Low Maintenance Verification

Valeria Bertacco
Advanced Computer Architecture Lab
The University of Michigan – Ann Arbor, MI
Email: valeria@umich.edu

***Abstract*— Guaranteeing the functional correctness of a digital integrated circuit is an unfulfilled challenge for even the most straightforward practical designs. The reality for even the simplest of today's designs is that they are released with latent bugs. Some of these bugs are innocuous, others are simply annoying, but still others are potentially dangerous to the users of the system or they might compromise its security, or adversely affect its performance. While the past few decades have witnessed significant efforts to improve verification methodology for hardware systems, these efforts have been far outstripped by the massive complexity of modern digital designs, to the point that today only a vanishingly small fraction of a design's possible behavior is verified to be correct before the manufactured system is delivered to the end user. Looking forward, the rise of highly complex chip-multiprocessors and heterogeneous systems-on-a-chip is deemed to only exacerbate this problem to new heights. It seems a reasonable prediction to say that, without out-of-the-box new thinking in verification research, it is only a matter of time before an escaped design error becomes the cause of a broad impact incident, perhaps worse than the Intel FDIV bug disaster of the mid 1990's. In this paper we present ideas to attack this problem from two opposite directions: on one hand, we present solutions which boost the coverage of design-time verification technologies, through closed-loop, hybrid semi-formal verification technologies. On the other hand, we provide techniques to craft novel hardware mechanisms which permit to deftly circumvent escaped bugs or even correct them after the device has been deployed in the field. Our ultimate vision for these technologies is to make hardware as malleable as software.**

I. Introduction

The semiconductor industry has been aware of its exposure to potential bugs escaping verification since its early days, however, the situation has become critical in the past few years due to the vast complexity of digital systems designed today, and to the first few alarming disasters caused by escaped bugs. The growing complexity of digital systems designs has lead to product releases for which an always smaller fraction of system's configurations has been verified. The news of escaped bugs in designs deployed in large-markets and/or safety critical domains is alarming because of their safety and cost (due to replacements, lawsuits, etc.) implications. Within the design teams developing integrated components, there is a sharp awareness of the lack of verification tools and methodologies that could provide strong guarantees of functional correctness. Hence, verification engineers are forced to hunt and peck for bugs in the vast design state space. Design houses respond by dedicating more and more resources and effort to verification: today the majority of semiconductor companies spend two thirds of their design resources on verification tasks. At the end, the result is the release of a system that is "as correct as possible", in the hope that no major flaw had escaped unnoticed to silicon.

Given this landscape, we believe that a necessary step to attain any complete verification solution is the acceptance of the fact that the problem's complexity is beyond what can be achieved by simply maximizing verification effort. If we can move beyond this deemed-to-fail approach, than we can begin to address the issue from alternative perspectives, to find more successful solutions. Within this assumption, we believe that a winning technology attacks the problem by budgeting up front what fraction of the design's functionalities is realistic to verify, and leaves all the remaining functionalities to be checked only after production, and only if and when the system is required to perform them.

A. *The challenges of verification*

Overlooked design flaws have been the cause of major replacement costs even in recent times, for instance, in 2001 the incident of the Cisco firewalls which would repeatedly hang. The lack of good verification methodology in complex systems has caused flaws to go undetected and, in some cases has costed the loss of lives, as in the Therac-25 radiation therapy machine [1], or the China Airlines B1816 crash due to inadequate system software. Today, a large part of the integrated circuit industry is moving towards system architectures which employ parallel chip-multiprocessor systems, with highly interconnected communication protocols, and distributed computation strategies, frequently embedded within a complex diverse system, including peripherals, software, analog components, etc. All these trends bring the unresolved challenge of "correctness guarantee" to new unreachable heights. In line with this perception, the influential, industry-generated International Technology Roadmap for Semiconductors has determined that, in order to maintain the current growth trends in the electronics and semiconductor industry [2]. verification barriers are the most important challenge to overcome. In contrast, decades of research effort devoted to guarantee the correctness of integrated circuits designs have failed in achieving any guarantee that a complex design is operating correctly in all of its execution scenarios. Figure 1 shows a schematic representation of how we see verification today: the pear shaped area represents the complete design state space, that is, the complete set of configurations or execution states a digital system can attain. In traditional verification methodologies, a small fraction of this area is verified before production, as it is represented by the "design-time verification" area on the right. However, the verified portion of the design is not quantified, particularly in methodologies where the metrics are based on bug rates [3], [4]. This aspect is depicted through a blurry transition to the unverified portion on the left, and by the question-marked line. The area on the left includes all those configurations which have not been verified, and which can hide potentially catastrophic bugs.

II. Why Mainstream Solutions Fall Short

The overarching goal of functional verification is to ensure that the system being designed fully adheres to its specification, that is, the planned intended behavior of the integrated circuit. Unfortunately, the growing complexity of the functional verification challenge has run rampant for the past decade, defeating the enormous effort put forth by armies of verification engineers and academic research efforts.

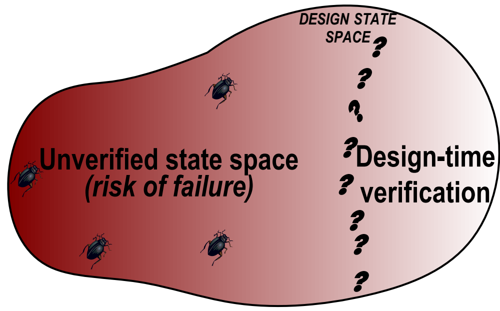

Fig. 1. **A design space view of traditional verification**. The pear-shaped area represents the complete set of distinct configurations which are possible for a design (design state space). The line traced with question marks indicates the boundary between configurations which have been verified before production (on the right), versus unverified execution scenarios (on the left). Note that this boundary is not actually known to the design team. Moreover, any execution pattern which visits any of the states in the unverified portion of the design has the potential to encroach an escaped bug (represented by the bug graphic).

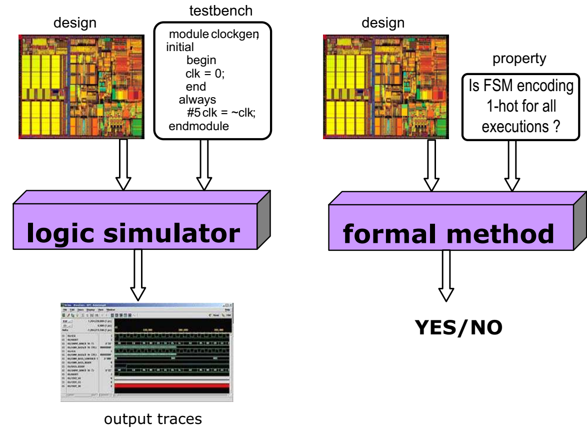

Fig. 2. **Mainstream philosophies in verification**. The figure contrasts two families of solutions in design-time verification. On one hand, logic simulation, a.k.a. validation, allows to check specific execution scenarios of a digital system by simulating them on a logic-level description of the design. The results of simulation are streams of logic 0s and 1s which must be visually inspected to evaluate if they match expected behavior. On the other hand, formal methods take a more comprehensive approach by proving that a specific aspect, or property, of the design holds under all circumstances. One catch of formal verification tools is the limited design size that they can handle, often requiring users to work with simplified design models or to carve small blocks from a full system.

The current practice in industry is a partial verification process, developed under tight time and cost constraints, where only a few aspects, or a few configurations of a design are addressed, checked, and verified; aspects which cumulatively constitute only a vanishingly small fraction of the entire universe of possible design behaviors. The underlying reasons for this unmanageable complexity lie in the inability of verification to keep up with the high-productivity trends of modern design solutions, targeting complex integrated system-on-a-chip (SoC) designs and parallel chip-multiprocessor systems, and paired with highly interconnected communication protocols implementing distributed computation strategies. Because of the extremely high costs of manufacturing flawed microchips, industrial developments devote a significant fraction of engineering resources to achieve functional correctness. Multiple sources report that approximately 70% of the design development effort is spent in verification [**?**], making it the main bottleneck to achieve short time-to-markets. Part of this high resource allocation is due to verification methodologies that are still largely ad-hoc and experimental, and to a lack of robust solutions targeting specific verification tasks.

A. Simulation-based methodologies

Current solutions adopted in industry are mostly simulation-based techniques, where the simulation is run on a behavioral or register-transfer level (RTL) description of the design. In large market microchip designs, verification is also carried on through hardware prototypes, which are also used as software development platforms [**?**]. This situation is in contrast with the critical importance of ensuring the functional correctness of a system: the short time-to-market forces the large fraction of a system's functionality to be left unexplored. Escaped bugs may trigger a costly design re-spin when found shortly after the first tape-out, or may be distributed in the final product, with implications ranging from mild to tragic.

In classic simulation-based validation the functional model of a design is simulated with meaningful input stimuli and the output is checked for the expected behavior. The simulation involves applying patterns of test data at the inputs of a logic-gate model, then using the simulation software to compute the simulated values at the outputs, and finally checking the correctness of the values obtained. Simulation-based techniques are favored in industry settings because of their linear scalability with design complexity. Moreover, to support the verification effort, simulation technology has been paired with methodologies which include support for fast testbench development (input stimuli and output checkers) using specialized languages [5], [6], [7], and coverage analysis tools that provide a quantitative evaluation of the progress of verification [3]. Commonly coverage is measured in terms of line of RTL description triggered by simulation, and functionality aspects that the team plans to verify. However, all the complex functionality aspects that the design team does not devise to check and their interdependencies are left aside by these coverage metrics: It should not come as a surprise that it is common for precisely these aspects to hide bugs.

The main limitation of simulation-based approaches is that they can explore only one design configuration per simulation cycle, which, for designs as complex as those developed today, translates practically to exploring only a very small fraction of the design's state space. As a conservative estimate, it takes approximately one year of simulation to reproduce just several seconds of operation of a complex microprocessor. One way to address this limitation is by exploiting emulation technology [8], [9] and hardware prototyping, however, both these techniques are very expensive and are commonly limited to systems with a large market distribution.

B. Formal and semi-formal methods

In contrast with the mainstream industry approach, academic settings have focused for in large part on developing formal verification solutions. In this context, the verification problem becomes one of guaranteeing, through some formal procedure, that a particular property of the design is always satisfied under every possible execution sequence. Formal verification techniques pertain to three main families: model checking, theorem proving

and symbolic simulation methods. The flow of symbolic simulation [10], [11] is very similar to logic simulation, with the main difference that the former uses Boolean functions where the latter relies simply on the logic 0 and 1 values. The intent is to boost the reach of simulation, by capturing in parallel all the possible behavior of the system. Because of added complexity of representing Boolean functions, much research has been poured in developing approximation and simplification techniques [12], [13], [14], [15]. In model checking [16], [17], a specific property of the design is proven by certifying that it holds for all of the legal configurations of a system. In most cases, this requires to first determine all such legal configurations, a task of exponential complexity on the number of storage bits of the design. Theorem provers [18], [19], [20] proceed by deriving a formal mathematical proof that a property holds using a simplified description of the system. The limitation here rests in the difficulty of crafting a simple enough description which captures all the relevant aspects. Most recently, a few attempts have been made in the direction of hybrid solutions. In [21], a new approach is proposed where a logic simulator alternate control over a design under verification with a range of formal engines to attain very high coverage on a selected set of state signals of a design. Simultaneously, approximated model checking techniques are used to rule out systems configurations that cannot be reached. The technology presented in [21] was the initial work for a major effort by Synopsys to create a product that addresses the issue of functional verification for large scale designs. Alternative approaches has been also suggested by Intel researchers in [22], [23], proposing alternative collaborative interactions between formal engines with the goal to prove properties efficiently. Additional recent work has explored techniques in the space of hybrid verification, where the integration between formal and simulation methods is much deeper, providing more scalable and efficient solutions [24], [25].

While formal and semi-formal techniques have the power to prove that a specific property holds for every possible execution of a system, they are not sufficient to provide any complete guarantee of correctness. A major limitation is the limited scalability of these solutions: only small components ($\sim$100K logic gates) of a system can be addressed by the most powerful semi-formal technique. Typical modern digital components use millions of gates, which explains why the industry has yet to fully embrace these solutions.

C. Hardware mechanisms

Very few solutions have been proposed which attempt to integrated verification into the final hardware product. One direction of research has attempted to synthesize and include design properties (or checkers) into the hardware itself. After system production, if an unverified property is violated during runtime operation, the user is notified and/or correcting mechanisms are put in place for recovery. Solutions in this space have been proposed by Nacif *et al.* in [26], where the authors use a hardware assertion library for runtime detection of bugs in on-chip networks. In [27] checkers are developed on a design-by-design basis. In some cases the synthesis of checker properties has the objective to detect bugs during emulation (and not in the final product). This is the case for work included in FoCs, a software tool by IBM [28], and for the approach presented in [29]. Another research direction was explored initially by DIVA [30]. DIVA proposes to insert an on-line checker component into the retirement stage of a microprocessor pipeline . The component is a simple complete microprocessor which fully validates all computation. If a functional error is detected, the execution results are provided by the simple microprocessor, guaranteeing correct functionality at a performance cost. The DIVA approach practically implements two versions of the same system next to each other, a simple low-performance one which is completely verified, and a complex high-performance version which may include escaped bugs. The solutions outlined in this paper share the same goal with DIVA, however, we derive techniques which do not require to implement a second version of the system, but rely on a simplified execution mode of the same high-performance implementation.

III. Rethinking the Verification Process

The ultimate goal of verification is to provide solutions that enable designers to deliver digital systems for which they can guarantee correct operation under all possible execution scenarios. We envision a viable route to this goal through multiple angles of attack, which together trespass the traditional boundary of forcing all bugs to be detected and corrected at design time, and allow for post-manufacturing verification solutions. We accomplish this through three novel research vectors, outlined in Figure 3:

1) **Re-energizing Traditional Solutions**. Existing verification methods have proven to be either too myopic and provide very low coverage, as in the case of simulation, or too weak to tackle complex systems, as in the case of formal methods. Successful solutions require taking advantage of both families of approaches in hybrid collaborative solutions. We have begun to develop powerful hybrid verification solutions which 1) take advantage of the strength of simulation techniques in attacking large, complex designs (hence the logic simulator is the underlying engine); 2) complement them with formal methods to provide direction in the state exploration; and 3) rely on feedback from the design itself to adjust the search, so that the system can autonomously navigate to critical portions of a design without requiring direct guidance from an engineer. Section IV presents two recent solutions in this area.
2) **Quantifying the Limits of Traditional Verification**. As mentioned earlier, most verification in industry is carried on with a "as long as time permits" approach, where the typical company devotes more than 70% of their resources for a finite amount of time (from project inception to product delivery) to verification, with the hope that no critical bug will escape this effort. A typical technique to track verification progress is that of counting how many bugs are discovered per week, and try to stabilize this metric as close to zero as possible. It is obvious that such approaches do not recognize the fact that much of the design is left to verify, nor how many bugs are left behind. We are developing techniques that permit verification engineers to gauge which fraction of the functionality remains unverified (through an abstract estimate of state space and verified state) and, based on dynamic and statistical estimates, evaluate how frequently the system, once deployed, will fall into one of those unverified configurations.
3) **Post-Silicon Hardware Mechanisms**. The third angle of attack closes the verification gap by addressing and correcting any escaped bugs, after silicon has been produced and delivered for use by a customer. We envision that many novel solutions are possible within this space which

can practically convert an escaped bug into a semantically equivalent operation that is simpler (and likely slower), but guaranteed correct. These solutions range from preventing a system to cross the boundary line of verified states, to providing hardware patching capabilities for fixing escaped bugs once a design has been released, as well as intermediate solutions between these two extremes. Section VI presents a few solutions along these lines, ranging from forcing the system to a verified "safe-mode" of execution, to isolating bugs whenever users expose them while adapting the system so that they can circumvent future occurrences. The most proper solution to adopt will depend on the context of where a system is deployed. For instance, medical, safety critical, mass transportation, or financial applications demand stronger guarantees of correctness. However, applications such as video/audio streaming, or gaming would probably be more focused on optimized performance, even at a correctness cost.

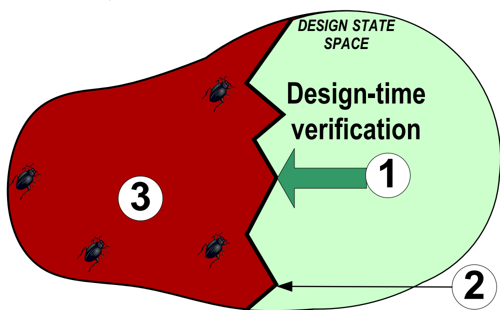

Fig. 3. **New directions in hardware verification**. To enable designers to deliver **perfectly correct** systems, that is, systems which are guaranteed to work correctly under every execution scenario, we propose three lines of attack: 1) pushing the limit of coverage that design-time verification can reach; 2) quantifying and delimiting the portion of a design which is left unverified; and 3) providing post-silicon mechanisms which can correct on the fly a system's execution, if and when it lands on an unverified state. The three circled numbered in the figure indicate how these three steps will change the verification scenario compared to the diagram of Figure 1.

Figure 3 shows the three angles of attack that we have explored. Through tight integration of simulation and formal and modeling techniques, the boundary of the verified fraction of a design can be pushed further, conquering a larger portion of the design. By quantifying the portion of the design's state space that *is* verified, it becomes possible to detect when a design transitions into unverified operation modes. Finally, post-silicon mechanisms are dedicated to manage situations in which an escaped bug is found after the design is deployed into the field.

Below we outline solutions that we recently developed in the three research vectors outlined, namely, in pushing the envelope of what design-time verification can accomplish, in quantifying the fraction of the design verified at run time, and in providing effective techniques to correct and possibly detect escaped bugs after customer release.

IV. Boosting the Power of Design-time Verification

Design-time verification is practically a race against time to explore and check as many diverse configurations as possible in the system being developed. Based on many frequent interactions with several hardware design teams, we strongly believe that powerful design-time verification solutions should be:

- *hybrid*, that is, leverage the horsepower of logic-simulation, but provide direction in the exploration through formal analysis and algorithms;
- *closed-loop*, using feedback from the design or the simulation data itself to tune and adjust the direction of the search so to diversify the exploration, or reach a specific verification target. This aspect is key in alleviating the effort that design teams pour into developing testbenches, and allowing them to refocus their goals on the high-level aspects of the task;
- *flexible* to adapt to a wide range of design types, and to produce high coverage results. Many semi-formal solutions today can only be applied effectively to some types of designs (for instance, blocks which do not include any multiplier operator), while they may be unable to tackle other designs, even very small ones.

A. Guided-search engines

A verification solution we have developed along these lines is the GUIDO guided-simulation tool [31]. GUIDO can be applied to any digital system design. It considers the design under verification (DUV) together with an associated coverage target expressed as a checker, then extracts automatically a small abstract model of the design and performs a reachability analysis on it. The results of this analysis are used to estimate the distance of any given design state to the target state (any of the states that fail the checker). Finally, this estimate is used in simulation to guide a logic simulator incrementally closer to the target, until it is reached. GUIDO has shown experimentally to reach targets much faster, and generate bug traces much more compact than the top commercially available semi-formal verification tools. Consequently, given the same time budget for verification, designers can expect to verify a larger and more relevant fraction of the design state space by applying GUIDO and targeting coverage goals of interest, compared to constrained random simulation .

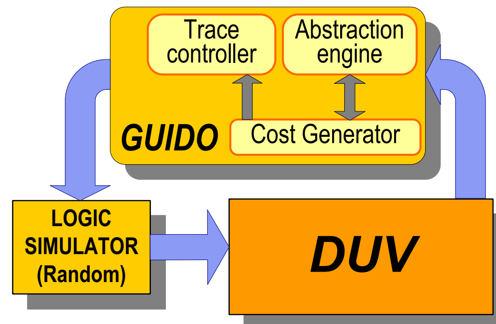

Fig. 4. **Closed-loop hybrid verification with GUIDO**. GUIDO implements a closed-feedback loop between a simulation engine and the current design's state, by first performing a static analysis on a design abstraction, and then using the results of the analysis to monitor the progress of the simulator towards a pre-set verification goal.

The development of integrated hybrid solutions in design-time verification has started only a few years ago, but it is a fertile ground in the research community because of its promising results. StressTest [25], [32] is another solution recently proposed which creates "smart" simulation streams by relying on a Markov model representing an abstraction of the system's communication protocol. We evaluated this solution both in the context of networking traffic and for microprocessor verification, where the protocol is the instruction set architecture itself. StressTest

has proven to reach coverage goals more than one order of magnitude faster than traditional constraint-random simulation methods. Other recent solutions proposed in this space explore an alternative mix of formal engines, and/or other metric for evaluating the quality of the search (and specifically, the distance from a verification goal) [24], [33].

B. Extracting protocols from simulation

An alternative approach to limit the effort spent in verification is that of creating tools which can automatically discern the design's functionality and distinguish between wrong and correct behavior. In the previous solutions, the designers themselves had to specify the verification objective: it is also possible to pursue techniques where the system autonomously identifies potential bugs by comparison with the mainstream activity observed in simulation. Based on this philosophy we developed a solution, called Inferno [34], which focuses on transaction-based systems. Inferno monitors critical control signals of a design in operation, and infers all the distinct transactions which can be observed. The transactions are presented to the user through compact graphs representing the various types of communication activity, and they are subject to inspection for approval. Figure 5 shows a new closed-loop verification flow which deploys Inferno. The core advantages of this approach is that there is no need for a designer to know and to reproduce the correct communication protocol for a system. In contrast, the user can just apply Inferno to the signals of interest, and check afterward that the activity observed is as expected. Moreover, the abstract transaction representation technique developed in Inferno facilitates the spotting behavioral anomalies, which frequently correspond to hiding spots of complex bugs.

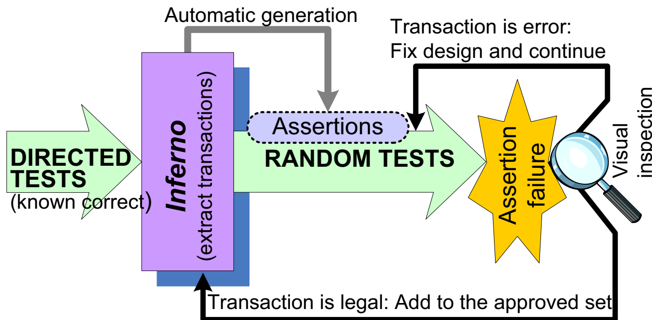

Fig. 5. **Closed-loop transaction-based verification with Inferno**. The diagram shows a closed-loop verification framework using Inferno: a first set of known-correct testbenches are run on the design, while Inferno extracts all the transactions observed. These transaction are added to an "approved list" and assertions are generated to detect any unseen transaction. The verification proceeds through random simulation, each time a new transaction is noticed, it is submitted for approval to the design team and one of two options can occur: either it is deemed correct and added to the approved list, or it is a buggy transaction and the design is corrected before verification continues.

Inferno fits well into the closed-loop methodology discussed above as it allows the design team to verify a system by pairing the tool with any constrained random test generator. The basic testbench suite (usually a reference of what is the correct behavior of the system) can be monitored by Inferno to infer what are the first set of accepted transactions. The process continues by monitoring the system running in random simulation mode. This flow is outline in Figure 5. In addition to presenting transactions to the user, Inferno can also generate corresponding assertions which fire whenever an unseen transaction is observed. These assertions are put in place in future random simulations. If a new transaction is detected that violates one of these assertions, it is an indication that the design has trodden onto unverified ground, thus the execution scenario is submitted for approval to the design team. At that point, it can deemed correct and added to the set of approved transactions as a new assertion, or determined illegal, in which case a bug is corrected in the design. This process can be iterated until no new transactions are found for a determined period of time.

V. Quantifying the Unverified Extent

A key requirement to enable complete verification solutions is to evaluate where in the design state space lies the boundary between verified and unverified state. Available methodologies have focused on assessing the progress of verification in terms of fraction of goals that have been attained, where the goals have been set by the design team upfront. In contrast, we are concerned with representing compactly which configurations are left unverified, in order to focus our post-silicon solutions on this portion. We address this task by using a small set of critical control bits of the design, which span the "critical state space". During design-time verification, we observe all the value combinations spanned by these critical bits and we base our assessment on this small state space, instead of the complete and too-complex design state space. For instance, in a recent effort [35], we selected 26 critical signals for an entire Alpha processor pipeline, signals which well represent the relevant control part of the system. Because of the small size of this set, it is possible to evaluate exactly which and how large of a fraction of the critical state space has been verified, and indirectly, which was potentially left to be explored. Note that by selecting just a small set of bits we are *de facto* considering a projection of the system over a few state bits. Hence, it is critical that the projection of choice provides a highly accurate, yet simplified model of the design.

An additional benefit of this techniques for quantifying verified design space is that it is possible to estimate how often the system will visit to an unverified configuration once deployed in the field. By comparing a baseline simulation and a long-range one, it becomes possible to estimate the fraction of time that a partially verified design will operate within unverified configurations. The frequency of this occurrence, paired with our post-silicon solutions, can be used to estimate the performance impacts of supporting operational modes in which a design is *not* allowed to enter unverified configurations. We explore techniques along these lines in the following section.

VI. Post-Silicon Verification Solutions

In this section we present some of the post-silicon hardware mechanisms that we have recently explored. All solutions in this group require two components:

1) a *detection mechanism* which identify when the system has reached or is about to reach an unverified configuration;
2) a *correction solution* which can intervene to correct the erroneous operation before it becomes finalized, which would corrupt the system's computation.

In the following we discuss two post-silicon correction mechanisms. The first, called *field-repairable control logic*, allows designers to fix a small number of hardware bugs *in the field* once the buggy hardware has been released to the customer.

The approach utilizes a hardware patch table that can selectively change control signal responses based on entries in the programmable patch table. The table can be updated any time a new escaped bug is reported, enabling each manufactured system to adapt to discovered escaped bugs. The technique in essence allows hardware designers to enjoy the same luxury software developers rely upon, in that it can be patched after it is released. The solution surgically eliminates the escaped bugs one by one, whenever they are identified. The surgical elimination approach is particularly suitable for systems that strive to work as correctly as possible, but for which the guarantee of correct operation is not a top or immediate requirement. It also provides the benefit of being suitable for improving the success rates of current designs and verification practices, without necessitating any major methodology changes.

A second technique we advocate along these lines uses a hardware matcher, called *semantic guardian*, to detect when the system transitions to an unverified configuration. Although unverified states do not necessarily correspond to escaped bugs, they are configurations at risk, which must be avoided for applications in which correctness is a paramount concern. Users that share this concern can optionally turn on a *trusted computing* mode which prevents the component from operating in states outside of the fully verified design space. When the system attempts to enter an unverified state, it is instead forced into a fully verified degraded mode operation, which allows the program to continue in absolutely correct operation, with a slight decrease in performance.

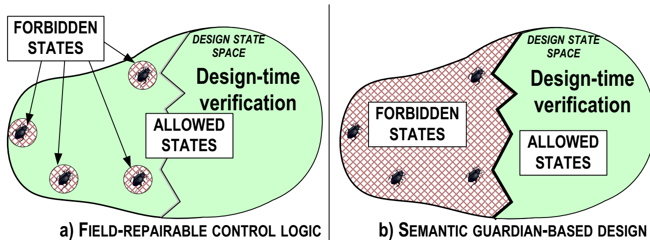

Fig. 6. **Schematic of two post-silicon detection mechanisms**. The diagrams represent two possible solutions in the space of post-silicon hardware verification. Field-repairable control logic lets the system execute normally, until when a configuration is reported to the design house as flawed. At this point the design team can issue a "patch", which can be downloaded to all the systems deployed in the field, and which prevents the hardware to ever enter the erroneous configuration again. The second technique, instead, avoids any potentially erroneous state *a priori*, by forcing the system to only operate within the boundary of the states which have been verified at design-time.

A. Field-repairable control logic

In surveying microprocessor's errata documents [36], [37], we found that the majority of reported bugs were due to incorrectly designed control logic. Hence, in [35] we developed an approach which employs a control state matching mechanism, called a *programmable matcher*, or soft matcher, which identifies when the processor has entered a control state associated with a design bug. Once a match occurs, the pipeline is flushed and forced into a *degraded performance* mode of operation to execute the next instruction. After the instruction is complete, normal pipelined execution is restored. For a microprocessor pipeline, the degraded performance mode is a barebone re-configuration of the system where only one instruction is executed at a time (that is, only one instruction is in the pipeline at any point in time), and most "performance-boosting" hardware is shut down (that is, no branch prediction unit, no instruction pre-fetching, no data forwarding, etc.) This degraded mode is sufficiently simplified that it can be formally verified for the full instruction set architecture. Since only one instruction is in flight at a time, no interdependencies between instructions can occur. In other words, we could formally verify that given any allowed initial configuration and executing any instruction, the system would transition to a correct final state. Hence, correct forward progress is *guaranteed* in degraded mode, although at a fairly low performance. The programmable matcher we developed is a content-addressable memory (CAM) which stores the "buggy" patterns and flags when one of those patterns occurs during normal operation. A schematic of a pipeline enhanced with a soft matcher is shown in Figure 7. We envision a use methodology whereby the soft matcher is, by default, empty. If an error is found after design tape-out, the set of states associated with the bug are encoded into a state matching pattern. This pattern is then distributed to customers, and loaded into the soft matcher. When a buggy state is detected by the matcher, the processor switches to the degraded mode which has been completely formally verified. Thus, we can rely on this mode for the processor to complete the next instruction correctly, and thereby guaranteeing forward progress. After the instruction is committed, the normal execution mode is resumed. Note that the cost of relying on the degraded mode for progress is a decrease in performance. In fact, on a n stage pipeline, each degraded-mode instruction would take n cycles to complete vs. the 1 cycle (approximately) of the fully capable pipeline. Fortunately, in practice, bugs that escape the verification effort will naturally lie in configurations that are rarely exercised, thus, one can expect the patched processor to be nearly as fast as the original design. In our experimental evaluation, we found that the added hardware for matcher (a 4- or 8-entries CAM) and recovery logic, ranges from 2% to 0.02% of the processor area. In addition, the performance impact of this solution is below 10% as long as erroneous configurations do not occur more frequently than 1 out of 100 instructions. Note that the solution proposed encompasses and moves beyond previous instruction and microcode patching techniques [38], because it can effectively address design errors that relate to a particular instruction, combination of instructions and, in addition, errors that are not associated with any specific instruction, for instance a non-maskable interrupt.

B. Semantic guardians to protect execution

Field-repairable control logic enables chip manufacturers to fix bugs found after the release of the product. However, this still does not guarantee that the system is performing exclusively correct operations. In contexts where reliable computation is a must, we need to explore alternative run-time correct solutions.

In an approach based on semantic guardians [39], we substitute the soft matcher with a *hard matcher*, a logic component which encodes **all the critical states** which have not been verified at design-time. When any of these combinations is a match with the ongoing computation, we resort to the use of a degraded operation mode to progress program state forward. In this sense the system is walking the line between verified and un-verified states, and the hardware matcher never allows the processor to fall over into the unverified space. This "trusted computing" mode of operation can be dynamically disabled or enabled, thus providing

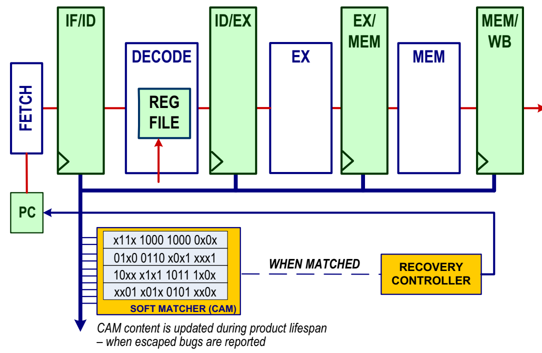

Fig. 7. **Field-repairable control logic for a microprocessor pipeline**. The pipeline has been augmented with a programmable (soft) matcher and a recovery controller block. The soft matcher is empty at manufacturing time. When an escape bug is reported to the design house, a bug pattern is developed and uploaded on all systems in the field. The soft matcher compares the "critical state" of the processor with the stored patterns, and when a buggy state is detected, the processor switches to a degraded mode with formally verified execution semantics, guaranteeing forward progress at a performance cost. After the instruction is committed, the normal, high-performance mode of execution is resumed.

a mechanism for the user to operate the system in a "correctness guaranteed" mode, with a potential performance cost, or in a "at risk" mode, at top performance. For instance, the "trusted mode" can be selected when the system is running safety critical applications, and deselected during more error-tolerant tasks, such as video and audio streaming.

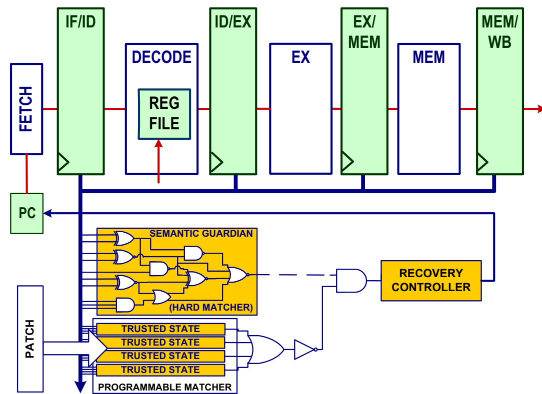

Fig. 8. **A semantic guardian-enhanced design**. This solution complements the pipeline with a hard matcher and a recovery logic block. The hard matcher is a logic block matching all the states of the design that are left to be verified at the time of release to the market. If the processor enters one of these configurations, the hard matcher detects it, and flags the recovery controller, which proceeds to flush the pipeline and switches the system to its degraded mode of operation.

The hard matcher is a direct result of the verification process for the full design. Although the full system is too complex to be handled by formal verification tools, it still can be partially verified with simulation-based and hybrid verification tools. The effort spent verifying the normal mode of operation can be measured in the number of control states that are observed and verified at design-time. We developed a system which automatically infers the states that were not observed or observed insufficiently often, and synthesizes a hardware component computing the logical-OR of those critical states. This component is then manufactured together with the system. Once in operation, whenever the processor enters one of the states encoded in the matcher, the hard matcher flags an alert to the recovery controller, which proceeds to flush the pipeline and switches to a fully-verified degraded mode (see also Figure 8). Evidently, the higher the simulation effort, the more states are verified, which usually translates to a smaller matcher and correspondingly less performance degradation (because the system enters degraded mode less frequently). This technique enables designers to easily compare and trade-off performance, area and verification time, resulting in a low-cost, trustworthy hardware component with optimal performance. During the development of a system, the verification flow would attack the full design with traditional methodologies, while the components which are part of the degraded mode of operation will have to be formally verified to guarantee correctness of this mode. A schematic of this development flow, including the automatic synthesis of automatic guardians, is shown in Figure 9.

Finally, note that the two solutions outlined above are complementary and can be deployed simultaneously on the same system, the former to correct escaped bugs which become known to the chip manufacturer, the latter to provide trusted computing capabilities.

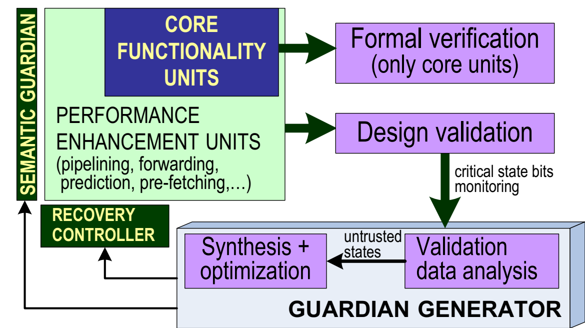

Fig. 9. **Verification flow for post-silicon correction mechanisms**. The diagram shows modified verification and design flows to include semantic guardians: core functionality blocks undergo formal verification, while the full design is addressed by validation techniques. In addition, the validation activity is monitored and analyzed, and the data is used to automatically synthesize a semantic guardian for the design.

C. Designing correction in place

In the examples described in Sections VI-A and VI-B the correction mechanism was a formally-verified, barebone version of the pipeline design, obtained by executing only one instruction at a time, and by eliminating all those blocks which improve the performance of a processor. In general, each family of designs will require the verification team to devise its specific *degraded mode*. The criteria for the degraded mode is to be sufficiently simplified to make the system amenable to formal verification. For instance, a network switch routing packets with high throughput, using advanced techniques such as directory-based routing could deploy a degraded which eliminates the complexity associated with smart routing choices. A simple degraded mode would outstrip the system from all the components dedicated to learn routes, and limit the operation to synchronous global broadcast where incoming packets are processed one at a time, and all packets are broadcasted to all outgoing ports.

VII. Conclusions

In this paper we outlined a novel approach to digital design verification which partitions the effort into design-time and post-manufacturing verification. Design-time verification focuses on the most frequently occurring executions scenarios, while rare, hard-to-reach corner-case situations are potentially left unchecked. For these latter configurations post-manufacturing mechanisms can intervene to still guarantee correct system execution. We outlined each of the components of this verification approach and, within each portion, we have provided example solutions which have been recently developed by our research group. It is our hope that these novel techniques will make a solid case that by extending verification to encompass both design-time and run-time, it becomes possible to significantly enhance the scope of verification, while maintaining the tight resources and time budgets necessary to meet market demands.

References

[1] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[2] International Technology Roadmap for Semiconductors. `http://www.itrs.net/reports.html`, 2005 edition.

[3] Michael Kantrowitz and Lisa M. Noack. I'm done simulating; now what? verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor. In *DAC, Proc. Design Automation Conference*, pages 325–330, June 1996.

[4] Andrew Piziali. *Functional Verification: Coverage Measurement and Analysis*. Kluwer Academic Publishers, 2004.

[5] Faisal I. Haque, Khizar A. Khan, and Jonathan Michelson. *The Art of Verification with Vera*. Verification Central, 2001.

[6] Yoav Hollander, Matthew Morley, and Amos Noy. The *e* language: A fresh separation of concerns. In *Technology of Object-Oriented Languages and Systems*, volume TOOLS-38, pages 41–50, March 2001.

[7] Michael Behm, John Ludden, Yossi Lichtenstein, Michal Rimon, and Michael Vinov. Industrial experience with test generation languages for processor verification. In *DAC, Proc. Design Automation Conference*, pages 36–40, June 2004.

[8] Srihari Cadambi, Chandra S. Mulpuri, and Pranav N. Ashar. A fast, inexpensive and scalable hardware acceleration technique for functional simulation. In *DAC, Proc. Design Automation Conference*, pages 570–575, June 2002.

[9] Chen Chang, Kimmo Kuusilinna, Brian Richards, and Robert W. Brodersen. Implementation of BEE: A real-time large-scale hardware emulation engine. In *FPGA, Proc. International Symposium on Field Programmable Gate Arrays*, pages 91–99, 2003.

[10] Valeria Bertacco. *Scalable Hardware Verification With Symbolic Simulation*. Springer, 2005.

[11] Douglas Perry and Harry Foster. *Formal Verification*. Mc-Graw Hill Professional, 2005.

[12] Valeria Bertacco and Kunle Olukotun. Efficient state representation for symbolic simulation. In *DAC, Proc. Design Automation Conference*, pages 99–104, June 2002.

[13] Chris Wilson and David L. Dill. Reliable verification using symbolic simulation with scalar values. In *DAC, Proc. Design Automation Conference*, pages 124–129, June 2000.

[14] In-Ho Moon, Hee Hwan Kwak, James Kukula, Thomas Shiple, and Carl Pixley. Simplifying circuits for formal verification using parametric representation. In *FMCAD, Proc. International Conference on Formal Methods in Computer-Aided Design*, pages 52–69, 2002.

[15] Valeria Bertacco, Maurizio Damiani, and Stefano Quer. Cycle-based symbolic simulation of gate-level synchronous circuits. In *DAC, Proc. Design Automation Conference*, pages 391–396, June 1999.

[16] Edmund Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000.

[17] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: a survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, 1999.

[18] David A. Cyrluk and Mandayam K. Srivas. Theorem proving: Not an esoteric diversion, but the unifying framework for industrial verification. In *ICCD, Proc. International Conference on Computer Design*, pages 538–544, October 1995.

[19] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *CADE, Int. Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

[20] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Lifted-FL: A pragmatic implementation of combined model checking and theorem proving. In *Theorem Proving in Higher-Order Logics*. Springer-Verlag, September 1999.

[21] Pei-Hsin Ho, Thomas Shiple, Kevin Harer, James Kukula, Robert Damiano, Valeria Bertacco, Jerry Taylor, and Jiang Long. Smart simulation using collaborative formal and simulation engines. In *ICCAD, Proc. International Conference on Computer Aided Design*, pages 120–126, November 2000.

[22] Scott Hazelhurst, Osnat Weissberg, Gila Kamhi, and Limor Fix. A hybrid verification approach: Getting deep into the design. In *DAC, Proc. Design Automation Conference*, pages 111–116, June 2002.

[23] Carl-Johan H. Seger, Robert B. Jones, John W. O'Leary, Thomas F. Melham, Mark Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, 2005.

[24] Per Bjesse and James Kukula. Using counter example guided abstraction refinement to find complex bugs. In *DATE, Design, Automation and Test in Europe Conference*, pages 156–161, March 2004.

[25] Ilya Wagner, Valeria Bertacco, and Todd Austin. StressTest: An automatic approach to test generation via activity monitors. In *DAC, Proc. Design Automation Conference*, pages 783–788, June 2005.

[26] J.A. Nacif, F.M. dePaula, H. Foster, C.N. Coelho Jr., F. Cortez Sica, D.C. da Silva Jr., and A.O. Fernandes. An assertion library for on-chip white-box verification at run-time. In *LATW, Proceedings of the IEEE Latin-American Test Workshop*, February 2003.

[27] Ali Bayazit and Sharad Malik. Complementary use of runtime validation and model checking. In *ICCAD, Proc. International Conference on Computer Aided Design*, pages 1052–1059, 2005.

[28] Yael Abarbanel, Ilan Beer, Leonid Glushovsky, Sharon Keidar, and Yaron Wolfsthal. FoCs: Automatic generation of simulation checkers from formal specifications. In *CAV, Proc. International Conference on Computer Aided Verification*, pages 538–542, 2000.

[29] Marc Boulé and Zeljko Zilic. Incorporating efficient assertion checkers into hardware emulation. In *ICCD, Proc. International Conference on Computer Design*, pages 221–228, 2005.

[30] Chris Weaver and Todd Austin. A fault tolerant approach to microprocessor design. *Int. Conf. on Dependable Systems and Networks (DSN-2001)*, 2001.

[31] Smitha Shyam and Valeria Bertacco. Distance-guided hybrid verification with GUIDO. In *DATE, Design, Automation and Test in Europe Conference*, pages 1211–1216, March 2006.

[32] Ilya Wagner, Valeria Bertacco, and Todd Austin. Depth-driven verification of simultaneous interfaces. In *ASPDAC, Proc. Asia South Pacific Design Automation Conference*, pages 442–447, January 2006.

[33] Stefan Edelkamp and Alberto Lluch-Lafuente. Abstraction in directed model checking. In *Workshop on Connecting Planning Theory with Practice*, pages 7–13, 2004.

[34] Beth Isaksen and Valeria Bertacco. Verification through the principle of least astonishment. In *ICCAD, Proc. International Conference on Computer Aided Design*, November.

[35] Ilya Wagner, Valeria Bertacco, and Todd Austin. Shielding against design flaws with field repairable control logic. In *DAC, Proc. Design Automation Conference*, July 2006.

[36] Dusko Koncaliev. Bugs in the Intel microprocessors. `http://www.cs.earlham.edu/~dusko/cs63/`.

[37] *IBM PowerPC 750GX and 750GL RISC Microprocessor Errata Notice*, July 2005.

[38] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, April 2004.

[39] Ilya Wagner and Valeria Bertacco. Engineering trust with semantic guardians. In *DATE, Design, Automation and Test in Europe Conference*, April 2007.