A Survey on Post-Silicon Functional Validation for Multicore Architectures

PADMA JAYARAMAN and RANJANI PARTHASARATHI, Anna University

During a processor development cycle, post-silicon validation is performed on the first fabricated chip to detect and fix design errors. Design errors occur due to functional issues when a unit in a design does not meet its specification. The chances of occurrence of such errors are high when new features are added in a processor. Thus, in multicore architectures, with new features being added in core and uncore components, the task of verifying the functionality independently and in coordination with other units gains significance. Several new techniques are being proposed in the field of functional validation. In this article, we undertake a survey of these techniques to identify areas that need to be addressed for multicore designs. We start with an analysis of design errors in multicore architectures. We then survey different functional validation techniques based on hardware, software, and formal methods and propose a comprehensive taxonomy for each of these approaches. We also perform a critical analysis to identify gaps in existing research and propose new research directions for validation of multicore architectures.

CCS Concepts: • **Hardware → Hardware validation**; *Functional verification*; *Post-manufacture validation and debug*; Bug detection, localization and diagnosis; Design for debug;

Additional Key Words and Phrases: Design errors, functional validation, multicore architectures

ACM Reference format:
Padma Jayaraman and Ranjani Parthasarathi. 2017. A Survey on Post-Silicon Functional Validation for Multicore Architectures. *ACM Comput. Surv.* 50, 4, Article 61 (August 2017), 30 pages.
https://doi.org/10.1145/3107615

1 INTRODUCTION

Ongoing demand in technology drives the need for designing multiple processors performing parallel tasks on a single multicore chip (Blake et al. 2009). The phenomenal growth in the number of cores makes the design more complex. Subsequently, the functional complexity of verifying these high-end processors to eliminate design bugs (hardware errors) also increases exponentially.

Design bugs or errors manifest themselves as electrical or timing errors, functional errors, and structural errors in the manufactured prototype. Based on this, there are different verification disciplines such as electrical verification, timing verification, and functional verification. This article surveys different functional verification techniques for multicore architectures.

To understand the significance of functional verification, let us consider two examples.

The first example is the design of an industrial robot for assembly operations. A cost-effective solution is to use a multicore where the motion control of a robot arm runs on one core and the

Authors' addresses: P. Jayaraman and R. Parthasarathi, Department of Information Science and Technology, College of Engineering, Guindy, Anna University, Chennai, TamilNadu, India 600025; emails: {padmab3, ranjani.parthsarathi}@gmail.com.

https://doi.org/10.1145/3107615

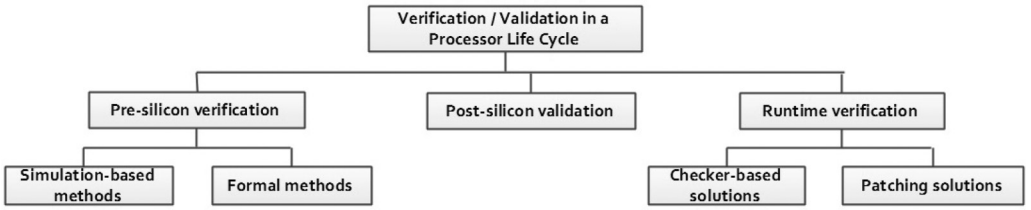

Fig. 1. Verification and validation during processor development.

human-machine interface runs on another core. The core that runs the motion control has to be more compute intensive than the other core. By using a real-time operating system, the robot responds quickly and predictably to real-time events. If the motion control arm of the robot is not functioning correctly due to an error in the control logic, then it can lead to outage of the whole production line. To address this issue upfront, it is required that coordination, calculation, and control aspects of the robot's arm be verified in a parallel manner.

The second example is the use of Many Integrated Core (MIC) architectures for accelerating high-performance computations in seismic image processing for the oil and gas industry. Seismic images are used to locate oil resources and reduce costs. Nearly 50,000 nodes in various seismic processing centers work in parallel to collect, process, and analyze seismic images together with reservoir modeling and simulation while drilling for oil. Parallel processing involves massive simultaneous two-dimensional (2D) and 3D calculations. Any functional error in the MIC could result in a system hang or shutdown of the nodes. This would lead to server downtime in locating the oil resources followed by revenue loss to the company. Therefore, functional verification of MIC includes verifying parallelization, thread affinity, and memory demands.

The aforementioned examples highlight the need for functional verification to avoid hazards or disasters in real-world scenarios.

As shown in Figure 1, verification and validation happen throughout the processor development lifecycle in three stages, namely pre-silicon, post-silicon, and runtime verification (Wagner and Bertacco 2010). Pre-silicon verification involves testing the design prototype (before it is fabricated) to prove that the design meets its specification. Post-silicon validation involves testing the first fabricated chip *(first silicon)* to check if it meets its design specification. Runtime verification is aimed at detecting design errors that happen over time. It also detects design errors that have found their way into the final product.

In the literature, the terms verification and validation are used synonymously as the problem of detecting design errors remain common to both of them (Wagner and Bertacco 2010). We also follow the same approach in this article.

Pre-silicon techniques detect bugs in the Register Transfer Level (RTL) model of a new design. These techniques are broadly classified into simulation-based methods and formal methods. In simulation-based methods (Wile et al. 2005), processor instructions are generated as test sequences in a random manner and fed to the RTL model of the design. The results computed by the RTL model are verified against a known reference model (golden model) (Mishra and Dutt 2005b). Their disadvantage is that the speed of simulation is actually several orders of magnitude slower than when the tests are run on the actual hardware. Due to their limited speed, they are used to verify only specific functionalities and therefore are not exhaustive in verifying a new design (Vasudevan 2006). As a result, these methods do not guarantee the absence of a design error in a design.

Formal methods (Bryant and Kukula 2003) use mathematical proofs to prove that a design meets its specification. They include equivalence checking (Mishchenko 2012), model checking (Clarke et al. 2000), automated theorem proving, and automata-theoretic techniques (Moore et al. 1998).

These techniques are predominantly state-based. Hence, in multicore designs, an exponential rise in the size of the state machine results in state space explosion. As a result, formal methods are used for verifying smaller units in a multicore design and not the chip as a whole.

Post-silicon validation begins when the first silicon is available in actual hardware. The key advantage is that the tests run faster on the bare metal when compared to pre-silicon techniques. But it has challenges like limited observability and controllability that make it difficult to monitor the internal signals in the hardware and to reproduce the bugs (Bertacco 2010). Due to the aforementioned limitations of both pre- and post-silicon techniques, it is quite difficult to provide a "totally bug-free" design and the bugs do escape into the market. Hence, runtime verification is needed.

Runtime or dynamic verification is classified into two approaches, namely checker-based solutions and patching solutions. Checker-based solutions (Austin 2000; DeOrio et al. 2007) deploy a separate hardware unit for monitoring and detecting the error. On the other hand, patching solutions (Sarangi et al. 2007) bypass an error by disabling specific units using programmable techniques. However, patching solutions only apply when the error has been discovered and a solution to bypass it has been identified within the capabilities of the patching engine. Thus, runtime verification is oriented toward solving a specific issue for a processor.

A study by Collett Research group in 2004 revealed that a leading cause of chip *re-spins* are functional issues (Collett 2004). A recent survey in 2014 concluded that nearly 60% of design errors are due to functional issues (Foster 2015). They throw light on the fact that functional validation gains more significance during post-silicon debug. Another study in 2014 by Mentor Graphics revealed that verification time exceeds design time by 57% (WilsonResearchGroup 2014). As the time-to-market shrinks to catch up with market share, the time spent on verification needs to be reduced. As pre-silicon verification is slow, and runtime verification addresses only some specific issues, these studies highlight the fact that post-silicon validation assumes greater significance for delivering a reduced bug-free design within a short time-to-market.

In recent times, both industry and academia have made efforts to understand post-silicon validation. To get a good perspective of all the work, this survey endeavours to provide a clear picture of the validation techniques that are practiced by the industry. We begin by examining the errors of a few multicore architectures to identify the regions that require more validation effort.

From the analysis, we identify that there is a shift in verification focus when moving from single core to multicore due to their non-deterministic behaviour. The non-deterministic behaviour arises from the (i) parallelism exhibited through distinct layers of memory hierarchy, (ii) heterogeneity in cores, and (iii) cross-core communication. Core unit is the prime area for single core verification, whereas uncore units gain importance in multicore verification. Based on this, we propose a detailed taxonomy of validation techniques for the core and uncore units of a multicore processor. We then highlight directions for further research in validation.

This article is structured in the following manner: Section 2 provides an analysis of the design errors. Section 3 proposes a high-level taxonomy of functional validation approaches. This is followed by a detailed taxonomy in Sections 4–9. The research areas are identified and brought to attention in Section 10. The entire work is summarized in Section 11.

2 OVERVIEW OF DESIGN ERRORS

To understand the importance of post-silicon validation, we begin with a discussion of a few significant examples of design errors in single core and multicore that have made their way into the market resulting in major chip recalls, delay in delivery of the product, or financial loss to the manufacturer.

The earliest one in this list is the single-core FDIV bug in the Intel Pentium (P5) processor reported in 1994. The FDIV bug (Coe 1995) was caused by an error in a lookup table that was

used to calculate the intermediate quotients necessary for the floating-point division. It affected the floating point FDIV instruction in division problems that had more than five digits. Its impact was that it brought a great loss to the company.

A more recent example is the Skylake bug (in the Intel sixth-generation Core i7-6700K and Core i7-6600K) reported in 2016. It made the chip to freeze under "complex workloads." When a Prime95 application ran on a Skylake CPU with a maximum number of threads, the system crashed. Hyper-threading was disabled to solve the issue (ExtremeTech 2016).

Similarly, a Transactional Synchronization Extensions (TSX) error in the Intel Haswell (Xeon) and early Broadwell processors were reported in 2014. It made the software hang when TSX instructions were used. These TSX instructions were used in server-class applications like transactional database servers. A microcode update was released by Intel to disable TSX instructions to ensure stable operation of the processor (PCWorld 2014).

A hypervisor-bursting bug in the AMD (AMD Opteron 3300 / 4300 / 6300) Piledriver processor created system instability. The unpredictable behaviour during non-maskable interrupt not only allowed a virtual machine (VM) to crash its host but also allowed a VM to take over the host (TheRegister 2016).

A Translation Lookaside Buffer (TLB) bug in the Phenom quad-core processor triggered AMD to stop its shipment and made them issue a microcode patch. Whenever nested cache writes occurred, there was a race condition where L2 (private) and L3 (shared) caches had incorrect values for the same data. This resulted in a hard lock on the system or a silent data corruption. However, a solution was provided in B3 Phenom to address this issue (AnandTech 2008).

2.1 Design Error Analysis

The industry makes it a practice to collect and record the design errors in documents called errata sheets. The errata sheet contains information about the root cause of the error, its implication, workarounds, and the significance of the error. The significance of the error is denoted by its frequency and severity (Narayanasamy et al. 2007).

Design error data are considered proprietary by the industries, but a few industries make their data available in the public domain (Van Campenhout et al. 2000). In this survey, the error analysis is carried out based on the publicly available data.

We look at the errata of four multicore architectures, namely Intel's Xeon, AMD's Opteron, Intel's Xeon Phi, and ARM Cortex-A9. We begin with an analysis of errors in coherent memory architectures like the Intel Xeon E5 v2 product family (12 cores) (Intel 2015a) and AMD's Opteron (AMD 2009).

2.1.1 Intel' Xeon E5 and AMD's Opteron. Intel's Xeon E5 v2 product family errata sheet (Revision date: April 2015) reports 164 errors and AMD's Opteron errata sheet (Revision date: June 2009) reports 77 errors. These errors fall under different buckets based on the source of occurrence: uncore units, faulty instructions, debugging support, monitoring events, launch of new features, and other miscellaneous events. This is illustrated in Figure 2 and discussed below.

Uncore units:

- *Interconnect networks:* The main category for a significant source of errors is the Interconnect networks. Errors of this type include issues in both off-die and on-die interconnects. In the Xeon processor, errors in Peripheral Component Interconnect Express (PCIe) and Quick Path Interconnect (QPI) contribute to nearly 26% of the total errors. In AMD, Hypertransport (Lightning Data Transfer) errors account for 13%.

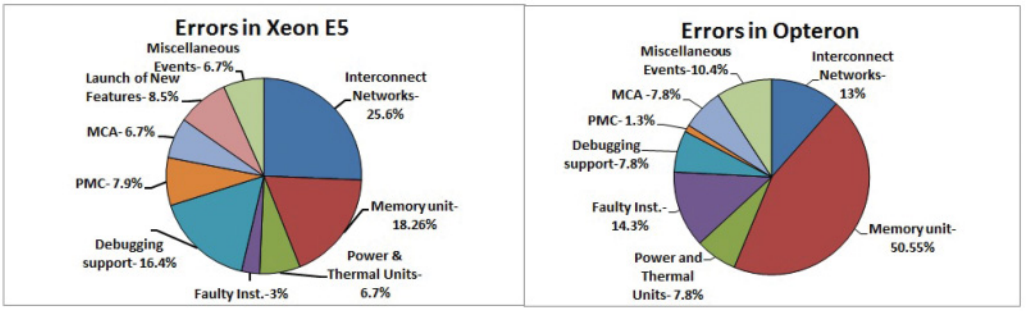

Fig. 2. Design error analysis for Intel Xeon Processor E5 v2 product family (Revision date: April 2015) (Intel 2015b) and AMD Opteron (Revision date: June 2009) (AMD 2009).

- *Memory unit:* Errors in the memory unit occur due to memory ordering violations, inconsistent memory types, memory latencies, page faults, memory scrubbing, TLB errors, cache parity errors using Error Correcting Codes (ECC), deadlocks (a combination of memory related events), cache coherency issues with hardware prefetching, and Direct Memory Access (DMA) errors.

  One category of a DMA error is the prevention of transactions, when one agent becomes the owner of another agent's resources. It is termed as lock quiescent flow (Intel 2015a). When this happens in DMA, it prevents the completed operations from entering the write queue. Hence, the DMA state machine gets deadlocked. Another scenario is when DMA processing fails to halt either on detecting 64-bit addressing errors or memory read-write collisions. These are a major source of errors (50.55%) in Opteron but contribute comparatively to a lesser percentage (18.26%) in Xeon.
- *Power and Thermal Units:* Errors in power and thermal units arise due to issues in power, voltage, and temperature. In Xeon, these errors are exposed via the Platform Environment Control Interface (PECI) and account for 6.7% of the total errors. In Opteron, these errors account for 7.8% of the total errors.

Faulty instructions: About 3% of the errors in Xeon and 14.3% of the errors in Opteron are under this category.

Debugging support: Errors occur due to issues in Advanced Programmable Interrupt Controller (APIC) (timer and interrupt) events, wrong error reports, interrupts, debug exceptions, and breakpoints. In Xeon, these errors add up to 16.4%, and in Opteron they account for 7.8%.

Monitoring Mechanisms: There are two monitoring mechanisms namely Performance Monitoring Counters (PMC) and Machine Check Architecture (MCA). Issues in performance monitoring counters account for 7.9% of the errors in Xeon and 1.3% of the errors in Opteron. Similarly, errors in machine check exception handling, lost values in Machine Check (MC) banks during a power-on reset, logging of uncorrectable errors in MC status registers, failure of platform recovery after MC, and loss of MC logs on warm reset account for 6.7% of the errata in Xeon and 7.8% in Opteron.

Launch of New Features: The release of new features like Quick Data Technology (QDT) and Virtualization Techniques (VT) have introduced new errors that add up to 8.5% in Xeon. Examples of errors in VT include system hang due to multiple VT translation requests, master abort during a VT memory check error on QDT and VM exits on various combinations of events. However, in Opteron, virtualization feature (AMD-V) has not reported any errors.

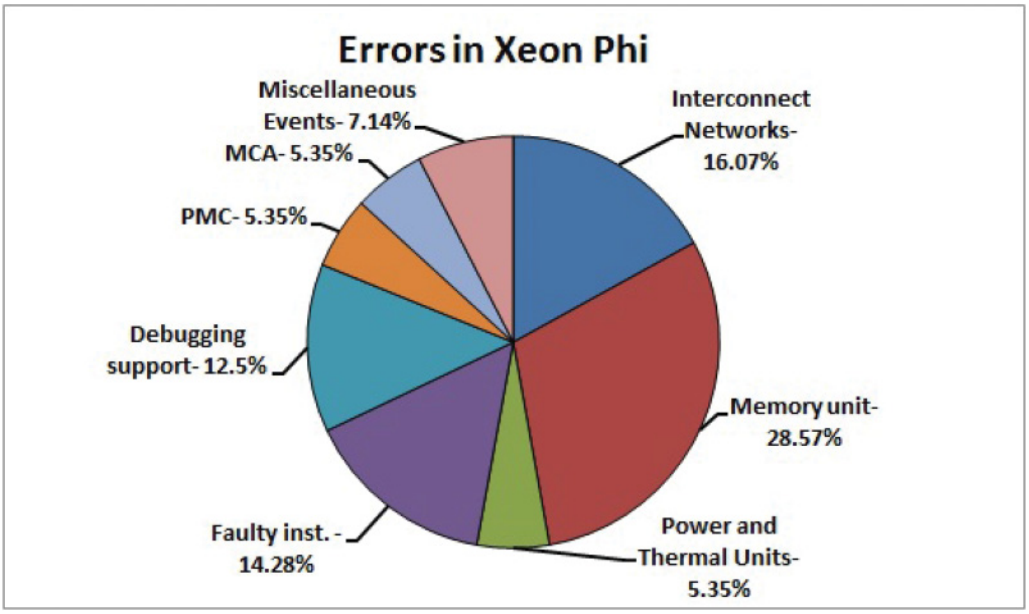

Fig. 3. Design error analysis for Intel Xeon Phi.

Miscellaneous Events: Events like warm or cold reset, Restore from System Management (RSM), delayed clock/frequency, system hangs (due to bypassed reads, stalled probe, core count greater than 14, etc.), and issues in micro patches, software prefetch, sequential prefetch account for 6.7% of the errors in Xeon and 10.4% of the errors in Opteron.

2.1.2 Intel's Xeon Phi. Next, we consider a mesh-based non-coherent architecture, that of Intel's Xeon Phi (72 cores). Its errata sheet reports a total of 56 errors (Intel 2015a), where 28.57% of the errors occur in the memory unit (including DMA), 16.07% of the errors occur in interconnect networks, 14.28% of the errors are due to faulty instructions, 12.5% of the errors occur due to debugging support, 5.35% of the errors each occur due to PMC and MCA, 5.35% of the errors is in power and thermal units, and 7.14% of the errors is due to other miscellaneous events (Figure 3). Thus, we find that majority of the errors occur in memory hierarchy and interconnect networks.

2.1.3 ARM Cortex-A9. We next examine the ARM Cortex-A9 family of processors. ARM Cortex-A9 is used in the i.MX6 DualPlus/Quad family of the application processor. The i.MX6 multicore platform has dual or quad ARM cores running at 1.2GHz. Its errata sheet reports 175 errors in various subsystems, of which 50 errors are in the ARM Cortex-A9. In the i.MX6 application processor, 28.5% of the errors are ARM processor errors, 48% of the errors are due to connectivity peripherals, 12% of the errors are due to multimedia peripherals, 5.14% of the errors are due to internal memory (ROM), 2.85% each due to security and timer issues, and 0.6% due to system control peripherals.

Of the 28.5% errors present in ARM Cortex-A9, 17.67% are in the memory unit, 4.56% are due to debug exceptions and interrupts, 3.42% are due to monitoring events, and 2.85% are due to clock issues, special instructions, and other miscellaneous events. This is shown in Figure 4.

In all the above cases, we find that the majority of the errors are in the memory and interconnect networks. We see that 48.05%, 17.67%, 16.46%, and 16.07% of the errors appear respectively in the

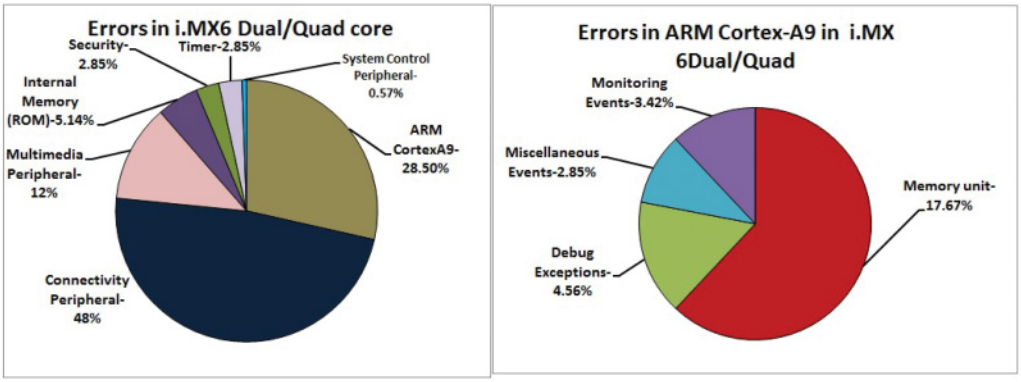

Fig. 4. Design error analysis for i.MX6 Dual/Quad Core and ARM Cortex-A9.

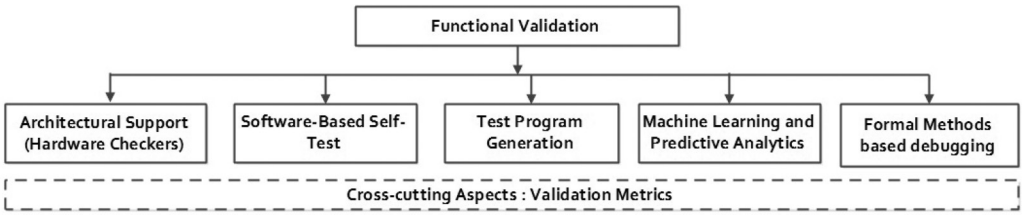

Fig. 5. High-level taxonomy of functional validation techniques.

memory units of Opteron, ARM Cortex-A9, Xeon and Xeon Phi. Likewise, 48%, 26%, 16.07%, and 13% of the errors appear respectively in the interconnects of i.MX6, Xeon, Xeon Phi, and Opteron.

This shows that, with multicore architectures, there are more issues in memory hierarchies and interconnect networks. This has prompted the industries to direct verification efforts in these areas to reduce the number of design errors.

In the following sections, we look at the various techniques that have been developed for the validation of these "uncore" components and the "core" components. Since there is a plethora of techniques, we classify them based on a new taxonomy.

3 TAXONOMY OF FUNCTIONAL VALIDATION TECHNIQUES

Although different taxonomies exist in the literature for functional validation (Bhadra et al. 2007), we propose a new taxonomy, keeping in mind the new dimensions that have to be considered in the multicore scenario. At a high level, we classify validation techniques as follows: Architectural support–assisted techniques, Software-Based Self-Testing techniques, Test Program Generation–based techniques, Machine Learning and Predictive Analytics–based techniques and formal methods–based debugging techniques (Figure 5). The low-level deeper taxonomies are presented in the following sections.

Architectural support–based techniques (hardware checkers) (explained in Section 4) involve modifying or embedding new hardware in the design for verification (Wagner and Bertacco 2010).

Software-Based Self-Testing techniques (Section 5) involve writing test programs in high-level language and running at normal speed of the processor to verify the new design. This supports online or in-field testing where the operating system tests without affecting the application running on the system. The operating system identifies the idle periods of the processor to execute the test program or makes use of programmable timers to execute the test program periodically at regular intervals (Paschalis and Gizopoulos 2005; Psarakis et al. 2010).

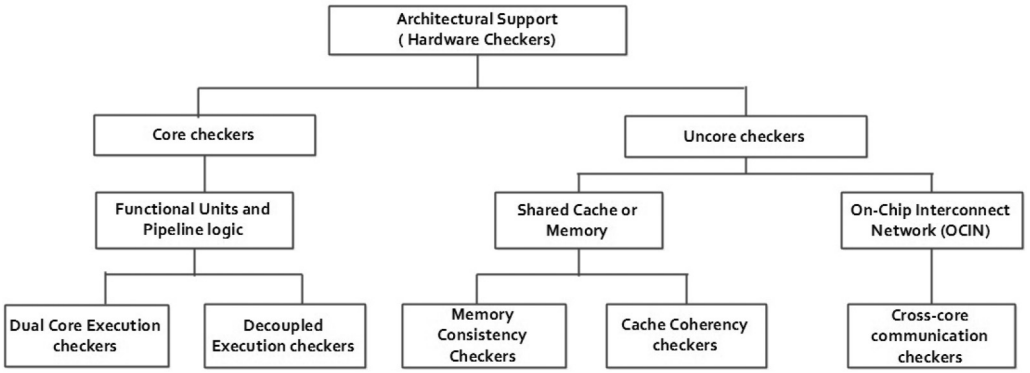

Fig. 6. Taxonomy of architectural support for functional validation.

Test-Program Generation– (Section 6) based techniques involve writing test programs in assembly language and executing them in parallel on multiple cores. This is followed by designing frameworks for automatic generation and execution of these programs.

Machine Learning and Predictive Analytics– (Section 7) based techniques consume huge amounts of test data from existing architectures to predict bugs in future designs.

Formal methods (Section 8) include techniques both with or without hardware support to assist the debugging process. These methods aid in reproducing the bug scenario by constructing an error trace.

A cross-cutting aspect that remains common for all the above techniques is the validation metric. The validation metric discussed in this article is functional coverage (Section 9).

In the following sections, we provide a detailed taxonomy for each category and explore the effectiveness of these techniques.

4 ARCHITECTURAL SUPPORT (HARDWARE CHECKERS)

Hardware checker is not a new concept in post-silicon validation, as it has its origin in the fault-tolerance domain. Hardware checkers are monitoring circuits that verify the operation of a processor. An exhaustive survey by Kalayappan et al. (2013) classified the checkers at various levels like circuit, architecture, and software for fault-tolerant architectures.

In fault-tolerance domain, the checkers are incorporated as self-checking circuits. They are packaged along with the main core when delivered to the market. On the other hand, in post-silicon validation, the checker is deactivated after validation is completed and prior to the chip being delivered to the market. This eliminates any performance degradation at the end-user site (DeOrio et al. 2009).

From a verification standpoint, we propose a taxonomy for checkers at the architectural level in Figure 6. At the highest level, we identify the target unit for validation and classify checkers as Core checkers and Uncore checkers. Core checkers verify the main core's functional units, pipeline control logic, and datapath logic. Uncore checkers verify the memory hierarchies, interconnect networks, and I/O units. We explore a set of techniques that have been used in the recent past.

4.1 Core Checkers

We note that fault-tolerant approaches (Ma et al. 2007) like Dual Core Execution (DCE) and Decoupled Execution (DE) can be exploited to assist functional validation of core units in multicore designs. In DCE, a separate checker core recomputes the instructions to verify the operations of the main core. In DE, an in-built checker unit is deployed inside the main core where the

instruction streams are decoupled and verified separately by the checker unit. Hence, we classify Core checkers as DCE checkers and DE checkers. In this subsection, we discuss the application of these techniques in single-core architectures that can be further extended for multicore designs.

4.1.1 Dual Core Execution Checkers. One of the first approaches of DCE (Zhou 2005) consists of two cores coupled with a queue. The main core pre-processes the instructions in a fast and accurate way. The retired instructions are kept in a result queue, and the checker core re-executes these instructions. The architectural states of the two cores are compared continuously, and if there is a mismatch, then the checkpoint state is rolled back for re-execution. Dynamic Implementation Verification Architecture (DIVA) (Austin 2000), one of the earliest works in dynamic verification of single-core architectures, operates on this principle. In DIVA, whenever there is a mismatch of results, an exception is raised by the checker core, and the incorrect value in the main core is replaced with the correct recomputed result of the checker core. The main core then enters into a degraded mode of operation where its pipeline is flushed to remove the erroneous results and it restarts execution of the next instruction. Here, the checker core's size and the design are kept small and simple.

An important issue in DIVA is that the main core executes the instructions at a faster rate but waits until commit happens in the checker.

There are a few similar notable approaches, such as SlipStream Processors (Sundaramoorthy et al. 2000), Filter Checkers (Yoo and Franklin 2008), Redundant Execution using Critical Value Forwarding (RECVF) (Subramanyan et al. 2010), and Redundant Execution using Simple Execution Assistance (RESEA) (Subramanyan 2010) that are based on the concept of DCE. In SlipStream processors, the leader core runs a shortened version of the program and the checker core runs the unmodified original program. Likewise, RECVF and RESEA use the second core to execute only the critical instructions in a program and verify the result with the first core. In line with this, another class of checkers termed as "filter checkers" deploy a checker hierarchy where the first-level checkers filter the critical instructions and pass the result to the second-level checkers. This checker hierarchy helps them to avoid performance degradation in scenarios where the throughput of the main core is very high but the checker's bandwidth is unable to handle it. Similarly, Fingerprinting (Smolens et al. 2006) is another technique where write operations are validated by exchanging the hash values as fingerprints between the two cores (Yoo 2007). The mismatch in fingerprints indicates an error. In the aforementioned approaches, the second core does not re-execute the entire program. It executes a skeleton of the main program to keep pace with the other core. The leader runs ahead and provides hints to the checker to accelerate its execution.

Another work in line with these is a resilient core that integrates error detection and recovery circuits to operate at a higher clock frequency (Bowman et al. 2011). An error control unit is integrated with the first five stages of the pipeline. Whenever there is a late timing transition, it suspects for the possibility of an error in that stage. The error signal is generated and pipelined to the write back stage to invalidate the errant instruction before a commit takes place. Later, the error recovery unit recovers the core from the error.

In Necromancer (Ansari et al. 2010), portions of a dead core or a faulty core are used to accelerate program execution on the active cores. The dead core cannot be fully trusted for the entire program execution, and only its valid architectural states are used to accelerate the other active cores. Although Necromancer is aimed to enhance system throughput, this idea can be extended to multicore verification where we can use the dead core as a checker core.

We can infer from the above approaches that, in a multicore environment, an idle or inactive core can act as a checker to do *verification in parallel for a cluster of active cores.* Also, a program's critical instructions alone can be verified by an idle core (checker) to reduce the congestion problem.

4.1.2 Decoupled Execution Checkers. In Decoupled Execution, the program functionalities are separated into the control stream, computation (execute) stream, and memory access stream. Each stream is verified separately by deploying either a checker core or checker unit, one for each functionality. Decoupled Execution occurs in two ways: (i) decoupled access/execute and (ii) decoupled control. Decoupled access/execute allows the access instructions to be processed ahead of execute instructions. However, in decoupled control, the control instructions are separated and processed ahead of access and execute streams (Sung et al. 2001). Apart from this, performance and correctness issues can also be decoupled (Garg and Huang 2008). Two prominent works of decoupled execution are discussed below.

4.1.2.1 Chico (Single Checker Unit). Chico (DeOrio et al. 2007) is designed to check the correctness of control operations in the pipeline of an Out-Of-Order (OOO) processor. To verify the control operations, it checks the program flow order by checking the address of two successive instructions. Chico is not a checker core, but it is a single checker block with two extra stages in the pipeline, namely setup stage and compare stage. It also includes a golden register file that holds the register values that have been verified.

Chico does not account for large error coverage, because it only detects control logic bugs. Experimental results show that the area impact of Chico is estimated to be less than 3%. DIVA and Chico focus only on checking pipeline errors and do not detect I/O errors.

4.1.2.2 Argus (Multiple Checker Units). Argus (Meixner et al. 2007) makes use of multiple checker blocks for verifying the functionality of the core. It checks the invariants rather than individual functional units in a processor. It performs runtime checking of control flow, computation, dataflow, and memory. It deploys four checkers separately for each of these activities. It follows a method similar to fingerprinting, where it generates a signature for each checker and verifies the same. One limitation is that it cannot detect some multiple error scenarios where an error in the checker core prevents detecting an error in the main core.

In summary, we can extend checker architectures like DIVA, Chico, and Argus to verify the core units of multicore designs.

4.2 Uncore Checkers

As mentioned earlier, there is a shift in validation focus toward uncore units when moving from single core to multicore (Lin et al. 2014). In this section, we provide an overview of the issues and solutions related to shared cache and interconnect networks.

Shared Cache or Memory: Issues in shared cache occur in large numbers due to the following reasons:

(1) Increase in the core count results in a corresponding increase in the number of shared and private caches giving rise to a number of issues in memory hierarchies (Nanehkaran and Ahmadi 2013).
(2) There are varying degrees of sharing of caches at different levels in multicore architectures. This leads to memory consistency and cache coherency issues.

As shown in Figure 6, hardware checkers for validating memory units focus on verifying two properties of memory operations, namely memory consistency and cache coherency. Based on these two properties, we classify memory checkers as Memory consistency checkers and Cache coherency checkers. Memory consistency checkers verify the constraints that are imposed on the order of memory operations for different memory locations during program execution. Cache coherency checkers verify whether the write operations to a particular location happen in the correct order so the last updated value is read back. These approaches are discussed in detail below.

4.2.1 Memory Consistency Checker. We explore two implementations of memory consistency checkers, namely DAtacoloring for COnsistency Testing and Analysis (Dacota) and Dynamic Verification of Memory Consistency (DVMC).

In a multicore system that deploys Dacota (DeOrio et al. 2009), an area in each core's L1 cache is reserved to record the memory accesses during program execution. Each cache line is associated with an Access Vector (AV). Whenever a store operation is performed in the L1 cache, a counter is used to increment the AV corresponding to the cache line. All the AVs of a cache are collectively arranged in order in an activity log that is available in each core's private cache. Once the activity log is full, program execution is temporarily suspended. The aggregated logs from all the cores are transferred to main memory. Finally, the logs are analyzed for memory consistency violations by running a software algorithm on any one of the cores. The algorithm constructs a directed graph. The presence of a cycle in the graph indicates a memory consistency violation. The graph is further analyzed to isolate the error specific to a core.

A few interesting aspects of Dacota are as follows:

(1) The data values are not stored in access vectors of each cache. Instead, a counter is used to store the incremented value whenever a store is performed. This reduces the storage space of the access vectors.
(2) The checker is disabled before shipment of the product, thereby leading to zero performance overhead.

Mammo et al. (2015) further extended Dacota by giving flexibility for the designer to decide where to run the software analysis algorithm when multiple threads are running on the CMPs. It has been identified that there are three possible locations to run the algorithm: the core under verification or a separate core on the same chip or offline on a separate machine. In this case, two types of memory access orderings are tracked for the different shared memory interactions:

(1) Consistency ordering ensures memory consistency that is enforced by the consistency model on memory accesses (within the core).
(2) Dependency ordering ensures ordering of data dependencies on memory accesses (between cores).

Meixner et al. (2009) designed a runtime verification framework named DVMC for verifying shared memory interactions. It verifies three invariants on memory operations, namely Uniprocessor Ordering, Allowable Reordering, and Cache Coherence. To verify the three invariants, each core is supplemented with the following set of checkers where the memory operations are replayed before they are committed:

(1) A verification cache to check uniprocessor ordering. Whenever write operations are made by a single core to the shared memory, it captures the operations and replays them in the checker. The replayed values are verified against the executed values.
(2) Ordering table checker to ensure that whenever a memory operation gets completed, it does not violate the reordering rules of the consistency protocol.
(3) Coherence checker to verify that the coherence property is satisfied.

4.2.2 Cache Coherency Checker. Coherence String Matching (CoSMa) is a solution for verifying cache coherence in multicore designs (DeOrio et al. 2008). An area in the L1 cache of each core is partitioned to store its local activity termed as Local History Logs (LHL). The L1 cache controller of each core has a Slave Checker (SC) hardware unit. During program execution, each cache line's coherence state is monitored by LHL. In a similr manner, an area in the shared L2 cache is also partitioned to store the global activity of all the cores and termed as Global History Logs (GHL). The

Table 1. Analysis of Dacota, Mammo's Work, DVMC, and CoSMa

Feature	*Dacota*	*Mammo's work*	*DVMC*	*CoSMa*
Hardware Provisions	A control block, a state machine, and an index table	A store counter and a log delay buffer	Deploys three checker units.	A master checker for L2 cache and a slave checker for each core's L1 cache.
Advantages	Low area overhead, high coverage, and fast debugging ability.	Scalable with increase in the number of cores.	Dynamically verifies a wide range of consistency models.	Offers a high coverage and less area overhead.
Limitations	Does not handle error recovery. Not scalable with increase in core count.	Cache coherence is not fully addressed. Test generation is not addressed.	Hardware costs and interconnect bandwidth are high.	Memory consistency bugs and deadlocks are not undetected.
Experimental Results	Implemented on OpenSPARC T1: area impact less than 0.01% and a 26% performance slowdown for applications.	Memory hierarchy is simulated for a 16-core multicore. It showed 83% bug detection rate over the three memory consistency models.	Implemented on SPARCv9 architecture: Performance slowdown up to 11%.	Implemented on Intel Core 2 Duo had a performance degradation of 1% to 23%. The area overhead was very minimal, only 0.002%.

L2 cache controller has a Master Checker (MC) unit associated with it. The MC periodically suspends program execution and interacts with the SC in each core to perform coherence validation. Here, cache coherence is verified through a string matching algorithm. The program execution takes place in the foreground while CoSMa operates in the background in a special mode called CoSMa mode. In CoSMa mode, the log sequences are compressed, partitioned at instances of the Invalid state of MESI protocol, and checked using the string matching algorithm. If a portion of an L1 string is present as a subset of an L2 string, then it is coherent else it results in a coherence error.

Table 1 shows an analysis of all the memory checkers that have been discussed above.

On the whole, we observe that a set of distributed checkers are deployed in various levels of cache to detect memory ordering violations. The bug detection capability of the checker depends on the workload running on the system; that is, to discover memory consistency and coherency bugs, the workloads have to use shared data. Hence, they do not completely guarantee the correctness of all possible non-deterministic behaviors in a multicore scenario.

Interconnect Networks: Other than memory, design errors occur in large numbers due to complex interconnect networks. Interconnects are of two types: off-die and on-chip. In Off-Die Interconnect Networks (ODIN), scan-based (Karpenske 1991; Hervé et al. 2009) and trace-buffer-based techniques (Xu and Liu 2010) are used to capture the internal states in the design. Functional bugs can manifest themselves thousands of clock cycles after they occurred. Hence, trace buffers store the data acquired from the crash point to the source of the error. The limiting factor is that the amount of data obtained is determined by the capacity of the buffer. One of the earliest techniques that leads to reduced test time is the boundary scan technique (Bleeker et al. 2011; Hassan et al. 1988). It is mainly used to test ODIN interconnects. It uses a boundary scan cell that reads the signal between two pins. In addition to this, data compression techniques (Anis and Nicolici 2007), state restoration techniques (Ko and Nicolici 2009), and multiple trace buffer based techniques (Basu and Mishra 2013) are also used.

An On-Chip Interconnect Network (OCIN) serves as a communication hub for cores inside a chip. Apart from traditional bus or crossbar interconnect fabrics, systems with large core counts use ring or mesh interconnects. These interconnects route the messages using a packet-based approach. The interconnects divide a message into multiple packets. The packets are partitioned

into small blocks called flits as a header flit, body flits, and a tail flit. After packets are injected into the network, they travel through a set of routers to reach the destination.

Today's multicore processors predominantly use mesh interconnects. For example, in Tile64, the Tile processor that contains 64 cores, its OCIN (termed iMESH) connects multicore tiles with five 2D mesh networks (Wentzlaff et al. 2007). The main aim is to provide high bandwidth through the use of mesh networks. However, latency and power are two major issues in designing such interconnect architectures.

Another example is Intel TeraFlop NoC, which has 80 tiles for its 80 cores, where the OCIN is the 2D mesh network (Dighe et al. 2011). The key communication fabric used for inter-tile communication is the mesochronous interface. A mesochronous interface allows clock phase-insensitive communication across tiles and synchronous operation within each tile. As a result, there is a synchronization latency penalty for distributing a single frequency clock across tiles. In such cases, verification includes handling of clock domains and task synchronization across tiles (Dighe et al. 2009).

4.2.3 Cross-Core Communication Checkers. Functional errors in interconnect fabrics include cross-core communication errors. Examples of such errors are as follows: packet data corruption, dropped and duplicated packets, network deadlock, starvation, packet misrouting, livelock, and dropped and duplicated flits (Peh et al. 2009). To overcome these errors, functional verification should ensure address mapping, non-dropping of packets, non-corruption of packet data, time-bounded packet delivery, non-generation of new packets within the network, non-breaking of the protocol under stressed conditions, ensuring secure transactions, and generation of errors for unsecure transactions.

Interconnects can be verified using checkers or software mechanisms. Checkers are deployed in the routers to verify the correct functionality of the interconnects. Unlike the memory unit or functional units, the checker does not use a separate core. Instead, a checker is a small built-in unit inside the router. This is discussed below.

4.2.3.1 Instrumenting Routers with Checkers. The observability of network traffic is enhanced by replacing a packet's original data content with debug information (Abdel-Khalek and Bertacco 2014). The debug information corresponds to the packet's current state during its traversal through every router. This is stored within each packet's body flits. Once the packet reaches its destination, it is stored at the local cache of each node. Each router is instrumented with a small checker to detect any error. Once an error is detected, the execution is stopped and the packet's debug data is analysed using a software algorithm running on the core or off-chip. The analysis process takes place locally as well as globally. Each core analyses the debug data of packets destined for itself. Then, the debug data from all the nodes are aggregated for a global view of the network's behaviour. Additional hardware to store the debug information includes a register and packet counter. The register is added for every input buffer within the router. The packet counter is added to every router and gets incremented whenever a packet is received. Table 2 shows the communication errors and the checker units employed to detect these errors (Abdel-Khalek and Bertacco 2014).

4.2.3.2 Traffic Capture Using Snapshots. Another technique to detect the errors takes snapshots of packets in-flight that in turn capture the traffic (Abdel-Khalek and Bertacco 2012). Error detection is achieved in two phases: the logging phase and the checking phase. During the logging phase, each router periodically takes snapshots of the traffic created by the packets in-flight. It then stores the snapshots in a temporary space inside each core's local cache. As soon as the logs are filled up in a core's local cache, it triggers the local check. During the checking phase, the stored snapshots are analysed periodically by a software algorithm to detect functional errors.

Table 2. Communication Errors and Checker Units

S.No.	Communication Error	Checker units
1	Packet data corruption—During transfer, the data contents of a packet are corrupted.	Error Detecting Code (EDC) is added to each packet. This EDC value in the source is verified with its value in the destination.
2	Dropped and Duplicated Flits—There is a loss of data flit or a new flit is added in the packet.	Verified using EDC. A counter and comparator are added to verify the number of flits in a packet.
3	Dropped and Duplicated Packets—There is a loss of data packet or a new packet is added in the network.	A packet counter per router verifies the number of incoming and outgoing packets.
4	Network Deadlock—When a packet gets blocked waiting for a resource and as a result becomes inactive, it indicates a deadlock.	Counters are added to each router and long periods of inactivity that exceed threshold are monitored.
5	Starvation Errors—When a packet requests for resources that are granted to other packets and therefore the packet keeps waiting infinitely.	A time-out counter is added to the routers input port to count the number of cycles the input port was not granted an output channel.
6	Livelock—Packet continues to move in the network but not toward the destination.	A Time-To-Live (TTL) counter is added to the header flit of a packet and the counter gets incremented on every hop. If the count exceeds the threshold, then it indicates the occurrence of a livelock.
7	Packet Misrouting—Packet is sent to the wrong destination or takes incorrect path to reach the right destination.	A comparator is added to each router in the network.

The packets that are not making forward progress are checked for errors like deadlock, starvation, livelock, and misroute. On detecting an error, the logs are aggregated, and the path taken by the packets is partially reconstructed to provide additional debug information. One limitation of this technique is that it fails to detect dropped packets. Also, the modifications done in the router have an area overhead of 9% as reported in Abdel-Khalek and Bertacco (2012).

In this section, we have presented a broad overview of different hardware checkers that are suitable for validating the core and uncore units of processor designs. In the next section, we explain software based approaches for validating a multicore system.

5 SOFTWARE-BASED SELF-TEST (SBST)

Validation can also be carried out in software without changes or with minimal changes in hardware (Constantinides et al. 2007) using software-based self-test. In SBST, self-test programs are generated and run at the normal speed of the processor (Chen et al. 2007) by exploiting its Instruction Set Architecture (ISA) (Foutris et al. 2010). Hence, SBST is also called Instruction-Based Self-Testing. The test programs and test data are loaded in the processor's memory. They are executed on the bare metal. This leads to reduction in test execution time. Unlike hardware checkers, they do not require a reference model (Psarakis et al. 2010; Chen et al. 2003). SBST methods are suitable for both structural (Chen et al. 2007) and functional testing (Kranitis et al. 2005). We classify SBST approaches into two major categories, namely Core SBST and Uncore SBST (Figure 7).

5.1 Core SBST

Core SBST is further classified based on whether it tests the full core, selective functional units, or pipeline logic in the core.

5.1.1 Full Core Testing. SBST for a complete core can be carried out based on the utilization level of each core in a multicore.

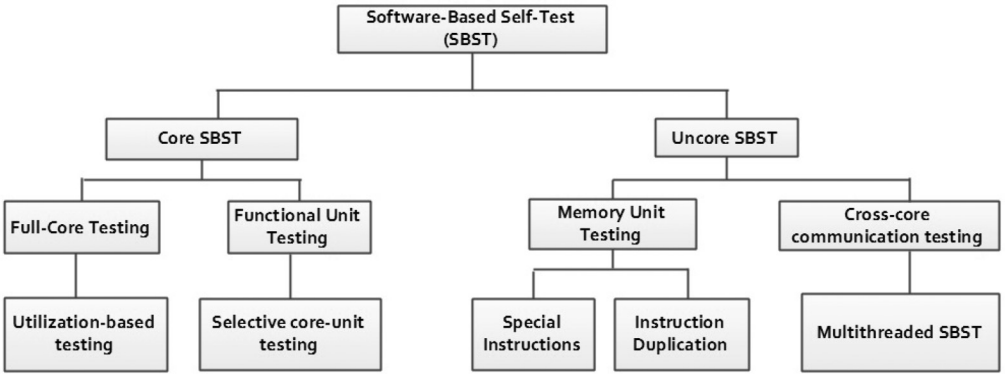

Fig. 7. Taxonomy of SBST Techniques.

Utilization-based testing: In a multicore System on Chip (SoC), all the cores are not fully utilized at all points of time. Only selective cores are utilized at certain points of time and selective units within a core are over-utilized (Skitsas et al. 2012). Full core testing is carried out based on the following:

(1) Total-core utilization: The number of instructions that are executed on each core is tracked and this is checked against the testing threshold. The cores that have a higher count above the testing threshold are identified for full core testing.
(2) Core-unit utilization: The number of instructions in each unit of the core is tracked and checked against a testing threshold. The units that have a higher count above the threshold are identified for testing.

This method avoids testing of under-utilized units in a core to save the test execution time. Apart from this, we discuss two approaches used in single-core verification that can be adopted for multicore as well.

Special Instructions: Constantinides et al. (2007) introduced a periodic checking technique where a firmware stalls the processor periodically and runs the checking mechanisms on the hardware. To support this, special instructions termed as Access-Control Extensions were introduced to provide accessibility and controllability to the processor states. However, its reliability depends on loading test patterns corresponding to these instructions, executing them, and validating their results. It has an area overhead of 5.8% and a performance cost of 5.5%.

Operation Reversal: Wagner et al. (2008) proposed a new methodology for test generation named Reversi where each operation in the ISA has an inverse operation. A sequence of operations and their inverse operations are fed to the processor in the form of a test program. Then, the processor is checked for correctness by comparing the initial register state with that of final register state in the test program. For the processor to be error-free, the initial state should be equal to its final state. The operation and its inverse operation together constitute a reversible test program.

5.1.2 Functional Unit Testing. In multicore SoC, certain functional units within a core are selectively identified and tested using Selective core-unit testing.

Selective core-unit testing: Skitsas et al. (2013) proposed an Operating System–assisted SBST termed "Selective core-unit testing" to test the functional units for multicores. To leverage this, a framework named DaemonGuard is loaded onto the operating system to observe the individual functional units and performs on-demand selective SBST of those units that are most stressed.

Table 3. Comparison of QED Transformations

Feature	PLC	CFCSS-V and CFTSS-V	EDDI-V
Working principle	Special operations inserted across memory spaces using targeted instructions	Block of instructions	Uses Check instruction
How is the error detected?	Self-consistency checks are done for a selected set of load operations.	CFCSS-V: For a selected pair of instructions, a window size is set and changing the window size helps to generate blocks of instructions and control flow is checked between these instruction blocks. CFTSS-V: The test code contains special signatures to track instruction execution.	Load and store values of original instruction are compared with that of duplicated instruction. If there is a mismatch, then it indicates an error.

The DaemonGuard is an active OS process and each daemon has test programs corresponding to a functional unit. Whenever a functional unit requires testing, the corresponding daemon is invoked by the DaemonGuard to run the test routine. This method performs frequent testing of highly utilized units and an infrequent periodic testing of under-utilized units. Their results show that there is a 30× reduction in testing time when implemented on an OpenSPARC processor.

5.2 Uncore SBST

In this subsection, we discuss SBST techniques for testing cache hierarchies and interconnect networks.

5.2.1 Memory Unit Testing. SBST for the memory unit is carried out through the usage of special instructions or duplicate (equivalent) instructions to test the cache arrays for read and write operations.

5.2.1.1 Special Instructions. A SBST for on-line testing of L1 cache arrays in multithreaded multicore architectures makes use of special instructions (Theodorou et al. 2011). The special load/store instructions denoted as ldxa/stxa instructions are used to directly access instruction and data cache arrays for March read/write (Van De Goor 1993) operations. In addition, performance counters are used to monitor the cache misses for validation of March read and verify operations.

Implementation is carried out for single-core multithreading (1 core/4 threads) and multicore multithreading (4 cores/16 threads). In multicore testing, the test program is not split among different cores as their L1 caches are not shared. But multiple test programs are run in parallel on different cores and each test program within a core is shared by 4 threads running on the core. Implementation results on L1 caches of OpenSPARC T1 processor (4 cores/16 threads) show a speedup of more than 1.7 in test execution time. This technique has also been extended to detect faults in TLBs (Theodorou et al. 2013).

5.2.1.2 Instruction Duplication. Another technique, named Quick Error Detetion (QED) (Lin et al. 2014), based on the concept of instruction duplication is used in multicore verification. QED is used to detect both electrical and logic bugs in the uncore units of the design specifically, in the memory hierarchy. Three different QED transformations are discussed in Table 3. These include Proactive Load and Check (PLC), Control Flow Checking using Software Signatures for Validation (CFCSS-V), Control Flow Tracking using Software Signatures for Validation (CFTSS-V), and Error Detection using Duplicated Instructions for Validation (EDDI-V) (Lin et al. 2014). The experiments

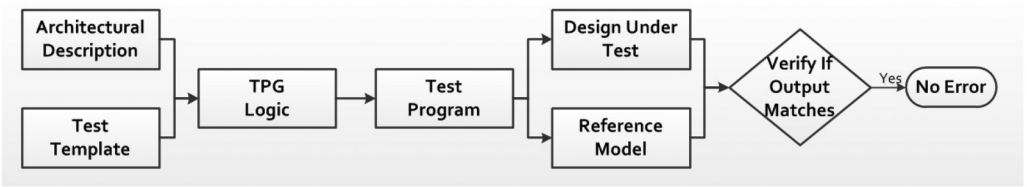

Fig. 8. An overview of test program generation.

were carried out on multicores like Intel Core i7 and OpenSPARC T2. The results demonstrated that QED shortened the error detection latency from nine orders of magnitude to a few hundreds of clock cycles.

5.2.2 Cross-Core Communication Testing. SBST for cross-core communication is achieved through thread-level parallelism and core-level parallelism. This is discussed below.

5.2.2.1 Multithreaded SBST. Multithreaded SBST focuses on optimizing the SBST program's execution time and improving fault coverage (Foutris et al. 2010). A single copy of the test program is stored on the shared cache instead of storing it privately in each core. The self-test programs are executed in parallel on multiple threads to analyze the thread switch logic (intra-core) and thread specific control logic (inter-core) among them. Thread switch logic is verified by splitting a test routine into different subroutines that are in turn executed by all the threads of a core. Thread specific control logic is verified by (a) executing the test routines in parallel by all the cores and (b) executing the test routine by maintaining a shared functional unit.

Thread-level optimum exploitation is achieved when the idle period of one thread is ovelapped by an active thread. The test routines in n cores are scheduled in different ways to achieve optimum utilization of shared cache and interconnects. Experimental results on OpenSPARC T1 (8 cores and 32 threads) show that the multithread scheduling algorithm speeds up test execution time both at the core level (up to 3.6×) and the processor level (up to 6.0×).

6 TEST PROGRAM GENERATION

We look at Test Program Generation (TPG) methods that involve generating test cases in assembly language or at the instruction set level. TPG (Sadasivam et al. 2012) deals with two types of tests, namely directed tests and random tests. Directed tests target specific functional units in the design and check for consistent results under normal as well as abnormal operating conditions. Random tests are parallel test programs that verify multiple internal and external signals in the control and data paths of the processor. One limitation is that the results of correct execution of the processor are not known *a priori*, and hence, the test programs have to be rerun using a golden model. The outputs of both the first silicon and the golden model are compared and if there is a mismatch, then it indicates an error (Figure 8).

Test program generators are used at pre-silicon and post-silicon stage. Some examples of pre-silicon test program generators for single-core designs are Genesys (Fournier et al. 1991) and Genesys-Pro (Adir et al. 2004) for IBM's PowerPC processors. Post-silicon test generator frameworks include Pseudo-random TPG (Sadasivam et al. 2012), Markov-Model-driven TPG (Wagner et al. 2007), Specification-driven TPG (Mishra and Dutt 2004), and Focused Test Suite (FTS) (Grandhi 2006).

Test program generators for multicore include multithreaded programs that in turn execute multiple instruction streams concurrently on all the cores. Initially, each thread creates its own

part of the test program. Then, all the threads coordinate and jointly create the test program. Finally, the threads synchronize and execute the test program as a whole (Adir et al. 2012).

Multicore TPG includes verifying collisions, thread coordination, and synchronization. Collision occurs when there is an access to a shared resource by different processes (or threads) from different cores or within the same core. Accesses that involve write operations to the shared resource are a general test case for collisions (Adir et al. 2012).

For the verification of Simultaneous Multithreading (SMT) in IBM's POWER5 and POWER6 processors (a dual-core processor) (Victor et al. 2005), three techniques were introduced for test program generation, namely Thread Irritation, Thread Merge, and Thread Replication.

Thread Irritation: There is a primary thread and a number of irritator threads. The irritator threads run in an infinite loop until the primary thread kills them. The Irritator thread interacts with the primary thread to perform a high degree of cross-thread interaction at the microarchitecture level. As a result, it detects bugs due to hangs, livelock, deadlock, thread starvation, and cache state transition.

Thread Merge: The Test program generator creates buckets of single-threaded test cases for each core and finally merges to get multithreaded test cases.

Thread Replication: Test program generator generates one single-threaded test case, and this gets replicated on multiple threads to simulate a multithreaded scenario. It is ensured that each test case is coherent and no duplicates are present.

Among the three techniques, Thread Irritation has been a great success in POWER6 and POWER7 processors. In POWER7 processors, it has been adopted for post-silicon TPG, where it exposed nearly 23 bugs in the design (Ludden et al. 2010).

During post-silicon validation, each test program needs to be loaded and executed separately on the processor. This takes considerable amount of time. To avoid this latency due to loading time, a technique called Exercisers on Accelerators (EoA) is used.

An Exerciser is a tool that is loaded on the processor by a builder application. Its role is to generate the test cases and run and verify the tests automatically on a chip. IBM's Threadmill (Adir et al. 2011), a post-silicon exerciser, has been used to verify multithreading in the POWER7 processor. Threadmill is loaded onto the silicon once. This is followed by test generation and execution for a number of times. Numerous test programs are generated to test inter-thread and inter-core computation and communication scenarios. Threadmill uses multipass consistency checking rather than a reference model for verifying the results.

In EoA, the exerciser shifts are run on an in-house acceleration platform. It plays a major role in verifying the POWER8 processor, as it is used both in pre-silicon and post-silicon stages. In the POWER8 processor (12 cores per chip), bare metal exercisers and hardware irritators are used for generating test cases. Bare metal exercisers employ multipass consistency checking where each test case is executed multiple times, and the results of each pass are compared against a reference pass. Hardware irritators use test-specific hardware that triggers microarchitectural events at random. This exposes the corner cases without creating explicit stimuli.

Another important aspect of POWER8 verification is that debugging the post-silicon failure is made easier through *cycle reproducible environment*. To debug the root cause of a failing test case, an exerciser image is executed among multiple cores and cycle-by-cycle latched data are collected from the cores. It results in reproducing a multicycle trace of all the cores. The collected data are analyzed to determine the exact cycle of the failing condition (Nahir et al. 2014).

In addition to this, today's multicore server systems need verification for Reliability, Availability, and Serviceability (RAS) features (Mitchell et al. 2005). Some examples of RAS features include parity or ECC protection of memory, generation of checksums (using cyclic redundancy check) for data transmission and storage, and avoiding single-point failure by duplicating components.

The RAS features can be verified using Focused Test Suite that performs algorithmic testing of components in a processor.

7 MACHINE LEARNING AND PREDICTIVE ANALYTICS

One of the recent approaches to post-silicon validation is the application of predictive analytics with a machine-learning paradigm to predict bugs in the post-silicon stage. This has become possible due to the availability of huge amount of data and big data processing algorithms and infrastructure. In this section, we discuss Anomaly Detection Algorithms, Bug Prediction Models, and Big Data Predictive Analytics.

7.1 Anomaly Detection Algorithms

Machine learning supports automatic diagnosis of inconsistent bugs through anomaly detection. Anomalies are patterns that deviate from normal behavior. To identify these anomalies (buggy patterns), training data (input data) are needed to learn the correct behavior of the system. The training data are labeled as passing (positive label) and failing (negative label) samples. After training is completed, the algorithm classifies the new data as passing or failing. The goal is to classify the data based on the time of occurrence of the bug and the critical signals. During processor execution, the number of failing samples is lower. Training occurs based on a large number of available passing samples. The learning is grouped under one-class learning, as it requires only positive labels (DeOrio et al. 2013).

In another work, machine-learning algorithms are adopted for server validation by building a learning set in a processor (Paidipeddi and Tomar 2014). Here, the learning set is constructed to test power management features in the Xeon processor.

7.2 Bug Prediction Models

Bug Prediction Models are mathematical models that are built to predict the occurrence of design errors in a new design. The models are used to predict the Mean Time To Failure (MTTF) of the chip. The important part of this process lies in selecting a proper model based on the failing cases. These models are constructed during pre-silicon and deployed in post-silicon to estimate the bug rate. There are two types of bug predictions: short-term bug prediction and long-term bug prediction.

Bugs in the reference model act as training data (input data) for the TPG framework of the new design. Machine-learning algorithms are applied on this training data to predict the probability of occurrence of bugs in the new design. Thus, the bug prediction serves as input when the verification plan is created for the new design.

Malka et al. (1998) performed a statistical analysis of bug discovery data on PowerPC processors to determine the processor release dates in the market and estimate the bugs in the post-release.

Guo et al. (2011) proposed the use of Artificial Neural Network (ANN) for bug prediction. Classification and regression models are constructed based on the relationship between attributes and the occurrence of bugs. The classification model identifies whether a module is prone with bugs and the regression model predicts the number of bugs in that module.

In another work, Guo et al. (2014) constructed a framework for pre-silicon bug forecast using Genetic Algorithms (GA). The module characteristics related to bug occurrence are collected and refined to select the important ones. Then, learning techniques are used to build the bug models from the training set. The bug predictive model that is built serves two purposes:

(1) The model built for reference revision can also predict the bug information of current revision.

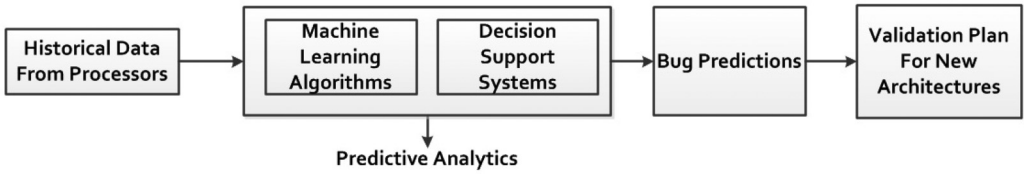

Fig. 9. Design validation using big data predictive analytics.

(2) When a verification plan is created, it aids in the allocation of verification resources as more resources should be dedicated for bug prone modules.

7.3 Big Data Predictive Analytics

There is a lot of potential and ongoing research work in big data predictive analytics for design validation. The histroical data (test data from existing architectures) is fed as input to machine-learning algorithms and decision support systems to make accurate predictions for detecting bugs on new architectures. From the existing large volumes of test programs and test scenarios, we can generate new test patterns for the futuristic multicore designs (Figure 9).

Post-silicon validation of Intel processors have made use of predictive analytics (Fania et al. 2013) in the following ways:

(1) to optimize the way bugs are handled.
(2) to eliminate 36% of test content during validation.
(3) to reduce post-silicon validation time by 25%.
(4) to proactively identify client issues and implement bug fixes (Chandramouly et al. 2013).

On the whole, we find that machine learning combined with predictive analytics would help the industry to detect the bugs in a more rapid, accurate, and automated manner.

8 FORMAL METHODS BASED DEBUGGING

All the validation approaches discussed so far require either hardware or software support. There is yet another approach, namely formal methods, that involves verifying a design mathematically. It uses the concept of "property verification" (McMillan 1994) where a set of properties that specify the behavior of the system are to be proved.

Properties are used to create assertions. An assertion is an instruction to prove that a given property holds good for the design. Properties are defined using temporal logics like Linear Temporal Logic (LTL), Computational Tree Logic (CTL) (Schlich et al. 2008), Property Specification Language (PSL), or System Verilog Assertions (SVA).

These approaches are exhaustive, as they cover all possible behaviors in the design. Therefore, they are used to narrow down an error to its exact location. But they become very complex as the design size increases. Hence, formal methods are suitable for verifying small designs or a single unit in a complex design.

There are different ways of verifying a design formally, namely *equivalence checking*, *model checking*, and *theorem proving* (Gajski et al. 2009). These three approaches verify a design mathematically by checking if a set of properties holds good for the design under test. In *equivalence checking* (Kuehlmann et al. 2002), two representations (RTL design and gate-level design) of a logic circuit are verified to check that they are functionally equivalent. This is done by reducing the designs to a canonical form by applying mathematical transformations (Gajski et al. 2009). The specification is equivalent to the implementation when the two canonical forms are identical. In

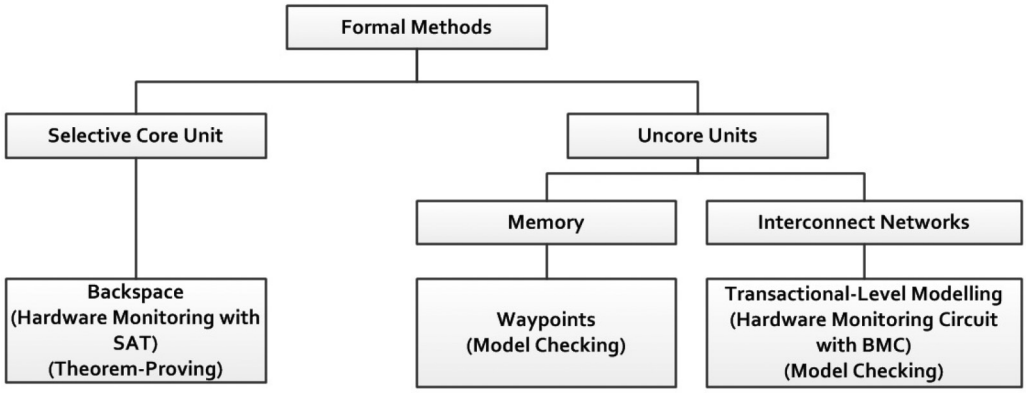

Fig. 10. Formal verification techniques for Post-silicon validation.

model checking (Clarke et al. 1986), the specification is represented as a set of properties, and the implementation is represented as a finite-state machine. To prove that the specification is functionally equivalent to the implementation, each property is verified by traversing all states in the state machine. In *Theorem proving* (Ray 2010), a formula is verified by showing that there exists a derivation of the formula (theorem) in the logic of the theorem prover. It proves that the theorem is valid for all models.

Formal verification has gained lot of importance after the detection of the Pentium FDIV bug that has been referred earlier. Formal methods are used in pre-silicon as well as post-silicon (Kern and Greenstreet 1999) validation. In pre-silicon, they detect an error in the RTL design. In post-silicon, they are used to find an error trace that helps in reproducing the error to make debugging faster. The scope of this article is limited to the use of formal methods for debugging in post-silicon.

In post-silicon, formal methods are implemented with or without hardware monitoring support. They are used to verify selective units in the core or uncore components (Figure 10). As soon as an error is detected using any of the hardware or software techniques, formal methods are used either to derive an error trace or perform backtracking to find its origin. Formal methods such as theorem-proving and model checking are widely in use for post-silicon. A few implementations are discussed below.

8.1 Selective Core Unit

Formal methods are used for verifying selective units in a core such as floating point unit, ALU, or pipeline logic. Backspace is a theorem-proving technique by Gort et al. (2012). It is a post-silicon debug method to backtrack (termed as backspace) from a crash state to the current state using a monitoring circuit. The monitoring circuit is a breakpoint circuit and the crash state where the error occurred is set as the breakpoint. This circuit incorporates a SAT (satisfiability) solver, a theorem proving technique to detect error traces and is programmed with a breakpoint state. Given the crash state and the error symptoms, the SAT solver creates an instance of the problem to check when the current state is equivalent to the breakpoint state. Then, the chip is stopped to find the predecessor states that led to the crash state. This is known as pre-image computation. A new breakpoint is set at that predecessor state, and the failing test is re-run on the chip to get a long set of error traces. When an error trace is detected that leads to its origin, backtracking is stopped. Otherwise, a new breakpoint is set by going back iteratively to the predecessor states until the history information is enough to debug the crash state. Using this method, backtracking

to hundreds of cycles is possible. A key issue with this approach is its high area overhead. When implemented using an OPENRISC 1200 processor, the full breakpoint circuit resulted in 61% area overhead and this was reduced by constructing a partial breakpoint circuit with 2% area overhead.

This issue is overcome in Trace Array Buffer (TAB) Backspace (De Paula et al. 2011), where there is no additional on-chip overhead except for the debug logic. There is no pre-image computation as in Backspace. Trace buffer recording and breakpoint capability are two features used in this design. It overcomes some limitations of Backspace like finding the right time window to capture the chip's partial state information by constructing abstract traces of the crash state.

The drawback in both approaches is that it is difficult to trigger the bug via the same execution scenario. This is due to the non-deterministic nature of the multicore. As a result, an exact trace or abstract trace cannot be constructed to debug the failure. To overcome this drawback, nuTAB-Backspace (De Paula et al. 2012) is proposed, where an equivalent execution trace is constructed that is not cycle-by-cycle identical but still triggers the bug. This is achieved by getting the rewrite rules from the user that specify the equivalent traces.

8.2 Uncore Units

8.2.1 Memory. Model checking technique is widely used for debugging memory-related errors. Counterexamples are generated in model checkers to prove the violation of a property within a model. But the time taken to discover counterexamples is very long. Ho et al. (2009) proposed a solution, termed *waypoints*, to reduce the time taken to discover long traces of counterexamples. Waypoints are a set of events (preconditions) that occur prior to the observed failure. The trace to the first waypoint acts as the initialization sequence for the next waypoint. Successive waypoints are determined in this manner. By traversing through a sequence of waypoints, traces (counterexamples) that lead to the crash state or error (property) are determined. However, there is no guarantee that the trace to an error signature will pass through all or any of the waypoints. This happens if the assertions are not predicted correctly. In such a scenario, waypoints are re-examined to alter their sequence.

8.2.2 Interconnect Networks. Transaction Level Modeling (TLM) (Gharehbaghi and Fujita 2012) is a formal approach, used for interconnect networks, based on Bounded Model Checking to detect the path that leads to the origin of the bug. The hardware part consists of a transaction extractor (monitoring circuit) and a trace buffer. It mainly focuses on detecting bugs that occur during the transaction between the cores. Each transaction is associated with information like initiator (sender), target (receiver), and the type of transaction (read or write) that occurs between the cores. This information is extracted using a transaction extractor. Based on the information from the transaction extractor, a Bounded Model Checker (BMC) constructs a finite-state machine termed a Transaction-Level State Machine (TLSM) that represents transactions of a core with another core. For each state in the TLSM, path analysis is performed based on the constraints, transactions sent and received for a state, and the assertions. Path analysis results in detecting error traces. As soon as an error trace is obtained from the extractor, it is stored in the trace buffer. In a similar manner, path analysis is done for all generated paths of the TLSM to collect the erroneous transaction patterns and prune the infeasible paths.

To summarize, formal methods are effective in debugging to find good error traces within a short period of time. Also, they are successful in hardware verification as they are cost effective.

9 CROSS-CUTTING ASPECTS -VALIDATION METRICS

Validation metrics are common for all the techniques discussed so far. There is a list of validation metrics used in the industry. A few such metrics are Coverage Models (Mammo et al. 2012), Test

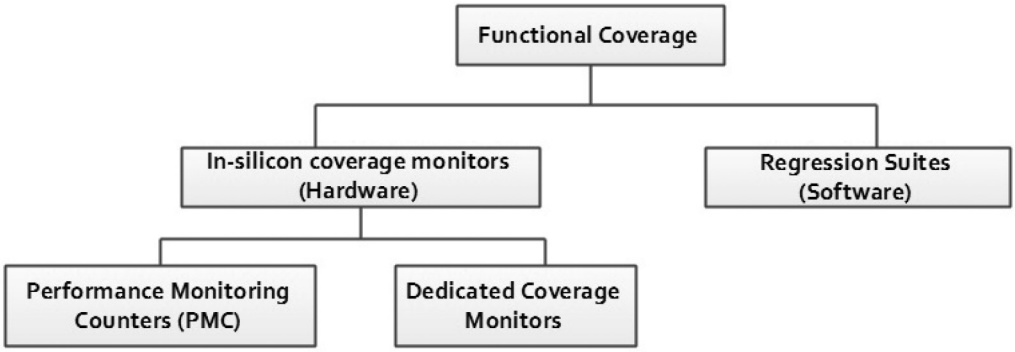

Fig. 11. An overview of functional coverage techniques.

Execution Time (Theodorou et al. 2011; Gizopoulos et al. 2008), Bug Detection Rate (Mammo et al. 2015), and Bug Detection Latency (Lin et al. 2014). Coverage is an important validation metric. Mainly, it is evaluated through Code Coverage (Balston et al. 2013; Piziali 2007) and Functional Coverage (Mishra and Dutt 2005a). Code coverage is used in post-silicon debug to measure statement and branch coverage. Functional Coverage (Bojan et al. 2007) measures verification progress on the functional requirements of the processor (Piziali 2007).We specifically look at functional coverage in the following section.

9.1 Functional Coverage

Pre-silicon functional coverage (Mammo et al. 2012) measurements cannot be directly adapted to post-silicon due to its limited observability. Post-silicon functional coverage can be measured either in hardware or software. Hence, we classify functional coverage as In-silicon coverage monitors (hardware) and Regression Suites (software) (Figure 11).

9.1.1 In-Silicon Coverage Monitors. These monitors (Adir et al. 2010) are deployed in the hardware. They are further sub-divided into performance monitoring counters and dedicated coverage monitors.

9.1.1.1 Performance Monitoring Counters (PMC). PMCs are hardware counters that measure events during program execution (Bandyopadhyay 2004). The counters do not interrupt program execution. Performance Monitoring (PMON) registers are available to monitor and count certain events. Events that can be monitored are the number of instructions, loads, stores, cache misses, TLB misses, and number of clock cycles. PMCs monitor these events that take place and then count the number of hits that occurred for that event. PMCs collect information about the frequency of occurrence of these events in an application. Using PMCs, the PMON registers capture different variations in the application. Each PMC group has a PMON ID. All variations of a PMON group corresponding to the PMON ID are programmed in a register termed the Model Specific Register (MSR) (Bojan et al. 2007).

Generally, PMCs are used to measure cache and TLB characteristics like L1, L2, and L3 cache misses, cache size, cache line size, TLB data misses, TLB size, and miss latencies (Dongarra et al. 2001).

Machine Check Architecture (MCA) (Montgomery and Tian 2010) is a mechanism where a set of Model Specific Registers (MSRs) detect and report machine errors that occur on the system bus, cache, and TLB. MSRs contain control and status information and register banks that report the

error. An ECC error or parity error is injected into the memory, and the correctness of data is checked while retrieving it back. If there is a change in value, then it indicates the occurrence of a memory error. Thus, correctness is ensured by validating the value of bits in specific MSRs.

Performance APIs (PAPIs) (Browne et al. 2000) specify a standard API for accessing the PMC on many processors. PAPI includes events for monitoring the cache and TLB misses, count of completed integer, the floating point, and load-store instructions.

9.1.1.2 Dedicated Coverage Monitors. Bojan et al. (2007) has discussed two different coverage monitors for Intel's Core 2 Duo multicore processor, namely Front-Side Bus (FSB) coverage monitors and Extended Execution Trace (EET) monitors. FSB monitors are used in post-silicon to log the requests and attributes. EETs use special microcode patches to monitor a microcode address and signal it to the user. Then, a data collection software collects and merges the coverage information for later analysis.

Apart from these hardware monitors, there are software based mechanisms like regression suites to monitor the coverage closure.

9.1.2 Regression Suites. Regression suites are software-based test generation programs or test templates that are used periodically to ensure verification coverage. The exerciser runs on a pre-silicon accelerator and coverage information is collected to check if coverage closure is reached. Then, the test templates are harvested as regression suites and replayed in silicon. There are two major types of regression suites: Deterministic and Probabilistic. Deterministic suites are built using test cases and Probabilistic suites are built using test templates. Regression suites are used in POWER7 processor validation (Adir et al. 2011) for reaching coverage closure.

10 RESEARCH DIRECTIONS

In the preceding sections, we presented an extensive survey of the different state-of-the-art techniques for functional validation of processors. Based on this survey, and the recent architectural developments in this field, we identify a few research questions that are to be addressed.

Architectural Support

Observability Enhancement: The foremost question in post-silicon validation is the observability of internal signals in a processor (Basu et al. 2013). Enhancing access to these internal signals is an open research question. In addition, there are a few challenges to be addressed in a multicore design:

Parallel verification: How can we use a checker core to perform parallel verification for a cluster of cores? In such a scenario, how can a small set of signals be used to observe a large number of states?

When the cores are heterogeneous and the sharing of caches also varies from one architecture to another, how can we exploit parallel verification?

Software-Based Methods. Existing techniques address major issues like on-line testing of multicore, building test generation framework and using predictive analytics for new upcoming designs. Some issues that need to be addressed here are as follows:

(1) *Lack of Golden Reference Model:* Can we built test generation frameworks for heterogeneous designs that lack a golden model?
(2) *Reuse test cases:* How do we select appropriate tests? How do we reuse the test cases for other multicore architectures?
(3) *Building a statistical model using machine learning:* Can we build a model that is capable of predicting the inconsistent bugs more accurately in a new design? Bugs that occur

in multiple scenarios and bugs of high dimensionality are some issues that need to be addressed in machine learning.

Formal Methods. Current techniques use formal methods to find an error trace using model checkers, state machines, and SAT solvers. When the number of states is enormously large in multicore designs, it leads to a few open issues in the existing techniques:

(1) *Improving coverage:* How can we increase the number of states that can be backtracked while finding error trace?
(2) *Storing Transaction-level states:* As there is an enormous number of interactions between cores and interconnect networks, how can we obtain and store these transactions or communications?

Other Emerging Areas. Before the appearance of multicore processors, most of the verification efforts were on the core functional units. With the arrival of multicore, there has been a shift in focus toward noncore components. With the architectural developments in 3D stacks and memories, there is bound to be a shift toward verification of multi-die 3D ICs. New validation techniques may be needed to address the issues unique to this technology. One such idea could be the development of a reusable verification environment where each die can be separately tested in a stack of dies without simultaneously operating all the dies in a stack.

Aspect Oriented Programming (AOP) is another new dimension that can make use of reusable verification environment (Kiczales and Hilsdale 2001). AOP focuses on verifying the cross-cutting concerns or aspects in a design. Using AOP, it is easy to modify the verification environment whenever a new component is added to the chip (Regimbal et al. 2003). Hence, this paradigm can be adopted for developing validation platforms (Aljasser and Schachte 2009).

In addition to this, application-specific validation is emerging for new architectures. In this context, validation is focused toward testing the design for a specific application by predicting all possible usage scenarios and stress testing the application on the new design. With many new applications being developed, this is a fertile field for new research.

Thus there are many new challenges and many new directions that need to be explored.

11 CONCLUSION

The goal of this survey has been the understanding of the state of the art in the functional validation of multicore designs. To this end, we have presented various approaches that have been proposed in the last decade using hardware, software, and formal methods. We find that there are many techniques in all these directions that now focus on the uncore components, in addition to the core components. We believe that this trend is likely to continue with multicore giving way to many-core architectures. Also, we intuitively identify certain trends that are likely to lay the road for the near future. Verification in parallel is promising to be the prime focus for multicore architectures. For multicore verification, decoupled execution would play a vital role as separate streams can be verified in a parallel manner. With on-line testing, scalability issues are inclined to get attention with an increase in core count. Big data predictive analytics and machine-learning techniques are other areas that are likely to see further growth. With this and a set of new possibilities, research in post-silicon validation promises to be exciting and challenging.

REFERENCES

Rawan Abdel-Khalek and Valeria Bertacco. 2012. Functional post-silicon diagnosis and debug for networks-on-chip. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'12)*. ACM, 557–563.

Rawan Abdel-Khalek and Valeria Bertacco. 2014. DiAMOND: Distributed alteration of messages for on-chip network debug. In *Proceedings of the 8th IEEE/ACM International Symposium on Networks-on-Chip (NoCS'14)*. IEEE, 127–134.

Allon Adir, Eli Almog, Laurent Fournier, Eitan Marcus, Michal Rimon, Michael Vinov, and Avi Ziv. 2004. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Des. Test Comput.* 21, 2 (Mar. 2004), 84–93.

Allon Adir, Maxim Golubev, Shimon Landa, Amir Nahir, Gil Shurek, Vitali Sokhin, and Avi Ziv. 2011. Threadmill: A post-silicon exerciser for multi-threaded processors. In *Proceedings of the 48th Design Automation Conference (DAC'11)*. ACM, 860–865.

Allon Adir, Amir Nahir, Gil Shurek, Avi Ziv, Charles Meissner, and John Schumann. 2011. Leveraging pre-silicon verification resources for the post-silicon validation of the IBM POWER7 processor. In *Proceedings of the 48th ACM/EDAC/IEEE Design Automation Conference (DAC'11)*. IEEE, 569–574.

Allon Adir, Amir Nahir, and Avi Ziv. 2012. Concurrent generation of concurrent programs for post-silicon validation. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 31, 8 (2012), 1297–1302.

Allon Adir, Amir Nahir, Avi Ziv, Charles Meissner, and John Schumann. 2010. Reaching coverage closure in post-silicon validation. In *Proceedings of the Haifa Verification Conference*. Springer, 60–75.

Khalid Aljasser and Peter Schachte. 2009. ParaAJ: Toward reusable and maintainable aspect oriented programs. In *Proceedings of the 32nd Australasian Conference on Computer Science*, Vol. 91. Australian Computer Society, Inc., 65–74.

AMD. 2009. *Revision Guide for AMD Athlon 64 and AMD Opteron Processors*. Retrieved from http://support.amd.com/TechDocs/25759.pdf.

AnandTech. 2008. AMD's B3 Stepping Phenom Previewed, TLB Hardware Fix Tested. Retrieved from http://www.anandtech.com/show/2477/2.

Ehab Anis and Nicola Nicolici. 2007. Interactive presentation: Low cost debug architecture using lossy compression for silicon debug. In *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 225–230.

Amin Ansari, Shuguang Feng, Shantanu Gupta, and Scott Mahlke. 2010. Necromancer: Enhancing system throughput by animating dead cores. *ACM SIGARCH Comput. Arch. News*. 38, 3 (Jun. 2010), 473–484.

Todd M. Austin. 2000. DIVA: A dynamic approach to microprocessor verification. *J. Instr.-Level Parallel.* 2, 5 (May 2000), 1–26.

Kyle Balston, Mehdi Karimibiuki, Alan J. Hu, Alexander Ivanov, and Steven J. E. Wilton. 2013. Post-silicon code coverage for multiprocessor system-on-chip designs. *IEEE Trans. Comput.* 62, 2 (Feb. 2013), 242–246.

Shibdas Bandyopadhyay. 2004. *A Study on Performance Monitoring Counters in x86-architecture*. Indian Statistical Institute.

Kaustav Basu and Prabhat Mishra. 2013. RATS: Restoration-aware trace signal selection for post-silicon validation. *IEEE Trans. VLSI Syst.* 21, 4 (Apr. 2013), 605–613.

Kaustav Basu, Prabhat Mishra, Prabir Patra, Amir Nahir, and Alon Adir. 2013. Dynamic selection of trace signals for post-silicon debug. In *Proceedings of the 14th International Workshop on Microprocessor Test and Verification (MTV'13)*. IEEE, 62–67.

Valeria Bertacco. 2010. Post-silicon debugging for multi-core designs. In *Proceedings of the 15th Asia and South Pacific Design Automation Conference (ASP-DAC'10)*. IEEE, 255–258.

Jayanta Bhadra, Magdy S. Abadir, Li-C. Wang, and Sandip Ray. 2007. A survey of hybrid techniques for functional verification. *IEEE Des. Test Comput.* 2 (Jun. 2007), 112–122.

Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. 2009. A survey of multicore processors. *IEEE Sign. Process. Mag.* 26, 6 (2009), 26–37.

Harry Bleeker, Peter van Den Eijnden, and Frans de Jong. 2011. *Boundary-scan Test: A Practical Approach*. Springer Science & Business Media.

Tommy Bojan, Manuel Aguilar Arreola, Eran Shlomo, and Tal Shachar. 2007. Functional coverage measurements and results in post-Silicon validation of Core 2 duo family. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLVDT'07)*. IEEE, 145–150.

Keith A. Bowman, James W. Tschanz, Shih-Lien L. Lu, Paolo A. Aseron, Muhammad M. Khellah, Arijit Raychowdhury, Bibiche M. Geuskens, Carlos Tokunaga, Chris B. Wilkerson, and Tanay Karnik. 2011. A 45 nm resilient microprocessor core for dynamic variation tolerance. *IEEE J. Solid-State Circ.* 46, 1 (2011), 194–208.

Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. 2000. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perf. Comput. Appl.* 14, 3 (Aug. 2000), 189–204.

Randal E. Bryant and James H. Kukula. 2003. Formal methods for functional verification. In *The Best of ICCAD*. Springer, 3–15.

Ajay Chandramouly, Ravindra Narkhede, Vijay Mungara, Guillermo Rueda, and Asoka Diggs. 2013. Reducing Client Incidents Through Big Data Predictive Analytics. Intel WhitePaper. Retrieved from http://www.intel.in/content/dam/www/public/us/en/documents/white-papers/reducing-client-incidents-through-big-data-predictive-analytics.pdf.

Chung-Ho Chen, Chih-Kai Wei, Tai-Hua Lu, and Hsun-Wei Gao. 2007. Software-based self-testing with multiple-level abstractions for soft processor cores. *IEEE Trans. VLSI Syst.* 15, 5 (May 2007), 505–517.

Li Chen, Srivaths Ravi, Anand Raghunathan, and Sujit Dey. 2003. A scalable software-based self-test methodology for programmable processors. In *Proceedings of the 40th Annual Design Automation Conference*. ACM, 548–553.

Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.* 8, 2 (Apr. 1986), 244–263.

Edmund M. Clarke, Orna Grumberg, and Doron Peled. 2000. Model Checking. MIT Press.

Tim Coe. 1995. Inside the Pentium-FDIV Bug. *Dr. Dobbs J.* 20, 4 (Apr. 1995), 129.

R. Collett. 2004. *IC/ASIC Functional Verification Study*. Industry Report from Collett International Research (2004), 34.

Kypros Constantinides, Onur Mutlu, Todd Austin, and Valeria Bertacco. 2007. Software-based online detection of hardware defects mechanisms, architectural support, and evaluation. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 97–108.

Flavio M. De Paula, Alan J. Hu, and Amir Nahir. 2012. nuTAB-BackSpace: Rewriting to normalize non-determinism in post-silicon debug traces. In *Proceedings of the International Conference on Computer Aided Verification*. 513–531.

Flavio M. De Paula, Amir Nahir, Ziv Nevo, Avigail Orni, and Alan J. Hu. 2011. TAB-BackSpace: Unlimited-length trace buffers with zero additional on-chip overhead. In *Proceedings of the 48th Design Automation Conference*. ACM, 411–416.

Andrew DeOrio, Adam Bauserman, and Valeria Bertacco. 2007. Chico: An on-chip hardware checker for pipeline control logic. In *Proceedings of the 8th International Workshop on Test and Verification (MTV'07)*. IEEE, 91–97.

Andrew DeOrio, Adam Bauserman, and Valeria Bertacco. 2008. Post-silicon verification for cache coherence. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'08)*. IEEE, 348–355.

Andrew DeOrio, Qingkun Li, Matthew Burgess, and Valeria Bertacco. 2013. Machine learning-based anomaly detection for post-silicon bug diagnosis. In *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 491–496.

Andrew DeOrio, Ilya Wagner, and Valeria Bertacco. 2009. Dacota: Post-silicon validation of the memory subsystem in multi-core designs. In *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture (HPCA'09)*. IEEE, 405–416.

Saurabh Dighe, Sriram Vangal, Nitin Borkar, and Shekhar Borkar. 2009. Lessons learned from the 80-core tera-scale research processor. *Intel Technol. J.* 13, 4 (2009), 1–15.

Saurabh Dighe, Sriram R. Vangal, Paolo Aseron, Shasi Kumar, Tiju Jacob, Keith A. Bowman, Jason Howard, James Tschanz, Vasantha Erraguntla, Nitin Borkar, and others. 2011. Within-die variation-aware dynamic-voltage-frequency-scaling with optimal core allocation and thread hopping for the 80-core teraflops processor. *IEEE J. Solid-State Circ.* 46, 1 (2011), 184–193.

Jack Dongarra, Kevin London, Shirley Moore, Phil Mucci, and Dan Terpstra. 2001. Using PAPI for hardware performance monitoring on Linux systems. In *Proceedings of the International Conference on Linux Clusters: The HPC Revolution*. 25–27.

ExtremeTech. 2016. Skylake bug causes Intel chips to freeze under 'complex workloads'. Retrieved from https://www.extremetech.com/computing/220953-skylake-bug-causes-intel-chips-to-freeze-in-complex-workloads.

Moty Fania, Parviz Peiravi, Ajay Chandramouly, and Chandhu Yalla. 2013. Predictive Analytics and Interactive Queries on Big Data. Intel. Retrieved from https://software.intel.com/sites/default/files/article/486302/ra-predictive-analytics-and-interactive-queries-on-big-data.pdf.

Harry D. Foster. 2015. Trends in functional verification: A 2014 industry study. In *Proceedings of the Design Automation Conference (DAC'15)*. IEEE, 1–6.

Laurent Fournier, Yaron Arbetman, and Moshe Levinger. 1991. Functional verification methodology for microprocessors using the genesys test-program generator. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 434–441.

Nikos Foutris, Mihalis Psarakis, Dimitris Gizopoulos, Andreas Apostolakis, Xavier Vera, and Antonio González. 2010. MT-SBST: Self-test optimization in multithreaded multicore architectures. In *Proceedings of the IEEE International Test Conference (ITC'10)*. IEEE, 1–10.

Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. 2009. *Embedded System Design: Modeling, Synthesis and Verification*. Springer Science & Business Media.

Alok Garg and Michael C. Huang. 2008. A performance-correctness explicitly-decoupled architecture. In *Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture (MICRO-41)*. IEEE, 306–317.

Amir Masoud Gharehbaghi and Masahiro Fujita. 2012. Transaction-based post-silicon debug of many-core system-on-chips. In *Proceedings of the 13th International Symposium on Quality Electronic Design (ISQED'12)*. IEEE, 702–708.

Dimitris Gizopoulos, Mihalis Psarakis, Miltiadis Hatzimihail, Michail Maniatakos, Antonis Paschalis, Anand Raghunathan, and Srivaths Ravi. 2008. Systematic software-based self-test for pipelined processors. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. 16, 11 (2008), 1441–1453.

Marcel Gort, Flavio M. De Paula, Johnny J. W. Kuan, Tor M. Aamodt, Alan J. Hu, Steven J. E. Wilton, and Jin Yang. 2012. Formal-analysis-based trace computation for post-silicon debug. *IEEE Trans. VLSI Syst.* 20, 11 (Nov. 2012), 1997–2010.

Satish Kumar Grandhi. 2006. *Post-silicon Validation of RAS Features for Next Generation Processors.* Master's Thesis. NITK, India.

Qi Guo, Tianshi Chen, Yunji Chen, Rui Wang, Huanhuan Chen, Weiwu Hu, and Guoliang Chen. 2014. Pre-silicon bug forecast. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 33, 3 (Mar. 2014), 451–463.

Qi Guo, Tianshi Chen, Haihua Shen, Yunji Chen, Yue Wu, and Weiwu Hu. 2011. Empirical design bugs prediction for verification. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'11).* IEEE, 1–6.

Abu Hassan, Janusz Rajski, and Vinod K. Agarwal. 1988. Testing and diagnosis of interconnects using boundary scan architecture. In *Proceedings of the International Test Conference.* IEEE, 126–137.

Marcos B. Hervé, Erika Cota, Fernanda L. Kastensmidt, and Marcelo Lubaszewski. 2009. NoC interconnection functional testing: Using boundary-scan to reduce the overall testing time. In *Proceedings of the 10th Latin American Test Workshop (LATW'09).* IEEE, 1–6.

C. Richard Ho, Michael Theobald, Brannon Batson, J. Grossman, Stanley C. Wang, Joseph Gagliardo, Martin M. Deneroff, Ron O. Dror, and David E. Shaw. 2009. Post-silicon debug using formal verification waypoints. In *Proceedings of the Design and Verification Conference.*

Intel. 2015a. Intel Xeon Processor E5 v2 Product Family Specification Update. Retrieved from http://www.intel.in/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-v2-spec-update.pdf.

Intel. 2015b. Intel Xeon Phi Coprocessor x100 Product Family Specification Update. Retrieved from http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-phi-coprocessor-specification-update.pdf.

Rajshekar Kalayappan and Smruti R. Sarangi. 2013. A survey of checker architectures. *ACM Comput. Surv.* 45, 4 (Aug. 2013), 48.

David S. Karpenske. 1991. Interconnect verification of multichip modules using boundary scan. In *Proceedings of the VLSI Test Symposium Chip-to-System Test Concerns for the 90's Digest of Papers.* IEEE, 85–91.

Christoph Kern and Mark R. Greenstreet. 1999. Formal verification in hardware design: A survey. *ACM Trans. Des. Autom. Electr. Syst.* 4, 2 (Apr 1999), 123–193.

Gregor Kiczales and Erik Hilsdale. 2001. Aspect-oriented programming. *ACM SIGSOFT Softw. Eng. Not.* 26, 5 (Jun 2001).

Ho Fai Ko and Nicola Nicolici. 2009. Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 28, 2 (Feb 2009), 285–297.

Nektarios Kranitis, Antonis Paschalis, Dimitris Gizopoulos, and George Xenoulis. 2005. Software-based self-testing of embedded processors. *IEEE Trans. Comput.* 54, 4 (2005), 461–475.

Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. 2002. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 21, 12 (Dec 2002), 1377–1394.

Dongyang Lin, Tianqi Hong, Yanjing Li, S. Eswaran, Sudhakar Kumar, Farzan Fallah, Nagib Hakim, Donald S. Gardner, and Subhasish Mitra. 2014. Effective post-silicon validation of system-on-chips using quick error detection. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 33, 10 (Oct. 2014), 1573–1590.

John M. Ludden, Michal Rimon, Bryan G. Hickerson, and Allon Adir. 2010. Advances in simultaneous multithreading testcase generation methods. In *Proceedings of the Haifa Verification Conference.* Springer, Berlin, 146–160.

Yi Ma, Hongliang Gao, Martin Dimitrov, and Huiyang Zhou. 2007. Optimizing dual-core execution for power efficiency and transient fault recovery. *IEEE Trans. Parallel Distrib. Syst.* 18, 8 (2007), 1080–1093.

Yossi Malka and Avi Ziv. 1998. Design reliability-estimation through statistical analysis of bug discovery data. In *Proceedings of the Design Automation Conference, Proceedings.* IEEE, 644–649.

Biruk Mammo, Jim Larimer, Mark Morgan, Deliang Fan, Eric Hennenhoefer, and Valeria Bertacco. 2012. Architectural trace-based functional coverage for multiprocessor verification. In *Proceedings of the 13th International Workshop on Microprocessor Test and Verification (MTV'12).* IEEE, 1–5.

Biruk W. Mammo, Valeria Bertacco, Andrew DeOrio, and Ilya Wagner. 2015. Post-silicon validation of multiprocessor memory consistency. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 34, 6 (Jun 2015), 1027–1037.

Kenneth L. McMillan. 1994. Fitting formal methods into the design cycle. In *Proceedings of the 31st Design Automation Conference.* IEEE, 314–319.

Albert Meixner, Michael E. Bauer, and Daniel J. Sorin. 2007. Argus: Low-cost, comprehensive error detection in simple cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07).* IEEE, 210–222.

Albert Meixner and Daniel J. Sorin. 2009. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. *IEEE Trans. Depend. Secure Comput.* 6, 1 (Jan 2009), 18–31.

A. Mishchenko. 2012. ABC: A System for Sequential Synthesis and Verification. Berkeley Logic Synthesis and Verification Group. Retrieved from http://www.eecs.berkeley.edu/~alanmi/abc/.

Prabhat Mishra and Nikil Dutt. 2004. Graph-based functional test program generation for pipelined processors. In *Proceedings of the Design, Automation and Test in Europe*, Vol. 1. IEEE, 182–187.

Prabhat Mishra and Nikil Dutt. 2005a. Functional coverage driven test generation for validation of pipelined processors. In *Proceedings of the Design, Automation and Test in Europe*. IEEE, 678–683.

Prabhat Mishra and Nikil Dutt. 2005b. *Functional Verification of Programmable Embedded Architectures: A Top-down Approach*. Springer Science & Business Media.

Jim Mitchell, Daniel Henderson, and George Ahrens. 2005. *IBM POWER5 Processor-Based Servers: A Highly Available Design for Business-Critical Applications. IBM White Paper* (Oct 2005).

Ashley Montgomery and Tian Tian. 2010. Debugging Machine Check Exceptions on Embedded IA Platforms. Intel White Paper-July 2010.

J. Strother Moore, Thomas W. Lynch, and Matt Kaufmann. 1998. A mechanically checked proof of the AMD5 K 86 TM floating-point division program. *IEEE Trans. Comput.* 47, 9 (Sep. 1998), 913–926.

Amir Nahir, Manoj Dusanapudi, Shakti Kapoor, Kevin Reick, Wolfgang Roesner, Klaus-Dieter Schubert, Keith Sharp, and Greg Wetli. 2014. Post-silicon validation of the IBM POWER8 processor. In *Proceedings of the 51st DAC Conference*. 1–6.

Yaser Ahangari Nanehkaran and Sajjad Bagheri Baba Ahmadi. 2013. The challenges of multi-core processor. *Int. J. Adv. Res. Technol.* 2, 6 (Jun 2013), 36–39.

Satish Narayanasamy, Bruce Carneal, and Brad Calder. 2007. Patching processor design errors. In *Proceedings of the International Conference on Computer Design (ICCD'06)*. IEEE, 491–498.

Pridhiviraj Paidipeddi and Dheerendra Singh Tomar. 2014. Machine learning adaptation in post silicon server validation. *Int. J. Appl. Inf. Syst.* 7, 11 (Nov. 2014), 11–14.

Antonis Paschalis and Dimitris Gizopoulos. 2005. Effective software-based self-test strategies for on-line periodic testing of embedded processors. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 24, 1 (Jan. 2005), 88–99.

PCWorld. 2014. Intel finds specialized TSX enterprise bug on Haswell, Broadwell CPUs. Retrieved from http://www.pcworld.com/article/2464880/intel-finds-specialized-tsx-enterprise-bug-on-haswell-broadwell-cpus.html.

Li-Shiuan Peh, Stephen W. Keckler, and Sriram Vangal. 2009. On-chip networks for multicore systems. *Multicore Processors and Systems*. Springer, 35–71.

Andrew Piziali. 2007. *Functional Verification Coverage Measurement and Analysis*. Springer Science & Business Media.

Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, and Matteo Sonza Reorda. 2010. Microprocessor software-based self-testing. *IEEE Des. Test Comput.* 27, 3 (Jan 2010), 4–19.

Sandip Ray. 2010. *Scalable Techniques for Formal Verification*. Springer Science & Business Media.

Sébastien Regimbal, Jean-François Lemire, Yvon Savaria, Guy Bois, El Mostapha Aboulhamid, and André Baron. 2003. Aspect partitioning for hardware verification reuse. In *System-on-Chip for Real-Time Applications*. Springer, 51–60.

Satish Kumar Sadasivam, Sangram Alapati, and Varun Mallikarjunan. 2012. Test generation approach for post-silicon validation of high end microprocessor. In *Proceedings of the 15th Euromicro Conference on Digital System Design (DSD'12)*. IEEE, 830–836.

Smruti Sarangi, Satish Narayanasamy, Bruce Carneal, Abhishek Tiwari, Brad Calder, and Josep Torrellas. 2007. Patching processor design errors with programmable hardware. *IEEE Micro* 27, 1 (Jan. 2007), 12–25.

Bastian Schlich, Thomas Reinbacher, Michael Kramer, and Martin Horauer. 2008. Challenges in embedded model checkinga simulator for the [mc] square model checker. In *Proceedings of the International Symposium on Industrial Embedded Systems*. IEEE, 245–248.

Michael A. Skitsas, Chrysostomos A. Nicopoulos, and Maria K. Michael. 2012. Toward selective software-based self-testing in multi-core microprocessors. In *Proceedings of the 1st MEDIAN Workshop*. 71–75.

Michael A. Skitsas, Chrysostomos A. Nicopoulos, and Maria K. Michael. 2013. DaemonGuard: O/S-assisted selective software-based self-testing for multi-core systems. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT'13)*. IEEE, 45–51.

Jared C. Smolens, Brian T. Gold, Babak Falsafi, and James C. Hoe. 2006. Reunion: Complexity-effective multicore redundancy. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 223–234.

Pramod Subramanyan. 2010. *Efficient Fault tolerance in chip multiprocessors using critical value forwarding*. MSThesis. Supercomputer Education and Research Center. Indian Institute of Science, Bangalore, India.

Pramod Subramanyan, Virendra Singh, Kewal K. Saluja, and Erik Larsson. 2010. Energy-efficient fault tolerance in chip multiprocessors using critical value forwarding. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'10)*. IEEE, 121–130.

Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. 2000. Slipstream processors: Improving both performance and fault tolerance. *ACM SIGPLAN Not.* 35, 11 (Nov. 2000), 257–268.

Michael Sung, Ronny Krashinsky, and Krste Asanović. 2001. Multithreading decoupled architectures for complexity-effective general purpose computing. *ACM SIGARCH Comput. Arch. News* 29, 5 (Dec. 2001), 56–61.

George Theodorou, Nektarios Kranitis, A. Paschalis, and Dimitris Gizopoulos. 2011. A software-based self-test methodology for on-line testing of processor caches. In *Proceedings of the IEEE International Test Conference (ITC'11)*. IEEE, 1–10.

Georgios Theodorou, Nektarios Kranitis, Antonis Paschalis, and Dimitris Gizopoulos. 2013. Software-based self test methodology for on-line testing of L1 caches in multithreaded multicore architectures. *IEEE Trans. VLSI Syst.* 21, 4 (Apr. 2013), 786–790.

TheRegister. 2016. AMD to fix slippery hypervisor-bursting bug in its CPU microcode. Retrieved from http://www.theregister.co.uk/2016/03/06/amd_microcode_6000836_fix/.

David Van Campenhout, Trevor Mudge, and John P. Hayes. 2000. Collection and analysis of microprocessor design errors. *IEEE Des. Test Comput.* 17, 4 (Oct. 2000), 51–60.

Ad J. Van De Goor. 1993. Using march tests to test SRAMs. *IEEE Des. Test Comput.* 10, 1 (Mar. 1993), 8–14.

Srivatsa Vasudevan. 2006. *Effective Functional Verification: Principles and Processes*. Springer Science & Business Media.

Dave W. Victor, John M. Ludden, Richard D. Peterson, Bradley S. Nelson, W. Keith Sharp, James K. Hsu, B.-L. Chu, Michael L. Behm, Rebecca M. Gott, Audre D. Romonosky, and others. 2005. Functional verification of the POWER5 microprocessor and POWER5 multiprocessor systems. *IBM J. Res. Dev.* 49, 4.5 (2005), 541–553.

Ilya Wagner and Valeria Bertacco. 2008. Reversi: Post-silicon validation system for modern microprocessors. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'08)*. IEEE, 307–314.

Ilya Wagner and Valeria Bertacco. 2010. *Post-Silicon and Runtime Verification for Modern Processors*. Springer Science & Business Media. DOI: http://dx.doi.org/10.1007/978-1-4419-8034-2

Ilya Wagner, Valeria Bertacco, and Todd Austin. 2007. Microprocessor verification via feedback-adjusted Markov models. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 26, 6 (Jun 2007), 1126–1138.

David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. 2007. On-chip interconnection architecture of the tile processor. *IEEE Micro* 27, 5 (Sep 2007), 15–31.

Bruce Wile, John C. Goss, and Wolfgang Roesner. 2005. *Comprehensive Functional Verification the Complete Industry Cycle*. Morgan Kaufmann.

WilsonResearchGroup. 2014. The 2014 Wilson Research Group Functional Verification Study. Mentor Graphics. Retrieved from https://blogs.mentor.com/verificationhorizons/blog/2015/01/21/prologue-the-2014-wilson-research-group-functional-verification-study/.

Qiang Xu and Xiao Liu. 2010. On signal tracing in post-silicon validation. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*. IEEE Press, 262–267.

Joonhyuk Yoo. 2007. *Harnessing Checker Hierarchy for Reliable Microprocessors*. Ph.D Thesis. University of Maryland.

Joonhyuk Yoo and Manoj Franklin. 2008. Hierarchical verification for increasing performance in reliable processors. *J. Electr. Test.* 24, 1–3 (Jun. 2008), 117–128.

Huiyang Zhou. 2005. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. IEEE, 231–242.

Received July 2016; revised April 2017; accepted June 2017