SoCGuard: A Runtime Verification Solution for the Functional Correctness of SoCs

Rawan Abdel-Khalek and Valeria Bertacco
Department of Computer Science and Engineering, University of Michigan
{rawanak, valeria}@umich.edu

***Abstract*—The system-on-chip design methodology is characterized by delivering a very high level of design complexity in a short development time. While this aspect is precisely what makes SoCs appealing, it also creates a unique challenge for their verification, requiring the system to be validated as a whole, besides checking the correctness of each of its components. In this paper, we propose a runtime correctness solution for SoCs where we equip each component with two modes of operation: a default high performance one, where all components capabilities are active, and a basic mode where only baseline modules are in operation, so that all basic functionality can be delivered, but performance is not optimized. By alternating between these two modes of execution on individual components, or on all components together, we can carefully sidestep both component and integration bugs that have escaped into the manufactured product. Our experimental evaluation on a LEON3 SoC shows that this solution incurs only 4% area overhead and less than 5% performance impact in most cases.**

I. Introduction

System-on-chip is a well known design paradigm that has been of interest for many years and constitutes the underlying structure of many electronic systems. SoCs integrate multiple IP cores onto a single chip; they are connected by standard interfaces such as buses and provide a wide range of functions in a highly compact form. The high level of integration of these different components makes SoCs a smaller size and a higher performance alternative to multi-chip solutions. One of the key reasons for the widespread adoption of SoC design lies in the ability to obtain and re-use existing IP blocks and directly integrate them into a new SoC solution, leading to much lower system development cost and time. In contrast, the development costs of ASICs have become prohibitively high, leading to a steady downtrend in the number of new ASIC developments started each year.

Despite the many advantages of SoCs, there are still many challenges that need to be resolved, particularly in their validation. Verifying a SoC requires verifying the individual IP blocks, their integration, and the system as a whole. With continuous scaling of silicon and increasing performance demands, IP cores themselves are becoming more complex. More functionality can also be integrated on a single chip, increasing the overall SoC system complexity as well. On the other end, current verification practices in the industry are far from ensuring complete functional correctness, due to time and computational resource constraints. Traditional pre-silicon verification approaches are incapable of scaling to the complexity of modern SoC designs or reaching good validation coverage. Because adequate verification of the individual components and of all system-level interactions is not attainable, bugs routinely escape into SoCs. While some functional bugs can be sidestepped by adapting the software layer (in the case of embedded systems), this approach is often not viable. Moreover, software workarounds my have a large impact on system performance and may not be viable if the platform is open source. As a result, bugs that escape into the manufactured SoC might manifest during runtime, potentially compromising its functionality, safety, and security and/or forcing a recall of the product.

In this paper, we propose SoCGuard, a novel approach that targets the correctness of the runtime operation of SoCs. Our solution ensures that individual components of an SoC are operating only within the functionally correct sub-domains that have been validated during the development phase. We achieve this goal by equipping each component with two modes of operation: a default high performance mode and a safe basic mode. The basic mode bounds the component to only execute behavior that has been previously verified, thus it is guaranteed to be correct. Integration issues may also be addressed by validating system integration only in the scenarios that occur when all components are in basic mode; and then forcing a system-wide basic mode whenever a potential integration bug is detected. By providing correctness guarantees at the component-level and bypassing bugs at the integration level, we enable a methodology where designers can release SoCs with high confidence in their functional correctness.

SoCGuard is a design-for-verification paradigm that can also be incorporated into the design of every IP unit. Vendors can provide basic modes of operation for their IPs, ensuring that they can always function within a safe subset of their operation. SoC developers can leverage this mode to provide a complete runtime correctness solution for their full system.

A. Contributions

SoCGuard is a novel, distributed verification approach that ensures that SoC's building blocks are operating only within the set of scenarios that have been exercised and verified during their development. At runtime, we monitor the state of each SoC component to detect when it is about to enter an unverified state, and then we force the component into a safer basic mode of operation. The basic mode is a simpler yet slower mode that uses only a subset of that component's functions or units, but still allows it to carry out all basic functionality. The basic mode is easier to verify during the development phase, because the range of concurrent activity

within it is minimal. Thus the correct operation of the IP can be guaranteed in this mode. During verification, we track the scenarios that are being validated, and we generate a combinational block compactly encoding those scenarios. The inputs to the block are a small set of critical control signals in the component, representative of its high-level activity.

Many integration issues can be avoided with this solution alone. If one component is in basic mode, the complexity of its system level interactions is also limited, and often this is sufficient to reduce the diversity of interactions at the system level and bypass escaped bugs. We also provide a basic mode solution for system integration. By monitoring the activity in the bus or interconnect of the SoC, we can detect when the system is about to enter a potential integration bug. The bus can then communicate to all components to enter their basic mode to bypass the issue. It is then necessary to prove that the system operates correctly when all components are in basic mode, a viable option, given the simplicity of the basic mode.

As we show in the experimental results, our approach can be applied to a wide range of IP blocks and it introduces a very small area (4%) and performance (less than 5% in most cases) overhead. The area costs are mainly due to the error detection blocks, while the performance impact is directly related to the quality of the design-time validation effort.

The rest of the paper is organized as follows: Section III describes our runtime verification solution. Sections IV and V discuss the error detection and recovery mechanisms in detail. Section VI presents our experimental evaluation and illustrates how SoCGuard can be applied to the LEON3 SoC.

II. Related Work

Mainstream verification methodologies for SoCs in industry rely heavily on simulation to test IP blocks and their integration. Most work in this area focuses on improving simulation tools and methodology. The contributions in [1]–[5] aim at improving test generation to achieve high verification coverage, and at developing better automated tools to facilitate the verification of large-scale systems. Other work explores techniques to better model the design and its specifications [6]. Finally, other approaches focus on formal verification, targeting only specific modules to bypass the scalability challenge. For example, [7] uses symbolic model checking to verify the PCI bus protocol.

However, SoC correctness is still a challenge exacerbated by design complexity, short development times, and limitations of current verification tools [8]–[10]. In this context, the work in [8], describes some practical approaches to SoC verification, and discusses the need for interface checkers, which ideally should be supplied with IP cores to detect interface protocol violations and provide test coverage metrics. Our work shares with this approach the idea of embedding checkers in the design; however our checkers are to be used at runtime instead of design-time, and provide a solution to avoid functional bugs post system manufacturing. Our embedded monitors track the runtime operation of each IP to ensure that they are operating only within previously verified states. Within the area of runtime verification, we are not aware of any previous work that targets the SoC design paradigm, but some solutions have been suggested for processor cores. DIVA [11] augments a high performance processor with a formally verified "checker core" that trails the execution of the actual core and overrides its results when it detects an error. Our solution shares with DIVA the idea of a runtime detection mechanism, however we target a broad range of components and do not require the addition of a simple copy of the design to bypass the error.

Other recent works in runtime validation rely on patching-based techniques applied to processor cores and cache coherence mechanisms, as proposed in [12]–[14]. Bug signatures are programmed into an added hardware module that triggers a recovery process whenever an error is flagged. The main disadvantage of these techniques is that they can only protect the system from bugs after they have been identified in a released product. In contrast, our work can prevent the occurrence of bugs, even if unknown, since it triggers a system recovery whenever a component enters an unverified state, which may or may not include a bug.

Finally, in [15], Wagner proposes adding a "semantic guardian" to a processor core to monitor its state, so that when an unsafe state is detected, its executions switches to a safe mode. The safe mode re-executes the instruction using only those processor units that have been previously formally verified. That solution focuses only on processor cores, whereas our work provides a novel general solution that can be applied to a wide range of IP components in an SoC. We also propose a technique to integrate the use of multiple runtime checkers within a system, and present a new design methodology that incorporates SoCGuard in the design process of third party IPs to assist in the verification of SoC integration.

III. SoC Runtime Verification

A. Solution Approach

Our runtime verification method provides error-free execution for individual components of a SoC. We augment each IP with a matcher circuit, which tracks that unit's runtime execution. When the matcher detects that its corresponding component is entering a potentially erroneous situation, it flags an error. We consider any state of the component that has not been observed during pre-silicon validation as an error state that could lead to a functional bug, and thus should be flagged. Once an error is flagged, the unit's operation switches from its high-performance mode to a basic mode of operation. Figure 1 outlines this execution flow in a simple SoC example. The basic mode disables the performance features of the design making it simpler and less prone to functional design bugs. One of the key features to disable is the ability to process multiple transactions or requests simultaneously. In addition, all modules and features within the unit that contribute only to its performance, but not to its baseline functionality, are disabled. The basic units of a design tend to be simple, amenable to verification and, in most cases, can be guaranteed to work according to specifications. In contrast, the performance enhancements introduce increasing complexity and

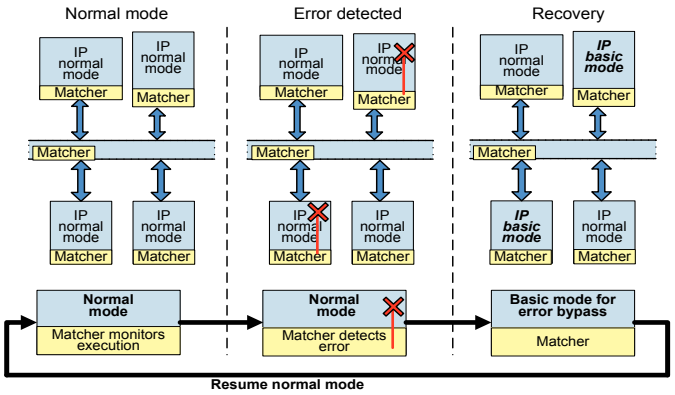

Fig. 1. **Overview of SoCGuard.** A matcher circuit is added to each IP component of the SoC. Any matcher can switch the corresponding IP to its basic mode upon detecting an error. In addition, the interconnect includes a matcher that allows it to toggle all units to basic mode. The bottom part of the figure shows the runtime execution flow, with snapshots of the system in each state shown above.

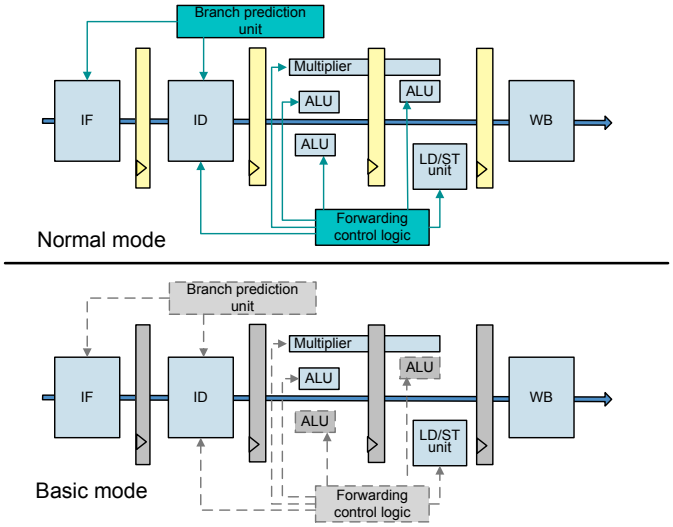

Fig. 2. **Basic mode of operation for a typical processor data path.** The basic mode is created by "switching-off" performance enhancement features such as branch prediction, pipelining, data forwarding, out-of-order execution, *etc.* This results in a simplified execution, with only one instruction in flight at a time, which is much easier to verify for error-free operation.

corner cases that might be missed during testing. Therefore, executing in basic mode allows the component to operate in a verified subset of its state space, thus bypassing the unverified state. In the case of errors detected in different components of the SoC, each of these components enters its own basic mode, independently of other components in the system. However, if an error is detected in the SoC interconnect, then all components are forced into their basic mode of operation.

As an example, Figure 2 shows how a processor may operate in basic mode. The basic pipeline stages and main functional units are sufficient for the execution of the complete ISA. Whereas features such as branch prediction, pipelining, data forwarding, out-of-order execution serve only the purpose of boosting a system's performance, and are therefore not active in basic mode. Similarly, the basic operation of a communication bus is to allow each component to read and write to the bus. Additional performance features, include complex arbitration, burst operations, and multiple units accessing the bus at a given time. Similar approaches can also be applied to designs based on controllers or state machines. Consider the cache controller example shown in Figure 3, whose normal mode of operation consists of handling cache accesses and transferring of data between memory, cache and microprocessor. A basic mode of operation for such a controller bypasses the cache and fetches data directly from memory, one word at a time. Therefore, it only uses a small portion of its complete state machine, which is much simpler and amenable to complete verification.

While an SoC component is operating in its basic mode, the number and frequency of its input and output signaling activity may become lower and simpler. For example, when a processor is executing in basic mode, it only fetches the next instruction when the current instruction is committed. Therefore, the frequency of data transfers between it and the cache and between the cache and memory decreases significantly, even when just the processor is in basic mode, simplifying the interconnect interactions and arbitration, as illustrated also in Figure 4. This simplification of system-level interactions creates a basic mode for the SoC integration. During the development and testing of a SoC, simple system level interactions are more likely to be exercised, whereas bugs tend to manifest in more complex scenarios or corner cases, when multiple activity is ongoing simultaneously. Therefore, at runtime, when an SoC switches to basic mode, the overall system tends to exercise functionality that reflects more closely what has been verified at design time.

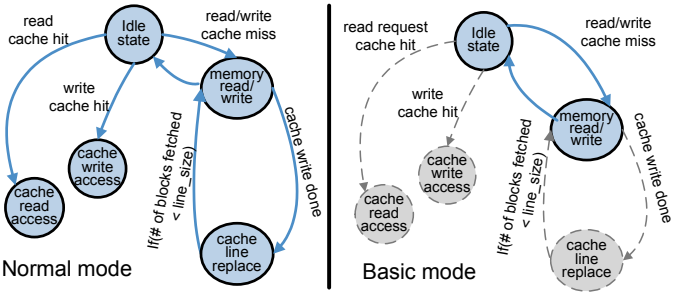

Fig. 3. **Basic mode for a cache controller FSM.** The cache controller's basic mode operation is to fetch data directly from memory, one word at a time, bypassing the cache. The figure shows an example of a complete cache controller on the left side, and its corresponding basic mode on the right. The basic mode effectively uses only a small fraction of the FSM's original states, making it much easier to verify.

B. Integration in SoC Development Flow

The SoCGuard approach is well suited for an effective design-for-verification paradigm. Verifying a SoC requires verifying that each IP is correct by itself and as part of the system. The IP blocks can themselves be very complicated and take up a lot of verification effort; system interactions are complex and difficult to verify as well. Our SoCGuard methodology has IP vendors providing a basic mode of operation for each of their components and, thus reducing their own verification effort and also that of their customers. First, verification of the component itself is simpler: the efforts can concentrate on the basic mode to validate it completely

and simulation-based validation can focus on typical execution scenarios. However, if an operation is missed during testing, it is less likely to expose an escaped bug at runtime, since the hardware can switch to basic mode. Second, this approach also facilitates the verification of system integration: SoC designers would have assurance that the IP units, even those from third parties, are functioning only within previously validated states. In addition, the basic mode of the units creates a simple basic mode for SoC integration in which pre-silicon verification of system integration becomes easier, resulting in higher confidence in its functional correctness. The SoC designers still need to verify system integration in normal mode for common behavior patterns, to the extent permitted by time and available verification resources, and, based on that, generate a matcher circuit for their interconnect bus or logic. If and when this matcher is triggered at runtime, it would switch all IP components into their basic mode, thus bringing the system back into the basic integration mode that has been thoroughly validated. Therefore, the ability to default at runtime to a verified basic mode, leads to a more relaxed correctness requirement for the system.

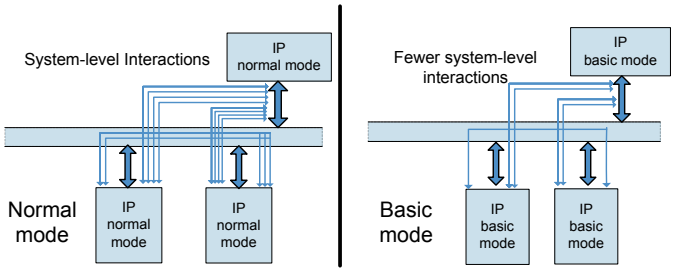

Fig. 4. **Basic mode for SoC integration.** A basic mode for the system-level SoC can be attained in two ways: i) often simpler system interactions occur because one or more of the individual IP units is operating in basic mode. ii) SoCGuard includes also a matcher associated with the SoC interconnect that can trigger a basic mode in all the system's IP units, thus resulting in much simpler and less frequent interactions.

IV. Error Detection

One of the central aspects of SoCGuard is detecting when a hardware component is encountering an error. In order to monitor an IP unit's operation, we first identify the most relevant control signals affecting its operation. We only need to consider signals that impact the behavior outside the basic mode, since we ensure formal correctness of the basic mode. During pre-silicon validation, we monitor these critical control signals and record the observed values. The sequence of vector values obtained is then synthesized and optimized into a combinational logic block, which constitutes the matcher for the IP unit. In our implementation, we encoded the vector values as a simple sum of products and optimized them using Espresso [16]. The matcher is added to its IP unit and connected to the selected control signals. During runtime, any monitored state outside the recorded set of vector values is flagged by the matcher as a potential bug. Figure 5 illustrates the development flow of a SoC with a SoCGuard-based verification methodology.

In choosing the control signals to monitor in an IP unit, we use a combination of manual effort with an automated process. We first run several benchmarks while that IP is forced to operate exclusively in basic mode and again while it is forced to operate only in normal mode. By comparing the switching activity of control signals during these two processes, we can identify candidate signals to observe: good candidates are signals with a much larger switching activity in normal mode than in basic mode. These are the signals that are mostly involved in optimization features or interactions between operations. For instance, in a processor, control signals involved in data forwarding or pipeline flushing exhibit this switching pattern. An additional group of candidates are signals with relatively few transitions or the same number of transitions in both normal and basic mode. These can represent signals that were not triggered often during validation, either because of poor quality testbenches or because they are involved in rare corner case situations. They can also include control signals that identify the type of operation being executed. For our processor core example, signals related to interrupts and exceptions and signals identifying the instruction opcode fall under this category. After narrowing down the list of candidate signals, we then run another automated process that eliminates control signals irrelevant to the actual design, such as VHDL or language-specific signals or debug signals. Finally, it is possible to further narrow down the list of signals to monitor through manual inspection. In our experiments, by applying this technique to the integer pipeline, we identified 600 control signals in the design. The automated process narrowed this set down to 190 potentially relevant signals, from which we manually picked 54 signals. Note that the higher the number of monitored signals, the better is the accuracy of the matcher, and the less frequently potential errors would be flagged, triggering recovery. However, encoding more complex states into a matcher increases its area overhead, resulting in a design trade-off that can be tuned based on external constraints.

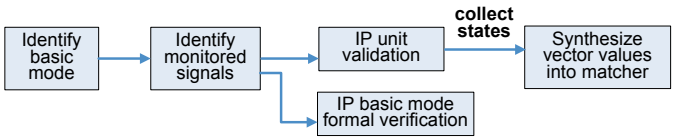

Fig. 5. **SoC development and verification flow using SoCGuard's methodology.** Designers of each IP unit first identify its basic mode of operation. Then, the control signals to be monitored are determined as described in Section IV. The IP undergoes a traditional validation process, while its basic mode is formally verified. The signal values collected are encoded into a combinational circuit and optimized to form the matcher.

V. Error Bypass and Recovery

When a matcher circuit detects a possible error state, it switches the SoC component from its normal, high-performance mode to the basic mode. As soon as a potential error is detected, the component is stalled and prevented from completing its current operation. The potentially unsafe operation is then executed and completed in basic mode. Since the basic mode is just a simpler execution framework that excludes all optimization features, it eliminates several complex interactions and corner cases that tend to be error prone

and easily missed during design-time validation. The units involved in basic mode are therefore easier to verify whether through extensive simulation or through formal verification, and completely guaranteed to be correct. By letting the unsafe operation complete in basic mode, we have a much higher guarantee that no other functional bugs will manifest during the recovery phase and that the unsafe operation will be carried out correctly. In general, an SoC and its IP components usually undergo significant simulation-based validation before they are released to the market. The scenarios that are validated tend to be the most common execution scenarios, those that occur with the highest frequency at runtime, since those are also the easiest ones to reproduce in validation. Therefore, at runtime, IP units enter in basic mode infrequently, because unverified states and corner cases that escaped design-time validation tend to rarely occur, which explains why they could not be reproduced in simulation. As a result, we expect SoCGuard to have a small runtime profile.

VI. Experimental Evaluation

As a case study of adopting the SoCGuard approach, we implemented our solution on LEON3, a VHDL-based model of a SoC. We chose to target the processor's integer pipeline unit and the instruction and data cache controllers, as examples of components in which we incorporated our runtime verification method. We simulated several benchmarks from the MiBench suite [17] on the LEON3 at the RTL level and evaluated the performance impact and area overhead of our approach.

A. Evaluation Platform

The LEON3 includes a 32-bit 7-stage in-order processor, compliant with the SPARC V8 architecture, in addition to a processor bus that connects it to multiple peripheral devices. We categorized the functionalities of the integer pipeline into those required for basic mode and those included only in the normal mode. The basic mode included all units that would allow it to execute one instruction at a time, which reduced the complexity that arises from having multiple instructions in flight and eliminated the use of data forwarding, stalling, and instruction nullifying logic. Using the method described in section IV, some of the critical signals that were chosen for the matcher to monitor were control signals involved in the stalling control logic, data forwarding, and exception handling, giving a total of 54 signals.

We also implemented our solution on the instruction and data cache controllers of LEON3. The cache controllers are similar to the one shown in Figure 3. The basic mode consists of fetching data directly from memory while bypassing the cache and eliminating burst requests. Some of the monitored control signals include cache access signals, request and grant signals between cache and memory or processor, control signals related to cache replacement policies, *etc.* The number of signals chosen to be monitored for the instruction and data cache controllers were 53 and 41, respectively.

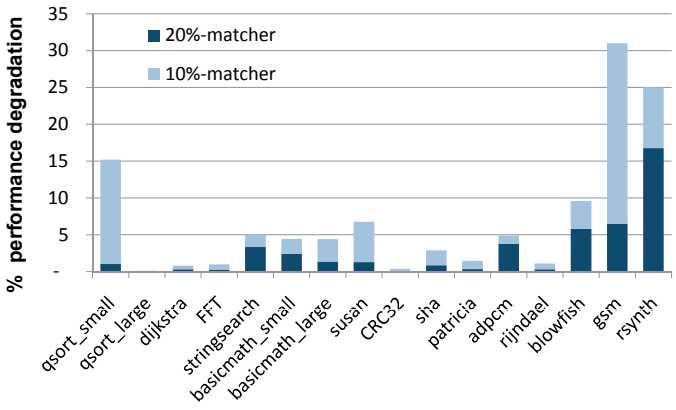

Fig. 6. **Performance impact of SoCGuard.** The graph plots the performance slowdown of the system running the MiBench benchmarks with SoCGuard vs. a baseline solution, with two different sets of matchers: one generated by collecting vector values over 10% of the testbench execution, and one by collecting 20%. Most benchmarks show <5% performance degradation. The performance impact is directly related to the frequency of occurrence of non-validated behavior and adding more vector values into a matcher decreases the frequency of basic mode operation, leading to better performance.

B. Results

Based on the control signals selected for each IP unit, we collected the vector values to encode into the matchers. These should represent the set of states observed during SoC design-time verification. Based on the idea of using statistical sampling proposed in [18], we collected the vector values by uniformly sampling a portion of each benchmark. The sample size was chosen such that the vector values collected from each benchmark accounted for 10% of the total dynamic vector values observed during that benchmark's execution. We sampled all the vector values for 1% of the dynamic execution time every 10% of execution. We then repeated the analysis using samples from 20% of execution.

The three matcher circuits were synthesized by combining the samples from all benchmarks together. The performance of the LEON3 equipped with SoCGuard is reported in Figure 6: the graph reports performance overhead normalized to a baseline LEON3 solution for the MiBench testbenches. As it can be noted, the impact is less than 5% for most benchmarks. The overhead is due to basic mode execution every time one of the matcher circuit flags an error. In `qsort_large` and `CRC`, there is almost no performance impact, since for these benchmarks the states encoded into the matcher were a good representative sample of their execution, and the basic mode was triggered rarely. In contrast, for `rsynth` and `gsm`, whose impact is notable, we found that the vector values collected from 10% of the benchmarks were not well representative of most of their execution. Note also how the matchers based on 20% samples impact overall performance: the overhead of all benchmarks decreases and in particular the overhead of `gsm` decreases to 7% and that of `rsynth` to 17%. We also evaluated the performance cost for one occurrence of the basic mode: when it is triggered by a single component, the basic mode last approximately 30 cycles, when it is system level triggered, it lasts on average 400 cycles.

To further analyze SoCGuard, we also inserted several

Bug name	Bug description
branch_stall	stalling data dependent branch
inst_nullify1	nullifying instruction following consecutive jumps
inst_nullify2	nullifying instruction following a jump and a branch
branch_delay	branch nullifying branch delay slot
store_double	write control signals of store double instruction
icache_tag	icache tag address

TABLE I
Design bugs inserted into LEON3.

Bug name	basicmath large	qsort large	dijkstra	FFT	CRC32
branch_stall	0.25% 4.06% 87.6%	0.28% 3% 73.05%	0.04% 2.02% 42.23%	0.27% 5.8% 116%	5×10^{-5}% 0.005% 0.37%
inst_nullify1	0.006% 0.18% 5.05%	25×10^{-5}% 8.5×10^{-4}% 0.14%	0.04% 0.17% 3.97%	0.01% 0.08% 3.21%	10^{-4}% 0.006% 0.38%
inst_nullify2	0.006% 0.17% 5%	10^{-4}% 6×10^{-4}% 0.13%	10^{-4}% 0.02% 0.82%	10^{-4}% 0.02% 0.99%	10^{-4}% 0.005% 0.37%
branch_delay	0.47% 0.16% 3.72%	0.4% 5×10^{-4}% 0.02%	0.03% 0.02% 0.36%	0.53% 0.03% 0.73%	0.002% 0.005% 0.37%
store_double	0.18% 1.82% 40.8%	0.22% 2% 45.93%	0.016% 0.07% 1.54%	0.21% 3.04% 58.53%	0.003% 0.02% 0.65%
icache_tag	0.02% 0.18% 3.48%	0.066% 7×10^{-4}% 0.13%	0.06% 2.3% 43.58%	0.02% 0.063% 2.22%	0.004% 0.005% 0.37%

TABLE II
Peformance overhead in the presence of design bugs. The values reported are the rate of occurrence of a bug, followed by the basic mode rate and the performance impact relative to normal mode.

functional design errors into the LEON3 (Table I), and we ran benchmarks on the "buggy" designs. With SocGuard, all benchmarks complete correctly, while in the "buggy" baseline, they crash or produce incorrect results. Table II reports the rate at which each bug is triggered, followed by the percentage of times the execution enters basic mode and the overall performance impact. The basic mode is triggered either because the bug manifests or because the execution encounters a state not encoded in the matchers. The more frequently the basic mode is triggered, the higher the performance cost incurred. This relation between performance and design-time validation is a trade-off that can be selected by the design team. As a result, with SoCGuard, a limited design-time validation effort no longer has catastrophic implications on the quality of the final product, but simply has a performance cost, which can be budgeted and controlled.

Finally, we synthesized the baseline LEON3 SoC and the three matcher circuits using Synopsys' Design Compiler and evaluated the area overhead of SoCGuard (Table III). We found that the 10%-matchers, all together, introduced an area overhead of 4.17%, while the 20%-matchers cost 4.25% extra silicon over the baseline.

	integer pipeline	icache controller	dcache controller	total overhead
10%-matcher	1.63	0.44	2.10	4.17
20%-matcher	1.64	0.46	2.15	4.25

TABLE III
Area overhead of the matchers relative to the LEON3 baseline, reported in % for the 10% and 20% matchers.

VII. Conclusion

In this paper, we presented SoCGuard, a runtime verification solution for SoCs. SoCGuard monitors the runtime operation of individual IP units in the SoC, and switches a unit into a basic mode of operation when it detects the occurrence of a state not validated at design time. The basic mode is a simple execution mode that excludes all complex performance and optimization features of the unit. Runtime correctness for SoC integration is attained by two means. First, when an IP unit is in basic mode, its system-level interactions are also simplified. Second, we equip the SoC interconnect itself with a monitor that tracks system-level communication and can trigger the basic mode for all IP units. These two techniques together enable the SoC design's validation effort to be traded-off with performance, thus reducing pressure on the design-time verification. In addition, SoCGuard allows IP vendors to ship their components with embedded matchers, which SoC designers can leverage to build a system-level matcher and to verify the basic mode for their integrated system.

References

[1] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin, "X-gen: a random test-case generator for systems and SoCs," in *HLDVT*, 2002.

[2] X. Xu and C.-C. Lim, "Using transfer-resource graph for software-based verification of system-on-chip," *Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1315 –1328, july 2008.

[3] K. U. Bhaskar, M. Prasanth, G. Chandramouli, and V. Kamakoti, "A universal random test generator for functional verification of microprocessors and system-on-chip," in *VLSID*, 2005.

[4] D. Geist, G. Biran, T. Arons, M. Slavkin, Y. Nustov, M. Farkas, K. Holtz, A. Long, D. King, and S. Barret, "A methodology for the verification of a "system on chip"," in *DAC*, 1999.

[5] M. Lajolo, M. Rebaudengo, M. S. Reorda, M. Violante, and L. Lavagno, "Behavioral-level test vector generation for system-on-chip designs," in *HLDVT*, 2000.

[6] Q. Zhu, T. Nakata, M. Mine, K. Kuroki, Y. Endo, and T. Hasegawa, "System-on-chip verification process using uml," in *UML Satellite Activities*, 2004.

[7] P. Chauhan, E. M. Clarke, Y. Lu, and D. Wang, "Verifying ip-core based system-on-chip designs," in *IEEE ASIC/SoC conference*, 1999.

[8] G. Mosenson, "Practical approaches to SoC verification," in *DATE user forum*, 2000.

[9] N. Bamford, R. K. Bangalore, E. Chapman, H. Chavez, R. Dasari, Y. Lin, and E. Jimenez, "Challenges in system on chip verification," in *MTV*, 2006.

[10] W. Stapleton and P. Tobin, "Verification problems in reusing internal design components," in *DAC*, 2009.

[11] T. M. Austin, "Diva: a reliable substrate for deep submicron microarchitecture design," in *MICRO*, 1999.

[12] I. Wagner and V. Bertacco, "Caspar: Hardware patching for multicore processors," in *DATE*, 2009.

[13] I. Wagner, V. Bertacco, and T. Austin, "Shielding against design flaws with field repairable control logic," in *DAC*, 2006.

[14] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas, "Patching processor design errors with programmable hardware," *IEEE Micro*, vol. 27, no. 1, pp. 12–25, 2007.

[15] I. Wagner and V. Bertacco, "Engineering trust with semantic guardians," in *DATE*, 2007.

[16] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, 2001, pp. 3–14.

[18] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. D. Bosschere, "Statistical simulation: Adding efficiency to the computer designer's toolbox," *IEEE Micro*, vol. 23, pp. 26–38, 2003.