Challenges and Solutions in Post-Silicon Validation of High-end Processors

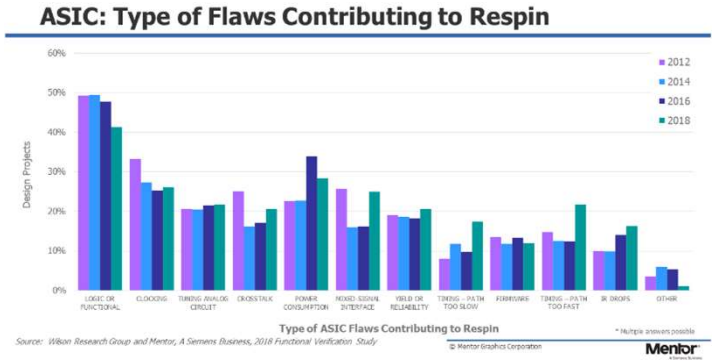
Avi Ziv
Verification and Quality Technologies
IBM Research - Haifa

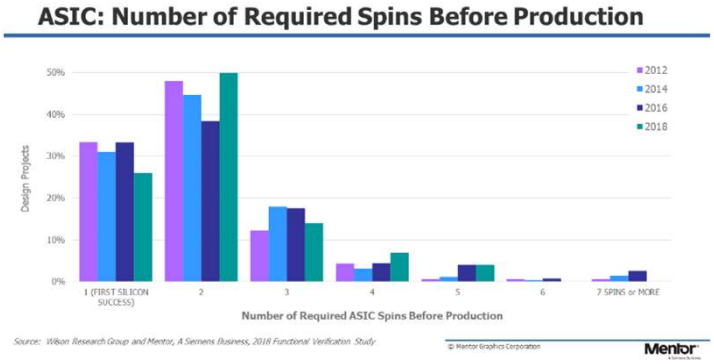
Welcome

Increasing complexity of hardware designs and shorter time to market mean that pre-silicon verification is not able (and in many cases not expected) to find all bugs prior to tape-out

Silicon, as a verification platform, is completely different from pre-silicon platforms

Unlike pre-silicon verification that utilizes well-established methodologies and technologies, post-silicon validation is still young and less organized domain





Tutorial Goals

- Explain the characteristics of silicon as a verification platform and how they affect the post-silicon validation process
- Describe solutions and advanced techniques for post-silicon validation of high-end processors
- Show how these techniques are applied in the post-silicon-validation of IBM high-end POWER processors

Presenter

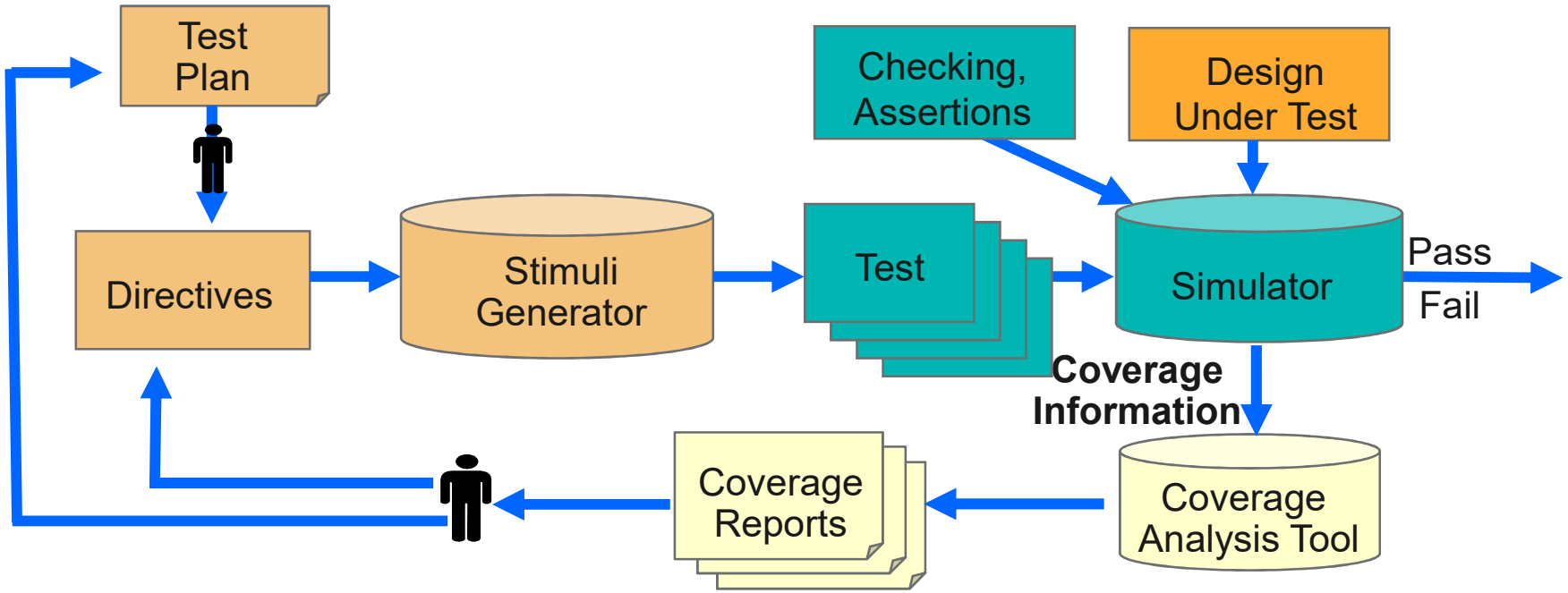
Avi Ziv (aziv@il.ibm.com)

- Research Staff Member, IBM Research – Haifa since 1996
- Researcher and project lead in all aspects of dynamic verification
- PhD in Electrical Engineering from Stanford University

Agenda

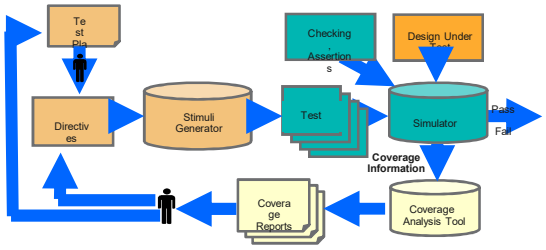
- Introduction
- The Pillars of Pre-silicon Dynamic Verification
- Silicon as a Verification Platform
- Post-silicon Validation Techniques
  - Stimuli
  - Checking
  - Coverage
  - Debug
- Case-study – IBM POWER Post-silicon Validation

Pre-silicon Dynamic Verification Flow



The Four (or Two or Three or Five) Pillars of Dynamic Verification

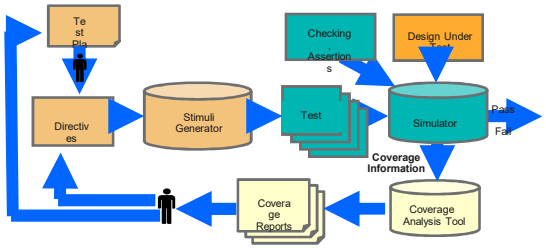
- Stimuli generation – create stimuli that exercise the DUV and reach all corners in it
- Checking – check that the DUV behaves as expected and detect when it does not
- Coverage – ensure that all aspects and features of the DUV are thoroughly verified
- Debug – identify and fix the root causes of fails



THE Pillar of Dynamic Verification

All these pillars are strongly affected by the platform on which the DUV is executed

- In software simulation, execution is slow and the DUV is accessible
  - We can do what we need and as much work as needed each cycle
  - We need to use the cycles wisely because we do not have many of them
- Other execution platforms (accelerators, emulators) improve speed at some cost
- And silicon …



Silicon as a Verification Platform

Speed

- Silicon is faster by 7-9 orders of magnitude than software simulation
  - The first hour of bring-up utilizes more cycles than the entire simulation of the project
- But
  - Very little verification work can be done each cycle
    - No time for complex computation
    - Many other limiting factors
- What about the verification per cycles x number of cycles?

Access

- Silicon is almost a true black-box
  - Virtually no control
  - Very limited observability
    - That can often be destructive

Silicon as a Verification Platform (Continued)

Load and initialization time

- It can take many seconds or minutes to load and initialize the silicon
  - So short tests are very inefficient
  - This is the main reason pre-silicon tests are not used as-is

Turn around time

- It take months to manufacture new batches of chips

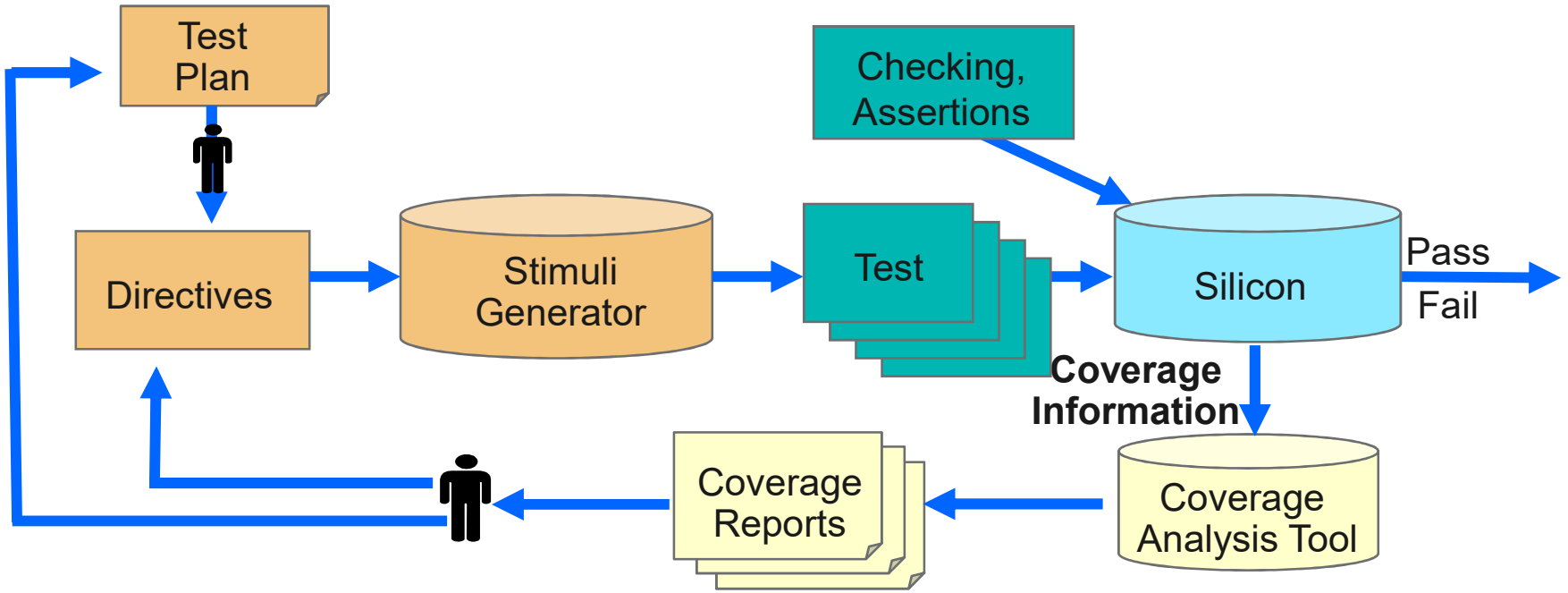
Cost

- And it costs millions of dollars

Determinism

- Silicon is not always (or almost never) deterministic
  - So observations may be hard to recreate

Post-silicon Validation Flow



The Four Pillars in Post-silicon Validation

Stimuli Generation in Post-Silicon Validation

Pre-silicon verification relies on complex and sophisticated stimuli generators

- High-quality tests that require long generation time
- Relatively short tests
- Many test-templates
  - Many of which are directed to very specific targets

This approach is not well suited to silicon

Available solutions for silicon include

- Using existing software
- Off-platform generation
- On-platform generation

Using Existing Software

The new processor needs to execute many applications in real-life

- Operating system, compilers, …

This software can be used also for post-silicon validation

- Zero generation time
- Long running time

But they have many limitations

- Not enough stress on the DUV
- Hard to target specific areas or features
- Very hard to debug

Off-platform Generation

Use tools similar to pre-silicon generators to generate stimuli

But

- The slow generation means that many servers need to generate tests for one chip
- Long load time and short tests generated by pre-silicon tools can lead to very low utilization
- Pre-silicon tools assume independent checking

Adaptation of the pre-silicon generators to post-silicon can improve efficiency

- Methods to increase length of tests with little generation effort
  - Permutation of instructions
  - Loops
  - Execute the same tests in different execution modes
- Add checking aids to the generated tests

On-platform Generation

The DUV is a processor that executes programs

- It can run a stimuli generation program and generate its own tests

This requires more than a simple stimuli generation program

- A sort of OS to activate the generator, execute the tests, etc.
- A way to check if a test executed correctly

But given this, the tool can be loaded once and run forever

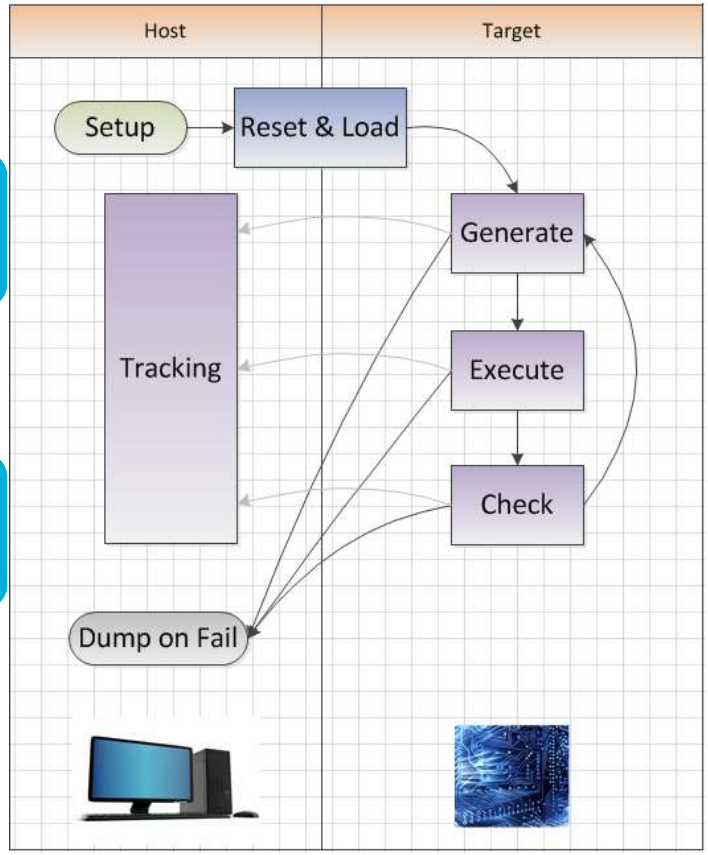
Exercisers

An exerciser is a program that:

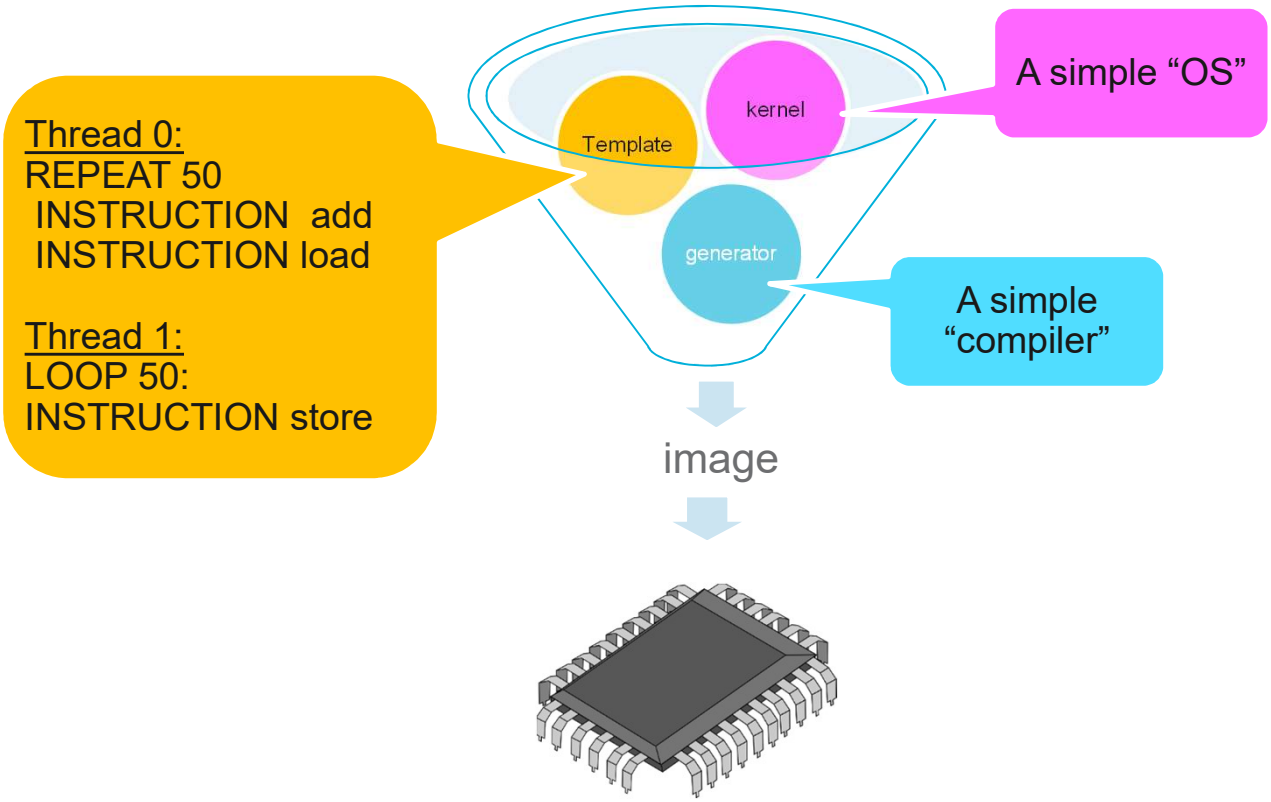
- Runs on the DUV
- “Exercises” it by testing interesting scenarios on it

It runs in an endless loop with three main phases

- Generate a test
- Execute the generated test
- Check that the test executed correctly



Exerciser Building Blocks



Exerciser Requirements

Include a random stimuli generation component (as in pre-silicon)

- Valid stimuli
- Adhere to user requests
- High quality stimuli
- Generate many test-cases from the same test-template

Simple and fast

- Can run on early bring-up silicon
- Eases debugging

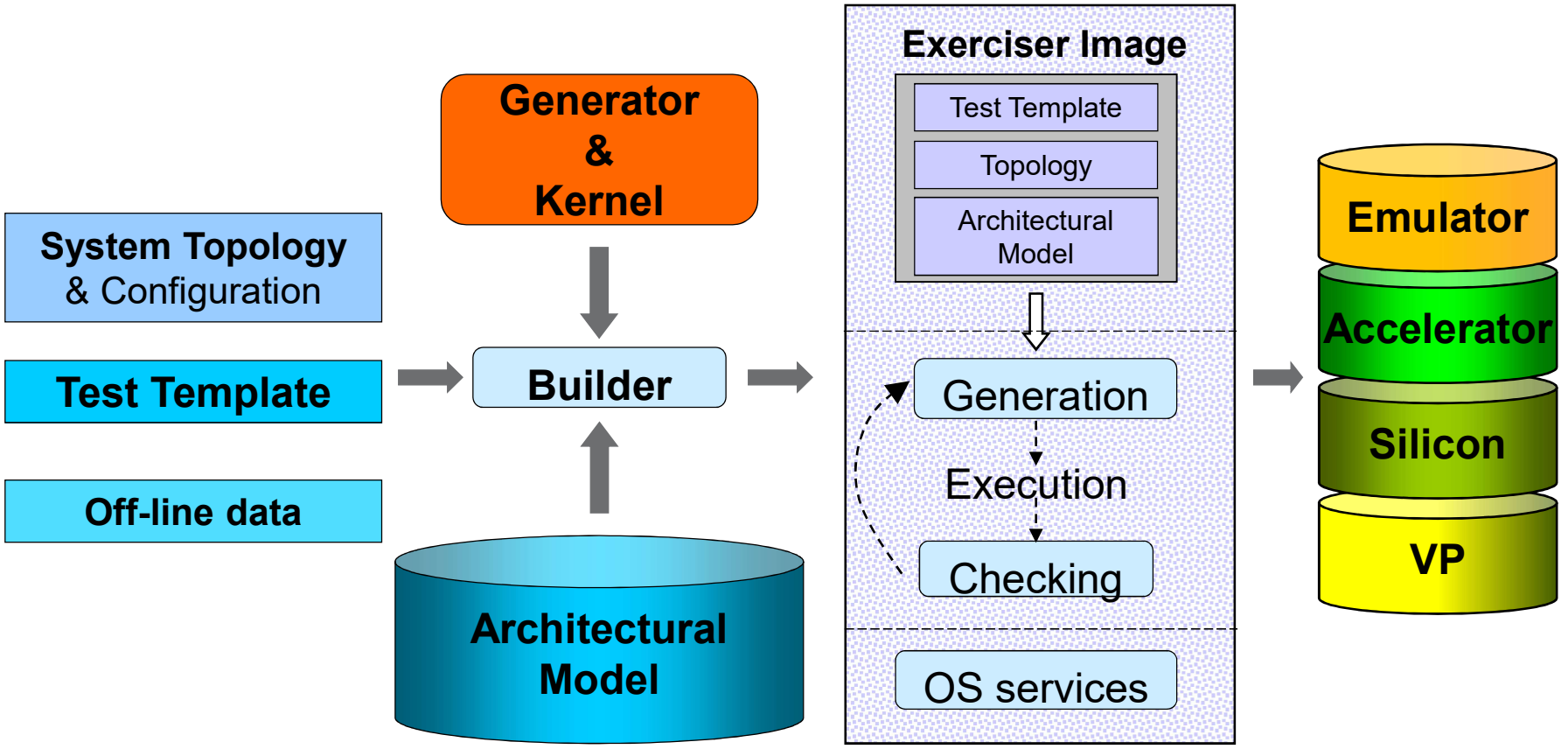
Self-contained

- Loaded once on the DUV, runs “forever”
- No interaction with the environment

Bare-Metal

- Contains OS services required by the test-cases
- Enables complete machine control

Threadmill – A State-of-the-art Exerciser



Threadmill Characteristics

Def language for test-templates:

- Rich language to describe the test-plan scenarios
- Multi-threaded support (each thread with its own scenario)

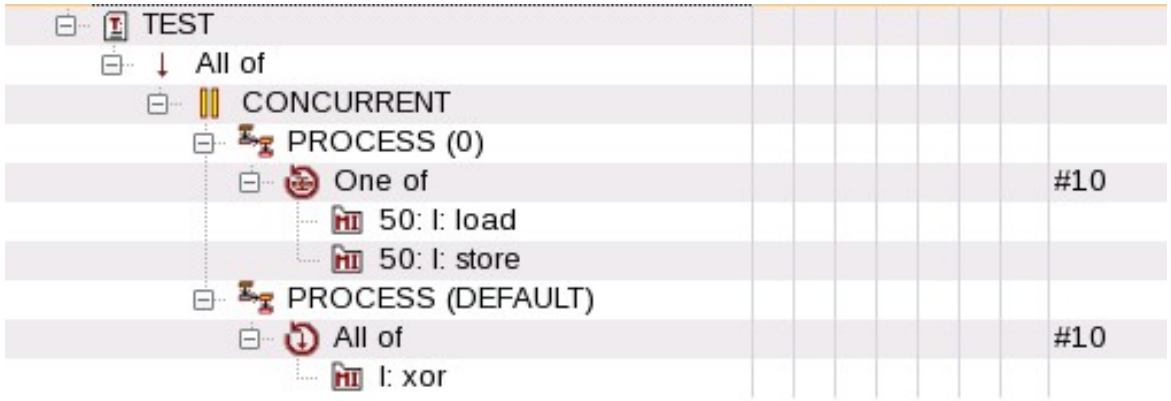
Generation:

- Concurrent multi-threaded generation
- Light-weight, on-platform
- No reference model and no state tracking
- Very fast (100s-1000s of tests per second on silicon)

Checking:

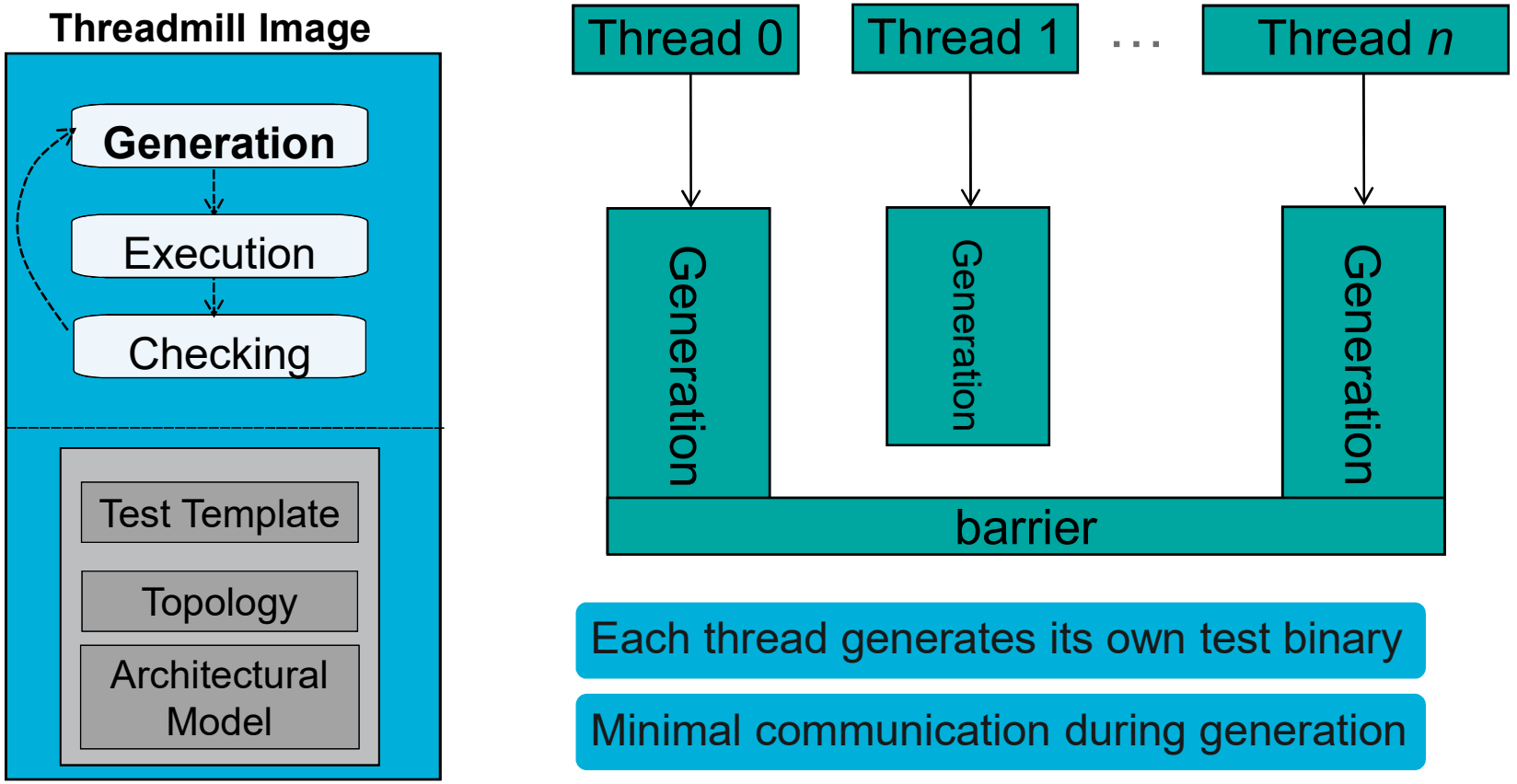
- Multi pass checking: comparing values of architectural resources (GPRs, SPRs, memory) between different executions of the same test-case
  - Variability originates from changes to the state of the design
- User ability to specify self checking
  - As part of the test-case
  - Through special attributes (over memory allocations)
- Rely on hardware checkers

Test Template Example

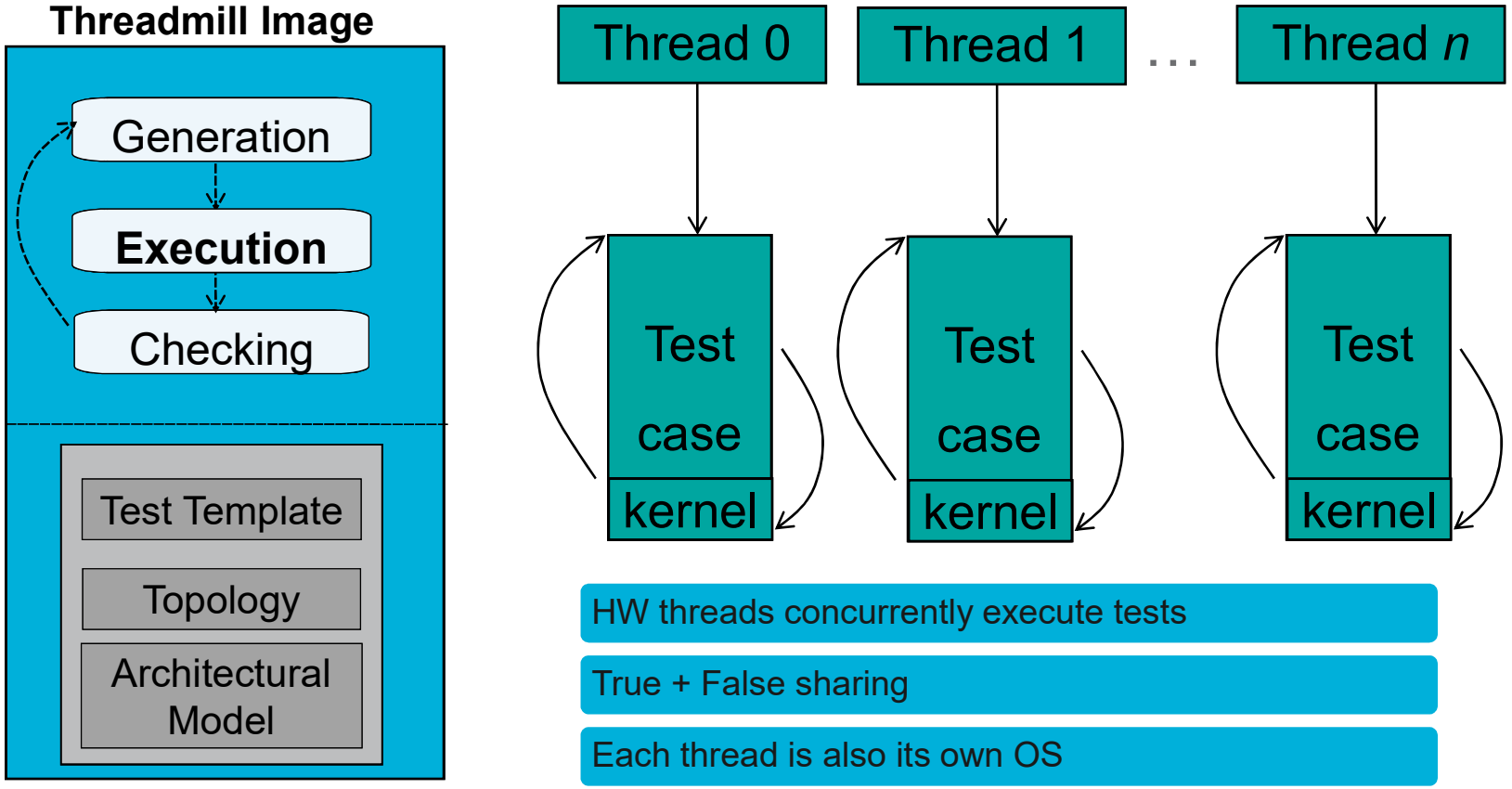


	Address	Opcode	Disassembly
➡	0x70E7E78532D0	d503201f	NOP
	0x70E7E78532D4	5cffffef	LDR d15,0x70e7e78532d0
	0x70E7E78532D8	79c066dc	LDRSH w28,[x22,#0x32]
	0x70E7E78532DC	78800e79	LDRSH x25,[x19,#0]!
	0x70E7E78532E0	780005fe	STRH w30,[x15],#0
	0x70E7E78532E4	783f6a9e	STRH w30,[x20,xzr]
	0x70E7E78532E8	bc000257	STUR s23,[x18,#0]

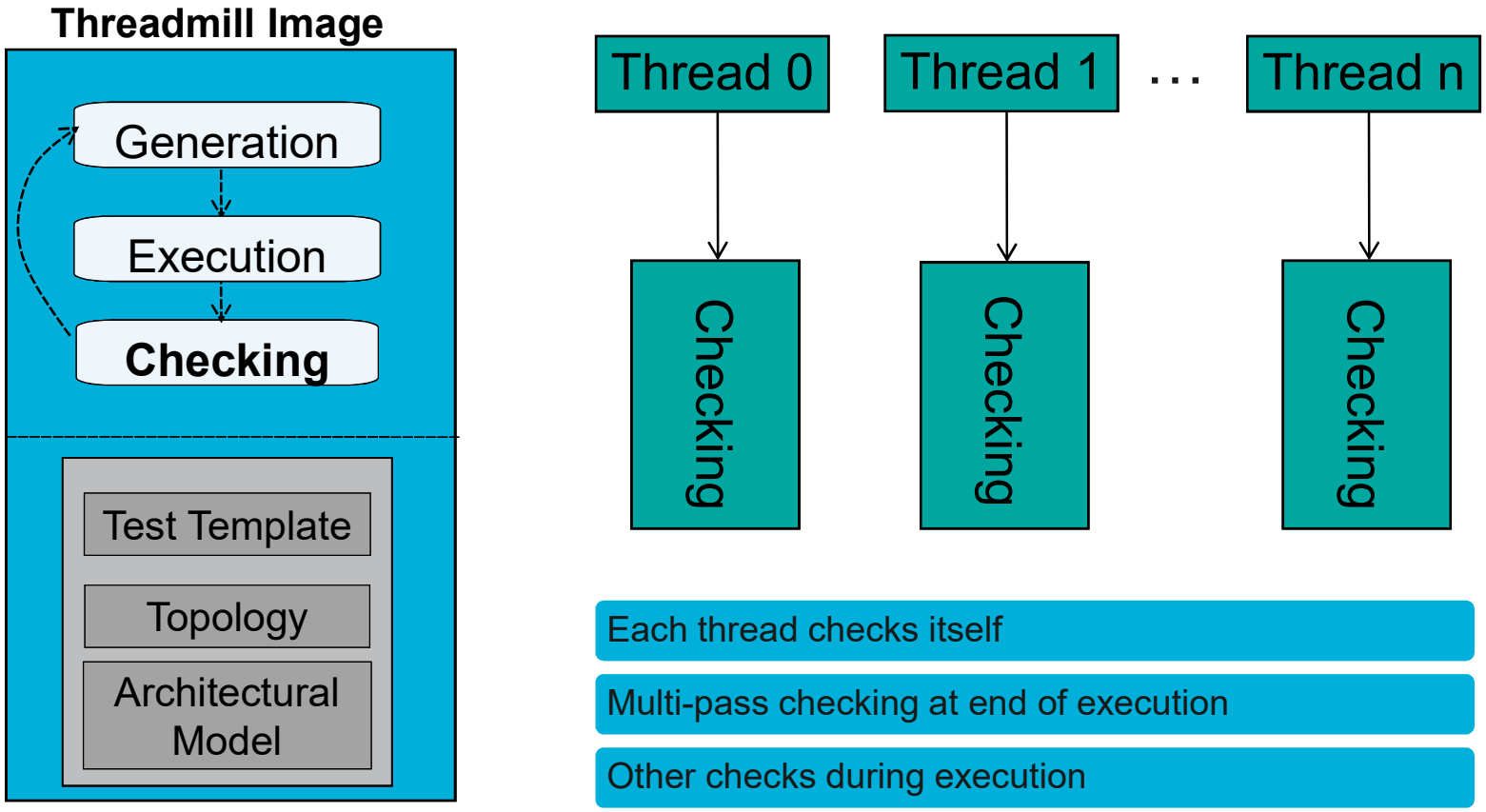
Threadmill On-platform Flow - Generation



Threadmill On-platform Flow - Execution



Threadmill On-platform Flow - Checking



Threadmill Off-platform Flow

Common code

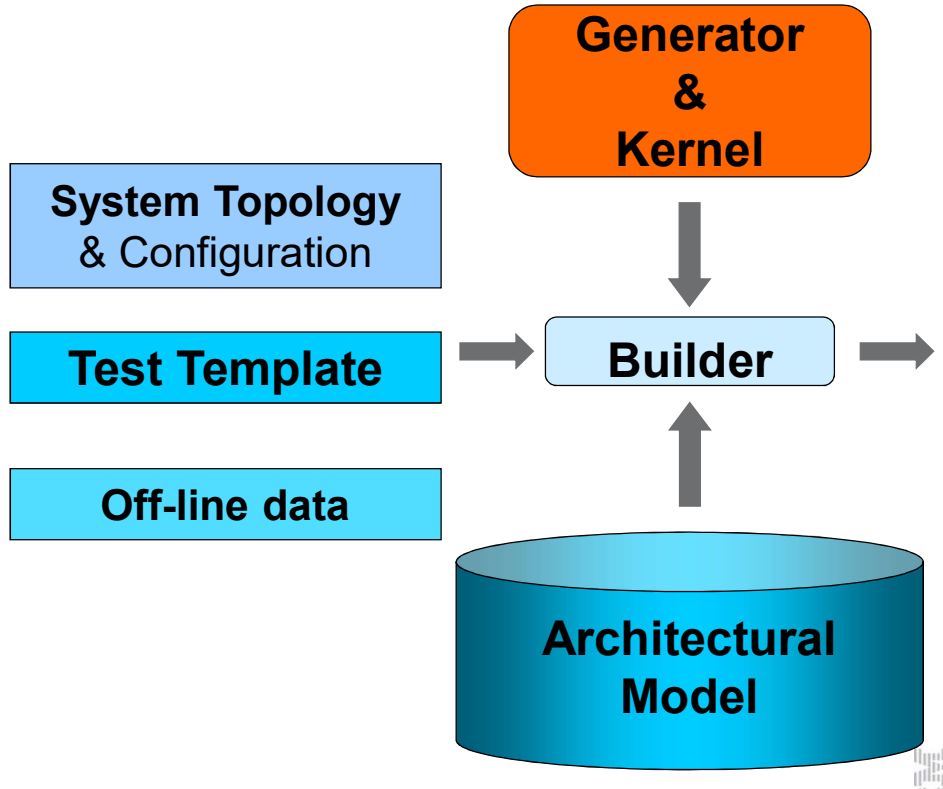
- Generation engine
- OS kernel
- Generic checking code

Architectural model

- Instructions
- Registers

Build-specific data

- Test template
- System configuration
- Other data items used by the generator

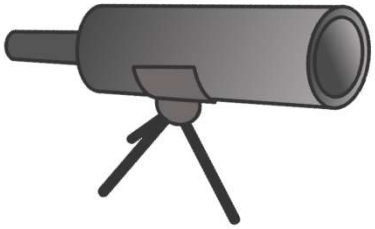


Off-platform or On-platform Generation?

- The off-platform builder can provide the on-platform ready-made data to assist in the random decisions during the generation process
  - For example, data for operands in arithmetic instructions
- At one extreme, the builder can make all the random decisions
  - That is, place deterministic tests instead of test-templates in the built image
- At the other extreme, everything can be left to the on-platform generator
- The actual operation point affects many aspects of the performance of the exerciser

Off-platform or On-platform Generation – Generic Trade-offs

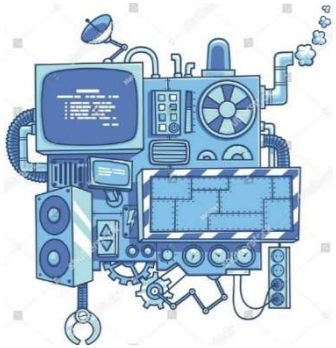
	Off-platform	On-platform
PROs	• Libraries • Long generation	• Fast • No overhead
CONs	• Large image • Choose once	• Custom c-code • Hard to debug



Instruction Generation and Checking

	Off-platform	On-platform
PROs	Reference model	Infinite tests
CONs	Fixed (small) number of tests per image	Multi pass checking

On-platform



Floating-point Data

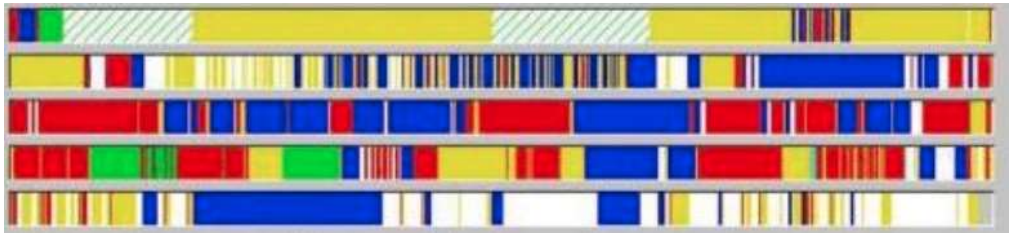
	Off-platform	On-platform
PROs	Sophisticated generation engine	Infinite amount of data
CONs	Fit between pre-fabricated data and tests	Hard to reach corner cases

Off-platform

Memory Management

	Off-platform	On-platform
PROs	• C++ Libraries • Knapsack heuristics	• Tiny image
CONs	• Large image	• Implement malloc • many locks

Off-platform



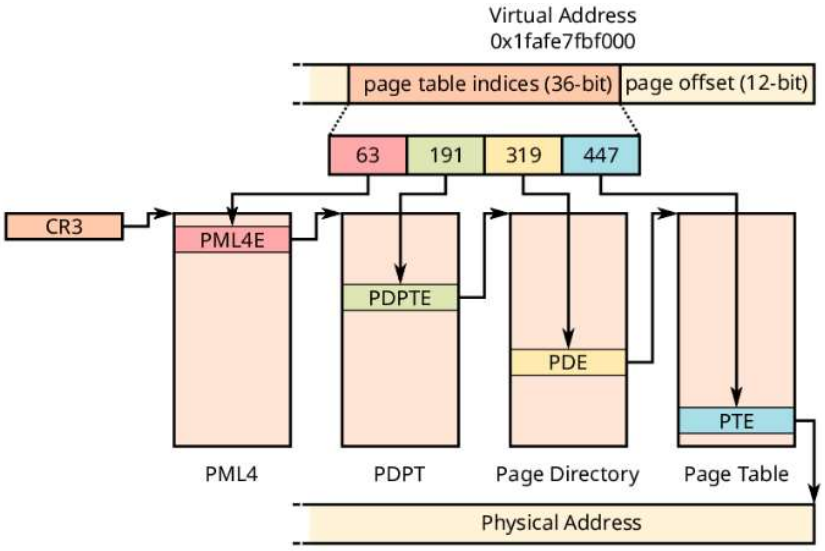
- Maximize fragmentation, collisions, and crossing
  - Cache Line, set, way, page, segment

Address Translation

	Off-platform	On-platform
PROs	• CSP Libraries	• smaller image
CONs	• Less variability	• Implement Linux translation

Off-platform

- Table sharing at all levels
- Virtual Address aliasing
- Physical Address modification (page migration)
- Context change (lpid/lpar)



Checking in Post-Silicon Validation

Pre-silicon verification utilizes many checking techniques

- Comparison to a reference model
- Scoreboards
- Behavioral rules and assertions

None of these techniques fits the silicon platform characteristics

- Need to observe the internal state
- Compute intensive

Silicon checking solutions utilize two main approaches

- Place the checkers inside the silicon
- Move the checkers to where and when they can operate

Place The Checkers Inside The Silicon

The main idea – Synthesize the checkers logic and fabricate them as part of the DUV logic

Advantages

- Problem detection is close to its occurrence
- No need to observe the signals feeding the checkers, only their output
- Checkers run at the silicon speed

Limitations

- Cannot cover all possible bugs
  - Not all checkers can be synthesized
  - Limit on how many checkers can be synthesized
- Effect on non-functional aspects
  - Area, power, timing
- Bugs in the checkers

Configurable Checkers

- We would like to place checker in problematic areas to detect cases where suspicious behavior occur
  - But we do not know where this area is and what behaviors to detect ahead of time
  - And once we know them, it is too late to place a checker
- Possible solution:
  - Place general-purpose configurable checking logic inside the chip and program it to the desired checker when needed
    - For example, programmable state-machine fed by important signals
- This suffers from all the limitations of embedded checkers, but smaller number of checkers can do the job
- We will talk more on this when discussing trace arrays and debug logic

Check Where and When Possible

The architectural state of the processor is visible

- But it can be time consuming and intrusive to observe and check it

There are points in time when such checking does not interfere with the testing

- For example, at the end of tests or in interrupt handlers

Two main issues

- Distance from failures to their detection
  - Many failures may completely disappear
- What is the correct architectural state to use
  - Or even, is there *a* correct state?

Reference Models in Post-silicon Checking

- When off-platform generation is used, a reference model can be used as a part of the generator
  - The expected architectural state can be placed as part of the test loaded to the chip
  - Or the actual state can be dumped at the end of the test
- The same reference model can also be used when on-platform generation is used
  - But at a high cost
- Instead, we can use the processor itself as its own reference model
  - The idea
    - Execute the same test more than once
    - Compare the architectural state of the different executions
- This technique is also called
  - Multi-pass checking
  - Self reference model

Self Reference Model Flow

- Generate a test
  - Including initial state
- Repeat *N* times
  - Load the test and initial state
    - Parts of the state that do not affect the architectural behavior can be changed

      For example, cache state, special execution modes
  - Execute the test
  - Save the architectural state in a dedicated memory
    - Registers + Memory + Interrupts count
- Compare the *N* saved states

Real Vs Self Reference Model

	Real	Self
Speed	Low	High
Architectural bugs	Yes	No
Development effort	High	Low

- Optimal intermediate solution
  - Use a self reference model when architectural bugs are not anticipated
    - This is true for most instructions
  - Use a real reference model when architectural bugs are expected
    - For example, instructions that perform complex data manipulation
    - This can be done by replacing the instruction with a call to the reference model

Quick Error Detection (QED)

Special technique for a self reference model

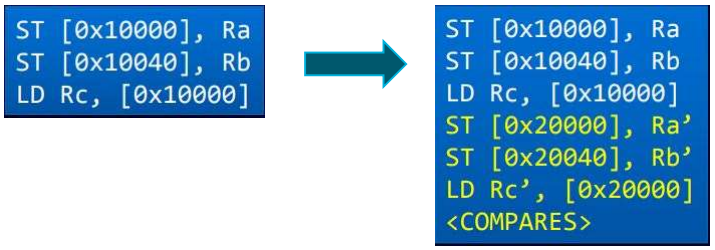
- Developed by Prof. Mitra and his students at Stanford University
- Successfully deployed in several industry projects

The main idea:

- Duplicate pieces of the test program
  - But use different resources in the duplication
- After the original code and its duplication are executed, compare the resources they modified

Advantages

- Faster detection than multi pass checking
- Can be applied to any test program



Self-checking Tests

When the results of a test are not deterministic, reference models cannot be used

- Because the result may not match the reference model but be correct

In many cases the results obey some behavioral rules

- In such cases it is common to add checking of these rules to the test itself

Example – Collier coherency test

	T0		T1
	G0 <- 0		G2 <- 0
Loop0:	ST G0, addr	Loop1:	LD G1, addr
	ADDI G0, G0, 1		CMP G1, G2
	BR Loop0:		BLT BUG:
			G2 <- G1
			BR Loop1:

Coverage in Post-Silicon Validation

Everything said about post-silicon checking is also true for coverage

But coverage is softer

- Much less damage in false detection and miss detection of coverage events
  - In fact, until not long ago coverage was not considered important in general and in post-silicon validation specifically

The same two checking solutions are the base for post-silicon coverage

- Synthesize and fabricate coverage monitors
- Observe coverage where and when possible
  - Architectural state, performance monitors

In addition, because of inaccuracies allowed in coverage, off-line approximated coverage is used

- More on this in the POWER9 experience later

A Debug Nightmare

A shift running in the lab fails after 30 minutes of running because the data in G7 is wrong

A second (and third and forth) run of exactly the same shift does not fail

- Same chip, same template, same seed, …

The goal of the debug team is to find the root cause of the fail

- Or, at least, re-create it on a more observable (and slower) platform

We can simplify the debug process and make the life of the debugger easier by providing debug aids

- Embedded in the silicon
- In the tools and environment around it

Silicon Debug Aids

Localize failures

- Embedded checkers

Improve observability

- Scan chains
- Trace buffers
- Immediate stop on failure

Other

- Cycle repeatable environment
- Hardware irritators

Scan Chains

Scan chains are serial connections between the state variables in the chip

- They are used to set and read the values of these variables using a non-functional serial interface

Scan chains can be used to dump a snapshot of the state of the chip when a failure is detected

But at a very high cost

- Destroy the state
- Stop the functional clock

They can be very useful when an embedded checker fires

- Because the state can provide significant information about the failure

But less effective when a software checker is used

- Because some history is often needed

Trace Buffers

Small buffers that store history of selected signals or events

- Typical size 100 entries of 100 signals
- A single chip can have many such trace buffers

Accompanied by (programmable) logic that controls

- Which signals or events to trace
  - Select 100 signals out of thousands of candidates
- When to sample the selected signals
- When to start and stop tracing

Which Signals To Trace?

The traced signals are used to reconstruct the important aspects of the chip state

- Or, more precisely, as much state as possible

The amount of reconstructed state strongly depends on the trace signals

- Therefore, it is important to trace the right signals
  - And finding these signals received a lot of attention in the research community

Signals are selected based on

- How much state variables they can fully reconstruct
- How much information they carry regarding the state variables
- Their connectivity
- How important they are

Immediate Stop on Failure

To get as much information from the scan chains and trace buffers, it is essential to get the data as close to the failure as possible

When an embedded checker fires, pervasive logic can immediately stop the clock in the clock domain of the checker

- And propagate the stop to near-by clock domains

When a failure is detected by software or an external tool, they need to stop the clocks as soon as possible

- This can be done using the non-functional interface to the chip or via a special command
- Parts of the state may already be ruined, but the trace buffers can still hold valueable information

Optimistic debug Scenario

An embedded checker fires

The debugger investigates the scanned data and the data from the trace buffers

Based on the findings, she decides to run the same scenario again

- Tracing different signals to get wider trace
- Stopping the trace before the previous trace started to get longer trace

The process is repeated until enough information is collected to identify the root cause

The problem – this process requires deterministic behavior of the chip

- And big complex chips are rarely deterministic
  - Access outside the chip
  - Different clock domains and asynchronous interfaces

Cycle Repeatable Environment

Deterministic, or cycle repeatable chip can significantly simplify and shorten the debug process

But building such chips is not feasible

- I/O
  - Including memory access
- Physical reasons
  - Clock propagation across the chip
- Logic reasons
  - Different elements operate at different speed

Design techniques can be used to create environment that is cycle repeatable

- For example, feed two clock domains from the same clock source
- In IBM – each core and parts of the caches are such cycle repeatable environment

Hardware Irritators

Special mechanisms embedded in the silicon that are designed to stress the hardware

- Can be activated using special purpose registers and / or during initialization
- Are used only in the lab
  - During bring-up or when trying to re-create a field problem
- Examples:
  - Error injection
  - Traffic generators
  - Artificially reduce resources sizes
    - For example, caches, TLBs

Exercisers Debug Aids

Do as much checking as possible

- Self-checking tests, multi pass checking, QED
- Shorten the time from failure to detection and reduce the risk of misdetection

Short tests

- Shorten the time from failure to detection and reduce the risk of misdetection

Small and simple

- Reduce the probability that the fail is caused by a software bug
- Increase the probability that the fail occurs during a test

Keep log of the generation process, the generated tests and the fail data

- For example, test #, seed, miscompare data

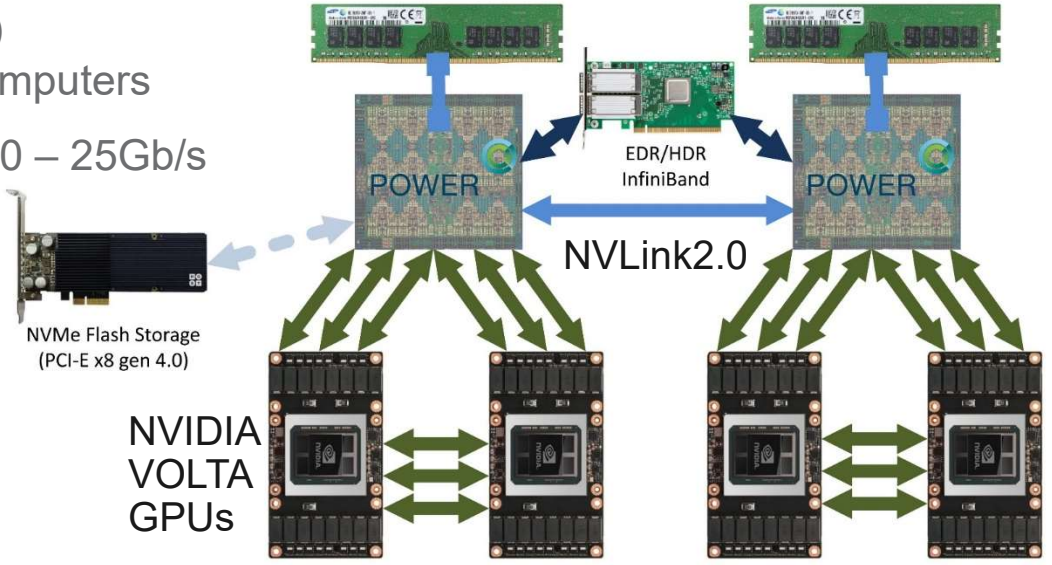
Easy to create variations of the failed tests

- For example, reduce number of participating cores, turn-off some features

The POWER9 Experience

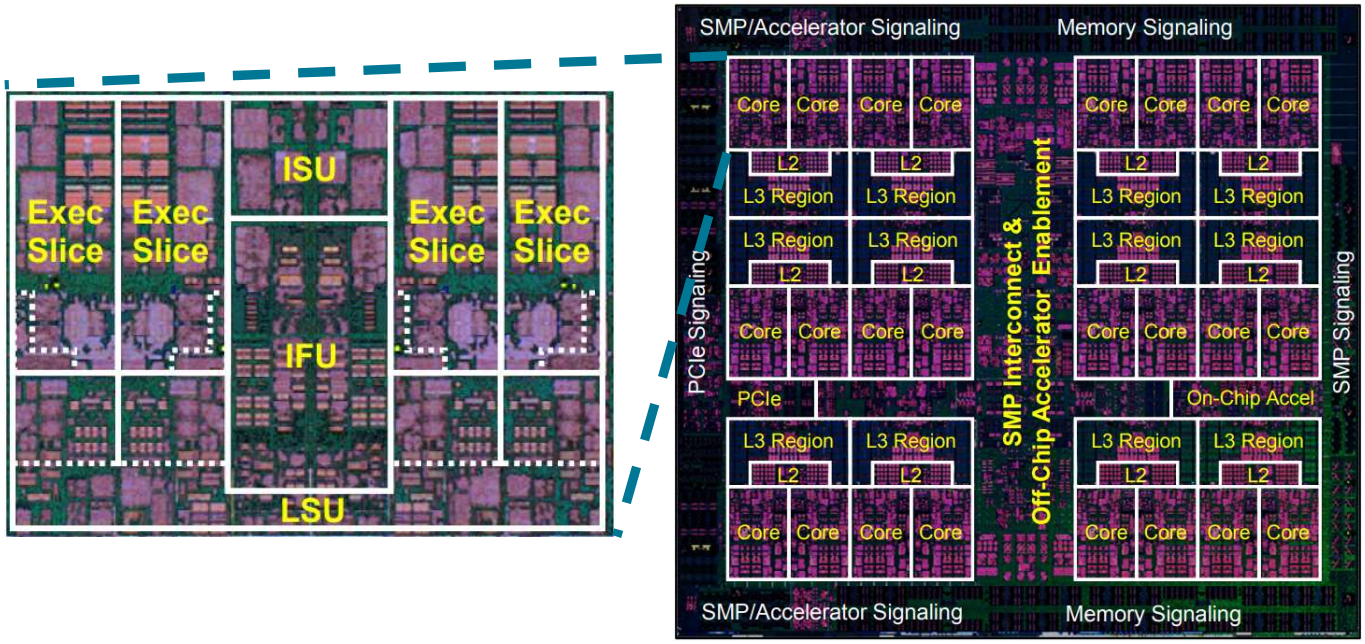
IBM POWER9

- POWER9 is IBM latest enterprise server chip
- Built to power the **Summit** supercomputer at Oak-Ridge
  - And its little brother **Sierra** at Lawrence-Livermore
- Ranked first and second (and tenth)
  in the top 500 list of fastest supercomputers
- 1st CPU with PCIE4.0 and NVLink2.0 – 25Gb/s



IBM POWER9

- 8B transistors
- 24 SMT4 cores
- Scale-Up – HPC
- Scale-Out - Cloud



IBM Post-silicon Validation Highlights

Heavy investment in embedded debug aids

- Cycle repeatable environment
- Many embedded checkers and trace buffers
  - Including a special trace buffer for coherency monitoring
    - With Hardware Trace Macro (HTM) that can direct traffic to main memory

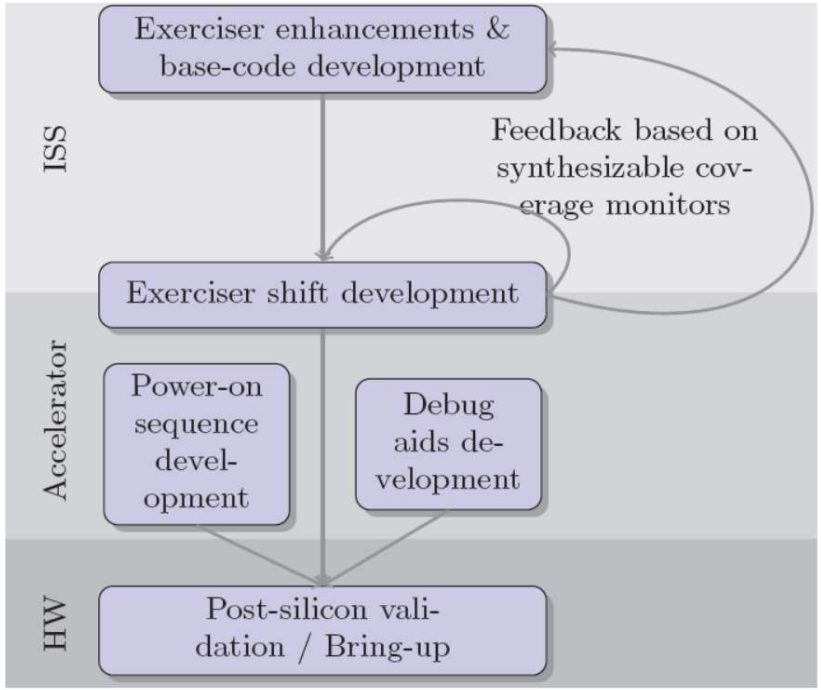
Bare-metal exercisers are the main source for stimuli

Unified pre- and post-silicon methodologies

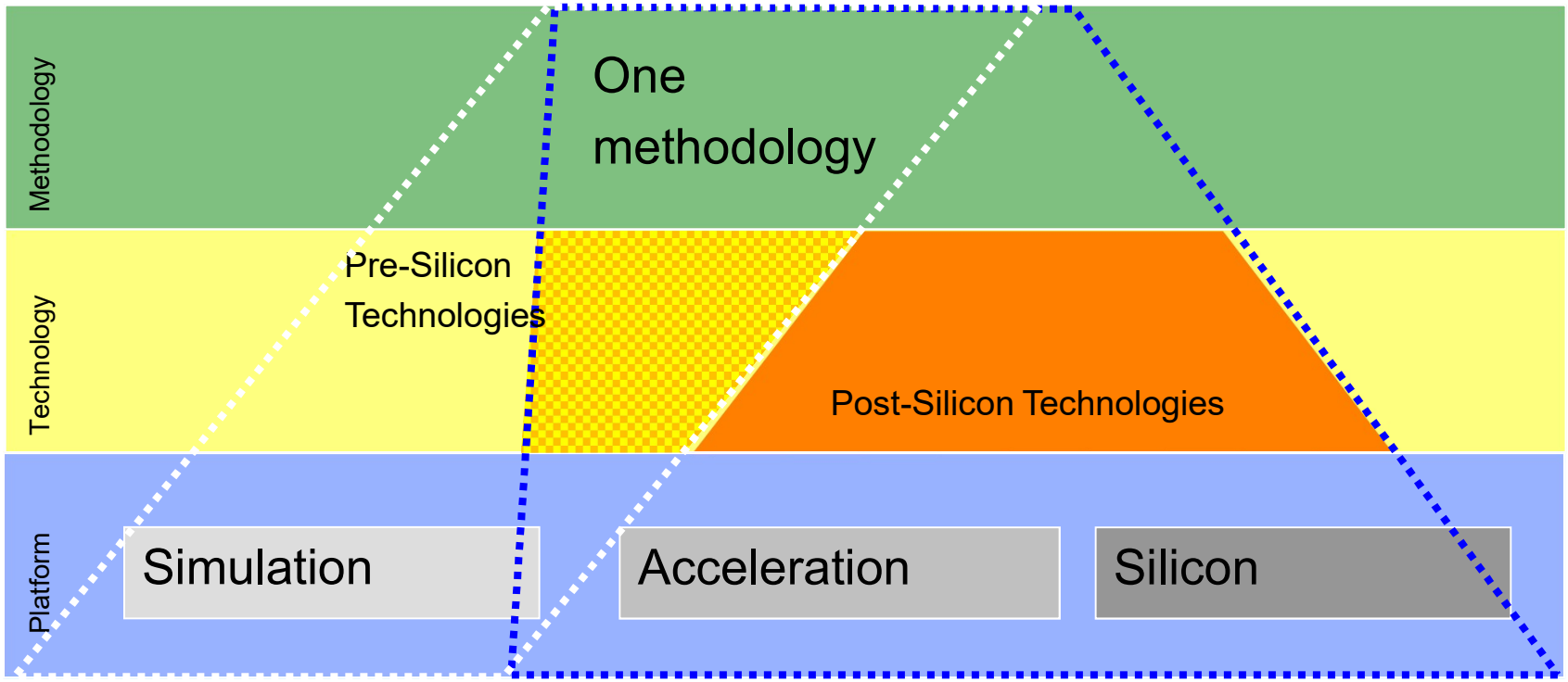
Approximated off-line coverage is used

Preparing for the Lab

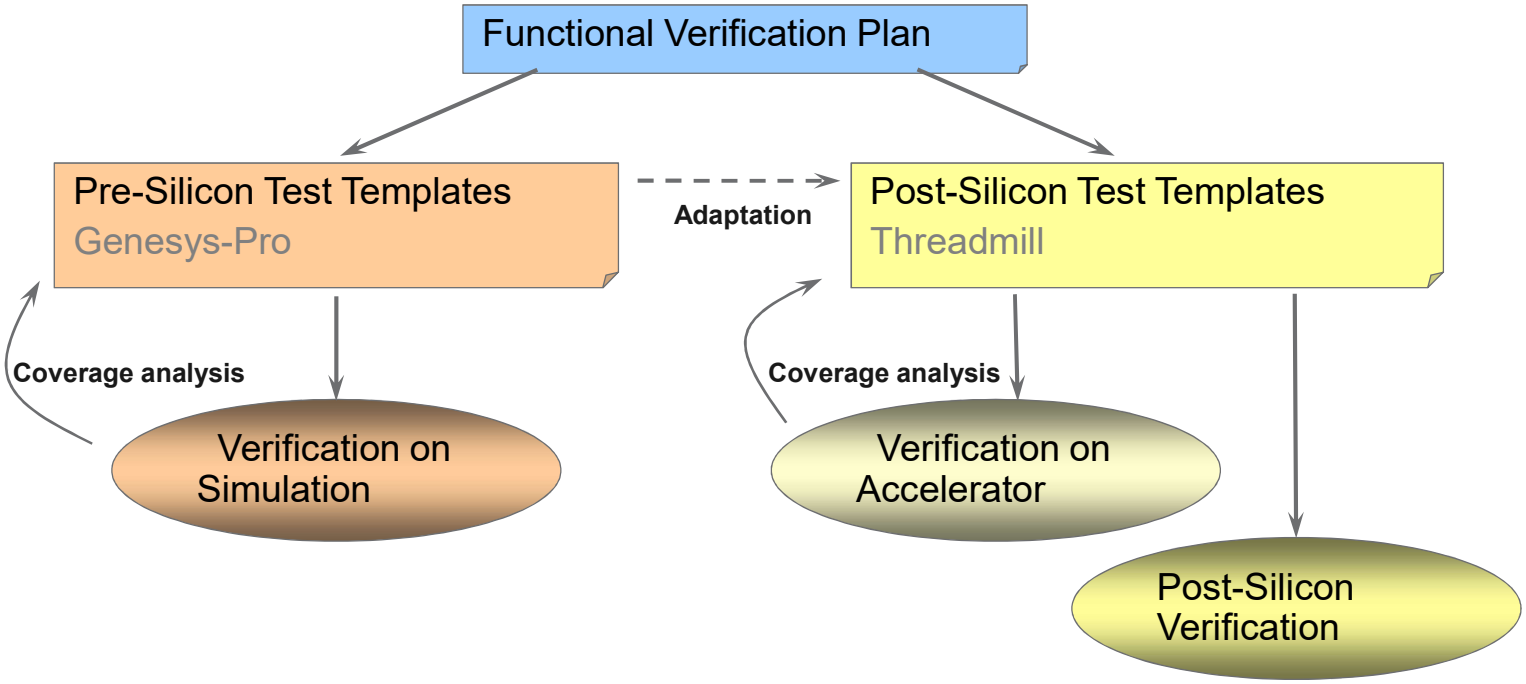
- The post-silicon validation process starts way before silicon is available
- Design and *verify* the embedded debug aids
- Build a test-plan for the post-silicon validation
- Prepare and *test* the lab tools
  - Add support for new architectural features
  - Add and tune shifts to fit the test-plan
  - Reach coverage closure



Unified Pre- and Post-silicon Methodologies



Unified pre- and post-silicon methodologies



Off-Line “Guaranteed” Coverage

The main idea

- Run post-silicon generation tools (exercisers) on a
- Collect coverage data and harvest interesting testc
  - For example, tests that contribute to coverage
- Use the harvested tests / test-templates as (p

Running the same test in simulation and sili same behavior

- Need “smart harvesting”
  - Instead of harvesting specific tests, harvest templates that provide non-negligent probability of hitting events
  - The large number of silicon cycles converts these probabilities to almost certainty

Assume

1. The execution speed ratio is 10^4
2. The probability of hitting the target coverage event in a 10-minute run is 0.1%

Then the probability of not hitting the event in 10 minutes on silicon is $(1 - 1/1{,}000)^{10{,}000} \cong 4 \times 10^{-5}$

Typical Debug Process

- Determine type of problem
  - Functional, electrical, manufacturing, …
  - Try to recreate on a different machine
    - Failure to do so indicates that this is not a functional problem
- Try to contain in a cycle repeatable environment
  - If successful, see optimistic debug scenario above
- Long set of experiments to
  - Simplify the failure
    - Reduce time to failure, reduce resources, more directed scenario
  - Gain understanding on the failure

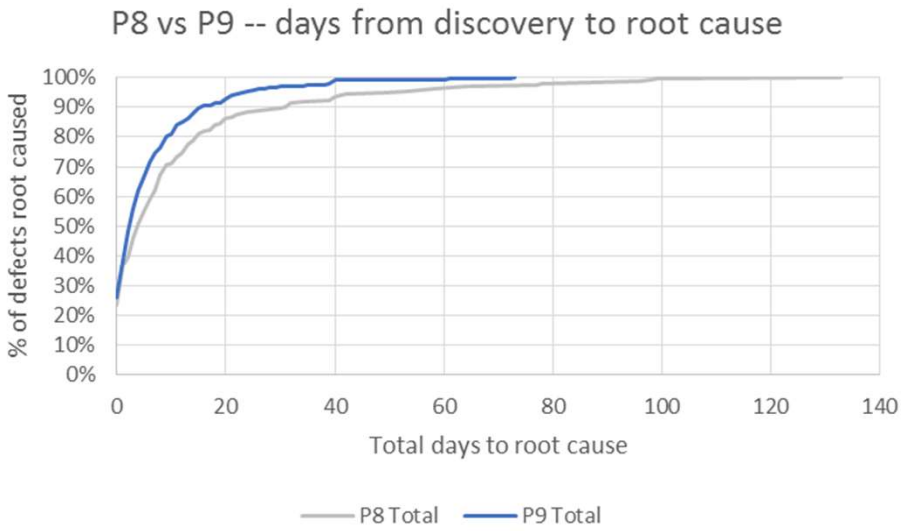
Typical Bug I - Deadlock

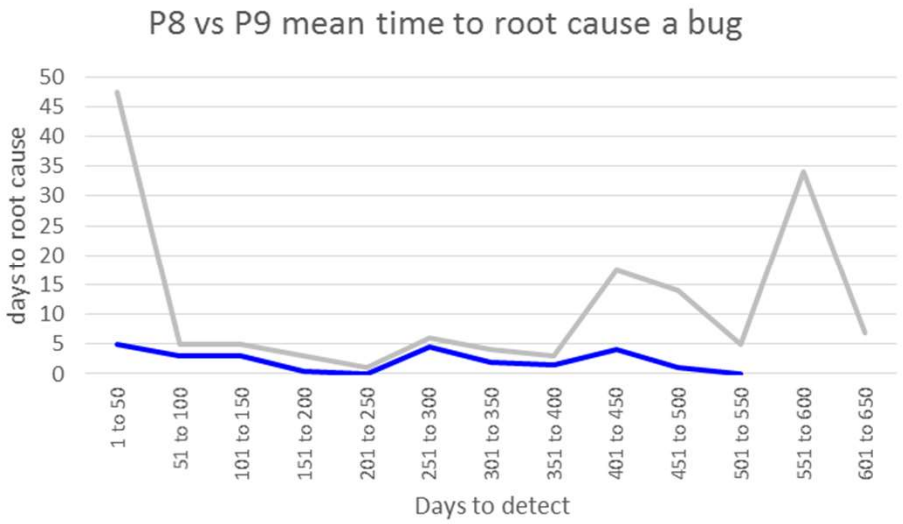
- Two methods for deadlock detection
  - A specific embedded checker for a specific type of deadlock
  - Generic checkers that detect timeout or lack-of-progress
- Luckily, even if the deadlock is detected late, it leaves important information behind
  - Specifically, who is locked and why
  - Therefore, scan dump may be enough to get a lot information to start the debug process
- From there long traces of the deadlocked resource may be required to get to the root cause
  - HTM is useful when the resource is a cache line

Typical Bug II – Bad Coherency State

- For example, two caches hold the same address in Exclusive or Modified state
- It is very hard to detect this type of problems
  - For example, reference model and multi-pass checking cannot detect them
- Special embedded checkers are added to detect the problem
  - Check that a read request does not receive too much data
- Even with such checker, detection can be far away from the bug
  - The checker is activated when the cache line is probed, not when the bad state occurred
- The two main debug actions are
  - Shorten the delay to detection
    - For example, with more reads
  - Collect as much information from coherency traces using HTM

Results





- P9 lab productivity 1.5X higher than P8 – 30% more logic bugs in 80% of the time
  - More complex and less mature design → more bugs
  - Better debug mechanisms and techniques
    - More bugs captured cache contained / cycle repro, faster to root cause

Concluding Remarks

Post-silicon validation is an essential part of the overall verification process

Silicon as a verification platform has significant different characteristics than pre-silicon platforms

- Speed
- Access
- …

Pre- and post-silicon validation share the same principals, use similar methodologies, but different technologies to reach their goals

The POWER9 experience

- Many reasons for successful bring-up
  - Long preparation for the bring-up process
  - Investment in embedded checking and debugs aids
  - Dedicated post-silicon validation tools
  - Experienced team