BackSpace: Formal Analysis for Post-Silicon Debug

Flavio M. De Paula[1], Marcel Gort[2], Alan J. Hu[1], Steven J. E. Wilton[2], Jin Yang[3]
[1] Dept. of Computer Science, University of British Columbia, {depaulfm, ajh}@cs.ubc.ca
[2] Dept. of Electrical and Computer Engineering, University of British Columbia, {mgort, stevew}@ece.ubc.ca
[3] Intel Corporation, jin.yang@intel.com

***Abstract*—Post-silicon debug is the problem of determining what's wrong when the fabricated chip of a new design behaves incorrectly. This problem now consumes over half of the overall verification effort on large designs, and the problem is growing worse. We introduce a new paradigm for using formal analysis, augmented with some on-chip hardware support, to automatically compute error traces that lead to an observed buggy state, thereby greatly simplifying the post-silicon debug problem. Our preliminary simulation experiments demonstrate the potential of our approach: we can "backspace" hundreds of cycles from randomly selected states of some sample designs. Our preliminary architectural studies propose some possible implementations and show that the on-chip overhead can be reasonable. We conclude by surveying future research directions.**

I. Introduction

Post-silicon debug (AKA post-silicon validation, silicon debug, silicon validation) is the problem of determining what's wrong when the fabricated chip of a new design behaves incorrectly. The focus of post-silicon debug is *design errors*, whereas VLSI test focuses on random *defects* introduced during the manufacturing process of each chip. Post-silicon debug currently consumes more than half of the total verification schedule on typical large designs, and the problem is growing worse.[1] Even worse, the schedule variability is greatest post-silicon, creating unacceptable uncertainty in time-to-market.

The problem is easy to understand from a typical scenario:

> *After a long development process, first silicon (or second, or third...) comes back from the fab. Yield is mediocre, but several "good" die pass the manufacturing tests and make it to the bring-up lab. Power-on reset and basic functionality tests work. But 30 seconds into booting the OS, the chip crashes. Worse yet, every single "good die" crashes in the same way.*
>
> *Scanning out the state of the crashed chips show an internal data structure (e.g., a routing table or a coherence directory) has been corrupted in an inexplicable manner. The logic analyzer dumps from just before the crash show routine traffic to/from memory and I/O devices. The stimulus to generate the crash (i.e., booting the OS for 30 seconds) is far too deep to replay in simulation or by single-stepping the die, and trying to hit the crash state with full-chip formal verification exceeds the capacity of the formal tools. Increasingly many engineers, from many different teams (pre-silicon verification, design, test, architects, OS, drivers,...) get sucked into the debugging effort. Yet debugging proceeds painfully slowly, because everyone is flying blind, trying to guess what happened.*

Obviously, the fundamental problem is observability — to debug a chip, we must know what is happening. With increasing technology scaling, higher speeds, and greater integration, there is simply no longer enough I/O to be able to debug effectively: pin counts are limited; I/O pads are too costly (area and power) and slow; and single-stepping and scanning out the state every clock cycle is far too slow.

This paper introduces a novel paradigm for using techniques from formal verification, augmented with some on-chip support logic, to greatly enhance observability of the execution leading up to the observed buggy behavior. Specifically, we allow the chip to run at full speed, yet provide the ability to "backspace" hundreds, perhaps thousands, of cycles from a crash state or a programmable breakpoint, to derive an error trace that led to the crash, which can then be replayed in a simulator or waveform viewer to help understand the bug. In a nutshell, the basic framework consists of adding circuitry to monitor architecturally key operating points and record a small signature of the monitored information. During debug, this signature, as well as the crash state, can be scanned out using existing on-chip test access mechanisms. By itself, the signature provides insufficient information (lossy) and would be meaningless to a human, but we can combine this information with formal analysis of what is possible in the design (by computing pre-images) to determine the unique, or nearly unique, predecessor states that led to the crash state. Such an approach can backspace only a limited number of cycles, so we also add circuitry to the chip to allow programmable breakpoints. By setting a predecessor state as a breakpoint, we can re-run the failing test on-chip (e.g., booting the OS), but crash some number of cycles earlier, then scan out the new crash state and signature, and iterate the entire process to compute arbitrarily long back traces.

This work can revolutionize post-silicon debug, but it is still in its infancy and not yet practical. This paper presents the basic framework and preliminary results demonstrating both potential and limitations. We conclude by surveying future research directions.

[1] Andreas Kuehlmann, personal communication, September 21, 2006. References [1], [6] also report a large fraction of total verification cost occurring post-silicon.

A. Related Work

There is no closely related work to what is presented in this paper. Under the broad rubric of using formal methods to aid post-silicon debug, however, there are a few related papers. Ahlschlager and Wilkins [2] describe their experience directly using a model checker for post-silicon debug: they write a formal property to describe the observed buggy behavior and ask the model checker to generate a trace. Such an approach is ideal when the model checker can verify the entire chip or when the debug engineer correctly maps the observed chip-level buggy behavior onto a block-level formal specification, neither of which can be counted on. Safarpour et al. [17] address the problem as rectification: they assume that design errors/fixes can be modeled by injecting corrected values at various points in the circuit and use a MAX-SAT solver to find a smallest set of locations to inject values to correct the behavior on a specific set of test cases. This work is simultaneously much more limited than ours (simplistic model of design errors, limited test cases, scalability issues), yet also much more ambitious (attempting to correct the design automatically). Several groups have proposed leveraging the intellectual investment into formal *specifications* during post-silicon debug, by compiling the formal specifications into on-chip monitor circuits (e.g., [13], [8], [4]). Such an approach provides enhanced observability into what went wrong if an on-chip monitor catches an assertion violation, but doesn't leverage formal *verification* technology to aid debugging.

Of course, industry has been doing (and struggling with) post-silicon debugging without the benefit of formal methods since the dawn of VLSI. Very little has been published about current techniques and methodologies, although anecdotally, we know that practices vary enormously between companies, e.g., a small company with a high-margin, low-volume product can't afford specialized high-end test equipment but can tolerate more die area overhead for debugging support, whereas a large company with a high-volume product might spend enormous up-front NRE costs to shave on-chip overheads to the bare minimum. On one extreme, some companies are not even using existing on-chip test access mechanisms to aid debugging, relying instead on manually modifying bring-up programs and observing the results; on the other extreme, some companies have purpose-built bring-up hardware and sophisticated logic analyzers that allow intercepting, recording, and replaying all traffic between the chip and its environment (e.g., [18]). Despite this variation, our work and existing methods share similarities, based on the common underlying constraints:

- It is possible to get a fairly complete snapshot of the internal state of a chip, albeit very slowly. The most basic mechanism is the scan chains [19] present on almost all chips to allow efficient manufacturing test. A chip with scan can be configured into test mode, in which most or all of the latches on the design are connected together into a small number of very long shift registers. At any point in time, the chip can be stopped, and the values of the latches can be shifted in or out. With hold-scan latches, it is even possible to scan out a snapshot of the state of the chip at one point in time, while allowing the chip to continue to execute during the scan-out process, at the cost of substantial on-chip overhead [10]. In our work, we assume the existence of full scan on the chip, but do not require hold-scan latches.
- A very limited history of some number of signals can be recorded at full speed on-chip, and this history can be read out (very slowly), e.g., via the scan chains. These mechanisms ("on-chip logic analyzers", "trace buffers") typically consist of a flexible mechanism to access desired signals on-chip and some way to store the signals for later read out (e.g., RAM or specialized cells [3]). The main trade-off is that considerable die area overhead must be used for each cycle's worth of history for each signal monitored, severely limiting how much history can be stored.
  The signature generation in our method can be viewed as a generalized, optimized trace buffer, using formal verification techniques to enable reconstruction of a fully detailed trace from compressed signatures. We rely on prior work on access mechanisms to observe on-chip signals. In this paper, we consider one efficient, flexible, and reconfigurable architecture that provides this access [15]. Abramovici et al. [1] propose a different reconfigurable architecture, also with the goal of providing efficient signal access for debugging. Also, many companies have their own, in-house access mechanisms to help in debugging (e.g., [7]), but published details are sparse. Nevertheless, the sort of on-chip access we assume are clearly very realistic. For signature storage, our area models assume SRAM (Section IV).
- In many debugging scenarios, standard off-chip logic analyzers are helpful. As mentioned above, in the extreme case, specialized hardware and a great deal of high-end test equipment can be used to record and replay *all* I/O signals between the chip and its environment, allowing deterministic repeatability of the stimulus that triggered a bug on-chip. The main drawbacks are the high cost of the test equipment, the extremely limited ratio of observable I/O pins and pads versus the internal state of the chip, the inability to debug internal IP blocks, and the ability to debug the chip only in the specialized bring-up system — sometimes, a bug will manifest itself only in some OEM system but not in the original bring-up system. Note also that even in the extreme case, the logic analyzer traces alone do not allow us to reproduce the bug in a logic simulator, since we don't know the internal state of the chip to start the simulation. Our method does not require off-chip logic analyzer traces and hence does not suffer the drawbacks. However, if such traces are available, we could use them to reduce our on-chip overhead.
- The only mechanism fast enough to run the actual bring-up of the chip, in an actual system, on actual data, is the chip itself. There is no way to simulate an extremely deep

trace (e.g., even 30 seconds of real execution time), and no way (without formal verification techniques, as in this paper) to go backwards from a state of interest on the actual silicon to determine what happened beforehand.

The goal, of course, is to determine what happened *before* the crash occurs, but we do not know *a priori* when that will be. Accordingly, current methods typically attempt to find some way to scan out a complete state snapshot some number of cycles before a crash state happens, and then use that state along with the recorded I/O behavior to simulate the chip from shortly before the crash up until the crash, e.g.,:

- Periodic Sampling. The chip can be stopped at regular intervals to scan out a snapshot of the internal state and then allowed to continue execution. With hold-scan latches, the chip need not even be stopped. When the crash occurs, the most recent snapshot and the logic analyzer traces of the I/O can be used to recreate the bug in simulation. An obvious problem is that stopping the chip disturbs system-level timing interactions, potentially changing the execution and hiding the bug. Furthermore, because the scan-out process is so slow, the interval between snapshots must be long, meaning that the most recent snapshot might be too far in the past.
- Cycle Counters. If the chip's behavior and the system environment are both deterministic (or the I/O behavior has been recorded and can be replayed), then a simple cycle counter can be used. When a crash occurs, we record when it happened, and then we re-run the system, but scan out a snapshot when the cycle counter is a convenient number of cycles prior to the crash. Non-determinism is the main difficulty for this approach, obviously in the system environment (e.g., when a disk or network interrupt occurs), but also on-chip (e.g., multiple independent clock domains, arbitration, PLL lock times).
- For a system implemented on FPGAs, the problem of system-level non-determinism can be eliminated by duplicating the entire design [11]. One copy of the design runs in the system as usual; the second copy has all of its inputs delayed in a FIFO. When the first copy hits the bug, it triggers trace recording on the second copy. For a non-FPGA design, it's obviously impractical to duplicate the design on-chip, but if two identical dies are available, both of which are fully deterministic in an identical manner, one could imagine building a specialized bring-up board that implements this solution.[2] Aside from the determinism restriction, the obvious problems with this approach are the cost of the purpose-built bring-up system and the likelihood that bug behavior will be different between that system and real OEM systems.

In contrast to the above methods, our method eliminates the challenge of trying to determine when to take a snapshot of the internal state just before the crash is about to happen; instead, the formal analysis allows us to compute the predecessor state directly.

[2]This idea was suggested to us by Igor Markov, June 30, 2008.

Our work is focused on design errors that escape pre-silicon verification and end up on the actual chip. A complementary post-silicon debug problem, for which there is also very little research, is to help identify and repair physical, electrical, and timing errors on-chip. Chang et al. [6] propose a methodological framework for this class of problems. Park and Mitra [14] also focus on electrical bugs and propose a processor-specific technique using summaries of in-flight instructions in the processor. A post-analysis over these summaries helps locate the source of the bug. The main similarity between their work and ours is the collection of information from the design via summaries (signatures), and then using that information in the post-analysis. Their approach demonstrates very low overhead, but is narrowly specific; our approach is not processor-specific, but currently has excessive overhead.

One insight behind our approach is that the actual silicon is so fast that it can be used to re-run some input stimuli *ab initio* repeatedly, to compute the state of the chip at different points in time. This insight echoes a similar computation used for "hardware modeling", in which an actual chip is used in a special modeling system to emulate its own behavior during system-level logic simulation [9].

II. Basic Framework

A. Intuition and Assumptions

The basic problem is that we have observed the chip in some buggy state, and we have no idea how that could have happened. The goal is to explain the inexplicable buggy state, by creating a "backspace" capability — iteratively computing predecessor states in an execution that leads to the bug. The resulting trace can be viewed like a simulation waveform, except it shows what actually happened just before the bug/crash on the real silicon.

We assume that the problem occurs at a depth and complexity not trivially solved by existing methods. For example, if the full chip can be handled in a model checker, we can simply ask the model checker to generate a trace to the observed buggy state. This solution is not realistic for complex designs, because of the capacity limits of model checkers. Alternatively, if the bug occurs extremely shallowly during bring-up, we could run the bring-up tests on the simulator, or via single-stepping the chip (scanning in a state, pulsing the clock, scanning out the state). Such an approach is also not realistic: the roughly billion-to-one speedup of the actual silicon versus full-chip simulation means that one second of runtime on-chip equals decades of run time in simulation, and within seconds of first power-on, the silicon has executed more cycles than months of simulation on vast server farms. Trying to reproduce the bug *ab initio* in simulation is clearly not feasible. Similarly, trying to monitor externally the full execution trace of the chip running full-speed is electrically impossible.

We start with a few simplifying assumptions:

- It must be possible to recover the state of the chip when an error has occurred. For example, this could be done with the chip in test mode, via the scan chain.

- The key assumption is that since we are focusing on *design errors*, we will assume that manufacturing testing has eliminated manufacturing defects. Therefore, we assume that the silicon implements the RTL (or gate-level or layout or any other model of the design that can be analyzed via formal tools).
- The bring-up tests can be run repeatedly and the bug being targeted will be at least somewhat repeatable (one out of every n tries, for a reasonably small value of n).

Later, we discuss how the framework changes when we relax these assumptions, e.g., partial scan, mixtures of design errors and defects, and non-deterministic errors due to marginal circuits, process variability, etc. Even without the relaxations, though, the problem is real, and a solution would be valuable.

Our framework consists of adding some debug support to the chip: a signature that saves some history information but otherwise has no functional effect on the chip's behavior, and a programmable breakpoint mechanism that allows us to "crash" the chip when it reaches a specified state. Given these, the approach repeats the following steps

1) Run the chip until it crashes or exhibits the bug. This could be an actual crash or a programmed breakpoint.
2) Scan out the full crash state, including the signature.
3) Using formal analysis of the corresponding RTL (or other model), compute the predecessor of the crash state. The signature must provide enough information to allow only one (or a few) possible predecessor state.
4) Set the computed predecessor as the new breakpoint.

until we have computed enough of a history trace to debug the design (or Step 3 fails). Each iteration of the loop is like hitting "backspace" on the design – we go back one cycle. The approach exploits the capabilities of different analyses: formal analysis is very slow with limited capacity, but can go forward or backwards equally well; simulation is too slow to run in a real system with actual software, but the visibility of a simulation trace is user-friendly and well-accepted for design understanding and debugging; the actual silicon runs full-speed, rapidly hitting bugs that may have escaped pre-silicon validation, but offers very poor visibility and no way to backspace to see how the chip arrived in some state.

B. Theory

We model the system in the usual manner as a finite state machine M, with S latches and I inputs, initial states Init $\subseteq 2^S$, and transition relation $\delta \subseteq 2^S \times 2^I \times 2^S$. We allow the transition relation to be non-deterministic, so the formalism can handle randomness in the bring-up tests as well as transient errors, race conditions, etc.

Given a state machine M, we can build an augmented state machine M' which has the same behavior as M (when projected onto the original S latches), but has an additional T latches of signature. The T signature latches are not allowed to affect the behavior of M, so the transition relation of M' is a pair of relations: the original $\delta \subseteq 2^S \times 2^I \times 2^S$ as well as a $\delta' \subseteq 2^S \times 2^T \times 2^I \times 2^T$. In other words, the next signature can depend on the signature as well as the state and inputs, but the next state cannot depend on the signature.

Definition 1 (Backspaceable State): A state (s', t') of augmented state machine M' is backspaceable if its pre-image projected onto 2^S is unique, i.e.,:[3]

$$\exists! s \exists t, i[((s, i, s') \in \delta) \wedge ((s, t, i, t') \in \delta')]$$

In general, a signature might contain enough information to allow computing multiple cycles of unique pre-images. In that case, the theory changes in the obvious manner to backspace multiple cycles from each run of the chip. Currently, we envision such multi-cycle signatures to be simply a series of single-cycle signatures, stored via pipelining in the signature collection circuitry or in efficient SRAM structures (Sect. IV). To simplify the exposition in this paper, we describe backspacing only a single cycle at a time.

Definition 2 (Backspaceable Machine): An augmented state machine M' is backspaceable iff all reachable states are backspaceable. A state machine M is backspaceable iff it can be augmented into a state machine M' for which all reachable states are backspaceable.

The algorithm to compute the states prior to the crash state starts from a given crash state and then iteratively computes its predecessors, going backwards in time:

Algorithm 1 (Crash State History Computation): Given a state (s_0, t_0) of a backspaceable augmented state machine M', compute a finite sequence of states $(s_0, t_0), \ldots, (s_k, t_k)$ as follows:

1) *Since M' is backspaceable, let s_{i+1} be the unique pre-image state (in the state bits S) of (s_i, t_i).*
2) *Run M' (possibly repeatedly) until it reaches a state (s_{i+1}, x). Define $t_{i+1} = x$.*

Theorem 1 (Correctness of Trace Computation): If started at a reachable state (s_0, t_0), the sequence of states $s_k, \ldots, s_0$ computed by Algorithm 1 is the suffix of a valid execution of M.

Proof Sketch: For any state (s_i, t_i) in the sequence, we must prove that two properties hold for s_{i+1}: first, that s_{i+1} is a predecessor of s_i in M, and second, that s_{i+1} is a reachable state in M. By the definition of pre-image, there exists x such that (s_{i+1}, x) is a predecessor of (s_i, t_i). By the definition of the augmented state machine, s_i cannot depend on x, so therefore s_{i+1} must be a predecessor of s_i. That establishes the first property. For the second property, because M' is backspaceable, s_{i+1} is the same for all predecessor states of (s_i, t_i). Therefore, any execution σ' of M' that reached (s_i, t_i) must have gone through a state (s_{i+1}, x) for some x. Because the signatures cannot affect the state latches, the projection of σ' onto the state latches is a valid execution of M and goes through the state s_{i+1}. Hence, s_{i+1} is a reachable state of M. ∎

If the state machine as well as the environment/testbench are deterministic, then Algorithm 1 not only gives *a* valid execution, but *the* execution of M that led to the crash state,

[3]The notation $\exists!$ denotes "There exists a unique...."

because the execution σ' will be the same when computing each state in the sequence. In the presence of randomness, the algorithm still works, but with a constant factor expected slow-down: if the bug appears in 1 out of n runs, then we expect to need to repeat n times step 2 per iteration of Algorithm 1. Similarly, if the pre-image is not unique, but there are k states in the pre-image, we can try step 2 for each of the k possible pre-image state, resulting in a constant factor $k/2$ expected slow-down. There is no combinatorial blow-up, as there would be for backward reachability. Definitions 1 and 2 generalize naturally to "k-backspaceable" for a given bound k.

An important caveat is that, under the assumption of true non-determinism, termination of the algorithm is not guaranteed. For example, it is conceivable that setting the programmable breakpoint hardware to target state s_a will result in an execution σ_a that reaches (s_a, t_a) from a state (s_b, x), but if we reprogram the breakpoint hardware to target state s_b, subtle electrical effects might cause the chip to follow a different execution σ_b that reaches (s_b, t_b) from some state (s_a, y). In this case, Algorithm 1 will still compute a valid execution of M, as indicated by the theorem, but this execution won't make any progress toward the initial states. Fortunately, if non-determinism in the model is really randomness, with non-zero probability of choosing all legal transitions, then we can prove termination with probability 1:

Theorem 2 (Probabilistic Termination of Algorithm 1): If we terminate Algorithm 1 when the computed sequence reaches an initial state of M, and if the executions σ' of M' are chosen randomly such that all valid transitions have non-zero probability, then termination occurs with probability 1.

Proof Sketch: At all times, the state being considered in the algorithm is reachable. Hence, there is an execution σ' of length l that reaches the target state, and this execution occurs with non-zero probability. If this execution gets chosen repeatedly l times (an event that also occurs with non-zero probability), then the algorithm will terminate after l iterations. Otherwise, the algorithm continues from another state. Hence, the algorithm is a random walk backwards on the state space, where the initial states are sink states and all states are reachable from the initial states. With probability 1, the random walk must terminate in a sink state.

∎

In practice, we do not expect these issues of non-determinism, randomness and termination to be a problem. The main difficulty with randomness will be the number of trials required to hit a breakpoint state when the chip runs — if the probability is low, many runs will be needed for each backspace step.

Algorithm 1 performs repeated pre-image computation, which can be expensive. We encountered problems in our initial experiments with BDDs and All-SAT. A key insight greatly improved efficiency:

> We need compute only whether a state has a unique pre-image state or not (or whether it has more than k pre-image states for k-backspaceability).

This insight means we can use a state-of-the-art, off-the-shelf SAT solver to search for *any* pre-image state. If one is found, then we add just one blocking clause to eliminate that solution and run the SAT solver a second time, to see if the first solution was unique.

To scale to full-size industrial designs, it is likely that not all state bits will be scannable and breakpointable. In that case, the above theory generalizes exactly as abstraction applies to model checking. Algorithm 1 will compute an abstract execution, over the subset of the state bits that are scannable/breakpointable. This execution will be the suffix of a valid execution of the conservative abstraction of M onto the subset of state bits. Hence, there will be the same risk of false abstract execution paths, exactly as in abstract model checking. Standard concretization heuristics from abstract model checking may help. It is also likely that the abstract trace may be suggestive enough of what actually happened on-chip to help the debug engineer understand what caused the crash/error, even if the trace can't be or hasn't been concretized. For example, an abstract trace might indicate that the error occurs when a certain type of transaction encounters a specific exception condition at the exact cycle that another event occurs, but without indicating the specific data values in the trace.

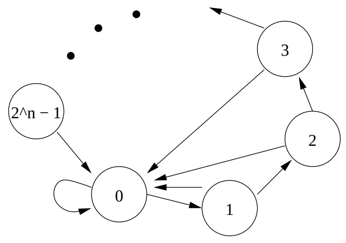


Fig. 1. State Machine Requiring $|S|$ Extra State Bits to Be Backspaceable

C. Backspace Coverage

Is it always possible to augment any state machine to make it backspaceable? The answer is yes. We can simply make $|T| = |S|$ and set up δ' to copy the values in the latches of S to the latches of T. In other words, we can always backspace to a unique predecessor state because we have stored that state.

Is it possible to do better, to make any state machine backspaceable using fewer than $|S|$ additional latches? Unfortunately, in the worst case, the answer is no. For a simple example, consider the state machine in Figure 1. This example is a simple n-bit counter, with a single input. If the input is low, the counter transitions to the 0 state; otherwise, it counts up. Almost all states have only a single predecessor, making them backspaceable with no additional signature. However, the 0 state has every state as a predecessor. To make the machine backspaceable, we must add the full n additional state bits, just to handle one particularly bad state. We call such states "convergence states" because many incoming transitions converge on them.

Figure 1 shows that in the worst case, we can do no better than by storing a copy of all the state bits. However, it also suggests that we might be able to do much better for *most* states. Is it good enough if we make most states backspaceable?

Definition 3 (Backspace Coverage): Given state machine M augmented into M', the backspace coverage of M' for M is the fraction of the reachable states of M' that are backspaceable.

Can we get good backspace coverage with much fewer than $|S|$ bits in the signature, or more to the point, can we backspace a long enough trace to be useful before hitting a convergence state? The convergence states are likely to be states that are easy to get to and easy to understand (like reset or idle states); backspacing to a convergence state may be sufficient for debugging purposes. In the next section, we explore whether we can make this theory work on some sample open-source designs.

III. Preliminary Experiments

A. Experimental Setup

We present our experiments on two small processors/microcontrollers. The research is still highly exploratory, so we have chosen to focus on a small number of design examples: we often needed detailed understanding of the designs to generate good research hypotheses. The designs also had to be small enough so that repeated experiments were feasible, and so that the supporting algorithms and tools that are not germane to this research did not need to be highly optimized. On the other hand, the designs must be realistic, to capture characteristics of real designs.

The two processors are a 68HC05 and an 8051. These are both open-source designs from opencores.org that are rebuilds from datasheets of the respective classic 8-bit microcontrollers from Motorola and Intel. The 68HC05 is smaller, with 109 latches. The 8051 implementation has 702 latches. In both cases, we developed a simulation testbench based on the testbenches supplied with the designs: the 68HC05 ran real LED and LCD controller applications, and the 8051 ran some small software routines.

For our experiments, we treated the design running on a commercial logic simulator as if it were the actual chip running on silicon. We simulated the designs for an arbitrary number of cycles and randomly selected 10 states each to serve as “crashed” states for our analysis. In addition, our testbench also recorded the immediate predecessor state before the crash state (which wouldn’t be possible in silicon); this predecessor state is the correct answer that our backspace analysis is trying to recover. Thus, we have 10 pairs of states per design to serve as testcases.

B. Signature Functions

As a first step, we needed to find some plausible signature functions. We concentrated on the 68HC05 and tried a variety of approaches. Fig. 2 summarizes the results of our experiments.

Our first idea was to try a quick experimental upper bound on the size of the signature. We created the signature as a randomly selected subset of all state bits. Unfortunately, this

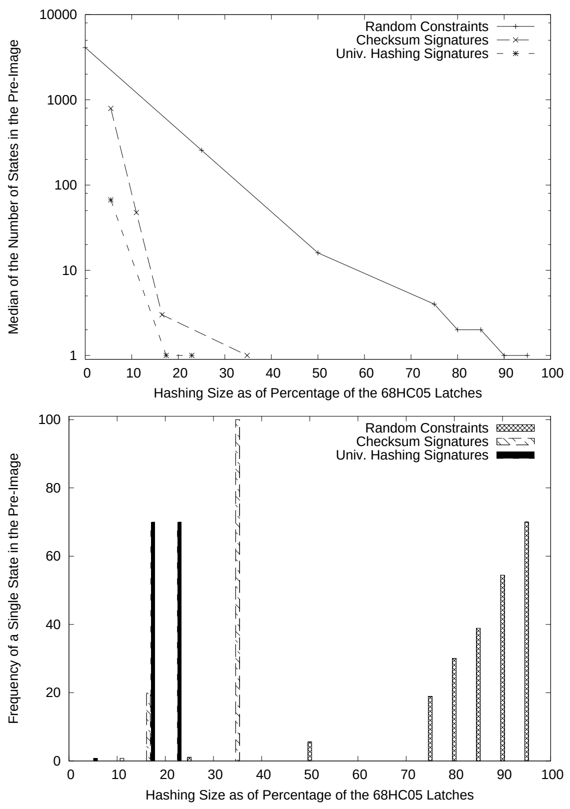


Fig. 2. Results for Compressed Signatures Based on Architectural Insight.

approach fared poorly: 90% of the state bits were needed in the signature before the median size of the pre-image was 1, and even with 95% of the state bits in the signature, only 7 out of 10 of our test states had unique pre-image states.

In real life, the designers understand their design, and architectural insight might allow selecting a particularly good subset of the state bits to use as a signature. Based on a careful study of the 68HC05, we identified 38 latches (35% of the design) to use as the signature. This approach was very successful, yielding unique pre-image states for all 10 test cases.

Spurred by that success, we tried some simple checksums on those 38 bits, reducing the number of bits used to 6, then 12, and then 19. These results were not very successful at getting unique pre-image states, but the plot suggested that better compression would be promising.

Accordingly, we tried a perfect hash function — universal hashing [5] (essentially the same as X-Compact [12], which is easier to implement on-chip) to compress the 38 bits to 6, 19, and 25 bits. These results demonstrated the promise of universal hashing.

In all of these experiments, computations were fast, and the SAT solver had no problem computing pre-image states.

C. BackSpacing

With some promising ideas for signature functions, we proceeded to the real test: can we backspace for hundreds of cycles from the random crash states? We created an automatic

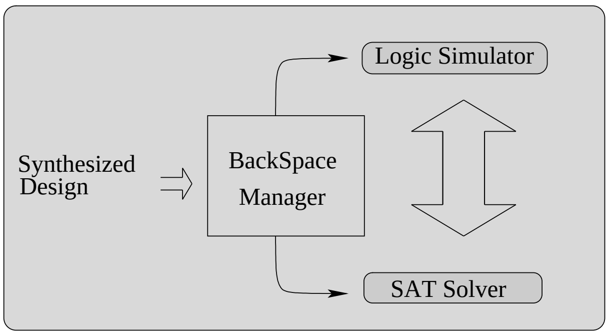


Fig. 3. BackSpace Framework

framework to experiment and explore the BackSpace paradigm (Fig. 3). The components of the framework are the BackSpace Manager, a commercial logic simulator, and a SAT solver. The input to the framework is a synthesized design (gate-level netlist). The logic simulator plays the role of the silicon: we use it to run our testbench, exactly as the real silicon would run bring-up tests. The SAT solver is the engine to compute the required pre-image states. The core of the framework is the BackSpace Manager.

The BackSpace Manager coordinates the logic simulation and the SAT solving tasks by dispatching each task and processing their intermediate results (shown as the double-headed arrow in Fig. 3). For logic simulation, the BackSpace Manager automatically generates a testbench instance based on the synthesized design, dispatches the logic simulation, awaits its termination, and captures the crash state and signature. For SAT solving, given the crash state and the signature, the BackSpace Manager generates a SAT problem instance. When the SAT solver finds a solution, it means there is one (more) state in the pre-image of the crash state. The BackSpace Manager generates a blocking clause based on this solution and asks the SAT solver for another solution. If another solution is found, this process repeats until there are no more solutions. At that point, a single state or a set of states is available as candidate states prior to the crash state. The task now is to find which candidate state is the actual one. The BackSpace Manager dispatches logic simulation, setting a candidate state as a simulation breakpoint. If simulation reaches the breakpoint, it means we have a new crash state and a signature. This process continues until we have "backspaced" some pre-determined number of cycles. If simulation does not reach the breakpoint, it means we need to try another candidate. For our logic simulator, we used Synopsys VCS (version 7.2), and for our SAT solver, we used Minisat (version 2.0). Due to VCS licensing issues and GCC compatibility problems, we had to run these tools on different machines: logic simulation was run on a Sun Fire V880 server (UltraSPARC III at 900Mhz); SAT solving was run on an Intel Xeon at 3.00GHz.

We ran experiments for both the 68HC05 and the 8051. For each, the goal was to see how far we could backspace before the pre-image set got too large or the computation blew-up.

Crash State	# of Cycles BackSpaced	Max States in PreImg	Sim Time	Sat Time	Manager Time
s1	54	4096	63.44	0.93	204.42
s2	1	65536	1.45	86.61	4.38
s3	37	4096	62.65	0.71	139.67
s4	7	4096	37.76	0.52	27.11
s5	53	4096	116.16	0.92	200.34
s6	500	1	1261.48	3.24	1884.31
s7	500	1	2384.29	3.15	1890.91
s8	500	1	4575.41	3.01	1893.89
s9	2	4096	22.93	0.51	22.93
s10	9	65536	2424.55	91.18	34.86

"Sim Time" is the time spent in the logic simulator. This time would be replaced by time running on the actual silicon. "Sat Time" is the time spent in the SAT solver. "Manager Time" is the time spent by the BackSpace Manager to supervise the framework and connect the various tools. Our BackSpace Manager implementation is very preliminary and can be optimized extensively.

TABLE I
68HC05 w/ 38-bit Subset Hand-Chosen Signature

Crash State	# of Cycles BackSpaced	Max States in PreImg	Sim Time	Sat Time	Manager Time
s1	500	2	1097.25	185.57	8524.15
s2	500	2	2011.04	187.21	8397.09
s3	500	2	2737.15	171.57	8335.45
s4	500	2	2988.38	242.89	8477.88
s5	500	2	3358.40	216.81	8398.14
s6	500	1	3176.94	31.89	8175.62
s7	500	1	6247.61	31.42	8280.93
s8	500	1	12207.49	38.58	8297.21
s9	500	2	15280.79	42.31	8173.19
s10	500	1	34084.53	36.63	8125.62

TABLE II
68HC05 w/ 38-bit Universal Hashing Signature

For the 68HC05, we reused the signature consisting of a hand-selected subset of 38 of the 109 total state bits, chosen based on our insight into the design. We also tried a 38-bit hash generated via universal hashing over the 109 state bits. For the 8051, we hand-selected a 281 bit subset of the 702 total state bits to be the "human architectural insight" signature. We also tried to use a 281 bit universal hash of the 702 state bits.

In these experiments, we used the k-backspaceable computation (i.e., pre-image sets are allowed to have up to k states), with k set to 300 states. To keep our experiments manageable, we also set an upper limit of 500 cycles of backspacing per test crash state.

Tables I and II show the results for the 68HC05. With the hand-chosen subset of bits, we hit our 500 cycle limit on 3 of the 10 test crash states. But on 4 of the 10, we cannot backspace more than a handful of cycles. With a universal hash of the same size, all 10 test crash states can be backspaced to our limit, and all of the pre-images are very small. In Section IV, we will see that a hand-chosen subset of bits is a very low-overhead signature, whereas universal hashing all bits of a large design appears to be prohibitively expensive. We can see the trade-off between quality and cost.

Table III presents the results for the 8051 using the hand-chosen subset of the state bits as the signature. The results are

Crash State	# of Cycles BackSpaced	Max States in PreImg	Sim Time	Sat Time	Manager Time
t1	205	512	2841.07	4.58	6048.46
t2	500	256	21759.74	9.70	14720.71
t3	500	257	8326.66	10.84	14746.10
t4	500	257	10342.40	10.77	14772.03
t5	500	256	11587.21	11.26	14742.81
t6	500	256	11581.93	8.72	14735.07
t7	500	255	25767.40	8.54	14742.60
t8	500	256	13581.20	11.57	14759.73
t9	500	257	22493.04	10.62	14735.48
t10	500	257	24793.42	10.81	14759.77

TABLE III
8051 W/ 281-BIT SUBSET HAND-CHOSEN SIGNATURE

Crash State	# of Cycles BackSpaced	Max States in PreImg	Sim Time	Sat Time	Manager Time
t1	500	8	138616.15	1379.21	55389.29
t2	500	4	497905.92	1350.15	55104.32
t3	500	4	191655.42	1378.15	55462.20
t4	500	4	183283.27	1383.10	55642.82
t5	500	8	431057.79	1377.87	55039.00
t6	500	4	151950.65	1399.62	55601.11
t7	500	4	506787.53	1388.94	55639.58
t8	500	8	506229.79	1368.52	55512.44
t9	500	4	488157.90	1379.14	55049.31
t10	500	4	534870.14	1378.37	55448.52

TABLE IV
8051 W/ 281-BIT UNIVERSAL HASHING SIGNATURE

excellent: we can backspace up to our 500 cycle limit in 9 out of the 10 test crash states. Initially, we were unable to complete results for the 8051 with a 281-bit universal hash. The SAT solver blew up (1 hour timeout and 1GB memory limit) on all 10 test cases. The universal hash function is essentially a matrix-multiplication over GF(2), with a random matrix, so it's not surprising that large instances are challenging for current SAT solvers. However, with some more thought and experimentation, we were successful with this experiment as well. The key is that any full-rank matrix provides correct universal hashing, but a sparse matrix will be easier for the SAT solver, and also reduce area overhead, too. If we generate the random hash matrix with a 0.985 probability of each entry being 0, we can backspace up to our set limit for all 10 test crash states. Furthermore, the number of states in the pre-image is 2 orders of magnitude smaller for all crash states. Table IV gives these results.

To summarize, the overall framework works. We can compute hundreds of cycles of error trace backwards from a crash state. Additional research will need to explore what sorts of signature functions work well, and at what hardware cost.

IV. ARCHITECTURE AND ON-CHIP OVERHEAD

This section describes the circuitry that must be added to the integrated circuit to implement the framework. It also estimates the area overhead of this circuitry.

A. Support Circuitry

Figure 4 shows how a Circuit Under Debug (CUD) can be instrumented with a Breakpoint Circuit, a Signature Creation circuit, and a Signature Collection circuit.

During debugging, as the circuit operates, the Signature Creation circuit monitors N_{mon} of the N_{state} state bits in the CUD. In general, $N_{mon} \leq N_{state}$, but in this analysis, we assume that all state bits are collected and used to form a signature, so $N_{mon} = N_{state}$. Each cycle, the Signature Creation circuit uses these state bits to construct a signature of size S_{width}; the construction of the signature will be described below. The signature is then stored in a memory within the Signature Collection circuit. The memory is arranged as a FIFO buffer composed of an SRAM block and read/write circuitry. The depth of this FIFO buffer dictates how many consecutive states can be stored. Meanwhile, the Breakpoint Circuit also monitors the state bits. When the state bits match a predetermined state (the target state), a signal is sent to stop the collection of signatures. The signature(s) stored in the buffer can then be read out and processed as described in Section II.

The heart of the architecture is the Signature Collection circuit. The simplest way to construct a signature is to simply use the state bits directly. If $N_{mon} = S_{width}$, then the history of all flip flops is stored, and the circuit becomes trivially backspaceable. If $N_{mon} > S_{width}$, then missing bits must be reconstructed using off-chip analysis as described in Section II.

If the set of N_{mon} signals cannot be determined at fabrication time, the selection of these signals can be made programmable at debug-time using a concentrator network [15]. Such a network would programmably connect a subset of the N_{mon} monitored signals for use in the signature. On-chip SRAM bits (similar to configuration bits in an FPGA) can be used to store the configuration of the concentrator. As debugging proceeds, the configuration can be changed, so that a different set of N_{mon} bits can be used in the signature. An example of the use of a concentrator in a debugging application can be found in [16]. Unlike the concentrators described in previous work in which each bit can be switched independently, we assume that the concentrator switches 8-bit wide words; this reduces the area of the concentrator by approximately 50%, while suffering only a small decrease in flexibility.

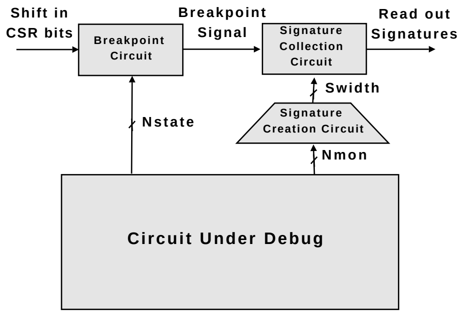
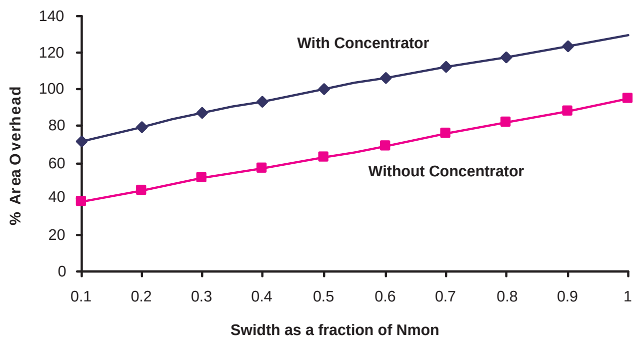
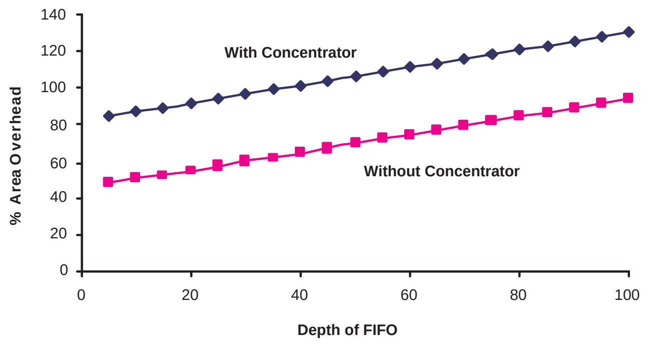


Fig. 4. Debugging Architecture



a) As a function of stored bits



b) As a function of FIFO depth

Fig. 5. Area overhead for instrumented LEON3 processor

As described in Section II, a Universal Hash Function can be used to compress the signature. This can be implemented as an array of XOR gates within the Signature Creation Circuit, and can be used with or without a concentrator network.

B. Area Overhead

In this subsection, we estimate the area overhead of our circuitry. To make our results concrete, we present an estimate of the area required to instrument a specific processor. In Section III, we used implementations of 68HC05 and 8051 processors to illustrate the technique, but these processors are too small to give meaningful area overhead estimates. Instead, in this section, we focus on a typical instantiation of the LEON3 open source processor,[4] because it represents a typical small-but-modern RISC processor, which is the natural next step beyond the small microcontrollers of our initial experiments. The LEON3 is a synthesizable, pipelined 32-processor that is certified SPARC V8 conformant. It is highly configurable, including support for multiprocessing, making it an attractive testbed as we scale this research to increasingly challenging designs. In this subsection, our LEON3 configuration has an area equivalent of 40,000 2-input nand-gates and 2,500 flip flops (so, $N_{state} = N_{mon} = 2500$). This area estimate does not include any RAM used by the LEON3 (but we will include the area of the SRAM used to store the signatures when computing the overhead of our method).

We first present results assuming that the Universal Hash function is not used, and then discuss the overhead of the hash function.

Figure 5(a) shows the area overhead as a function of the ratio between S_{width} and N_{mon}, for an architecture without the Universal Hash function. Intuitively, if this ratio is 1, all state bits are stored as a signature, and so the area is maximum. As the ratio drops, the size of the memory decreases, reducing the area overhead. The figure shows results for an architecture with and without a concentrator; as described above, if a concentrator is not present, the decision of which N_{mon} bits must be fixed before fabrication, while if a concentrator is present, this decision can be made during debugging. The difference between the two lines in Figure 5(a) indicates the area cost of this post-fabrication flexibility.

[4]http://www.gaisler.com

Figure 5(b) shows the area overhead results as a function of the depth of the FIFO, assuming $S_{width} = 0.3N_{mon}$. The larger the depth, the higher the area overhead, but the more cycles that can be backspaced per run of the CUD.

Much of the area overhead in our architecture is due to the Breakpoint Circuit. This circuit requires storing each bit of the target state. As described in Section II, we could use abstraction and match only a subset of the flip flops. Doing so could greatly reduce the size of the circuit, at the cost of losing precision in the debug traces.

Adding the universal hash circuitry to the Signature Creation circuit increases the overhead dramatically. For small designs (such as the 68HC05 described earlier), such circuitry may be feasible. However, the size of a straightforward implementation of the hash circuit grows quadratically with the number of inputs. For the LEON3, if we use a hash circuit similar to what we used for the 8051, the estimated area overhead would be unacceptable (almost 150% *overhead*). By intelligently combining signals (such that the number of XOR gates grows linearly with respect to the number of inputs), we might be able to reduce the area overhead of this structure considerably.

More generally, we believe that signatures can be made much smaller, likely by exploiting architectural insight and design-specific characteristics. For example, Park and Mitra [14] produce information-rich signatures with only 2% area overhead, but narrowly tuned for an Alpha-like processor. We are hopeful we will be able to achieve similarly low overheads in future research.

V. Extensions and Research Directions

This paper introduces a novel paradigm for using formal analysis in post-silicon debug and demonstrates its potential. However, it is only a start. As with any new paradigm, considerable further research remains to be done.

The primary direction for further research is scalability and reducing overhead. As mentioned, we have started work on implementing the BackSpace paradigm on physical hardware, with a multi-thousand latch design. As the idea scales to larger designs, new research challenges will reveal themselves. Some ideas already mentioned include abstraction (with research questions of finding effective abstraction and concretization techniques), better signature functions (that are effective at constraining the pre-images, scalably solvable via SAT, and efficiently realizable in hardware), and reconfigurable BackSpace architectures (how much smaller can the signatures be if they are tailored to a specific target state).

A specific idea, which we have yet to explore but which we believe can greatly reduce signature size, is to use external constraints to prune the pre-images. For example, on toy benchmark circuits, constraining the pre-image by the reachable states of the circuit results in vastly smaller pre-images (equivalently, allows a smaller signature size). Obviously, if we could compute the exact reachable state set, then we would use pure formal verification to hit the target state, obviating the need for the BackSpace approach. But a crude over-approximation of the reachable states might be efficiently computable even for a large design, yet provide useful pruning of the pre-images. Similarly, we are not currently using constraints on the primary inputs to reduce pre-image size. For example, logic analyzer traces or knowledge about the bring-up tests could be used to constrain the primary inputs when computing pre-images.

The other main direction for further research is better handling of realistic designs: partial state monitoring, randomness/non-determinism, multiple clock domains, and circuit marginality and faults. For each of these, there is a straightforward, brute-force attack (partial states and non-determinism were described already, multiple clock domains can be conservatively approximated by a single state machine, and circuit marginality/faults can be handled analogously to fault simulation: for each postulated circuit fault, we repeat the entire BackSpace framework), but much more elegant and efficient approaches are likely necessary and possible.

ACKNOWLEDGMENTS

This work was funded by the Semiconductor Research Corporation (SRC) Task ID: 1586.001. We would also like to thank Brad Quinton for helpful advice on the circuit modeling.

REFERENCES

[1] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, “A Reconfigurable Design-for-Debug Infrastructure for SoCs,” in *43rd Design Automation Conference*. ACM/IEEE, 2006, pp. 7–12.

[2] C. Ahlschlager and D. Wilkins, “Using Magellan to Diagnose Post-Silicon Bugs,” *Synopsys Verification Avenue Technical Bulletin*, vol. 4, no. 3, pp. 1–5, September 2004.

[3] ARM, *Embedded Trace Macrocell Architecture Specification*, July 2007, vol. 20, ref: IHI0014O.

[4] M. Boulé, J.-S. Chenard, and Z. Zilic, “Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug,” in *International Conference on Computer Design*. IEEE Computer Society Press, 2006, pp. 294–299.

[5] J. L. Carter and M. N. Wegman, “Universal Classes of Hash Functions,” in *9th ACM Symposium on Theory of Computing*, 1977, pp. 106–112.

[6] K.-H. Chang, I. L. Markov, and V. Bertacco, “Automating Post-Silicon Debugging and Repair,” in *International Conference on Computer-Aided Design*. IEEE/ACM, 2007, pp. 91–98.

[7] T. Glökler, J. Baumgartner, D. Shanmugam, R. Seigler, G. Van Huben, B. Ramanandray, H. Mony, and P. Roessler, “Enabling Large-Scale Pervasive Logic Verification through Multi-Algorithmic Formal Reasoning,” in *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2006, pp. 3–10.

[8] A. J. Hu, J. Casas, and J. Yang, “Efficient Generation of Monitor Circuits for GSTE Assertion Graphs,” in *International Conference on Computer-Aided Design*. IEEE/ACM, 2003, pp. 154–159.

[9] N. F. Kelly and H. E. Stump, “Software Architecture of Universal Hardware Modeler,” in *European Design Automation Conference (EDAC)*. IEEE, 1990, pp. 573–577.

[10] R. Kuppuswamy, P. DesRosier, D. Feltham, R. Sheikh, and P. Thadikaran, “Full Hold-Scan Systems in Microprocessors: Cost/Benefit Analysis,” *Intel Technology Journal*, vol. 8, no. 1, pp. 63–72, February 2004.

[11] M. Larouche, *Infusing Speed and Visibility into ASIC Verification*, January 2007. [Online]. Available: www.synplicity.com/literature/whitepapers/pdf/totalrecall_wp_1206.pdf

[12] S. Mitra and K. S. Kim, “X-Compact: An Efficient Response Compaction Technique for Test Cost Reduction,” in *International Test Conference (ITC)*. IEEE, 2002, pp. 311–320.

[13] J. A. M. Nacif, F. M. de Paula, C. N. Coelho, Jr., F. C. Sica, H. Foster, A. O. Fernandes, and D. C. da Silva, “The Chip is Ready, Am I done? On-chip Verification using Assertion Processors,” in *International Conference on Very Large Scale Integration of System-on-Chip (VLSI-SoC)*. IFIP WG 10.5, 2003, pp. 111–116.

[14] S.-B. Park and S. Mitra, “IFRA: Instruction Footprint Recording and Analysis for Post-Silicon Bug Localization in Processors,” in *45th Design Automation Conference*. ACM/IEEE, 2008, pp. 373–378.

[15] B. Quinton and S. Wilton, “Concentrator Access Networks for Programmable Logic Cores on SoCs,” in *IEEE International Symposium on Circuits and Systems*, 2005, pp. 45–48.

[16] ——, “Programmable Logic Core Based Post-Silicon Debug For SoCs,” in *4th IEEE Silicon Debug and Diagnosis Workshop*, Germany, May 2007.

[17] S. Safarpour, H. Mangassarian, A. Veneris, M. H. Liffiton, and K. A. Sakallah, “Improved Design Debugging Using Maximum Satisfiability,” in *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2007, pp. 13–19.

[18] R. Singhal, K. S. Venkatraman, E. R. Cohn, J. G. Holm, D. A. Koufaty, M.-J. Lin, M. J. Madhav, M. Mattwandel, N. Nidhi, J. D. Pearce, and M. Seshadri, “Performance Analysis and Validation of the Intel Pentium 4 Processor on 90nm Technology,” *Intel Technology Journal*, vol. 8, no. 1, pp. 34–42, February 2004.

[19] M. J. Y. Williams and J. B. Angell, “Enhancing Testability of Large-Scale Integrated Circuits via Test Points and Additional Logic,” *IEEE Transactions on Computers*, vol. C-22, no. 1, pp. 46–60, January 1973.