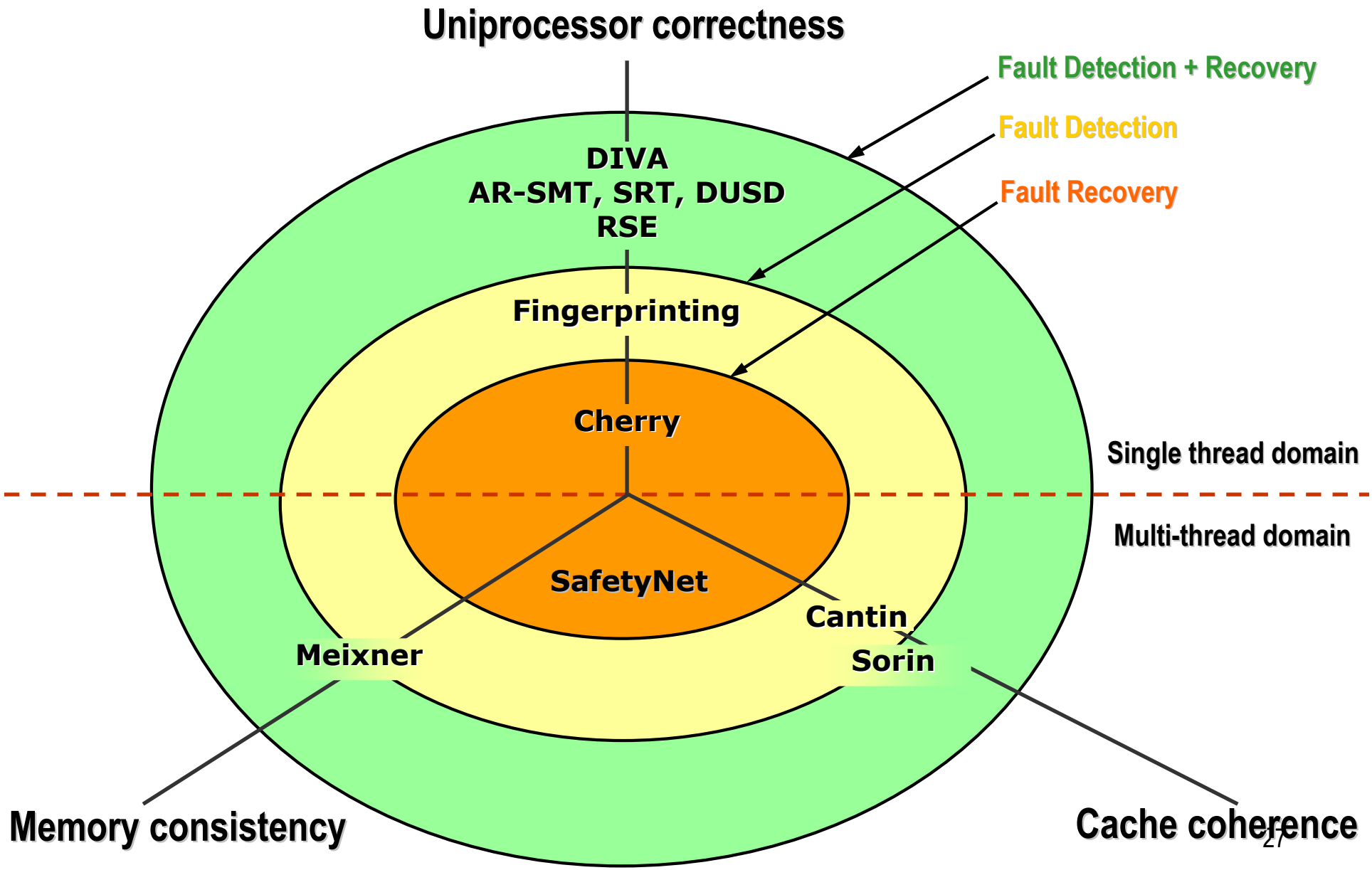
The Integrated Runtime Validation Solution



Experimental Results



Microarchitectural Support for Validation



Microarchitectural Support for Validation - References

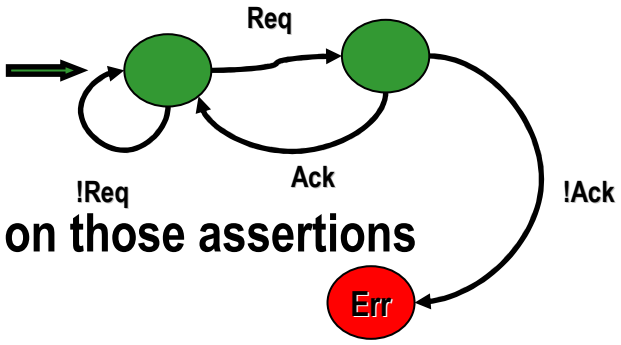
- [DIVA] Todd Austin. “DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design”. ACM/IEEE 32nd Annual Symposium on Microarchitecture (Micro), 1999
- [AR-SMT] E. Rotenberg. “AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors”. 29th Int’l Symposium on Fault-Tolerant Computing, June 1999.
- [SRT] S. K. Reinhardt and S. S. Mukherjee. “Transient Fault Detection via Simultaneous Multithreading”. 27th Annual Int'l Symposium on Computer Architecture (ISCA), 2000
- [DUSD] Joydeep Ray, James C. Hoe and Babak Falsafi. “Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery”. Proceedings of International Symposium on Microarchitecture (MICRO), December 2001
- [RSE] Nithin Nakka, Jun Xu, Zbigniew Kalbarczyk and Ravishankar K. Iyer. “An Architectural Framework for Providing Reliability and Security Support”. IEEE Int’l Conf. on Dependable Systems and Networks (DSN), 2004
- [Fingerprinting] Jared C. Smolens, Brian T. Gold, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. “Fingerprinting: Bounding Soft-Error Detection Latency and Bandwidth”. (ASPLOS), October 2004
- [Cherry] J.F. Martínez, J. Renau, M.C. Huang, M. Prvulovic, and J. Torrellas. "Cherry: Checkpointed early resource recycling in out-of-order microprocessors". In Int’l Symposium on Microarchitecture (Micro), Nov. 2002
- [Safetynet] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. “SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery.” 29th Inter-national Symposium on Computer Architecture (ISCA), May 2002
- [Meixner] Albert Meixner and Daniel J. Sorin. "Dynamic Verification of Sequential Consistency." 32nd Annual International Symposium on Computer Architecture (ISCA), June 2005.
- [Cantin] J. Cantin, M. Lipasti, J. E. Smith. “Dynamic Verification of Cache Coherence Protocols”. Workshop on Memory Performance issues, Gothenburg, Sweden, June 2001.
- [Sorin] Daniel J. Sorin, Mark D. Hill, and David A. Wood. “Dynamic Verification of End-to-End Multiprocessor Invariants”. International Conference on Dependable Systems and Networks (DSN), June 2003.

Outline

- The Verification Gap/Crisis
- Different Failure Modes and Runtime Validation
- Microarchitectural Solutions using Runtime Validation
- Runtime Property Checking
- A General RTL Methodology
- Summary and Conclusions

Assertion Based Runtime Validation

- Terminology
  - Property: A quality or trait belonging to a design (based on specifications)
    - e.g. Any request should be acknowledged eventually
  - Assertion: Typically syntactical statement of a property that should hold
    - PSL: assert always (req → eventually! ack);
    - Assertions available from design/simulation/formal verification processes
  - Runtime Checker: Hardware responsible for validating a property or properties
    - Assertions can be synthesized [Abarbanel99]
- Runtime Validation based on Assertions/Specifications
  - Find/Write the assertions
    - E.g. G(Req →X Ack)
  - Generate hardware models for error detectors based on those assertions
  - Implement recovery mechanism (design specific)
    - e.g. Invalidate all requests

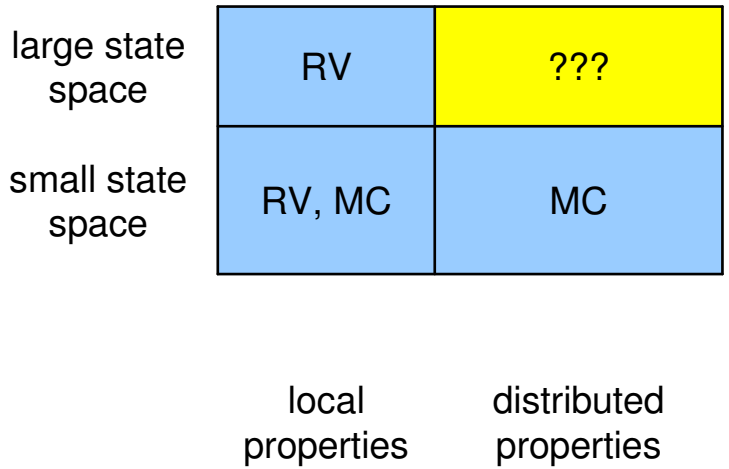


A Classification of Properties

- Based on time
  - Liveness: Good things will *eventually* happen
    - e.g. G (Req → F ack)
    - Bounded liveness: fix time bound for when this will happen
  - Safety: Bad things should never happen
    - e.g. G (Req → XX Ack)
- Based on spatial distribution
  - Local: All necessary information can be gathered easily (e.g. one clock cycle)
    - e.g. G (req → X !req)
  - Distributed: Signals separated in space and difficult to gather
    - e.g. dual-ownership of cache lines in a shared memory multi-processor system
- Based on recovery requirement
  - Soft: Only control bits may be corrupted
    - e.g. deadlock situation, bad control state, data is safe
  - Hard: Both data and control bits may be corrupted
    - e.g. dual-ownership of cache lines in a shared memory multi-processor system

Compositional Reasoning using Runtime Validation

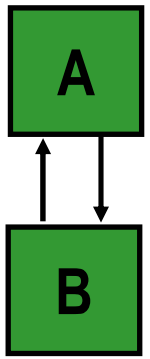
- Complementary relation between Runtime Validation (RV) and Model Checking (MC)
  - RV does not suffer from state space explosion
  - MC handles distributed properties easily
- Basic Idea
  - Runtime Validate some property R
  - Use it in off-line model checking of property P
  - Proving P with assumption R using MC guarantees ($R \rightarrow P$) at runtime
  - As long as R holds at runtime, P holds at runtime



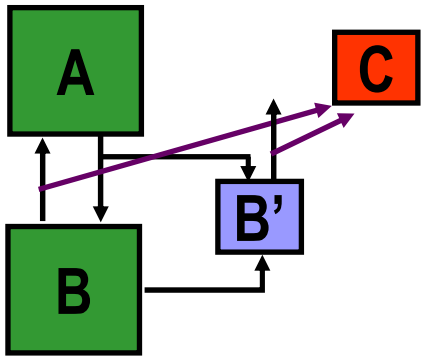
- When does this help?
  - MC does not "complete" checking P without assumption R
  - MC does not "complete" checking R, i.e. RV is necessary for validating R

Using Runtime Abstracters

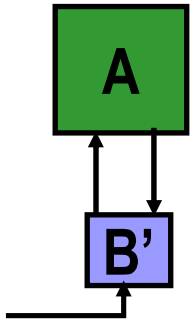
- **Runtime Abstraction:** Using abstraction in model checking and validating correctness of abstraction at runtime.
  - Allows for wider range of abstractions (e.g. property specific and not just parametric)



Design



Checker C checks Abstracter B' at runtime



Model Checking uses Abstracter B'

Experimental Results

Model Checking Single-cluster CCP

Proccessor Count	with Assumptions	without Assumptions	Checking Assumptions
4	0.08	0.51	9.87
6	0.33	8.94	143.57
8	0.61	29.06	348.95
10	0.8	68.86	1606.61
12	5.13	141.33	8008.13
14	3.07	183.51	145989.05
16	13.2	656.41	TIME-OUT

Model Checking Multi-cluster TokenShare

Unit #	Safety/A	Safety	Check A	Liveness/A	Liveness
4	2.03	4.45	27.11	67.19	385.72
6	8.16	14.84	105.47	294.94	85428.15
8	50.12	74.56	448.99	1840.63	TIME-OUT
10	66.09	57.02	448.32	2007.45	TIME-OUT
12	133.82	82.88	723.48	4212.76	TIME-OUT
14	285.83	120.39	1110.78	9053.72	TIME-OUT
16	408.03	158.98	1423.31	13810.5	TIME-OUT

Results for Using Runtime Assumptions

Time unit is second. TIME-OUT is 2 days.

- Runtime abstraction: Replacing one 16-unit cluster with the runtime abstracter drops verification time to 56.2 seconds from 408 seconds in TokenShare
- HDL implementation of CCP: 1600 lines of Verilog code. No memory overhead for safety checkers. Bounded liveness checkers use 10-bit counter for 4-cluster, 16-processor configuration.

Ali Bayazit and Sharad Malik, "Complementary Use of Runtime Validation and Model Checking," in *ICCAD'05: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2005.

Outline

- The Verification Gap/Crisis
- Different Failure Modes and Runtime Validation
- Microarchitectural Solutions using Runtime Validation
- Runtime Property Checking
- **A General RTL Methodology**
- **Summary and Conclusions**

General Design with Integrated Runtime Validation

- Conceptually, runtime validated systems have 3 essential components
  - Primary implementation of design (D)
  - Runtime checkers (C)
  - Design-specific runtime error recovery (R)
- Usually, the designer has to worry about *interactions* between D, C, and R and ensure that:
  - Checkers and recovery are “silent” under normal operation
  - Design halts when an error is detected
  - Recovery kicks in safely when an error gets detected
  - Recovery occurs correctly
  - Design continues execution safely post-recovery
- Given D, C, R can we relieve the designer of correctly implementing their interactions?

A Proposal for an RTL Methodology Solution

- Clearly separate D, C and R in specification
- Language semantics should enforce useful properties of interactions between D, C and R
  - Checkers and recovery are “silent” under normal operation
  - Design halts when an error is detected
  - Recovery kicks in safely when an error gets detected
  - Recovery occurs correctly
  - Design continues execution safely post-recovery
- Leave actual implementation of the interactions between D, C and R to synthesis
  - use a generic hardware implementation template that respects these semantics
- Additional checking/recovery specific language features
  - e.g. check-pointed register data type for rollback and recovery, implemented using automated checkpointing

Example

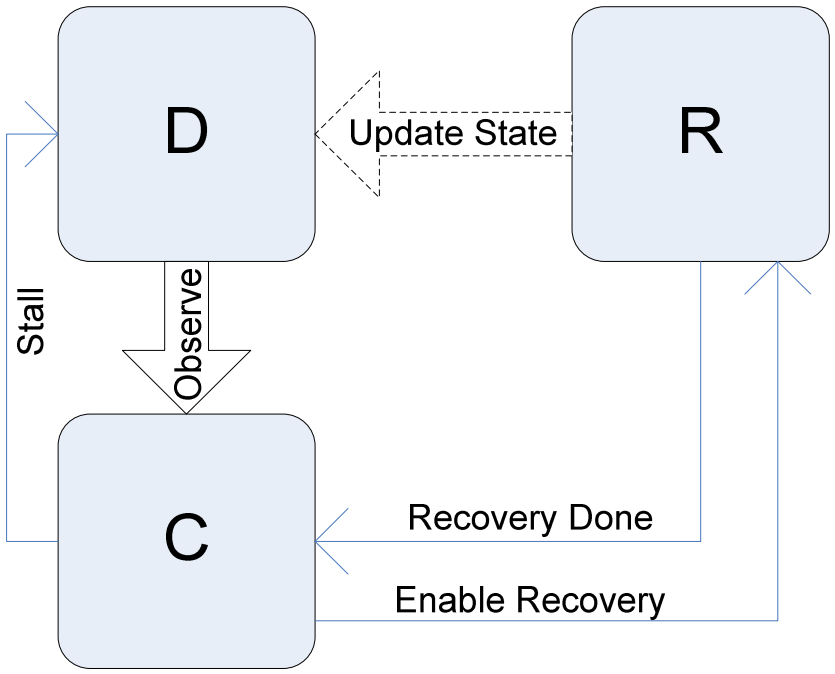
- A possible specification could look like this

```
{design D
    usual HDL description of design
}while {checker C
    monitor property at runtime
}else {recovery R
    recovery procedure
}
```

Example (contd.)

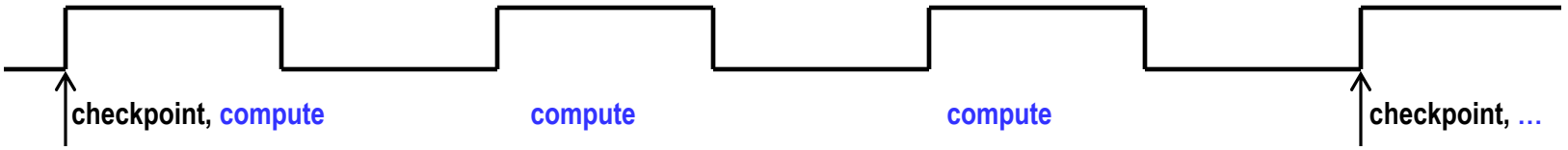
General Design Template

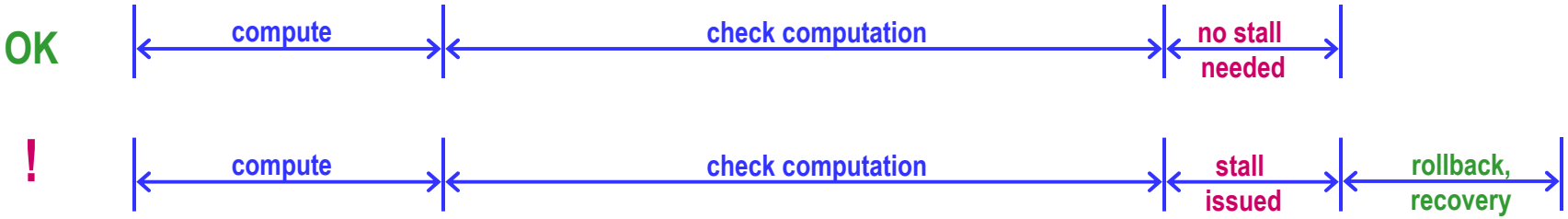
- C and D operate in parallel
- C can examine D's state
- On detection of error, D gets stalled, and R is triggered
- R permitted to change D's state to perform recovery
- D continues operation after recovery



Backward Error Recovery

- Checkpoint the state regularly
- In case of error detected by C
  - C stalls D
  - D is rolled back using checkpointed state
  - R performs the specified recovery

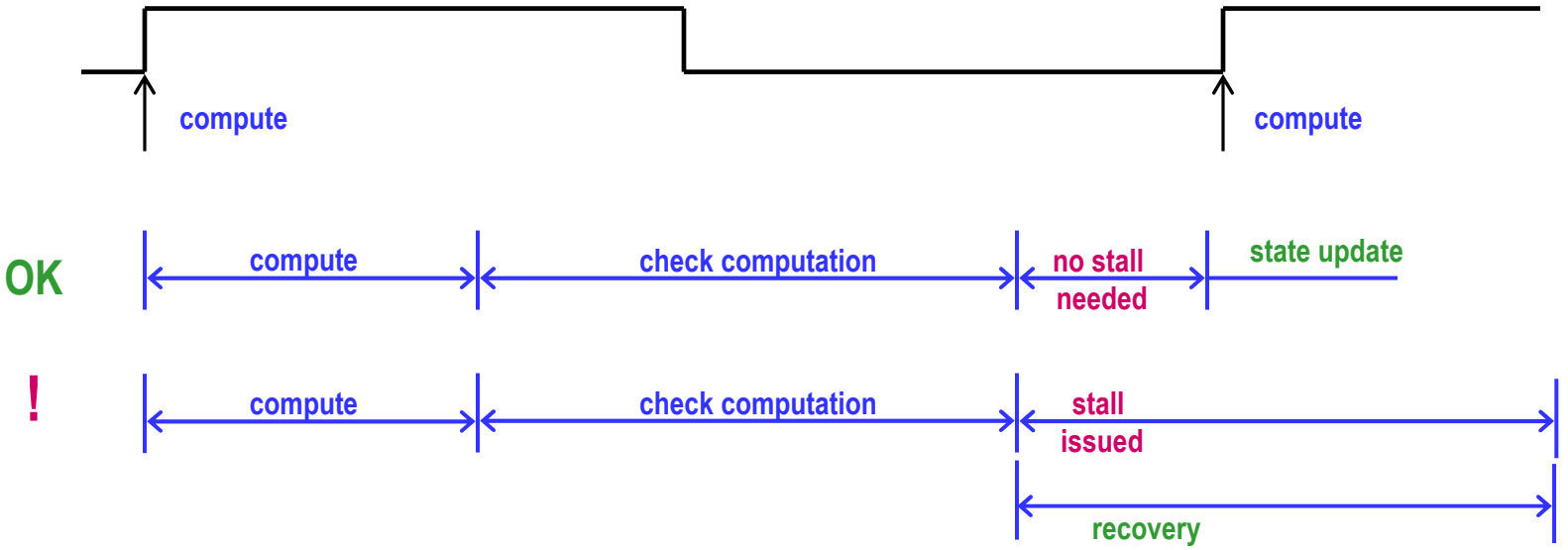




- Stall can arrive after multiple computation cycles
- Pipeline for throughput

Forward Error Recovery

- State in D only committed after checker approves
  - Checker “enables” or “stalls” design
- In case of error detected by C
  - C does not enable state update of D
  - R is triggered to perform prescribed recovery



- Stall arrives before the next computation cycle
- Pipeline for throughput

Synthesis Tasks

- Coordinate actions of D, C and R
- Guarantee the timing aspects of the recovery semantics
- Identify parts of D that need to be stalled during recovery
- Make R electrically robust and less susceptible to process variations and noise (compared to D)
- Help with generating checker/recovery circuits?

Advantages

- Separation between the blocks by intent makes the checking and recovery easy to reason about
  - Implementation guarantees language semantics
  - Generic reusable implementation template
- Tools can handle different blocks differently
- Easier to maintain the design

Outline

- The Verification Gap/Crisis
- Different Failure Modes and Runtime Validation
- Microarchitectural Solutions using Runtime Validation
- Runtime Property Checking
- A General RTL Methodology
- **Summary and Conclusions**

Summary and Conclusions

- Design complexity increasing exponentially faster than our ability to handle it
  - Increasing cost of verification
  - Increasing bug escapes
- Runtime Validation inevitable for increasing operational failures
  - Consider functional failures in the same framework
- Runtime Validation as an insurance policy for functional failures
  - Learning to live with bug escapes
- Already being considered for specific instances
- Possible use in property checking
- Can we bring this into general RTL design?
  - Clean separation of design, checking and recovery through language semantics and synthesis support