Post-silicon Debugging for Multi-core Designs

Valeria Bertacco
Dept. of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109
valeria@umich.edu

Abstract— Escaped errors in released silicon are growing in number due to the increasing complexity of modern processor designs and shrinking production schedules. Worsening the problem are recent trends towards chip multiprocessors (CMPs) with complex and sometimes non-deterministic memory subsystems prone to subtle, devastating bugs. This deteriorating situation is causing a growing portion of the validation effort to shift to post-silicon, when the first few hardware prototypes become available and where validation experiments are run directly on newly manufactured prototype hardware. While post-silicon validation enables much higher raw performance in test execution, it is a much more challenging environment for bug diagnosis and correction. In this work we briefly overview some of the current methodologies used in industry. We then discuss some recent ideas developed in our research group to leverage the performance advantage of post-silicon validation, while sidestepping its limitations of low internal node observability and expensive bug fixing. Finally we present some of today's general trends in post-silicon validation research.

I. Introduction

Post-silicon validation encompasses all that validation effort that is poured onto a system after the first few silicon prototypes become available, but before product release. While in the past most of this effort was dedicated to validating electrical aspects of the design, or diagnosing systematic manufacturing defects, today a growing portion of the effort focuses on functional system validation. This trend is for the most part due to the increasing complexity of digital systems, which limits the verification coverage provided by traditional pre-silicon methodologies. As a result, a number of functional bugs survive into manufactured silicon, and it is the job of post-silicon validation to detect and diagnose them so that they do not escape into the released system. The bugs in this category are often system-level bugs and rare corner-case situations buried deep in the design state space: since these problems encompass many design modules, they are difficult to identify with pre-silicon tools, characterized by limited scalability and performance. Post-silicon validation, on the other hand, benefits from very high raw performance, since tests are executed directly on manufactured silicon. At the same time, it poses several challenges to traditional validation methodologies, because of the limited internal observability and difficulty of applying modifications to manufactured silicon chips. These two factors lead in turn to critical challenges in error diagnosis and correction.

This situation is further exacerbated by modern multi-core processor systems, where multiple processor nodes share one or more memory blocks through an interconnect network. In most architectures, the communication between these nodes and the central memory is regulated by cache coherence and consistency protocols. The implementation of these protocols in a hardware system is much more complex then their high level specification, because of many possible intermediate situations that may arise when several cores are communicating at the same time. In addition, memory protocols pose a few constraints on the ways processor cores may share data, but in general they do not enforce a unique execution flow. As a result, deterministic behavior by the system can no longer be expected, and something that used to be as simple as determining the correctness of a test by checking a unique correct output, is now a challenging problem in itself, since several correct outcomes are possible. Due to the complexity of even basic cache coherence models and the slow simulation speed of pre-silicon verification, large portions of the design space go unverified at the pre-silicon stage and an increasing number of functional errors escape to silicon, hopefully to be detected in the post-silicon validation phase. It is not a surprise then that, within several large microprocessor design houses, post-silicon validation has become the fastest growing cost item of the entire development effort [1].

In this paper we provide a brief overview of current approaches for post-silicon validation and efforts to overcome its limitations in bug detection and diagnosis. We discuss current industry practices and cover some of the recent solutions proposed by our research group in this space, specifically targeting multi-core designs. We conclude the paper by pointing at some of the trends in post-silicon validation research.

II. Post-silicon validation and its challenges

Functional post-silicon validation strives to establish if the prototype adheres to its initial specification and truly embodies the designer's intent. It relies on a concept similar to simulation: the hardware prototype executes as many randomly generated input vectors as possible. However, there are a few key differences between this approach and pre-silicon validation. First, the execution on a hardware prototype is several orders of magnitude faster than any functional simulator, therefore, more and longer test sequences can be tested, and higher coverage may be obtained. To put things in perspective, consider that during the validation of the Pentium 4 processor approximately 200 billion cycles were simulated before tape-out [2]. That effort accounted for only 3 minutes of runtime in the actual processor operating at 1GHz frequency.

This high speed, however, comes at the price of limited observability: the internal state of the prototype cannot be easily nor fully observed, forcing engineers to diagnose errors by accessing only architectural state registers.

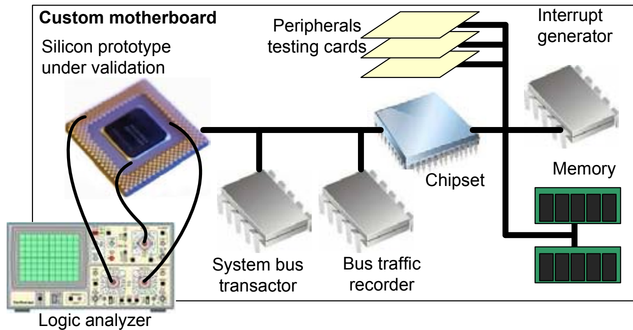

Fig. 1. **A sample post-silicon validation platform.** The prototype processor is verified on a custom-built motherboard equipped with a number of testing and debugging modules, including tester cards, off-chip communication traffic snoopers/recorders, *etc.*

Tests in the post-silicon domain consists of directed tests checking specific features of the processor, automatically generated random tests, as well as compatibility checks, such as operating system boot-up and tests with legacy software [2, 3]. Directed tests demand much engineering effort and are limited to short sequences or basic system activities. Random-generated tests have a much broader scope but present challenges in establishing whether the outcome of a test is correct or not. Due to the unpredictable outcome of these random programs, engineers must simulate them on a known-correct model of the design to obtain the correct final state, which is then compared with that of the hardware prototype to identify discrepancies. While tests can be run at-speed on the hardware, test generation and simulation constitute the bottleneck in this process, almost reducing it to the performance level of pre-silicon simulation. Consequently, design houses are forced to spend enormous computational resources on test simulation server farms [3]. Nevertheless, verification with randomized programs remains a central component of the post-silicon validation process, since it can expose many unexpected behaviors of the system missed by directed tests. For instance, in the context of microprocessors, designers often randomize the timing of certain input events, such as interrupts, during a test and alter delays of messages sent to and from the processor on the system bus. Typically, this is done with custom-made hardware engines, which reside on the same testing board as the prototype and can be programmed to exhibit a variety of behaviors [4]. Fig. 1 illustrates some of the components that are usually part of a post-silicon validation platform.

In addition to the functional validation of the prototype itself, in the post-silicon phase, the compatibility of the processor with a number of deployment platforms is also evaluated. For this purpose, the device is plugged into a system test board with commercially available peripherals, and a broad range directed benchmarks and applications are executed. These include boot up of operating systems, commercial applications, performance benchmarks, legacy software, and so on. The operating conditions of the prototype in this case are not as stressful as with the randomized tests.

Identifying an erroneous behavior in functional post-silicon validation is only the beginning of a long and arduous effort to establish the root cause and propose a fix. As with electrical defects, the process begins with trying to reproduce the bug and determine the conditions under which it occurs. Throughout this activity, design-for-test techniques [5, 6] play a vital role, since they enable designers to sample the internal state of the prototype and unwind the events that caused the erroneous output behavior all the way back to the bug . In this domain, in addition to generic state acquisition solutions, such as scan chains [7] and on-chip logic analyzers [8], engineers often deploy domain-specific techniques, by taking advantage of built-in performance counters and special interrupt modes. In the latter case, for instance, an interrupt may be periodically asserted during the test execution, each time invoking a software routine to dump the processor's state to memory [4].

When traces leading to an error are obtained, the validation team can leverage again pre-silicon tools, trying to reproduce the bug in simulation or using formal tools. A number of recent works have proposed ideas to support this diagnosis effort, by proposing tools that can automatically minimize a trace's length [9, 10] (thus making it more amenable to simulation), identify the module(s) responsible for the bug [11, 12, 13], and also suggest logic modifications to correct the problem [12]. Finally, corrections are evaluated in pre-silicon and then included into new versions of a prototype. Because of the high costs of silicon manufacturing, effort is also dedicated to bypass the issue in test generation, so that the validation of the prototype may continue as long as possible without a re-spin.

The following two sections outline two post-silicon solutions developed in our research group. The first is a technique to validate individual cores with post-silicon tests, but without the need for a separate simulator to determine the correctness of test responses: Reversi [14] is an automatic test generator that creates self-checking tests for post-silicon validation. The second solution focuses on the validation of the memory subsystem in multi-core designs, specifically of the cache coherence protocol. Because in this domain test correctness cannot be uniquely specified, our solution [15] executes a software algorithm in the background that checks all memory activity observed during test execution against the protocol requirements.

III. Validating processor cores with Reversi

As mentioned above, one of the key challenges of post-silicon validation with randomized tests is in determining the correct final state of the system. Typical solutions entail simulating the design's golden model to compute the final processor state and check it against that of the actual hardware prototype (as illustrated in Fig. 2.a). Because of the relatively slower performance of architectural simulation, the computation of this final state becomes a bottleneck for the entire effort. Reversi addresses this issue by generating test programs that sidestep simulation and greatly boost the performance of the overall validation flow (Fig. 2.b). This is accomplished by generating test sequences where the initial state of the system is restored at the end of the test, if and only if the test executes correctly. Moreover, Reversi can run on the prototype itself, after a few basic functionality aspects of the hardware are verified.

To generate tests whose final state matches the initial one, Reversi uses a bottom-up approach by noting that many instructions in a processor's instruction set architecture (ISA) have counterparts, *i.e.*, operations whose functionality is the inverse of the other instructions. For instance, it is possible to restore the original value in a register that was the destination

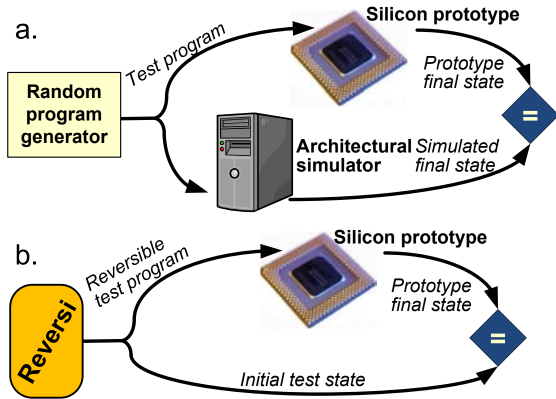

Fig. 2. **Post-silicon validation of random tests vs. Reversi. a.** Random tests produced by a test generator are fed to both a silicon prototype and an architectural simulator. When the outcome of the two platforms differ, a bug is flagged. **b.** In a Reversi-based flow, tests are generated using the prototype itself, and their outcome is known by construction, thus the architectural simulator can be bypassed, greatly boosting overall testing performance.

of an *add* instruction by using a *sub*tract instruction. Similar goals can be achieved with logic operations, in setting and clearing flags, with branch instructions by creating proper jump sequences, *etc*. In general, operations and/or their inverse may require several instructions, instead of a single one. By combining pairs of operations and their reverse, one can generate tests for which the final register values match the initial ones.

A. Block database

In Reversi, a *block database* specific to the ISA under study is first prepared. This database contains pairs of *functional blocks*: for each operation block, there is a corresponding inverse block, the former modifies the value of a register, called the *focus register*, while its inverse restores its initial value. Since the focus register is a parameter set dynamically during test generation, a same block may appear in the test program multiple times, each time modifying a different register. Note that blocks operate only on a single focus register at a time to maintain the reversibility of the program. Thus, for instructions with multiple operands, only one of the registers is the focus register, while other operands are randomly generated by Reversi according to the instruction format. The block-pair structure is sufficiently powerful to allow Reversi to exercise most aspects of a processor's functionality, including loops, procedure calls, *etc.*, and create elaborate tests representative of real software. Moreover, because of this parametric setup, Reversi is agnostic to the functionality of the underlying ISA, making the framework readily adaptable to different architectures.

B. Reversi generator

The Reversi generator combines block pairs from the database to create interesting test sequences. It first prepares several block sequences, which we call *stacks*. A stack is a sequence of instruction blocks, followed by the reverse blocks in reverse order, all using the same focus register and a handful of support registers (these are needed to support the operation on the focus register and are not part of the correctness check). Each stack generated alters a different focus register. Thus, on a properly working processor the focus register should be re-

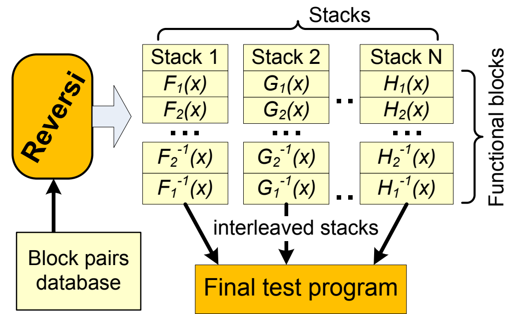

Fig. 3. **Reversi test generator.** Given a functional block database, Reversi creates a set of stacks, each consisting of several blocks and inverse operations. Stacks are then interleaved into a program with predictable outcome.

stored to its original value once a stack execution completes. In generating multiple stacks, Reversi selects a different focus register for each of them. In addition, it also allocates completely disjoint sets of support registers to each stack: while this is not a requirement, it does simplify the subsequent phase of stack interleaving. All stacks generated are then interleaved into a complex reversible test program, as Fig. 3 illustrates. This is accomplished by selecting instructions from all stacks and chaining them together to form a single test program.

It is important to note that Reversi programs provide more aid in debugging than traditional randomly generated programs: bugs can be isolated by checking if an exposing instruction sequence is located in an individual stack, and by "peeling" operations and inverse blocks from the program. Therefore, a reversible program exposing a bug can be dramatically shortened to alleviate debugging. In contrast, in a traditional flow a costly re-simulation is required to obtain the new golden state after each change in the test program.

IV. Memory subsystem validation

Once individual cores are sufficiently validated, their interactions must also be checked. In most chip multiprocessor (CMP) designs, the communication between cores is controlled by cache coherence and consistency protocols. In this section we discuss a solution developed in our group to provide post-silicon validation of cache coherence protocols. This solution, called CoSMa, can be deployed on systems employing a wide range of coherence mechanisms. It is not a test generation solution, rather it observes tests executing on the CMP platform and checks that all data sharing occurs in respect of the coherence protocol in use. We found experimentally that it enables high coverage verification while incurring a small performance overhead ($<$23%) and a negligible area impact ($\ll$ 1%). For the sake of brevity, we only highlight the cache coherence validation capabilities of CoSMa. In a later effort, we also implemented a technique to validate cache consistency, in a technology called Dacota [16].

In a CoSMa-augmented design, a multi-core processor is extended to include a special mode of operation to be activated only during post-silicon validation. While in this mode, time is organized into epochs: each epoch includes a phase of normal execution, when relevant memory activity information is logged in the background, followed by a checking phase, during which all the activity is checked against the coherence pro-

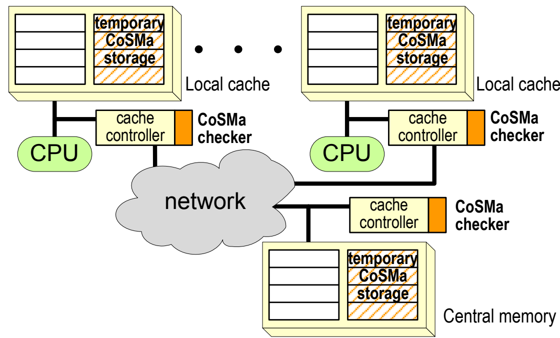

Fig. 4. **CoSMa architecture**. Small checkers residing on each cache controller are responsible for logging the relevant memory activity and for coordinating data transfers during the checking phase. Storage for the activity logging (hashed areas) is temporarily allocated only during post-silicon validation using the existing memory resources of the system. Note that the solution is agnostic to the network topology.

tocol in use in the system. The activity logged is stored in a portion of the cache resources, temporarily reclaimed for this purpose during post-silicon validation. This cache reclamation technique keeps the the area overhead at a bare minimum (Fig. 4). The logged activity includes a footprint of the coherence protocol activity, and possibly processor and timing information. CoSMa offers two variants of this mechanism, one optimized for minimal performance overhead, the other for high coverage. During the checking phase, coherence is checked by a distributed software algorithm implementing a variant of string matching.

A. Runtime operation

During each epoch, normal execution extends until the storage reserved for CoSMa has been exhausted. At this point, all pending memory operations are allowed to complete, ensuring a consistent system-level state and emptying all queues and buffers for data transfers. Then the checking phase begins: the controller in the central memory (or level-2 cache) broadcasts the activity log of each of its valid address lines. Local caches that have the same address line marked as valid compare their local activity log against the one received to expose potential coherence violations. The process completes with the local caches verifying that all local address lines that did not go through this check are indeed invalid: assuming an inclusive central memory, all lines in a local cache must also reside in central memory. If an error is detected in this process an exception is raised and debug information is readily available for the verification team in the activity log storage.

B. Checking Algorithm

The checking procedure analyzes two activity logs corresponding to the same cache line: the global history from central memory and the local history from the local cache. Two algorithms are available for this task: a low-overhead algorithm optimized for low performance overhead and minimal perturbation to the system under test. This algorithm logs compact, encoded sequences of cache coherence states. The second algorithm targets high validation coverage but has higher performance impact. In this case, additional timestamps and processor IDs are stored in the activity logs. The basis of the checking algorithms is that, in a correctly functioning cache coherence scheme, the state of each valid local cache line must agree with the corresponding central memory line. The activity logs are deemed compatible if there exists at least one valid sequence of operations that could have generated both.

V. Current trends in post-silicon research

The challenges of modern system verification have led to a demand for more effective and broad-reaching post-silicon validation technologies. While until a few years ago, post-silicon validation was a burden confined to industry effort, in the past five years a growing number of research efforts have appeared, striving to provide improved and automated solutions for error detection, diagnosis and even correction in post-silicon. The number of research ideas appearing in top quality literature forums is growing quickly: researchers are leveraging simulation-based and abstracted formal techniques to support these tasks in the context of a structured methodology. This effort promises to hold a radical change from traditional post-silicon flows where bugs are difficult to identify and reproduce, and may take several weeks to be diagnosed using ad-hoc, trial-and-error techniques. Additional efforts are also invested in supporting the post-silicon process, for instance through automatic trace minimization to ease the diagnosis phase. All these efforts hold great promise to narrow the gap between design complexity and verification capability.

References

[1] S. Yerramilli, "On the need for convergence between design validation and test," in *Proc. ITC*, Oct. 2006, pp. 14–14.

[2] B. Bentley, "Validating the Intel® Pentium® 4 microprocessor," in *Proc. DAC*, Jun. 2001, pp. 224–228.

[3] H. Rotithor, "Post-silicon validation methodology for microprocessors," *IEEE Design & Test of Computers*, vol. 17, no. 4, pp. 77–88, Oct. 2000.

[4] I. Silas, I. Frumkin, E. Hazan, E. Mor, and G. Zobin, "System-level validation of the Intel® Pentium® M processor," *Intel Technology Journal*, vol. 07, pp. 38–43, May 2003.

[5] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*, revised ed. Wiley-IEEE Press, Sep. 1994.

[6] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-signal VLSI Circuits*. Springer, 2000.

[7] R. Kuppuswamy, P. DesRosier, D. Feltham, R. Sheikh, and P. Thadikaran, "Full hold-scan systems in microprocessors: Cost/benefit analysis," *Intel Technology Journal*, vol. 08, pp. 63–72, Feb. 2004.

[8] W. Corti, R. Kenny, J. Marsh, S. Parker, F. Scanzano, and M. Won, *U.S. Patent no. 6834360: On-chip logic analyzer*, IBM, Dec. 2004.

[9] K.-H. Chang, V. Bertacco, and I. Markov, "Simulation-based bug trace minimization with BMC-based refinement," vol. 26-1, 2007.

[10] S. Safarpour, A. Veneris, and H. Mangassarian, "Trace compaction using SAT-based reachability analysis," Jan. 2007, pp. 932–937.

[11] Y.-S. Yang, N. Nicolici, and A. Veneris, "Automated data analysis solutions to silicon debug," in *Proc. DATE*, Apr. 2009, pp. 982–987.

[12] K.-H. Chang, I. Markov, and V. Bertacco, "Automating post-silicon debugging and repair," in *Proc. ICCAD*, Nov. 2007, pp. 91–98.

[13] K.-H. Chang, I. Wagner, V. Bertacco, and I. Markov, "Automatic error diagnosis and correction for RTL designs," in *Proc. HLDVT*, Nov. 2007, pp. 65–72.

[14] I. Wagner and V. Bertacco, "Reversi: Post-silicon validation system for modern microprocessors," in *Proc. ICCD*, Oct. 2008, pp. 307–314.

[15] A. DeOrio, A. Bauserman, and V. Bertacco, "Post-silicon verification for cache coherence," in *Proc. ICCD*, Oct. 2008, pp. 348–355.

[16] A. DeOrio, I. Wagner, and V. Bertacco, "Dacota: Post-silicon validation of the memory subsystem in multi-core designs," in *Proc. HPCA*, Feb. 2009, pp. 405–416.