Post-Silicon Bug Diagnosis with Inconsistent Executions

Andrew DeOrio, Daya Shanker Khudia and Valeria Bertacco
University of Michigan
{awdeorio, dskhudia, valeria}@umich.edu

***Abstract*—The complexity of modern chips intensifies verification challenges, and an increasing share of this verification effort is shouldered by post-silicon validation. Focusing on the first silicon prototypes, post-silicon validation poses critical new challenges such as intermittent failures, where multiple executions of a same test do not yield a consistent outcome. These are often due to on-chip asynchronous events and electrical effects, leading to extremely time-consuming, if not unachievable, bug diagnosis and debugging processes.**

In this work, we propose a methodology called BPS (Bug Positioning System) to support the automatic diagnosis of these difficult bugs. During post-silicon validation, lightweight BPS hardware logs a compact encoding of observed signal activity over multiple executions of the same test: some passing, some failing. Leveraging a novel post-analysis algorithm, BPS uses the logged activity to diagnose the bug, identifying the approximate manifestation time and critical design signals. We found experimentally that BPS can localize most bugs down to the exact root signal and within about 1,000 clock cycles of their occurrence.

I. Introduction

Diagnosing and debugging failures in large, complex modern digital designs is a difficult task that spans the entirety of the design process. Recently, the role of post-silicon validation has increased, particularly in the microprocessor design industry, in light of the scaling problems of pre-silicon methodologies and tight time-to-market development schedules.

Pre-silicon verification operates on an abstract design model and has the advantage of being fully deterministic and fully observable, but is limited by its slow speed and low coverage. Failing testcases can be reliably reproduced to diagnose functional bugs, which in turn manifest consistently. By contrast, real silicon lacks observability, controllability and deterministic repeatability. As a result, some tests may produce the same outcome over multiple executions, due to the interaction of asynchronous clock domains and varying environmental and electrical conditions. *Bugs that manifest inconsistently over repeated executions of a same test are particularly difficult to diagnose.* Furthermore, the number of observable signals in post-silicon is extremely limited, and transferring observed signal values off-chip is time-consuming. This work addresses precisely this post-silicon validation platform and focuses on the localization of these difficult, inconsistent bugs to ease their debugging.

During post-silicon validation, tests are executed directly on silicon prototypes. A test failure can be due to complex functional errors that escaped pre-silicon verification, electrical failures at the circuit level, and even manufacturing faults that escaped testing. The failed test must be re-run by validation engineers on a post-silicon validation hardware platform with minimal debug support. Post-silicon failure diagnosis is notoriously difficult, especially when tests do not fail consistently over multiple runs. The limited observability and controllability characteristics of this environment further exacerbate this challenge, making post-silicon diagnosis one of the most challenging tasks of the entire validation effort.

A. Contributions

To address this problem, we propose a novel solution called BPS, ("Bug Positioning System"). BPS leverages a statistical approach to address the most challenging post-silicon bugs, those that do not manifest consistently over multiple runs of a same test, by localizing them in space (design region) and time (of bug manifestation). BPS leverages existing on-chip trace buffers or a lightweight custom hardware component to record a compact encoding of observed signal activity over multiple runs of the same test. Some test runs may fail, while others may pass, leading to different activity observations. In addition, observations may be affected by variations introduced by the operating environment – both system-level activity and environmental effects. Finally, a post-analysis software algorithm leverages a statistical approach to discern the time and location of the bug manifestation. Overall, BPS eases debugging in post-silicon validation by:

- **Localizing inconsistent bugs** in time and space, often to the exact problem signal, thus reducing the engineering effort to root-cause and debug the most difficult failures. BPS targets a wide range of failures, from functional, to electrical, to manufacturing defects that escaped testing.
- **Tolerating non-repeatable executions** of the same test, a characteristic of the post-silicon environment, and thus not part of any mature pre-silicon methodology. BPS does not require any a-priori knowledge of the design or failures.
- **Providing a scalable solution** with minimal engineering effort, able to handle the complexity of full chip integration typical of post-silicon validation, while minimizing off-chip data transfer through the use of compact encodings of signal activity.

II. Related Work

In industry practice, the post-silicon validation process begins when the first silicon prototypes become available. These chips are then connected to specialized validation platforms that facilitate running post-silicon tests, a mix of directed and constrained-random workloads. Upon completion of each test, the output of the silicon prototype is checked against an architectural simulator, or in some cases, self-checked [1], [2].

When a check fails, indicating that an error has occurred, the debugging process begins, seeking to determine the root cause of the failure. On-chip instrumentation can be used to observe intermediate signals. Techniques such as scan chains, on-chip logic analyzers [3] and flexible logging infrastructures [4] are configured to trace design signals (only a small number can usually be observed) and periodically transfer data off-chip. Traces are then examined by validation engineers to determine the root cause of the problem. This process is time-consuming and engineering intensive, and is further exacerbated by bugs with inconsistent outcomes. Additionally, off-chip data transfers are very slow, which further hinders observability due to limited transfer time. BPS strives to reduce debugging effort, automatically diagnosing the time and location of bugs, while minimizing off-chip transfers.

The debugging process of non-deterministic failures can be aided by deterministic replay mechanisms [5], [6]. However, these solutions perturb system execution which can prevent the bug from manifesting, and often incurs significant hardware and performance overheads. In addition, in an effort to automate the failure diagnosis process, methods based on formal verification techniques have been proposed [7]–[10]. These solutions require deterministic execution

and a complete golden (known-correct) model of the design for comparison. However, the fundamental scaling limitations of formal methods preclude these techniques from handling industrial size designs. By contrast, BPS does not rely on formal methods, but rather leverages a statistical approach. Furthermore, it requires no a-priori knowledge of the design or the failure. Methods that target small fully deterministic systems [11]–[13] are capable of identifying limited types of electrical errors in small circuits. Other solutions targeting different types of errors leverage Bayesian approaches [14] for transient errors and analysis of bug reports for software errors [15]. On the other hand, functional failures have been approached by recording system state using a scan chain, and then comparing passing and failing tests [16]. Early work in troubleshooting circuit boards used signatures to achieve compact observations [17], [18]. While BPS shares common goals with many of these works, it focuses on targeting large designs, since post-silicon validation is typically carried out on the largest industrial microprocessors.

Finally, specialized post-silicon debugging approaches often add dedicated hardware units for debugging specific areas, such as the memory subsystem [19] or speed paths [20]. Solutions such as BLoG/IFRA [21] can localize electrical bugs in the processor core, as long as errors are detected on-chip within about 1,000 cycles. Additionally, a bug is localized by IFRA only down to an architectural block comprising approximately 10,000 gates, thus requiring significant additional manual effort to locate the exact signal(s) responsible for the failure. In contrast, BPS is capable of identifying exact bug signals at any location or time in complete industrial-size chips. Table I highlights the main differences between BPS and IFRA.

	IFRA [21]	**BPS**
description level	architectural only	✓logic and architectural
design complexity	single core only	✓whole chip
manual effort	required for setup	✓fully automated
type of bugs	electrical	✓functional, electrical, mfg.
bug depth	at most 1,024 cycles	✓flexible
temporal localization	✓exact cycle	cycle window
spatial localization	architectural block	✓exact injection signal
bug signal specificity	˜10,000	✓˜60
search space reduction	99.8%	✓99.999%

TABLE I
BPS compared with IFRA [21] shows that BPS is a general solution capable of localizing bugs with minimal manual effort.

III. BPS Overview

BPS diagnoses the time and location of functional, electrical and manufacturing bugs during post-silicon validation; in particular, those bugs that manifest through inconsistent test outcomes. In these situations, the same post-silicon test may pass for some of its executions and fail other times, due to asynchronous events or electrical and environmental variations on-chip.

To locate these difficult bugs, BPS leverages a two-part approach: logging compact observations of signal activity with an on-chip hardware component, followed by an off-chip software post-analysis. The compact size of observations produced by the hardware are essential for minimizing expensive off-chip data transfers. These signal observations are gathered and reduced to a compact encoding for a number of executions of the same test, some passing and some failing, but usually all slightly different. Finally, the collected data is analyzed by the BPS post-analysis software, leveraging a statistical approach that is insensitive to the natural variations over several executions, but it is capable of detecting the more dramatic differences in signal activity typically caused by bugs. The result is the localization of the bug through the reporting of an approximate clock cycle and the set of signals most closely related to the error.

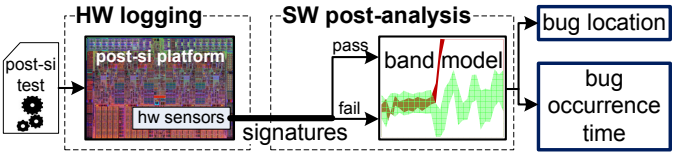


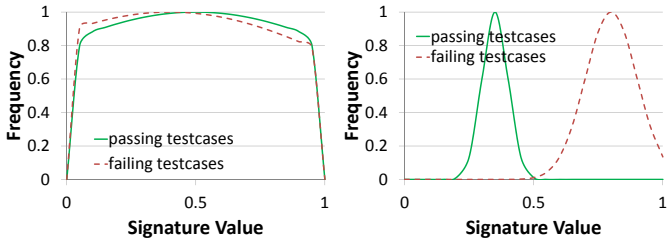
Fig. 1. **BPS Operation.** BPS operates in two phases: first, hardware sensors collect compact encodings of signal activity on the post-silicon platform for a number of executions of the same test: some may pass, while others fail. These observations are then analyzed by post-analysis software, which locates functional, electrical and manufacturing failures in time and space.

A. BPS Hardware

The hardware component of BPS logs *signatures*, compact encodings of observed activity on a set of target signals, which are later used by BPS' post-analysis software to locate failures. Signals available for observation are selected at design time, and the most effective choices are typically control signals. Signatures are recorded at regular intervals, called *windows*, and stored in an on-chip buffer. Windows can range in length from hundreds to millions of cycles, and are later used to determine the occurrence time of a bug. Logged data is periodically transferred off-chip for analysis by the BPS software. Simple signatures can often be collected using existing debug infrastructures, such as on-chip logic analyzers [3], flexible event counters [4], [22], [23] or performance counters.

An ideal signature is compact for dense storage and fast transfer, and represents a high-level view of the observed activity. Furthermore, the signature must exhibit a statistical separation between passing and failing cases, as shown in Figure 2. In order to differentiate erroneous behavior from correct behavior, BPS characterizes activity using distributions of signatures. Throughout the development process of BPS, we considered a variety of signatures, including various codes and counting schemes. We found that many traditional codes, such as cyclic, hamming distance and multiple input shift registers (MISR), exhibited a wide range of output (Figure 2a) and are very susceptible to noise: small variations among executions led to severe variations in the signature value. Thus, it is difficult to distinguish erroneous from correct behavior with these signatures. This led us to counting schemes, where the amplitude of changes in signal activity leads to approximately proportional changes in signature values. The result is a discernible difference in the distribution of signatures for passing vs. failing testcases (Figure 2b) and less vulnerability to noise.

Signatures based on counting schemes include toggle count, time at one and time at zero. We chose a variation of time at one for BPS: the probability of a signal being at one during a time interval (window), $P(time@1)$. This signature is compact, simple and encodes notions of switching activity, as well as timing. By contrast, toggle count expresses the logical activity of the signal, but it does not provide



(a) Poor statistical separation (b) Good statistical separation

Fig. 2. **Comparison of typical distributions with different signatures.** A signature that exhibits a wide, evenly distributed output (a) does not allow BPS to differentiate correct behavior from incorrect. In contrast, signatures that exhibit good separation (b) are effective.

any temporal information. Figure 3 shows an on-chip hardware sensor implementation for measuring $P(time@1)$. Signals from the design are connected to counters via muxes, allowing the selection of a subset of the signals to be monitored. Note that we can calculate this signature by simply counting the number of cycles when a signal is at 1 and normalizing to the window length. Furthermore, we noted experimentally that approximately 9 bits of precision are sufficient for accurately locating bugs, offering precision similar to a window size of 512 cycles. Thus, the resulting probability can be truncated and stored with fewer bits. The final result is copied to a memory buffer at the end of each window.

Note that it is not necessary to collect signatures for every signal in the design. BPS leverages signals high in the module hierarchy, those most likely to be available for observation in a post-silicon validation platform. To further reduce the amount of data that must be transferred off-chip, BPS uses two signal selection optimizations: first, it excludes data signals, often identified as busses 64-bits wide or more for a 64-bit processor design. Depending on hardware resources, signatures can be collected all at once or in groups. If post-silicon debugging hardware resources are scarce, then multiple executions of the test can be leveraged to complete the signature collection, even if those executions are not identical, since BPS' post-analysis software is tolerant to variation.

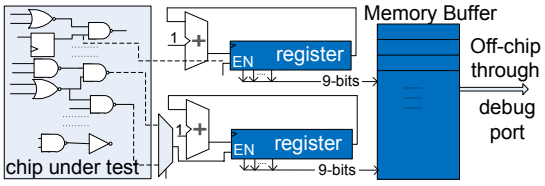


Fig. 3. **BPS Hardware** collects signatures for a subset of the design's signals. An observed signal's time@1 is tracked at each cycle for the duration of a window; the sum is then truncated to limit its size and saved to a buffer.

B. BPS Post-Analysis Software

After on-line signature collection is completed, off-line software analysis identifies a set of signals indicating where the bug occurred and at what time. BPS uses the signatures from passing runs of the test to build a model of expected behavior, and then determines when failing executions diverge from the model, revealing a bug.

BPS' software begins by partitioning a test's signatures into two groups: those where the test passed, and those where the test failed, as illustrated in Figure 4 (top and bottom portions). The signatures in each group are organized by window and signal: for each window/signal combination, BPS considers multiple signature values, the result of multiple executions of the test. Next, passing signatures are used to build a model of acceptable system behavior for each observed signal: the algorithm goes through all the signatures related to one signal, building the model one window at a time.

The middle part of Figure 4 illustrates the model built for signalA as a green (light gray) band. Representing the expected behavior as a distribution of values enables BPS to tolerate variations in signature values since, as we discussed above, post-silicon validation is characterized by non-identical executions due to naturally occurring variations among distinct executions. Figure 5 illustrates how distributions are used to build a model of observed behavior. The passing band for one signal is generated by computing the mean (μ_{pass}) of the signature values for each window, surrounded by k_{pass} standard deviations (σ_{pass}), where k_{pass} is a parameter. Thus the band representing the passing signatures is bounded by

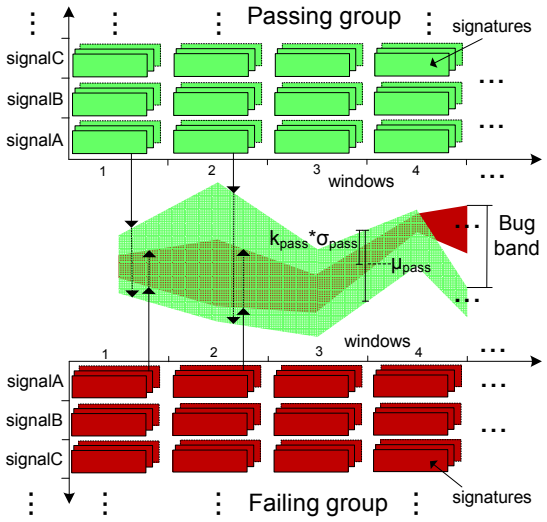


Fig. 4. **BPS post-analysis algorithm.** Using the passing group of signatures from a test, BPS builds a model of the expected behavior for each signal, shown by the green (light gray) band. The red (dark gray) band shows the behavior of the failing test runs, constructed from the failing group.

$\mu_{pass} \pm k_{pass} * \sigma_{pass}$. In order to represent over 95% of uniformly distributed data points, we used $k_{pass} = 2$ for our experiments.

The BPS software now adds the failing group to the model, once again considering each signal in turn and building the model window-by-window. The failing group is plotted in Figure 4 as a red (dark gray) band. Similar to the passing group, the failing group is modeled as the mean surrounded by k_{fail} standard deviations ($\mu_{fail} \pm k_{fail} * \sigma_{fail}$). When the failing band falls inside the passing band, we deem the corresponding signal's behavior to be within an acceptable range, indicating that a test failure has not yet occurred or, possibly it is masked by noise. When it diverges from the passing band, we identify this as buggy behavior.

Using this band model, BPS determines when failing signatures diverge from passing signatures: we call the divergence amount a *bug band*. Starting at the beginning of a test execution, the algorithm considers each window in turn, calculating the bug band one signal at a time. The bug band is zero if the failing band falls within the passing band, otherwise it is the difference between the two top (or bottom) edges. As an example, Figure 5 shows the model obtained and the bug band calculation for a signal in the memory stage of a 5-stage pipelined processor.

The set of bug bands (one for each signal) is ranked and compared

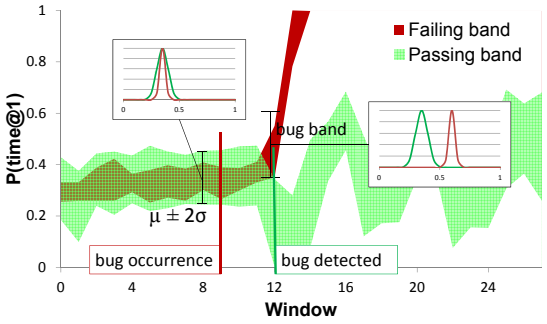


Fig. 5. **BPS band model.** Band model for a memory control signal from a 5-stage pipelined processor. Each slice of time in the model represents two distributions (passing and failing). The bug is detected when the failing band diverges from the passing one, representing diverging signature distributions. The quantitative amount is measured by the bug band.

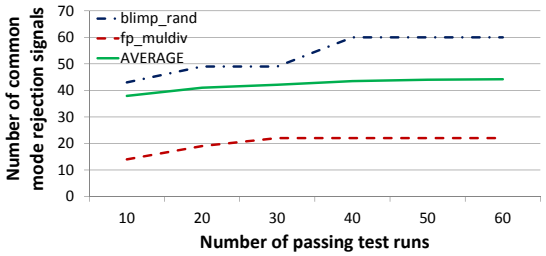


Fig. 6. **Number of common mode rejection signals** in the OpenSPARC T2 design. As the number of passing tests increases, the average number of signals stabilizes to 44, only 1% of the signals monitored by BPS.

against a threshold that varies with the design (see Section IV-B). If no bug band exceeds the threshold, BPS moves on to the next window. When one or more bug bands exceed the threshold, BPS notes the time (represented by the window) and the signals involved, reporting them as the bug time and location.

As an additional filtering step, a set of common mode rejection signals is leveraged by BPS to mitigate the noise present in large designs. To generate this filter, BPS is run with two passing groups of a same testcase, rather than a passing and a failing group. The signals identified in this process are removed from BPS' candidate bug signals list; this helps to minimize the number of false positives. We found that in a complex design (OpenSPARC T2), as the number of runs used for identifying common mode rejection signals increased, the resulting list stabilized. Figure 6 shows this asymptotic trend for the testcases exhibiting the largest and smallest common mode rejection signal lists, as well as the average. The size of the list is typically small, only 44 signals (some of which are buses), comprising 1% of the design's monitored signals.

C. Tuning Parameters

A number of parameters affect the quality of the results produced by BPS: the bug band threshold, the window length, the number of test executions in both the passing and failing groups and a set of common mode rejection signals.

The **bug band threshold** is used to determine which signals BPS detects, and also causes BPS to stop looking for bugs. Changing this value changes BPS' sensitivity to bugs and noise. In some cases, the design perturbation caused by a bug can be amplified by neighboring logic over time: a higher bug band threshold can cause BPS to detect these neighboring signals after searching longer (more windows) for errors. The result is often a reduction in the number of signals detected, since few signals have a bug band that exceeds the threshold. However, this can also lead to signals that are less relevant to the error, as well as longer detection times. On the other hand, a bug band threshold that is too small can result in prematurely flagging irrelevant signals, halting the search for the bug. In our experiments, we found that a single threshold value could be used for each design. Thus, in practice, the proper bug band threshold is determined when running the first tests, and then reused for the rest.

The **window length** is the time interval (in cycles) of signature calculation, and affects the precision of BPS' timing. Increasing the window length increases the number of cycles that must be inspected after BPS reports the bug detection window. However, large window lengths have the advantage of allowing longer periods of execution between signature logging and thus decrease the volume of data that must be transferred off chip. Thus, the choice of window size is a trade-off between off-chip data transfer times and the precision of bug localization timing.

The **population size of passing and failing groups** primarily affects false negative and false positive rates. When the population of failing runs is small, variations in the failing group have greater impact on the mean. Thus, bugs are triggered more easily, resulting in increased false positives. Conversely, when the number of passing testcases is small, variations impact the mean of the passing population, this time increasing the false negative rate.

IV. Experimental Evaluation

In order to evaluate the effectiveness of BPS, we employed it to find bugs on two microprocessor designs with a variety of failures, including electrical, manufacturing and functional bugs. Each processor ran a set of 10 distinct application workloads. The designs are a 5-stage pipelined processor implementing a subset of the Alpha ISA, comprising 4,901 lines of code and 4,494 signals (bits). After excluding data signals, BPS was left with 525 signals for analysis. Our larger industrial design, the OpenSPARC T2 [24] system, has 1,289,156 lines of code and 10,323,008 signal bits. We simulated the system in its single core version (cmp1), which consisted of a SPARC core, cache, memory and crossbar. BPS monitored the control signals at the top level of the design for a total of 41,743 signal bits, representative of the signals that would likely be available during post-silicon debugging of such a large design. Both designs were instrumented to record signatures during logic simulation; execution variations were introduced with variable and random communication latencies. BPS requires only these compact signatures and pass/fail status of the test to operate.

Table II shows the bugs introduced in 10 different variants of the design, with one bug in each variant. The failures included functional bugs (design errors), electrical failures and manufacturing errors. Functional bugs were modeled by modifying the design logic, and electrical failures were simulated by temporary single bit-flips persisting for a number of cycles. Manufacturing errors were modeled as single bit stuck-at faults lasting for the duration of the test. Each design variant executed several tests a number of times, and a checker would determine if the final program output was correct. The workloads used as test inputs for the two processor designs included assembly language tests, as well tests from a constrained-random generator. There were 10 tests for each design, ranging in size from about 20K cycles to 11M cycles. Each test was run 10 times for each bug, using 10 random seeds with varying impact on memory latency. Additionally, each test was run 10 times (with new random seeds) without activating the bug to generate the passing group.

5-stage pipeline bugs	**description**
ID fxn	functional bug in decode
EX fxn	functional bug in execution unit
fwd fxn	functional bug in fwding logic
EX SA	stuck-at in execution
cache SA	stuck-at in cache to proc ctrl
ID SA	stuck-at in decode
MEM SA	stuck-at in memory
WB elect	electrical error in writeback
ID elect	electrical error in decode
EX elect	electrical error in execute
OpenSPARC T2 bugs	**description**
PCX gnt SA	stuck-at in PCX grant
XBar elect	electrical error in crossbar
BR fxn	functional bug in branch logic
MMU fxn	functional bug in mem ctrl
PCX atm SA	stuck-at in PCX atomic grant
PCX fxn	functional bug in PCX
XBar combo	combined electrical errors in Xbar/PCX
MCU combo	combined electrical errors in mem/PCX
MMU combo	combined functional bugs in MMU/PCX
EXU elect	electrical error in execution unit

TABLE II

Designs and modeled failures. The bugs introduced in each design variant were functional, electrical and manufacturing (stuck-at) failures.

5-stage	ID fxn	EX fxn	fwd fxn	EX SA	cache SA	ID SA	MEM SA	WB elect	ID elect	EX elect
bubblesort	✓+	✓+	✓	✓	✓	✓+	✓+	✓+	✓+	✓+
combRec	n.b.	✓	✓	✓	✓	✓+	✓+	✓+	✓+	✓+
fib	✓+	n.b.	✓	✓	✓	✓+	✓+	✓+	✓+	✓+
hanoi	n.b.	n.b.	✓	✓	✓	✓+	✓+	✓+	✓+	n.b.
insert	✓+	✓+	✓	✓	✓	✓+	✓+	✓+	✓+	✓+
knapsack	✓	✓+	✓	✓	✓	✓+	✓+	✓+	✓+	✓+
matmult	✓	✓+	✓+	✓	✓	✓+	✓+	✓+	✓+	✓+
merge	✓+	✓	✓	✓	✓	✓+	✓+	✓+	✓+	✓+
quick	✓+	✓+	✓	✓	✓	✓+	✓+	✓+	✓+	✓+
saxpy	✓	✓+	✓+	✓	✓	✓+	n.b.	✓+	✓+	✓+

OpenSPARC	PCX gnt SA	XBar elect	BR fxn	MMU fxn	PCX atm SA	PCX fxn	XBar combo	MCU combo	MMU combo	EXU elect
blimp_rand	✓+	✓+	✓	✓	✓+	✓+	✓+	f.n.	✓+	f.n.
fp_addsub	n.b.	f.p.	✓	✓	✓	✓+	f.p.	n.b.	✓+	f.p.
fp_muldiv	n.b.	f.p.	✓	✓	✓	✓+	f.p.	f.p.	✓+	f.p.
isa2_basic	n.b.	f.n.	✓	n.b.	✓+	✓+	✓+	✓+	n.b.	f.n.
isa3_asr_pr	n.b.	✓	✓	f.n.	✓+	✓	✓+	✓+	✓	✓
isa3_window	n.b.	✓	✓	n.b.	✓+	✓	f.n.	f.n.	n.b.	✓
ldst_sync	n.b.	✓+	✓	✓	✓+	✓+	✓+	✓+	✓+	n.b.
mpgen_smc	n.b.	✓+	✓	✓	✓+	✓+	✓+	✓+	✓+	✓+
n2_lsu_asi	n.b.	f.n.	✓	f.n.	✓+	✓+	✓+	✓+	✓+	n.b.
tlu_rand	n.b.	✓+	✓	✓	✓+	✓+	✓+	✓+	✓+	✓+

TABLE III

BPS signal localization. Checkmarks (✓) indicate that BPS identified the bug; the exact root signal was located in cases marked with ✓+. Each design includes two bugs involving signals not monitored by BPS (light shading). In these cases, BPS could identify the bug, but not the root signal. "n.b." indicates that no bug manifested for every run of the test; false negatives and false positives are marked with "f.p." and "f.n.".

A. Bug Localization

Table III shows the quality of BPS bug detection for the 5-stage pipeline and OpenSPARC T2 designs: eventually, BPS was able to detect the occurrence of every bug. Often, the exact root signal was detected, a few exceptions include 5-stage's `EX SA` and `cache SA`, as well as OpenSPARC's `BR fxn` and `MMU fxn`, where the root bug signal was deep in the design and not monitored by BPS (indicated by light shading). In these situations, BPS was still able to identify signals close to the bug location. In a few cases with the OpenSPARC design, BPS did not find an injected bug, a false negative. Finally, we observed false positives in two testcases, instances where the system detected a bug before it was injected: both were floating point testcases (`fp_addsub` and `fp_muldiv`). Upon further investigation, we found the cause to be three signals that exhibited noisy behavior, but were not included in the common mode rejection filter. When these three signals were added to the filter, the false positives were correctly avoided, highlighting the effectiveness of rejecting noisy signals.

Some bugs were easier to detect than others, for example BPS was able to detect the exact bug root signal in 8 out of 10 testcases with the `PCX atm SA` bug, while a seemingly similar bug, the `PCX gnt SA`, did not manifest in 9 out of 10 cases. `PCX atm SA` had wider effects on the system, and thus manifested more frequently and was easier to detect. By contrast, the `PCX gnt` signal was not often used and thus the related bug did not manifest as frequently.

The number of signals and the time between bug occurrence and bug detection are also a consideration in post-silicon validation: it is easier to debug a small number of signals that are close to the bug's manifestation. Figure 7 shows the number of signals identified by BPS for the bugs in each design. Each bar of the figure represents one bug, averaged over all tests used in BPS, using a window length of 512 cycles. We found that the number of signals is highly dependent

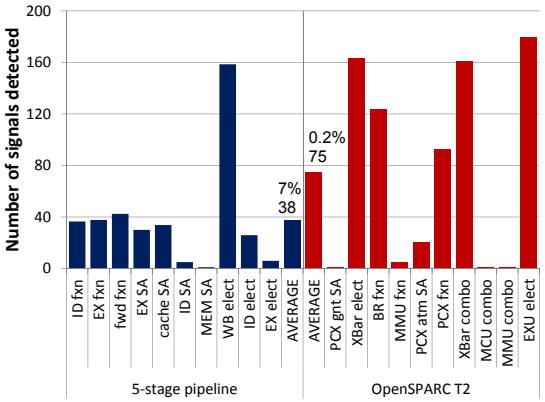


Fig. 7. **BPS spatial localization,** number of signals identified as closely related to the bug when using a 512 cycle window.

on the bug, with BPS detecting a single signal for some bugs, such as 5-stage's `MEM SA` and OpenSPARC's `MCU combo`. Other bugs were more challenging, for example, with the 5-stage pipeline's bug `WB elect`, BPS detected 158 signals on average: this was due to very wide-spread effects of this single bug throughout the design. We also noted that this catastrophic bug was caught by BPS very quickly, less than 750 cycles after the bug's manifestation. While BPS monitored 80x more signals in the OpenSPARC experiments, the number of detected signals increased by only 2x, on average. This demonstrates BPS' ability to narrow a large number of candidate signals (nearly 42,000) down to a smaller pool amenable to debugging.

The time to detect each bug is reported in Figure 8, expressed as the number of cycles between bug injection and detection. Each bar of the figure represents one bug, averaged over all tests, using a window length of 512 cycles. The error bars indicate the error window in the BPS reporting, which corresponds to the window length. The average detection time was worse for the 5-stage pipeline; mostly due to three bugs: the `EX SA` and `cache SA` stuck-at bugs were both inserted into data busses, and thus could not be directly observed by BPS. The effects of the bug required many cycles before observable control signals diverged. In the case of the ID functional bug, the effects of the bug were masked for many cycles in the fib testcase, thus, the bug went undetected until later in the program's execution. In the OpenSPARC design, we noted that most bugs were detected within about 750 cycles, on average. Two bugs were an exception to this rule, both involving the MMU, where bugs involving signals deep in the design remained latent for a time before being detected.

Overall, BPS was successful in narrowing down a very large search

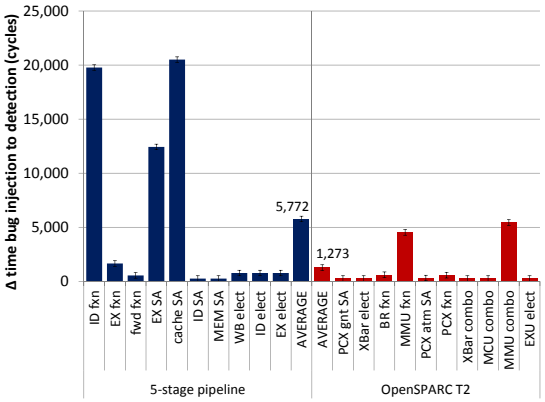


Fig. 8. **BPS temporal localization,** in cycles between bug injection and detection. Error bars indicate the localization range.

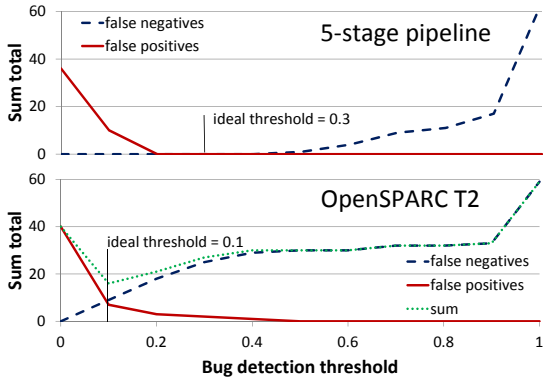


Fig. 9. **Bug band threshold and quality of results.** As the threshold increases, false negatives increase, while false positives decrease. Thus, we select a threshold by minimizing the sum.

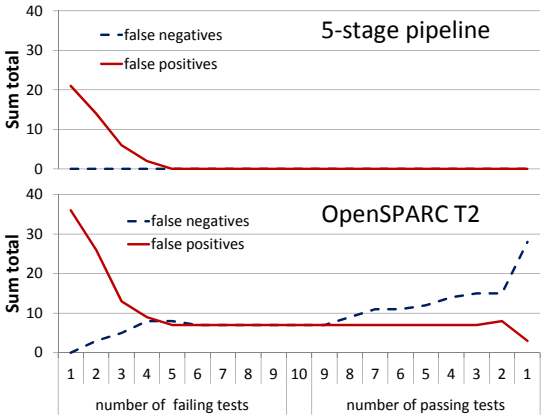


Fig. 10. **Population size and quality of results.** The plot shows false negatives and false positives as the population of passing and failing runs changes:. 10 passing runs on the left and 10 failing runs on the right.

space (number of signals * test length) to a small number of signals and cycles. Our experiments show that it was able to correctly reject over 99.999% of the candidate $\langle location, time \rangle$ pairs. By contrast, IFRA [21] achieves 99.8% by this metric.

B. Bug Detection Quality

The quality of detection results is evaluated by both the number of signals detected and the ability to detect the source signal of the bug. The accuracy of the time at which the bug is detected is also a consideration, as are the rates of false negatives and false positives. Additionally, the number of signals must be manageable by validation engineers. A variety of factors affects the quality of BPS results, including the bug band threshold, number of executions (of the same test) and window length.

Bug Band Threshold. Figure 9 shows the effect of threshold on the number of false negatives and false positives, reporting the sum total over all bugs and testcases for each threshold. We note that for both designs, the number of false positives starts high and decreases as the threshold increases, the result of a tighter filter for discerning a bug occurrence over system's noise. However, when the bug band threshold is high, the subtler effects of bugs are overlooked by BPS, resulting in more bugs being missed. In contrast, the number of false negatives increases as the threshold increases. Thus, there is a trade-off between false positive and false negative rates, indicated by the minimum of their sum, occurring at 0.3 for the 5-stage pipeline and 0.1 for OpenSPARC T2. We also noted that the number of signals detected decreased as the threshold increased: with high thresholds, BPS detects neighboring signals after searching longer (more windows) for errors.

Population Size. The ratio of the number of passing vs. failing runs determines the weight of signatures that may vary from the mean. Figure 10 plots the number of false positives and false negatives as the balance of passing vs. failing testcases changes. At the center of the X-axis, the balance is equal, with 10 passing and 10 failing tests. In the case of the 5-stage pipeline, we note that false negatives are not present, regardless of the number of runs, indicating that the wide-spread effects of a bug in a small design will always be caught eventually. In both designs, the number of false positives decreases with more failing testcases. When the number of failing runs is small, only a few data points that differ from the mean can exceed the bug band threshold, thus false positives are more prevalent. Conversely, we see an increase in false negatives as the balance tips towards more failing. Note that for our other experiments, we used an even balance of 10 passing and 10 failing runs.

Window Length. Next, we examine the interaction of window length with quality of results. BPS measures signatures over a time interval and the interval length affects the time at which bugs are detected. We found that the window length had a significant effect on the number of signals detected and the detection time. Figure 11 plots the number of signals detected, as well as the time between bug injection and detection as window length increases. First, we observed that shorter windows yield more accurate time localizations: for long windows, the lag is mostly due to the length of the window itself. Note that failing runs may execute long past the bug detection, since the cycle is identified during post-analysis. Additionally, the number of signals detected increases as the window length increases. This is due to the increased time for the effects of the bug to spread to many signals in the system. Thus, a smaller window length yields more accurate results. We note however, that it would be possible to run BPS multiple times with an iterative approach, strategically decreasing the window size in order to narrow down a bug's location.

C. Performance and Area Overhead

BPS' software post-analysis was run on Xeon Core i7 2.27GHz servers, and the time for running each analysis was approximately 412s for OpenSPARC T2 and 5s for the 5-stage design. This time varies with the number of signals under observation, as BPS must consider more data. CPU time is also linearly dependent on the bug detection window, since BPS must sift through more windows searching for bugs located deep in a test execution.

The hardware logging of BPS' simple signatures can be obtained using either standard flexible debugging infrastructures, or with custom hardware. In the case of pre-existing debugging hardware, signatures can be gathered with no area overhead.

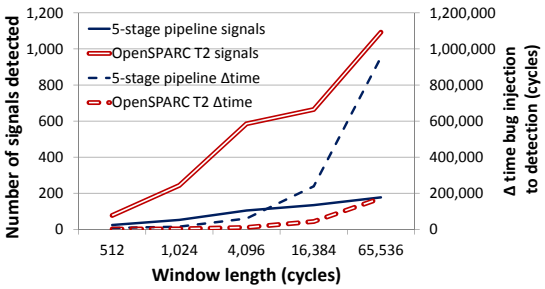


Fig. 11. **Impact of window length on quality of results.** The plot reports the number of signals detected, and the time between bug manifestation and bug detection as window length increases. Shorter window lengths yield faster, more precise time localization.

With its unique ability to leverage data from non-repeatable test executions, BPS enables a trade-off between area overhead and the time required to gather signature data. With a small area budget, the signatures for a set of signals can be gathered a few signals at a time. Leveraging fast post-silicon execution, a test is run multiple times, recording signatures from a different subset of signals with each run. Variation among different runs averages out in BPS' statistical approach, and thus does not impact the diagnosis quality.

We evaluated the area overhead of a BPS hardware sensor implementation in Verilog HDL, synthesized with a 65nm TSMC target library. A signature recording unit capable of 100 signals over 100 windows, recording signatures for the 41,743 signal bits in the OpenSPARC design would require 418 test executions, a reasonable demand at fast post-silicon execution speeds. With 9-bit precision for signature storage, full precision for a window length of 512 cycles, the resulting memory buffer comprises 1.33 mm^2. The hardware to generate these signatures occupies 23,240 μm^2, resulting in a total area of 1.35 mm^2. Compared to the OpenSPARC T2 system (342 mm^2 [25]), the area overhead of BPS is 0.396%, less than half the overhead of IFRA [21]. When comparing storage, the dominant factor in both BPS' and IFRA's overhead, BPS requires 11KB with this configuration, compared to 60KB for IFRA.

D. Limitations

While BPS is effective in localizing a wide variety of functional, electrical and manufacturing failures, it has a few limitations.

The signals available to BPS for observation play a role in its ability to accurately localize bugs. The scope of signals available for observation during post-silicon validation varies with the quality of its debug infrastructure. When the signals involved in a bug are monitored by BPS, it is highly effective in identifying failures down to the exact source signal, illustrated qualitatively in Section IV-A. However, when the source signal is deep in the design and not monitored, the accuracy of BPS is reduced. This results in an increased number of signals detected, as well as increased detection time. Thus, BPS is able to identify bugs that originate either within or outside of its observable signals, but it can only identify the exact signal when this signal is monitored.

The relationship between window size and the duration of a bug also affects BPS. A bug's duration comprises the perturbation in the source signal and the after-effects that may spread to nearby connected logic. When the bug duration is small relative to the window size, its effect on the signature recorded for a window is proportionally small ($bugband < 2\sigma$), sometimes resulting in false negatives, depending on the bug band threshold. The effect of short bug durations can be counteracted by a smaller threshold, as well as by smaller window sizes. We most often observed this phenomena when identifying bug root signals. As window sizes increased, the number of cases where BPS detected the exact root signal decreased, despite being able to detect other signals related to the bug. Upon further investigation, we found that in many cases, the duration of the perturbation of the bug's root signal was small compared to the window size, while the secondary effects of the bug remained observable in the design's behavior.

V. Conclusions and Future Work

We have presented BPS, a solution for locating the most challenging bugs during post-silicon validation: functional, electrical and manufacturing bugs with inconsistent program outcomes. BPS has two components: hardware structures that log a compact encoding of observed signal activity and companion post-analysis software. BPS can localize bugs in time and space while tolerating non-repeatable executions of the same test. It provides a fast solution, reducing off-chip data transfers with compact signatures and scales to industrial-size designs. BPS is effective in locating bugs under many different workloads, often to the exact signal.

Moving forward, we plan to couple BPS with a technique to automate signal selection: choosing those signals with high potential for exposing bugs would increase the accuracy of BPS. Finally, while BPS was evaluated on bare-metal tests, we plan to explore the type of variations that an OS-based environment may introduce.

Acknowledgments

This work was developed with partial support from the National Science Foundation and the Gigascale Systems Research Center.

References

[1] A. Adir, M. Golubev, S. Landa, A. Nahir, G. Shurek, V. Sokhin, and A. Ziv, "Threadmill: A post-silicon exerciser for multi-threaded processors," in *Proc. DAC*, 2011.

[2] I. Wagner and V. Bertacco, "Reversi: Post-silicon validation system for modern microprocessors," in *Proc. ICCD*, 2008.

[3] L. Whetsel, "An IEEE 1149.1 based logic/signature analyzer in a chip," in *Proc. ITC*, 1991.

[4] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs," in *Proc. DAC*, 2006.

[5] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously recording program execution for deterministic replay debugging," in *Proc. ISCA*, 2005.

[6] S. Sarangi, B. Greskamp, and J. Torrellas, "CADRE: Cycle-accurate deterministic replay for hardware debugging," in *Proc. DSN*, 2006.

[7] F. De Paula, M. Gort, A. Hu, S. Wilton, and J. Yang, "Backspace: formal analysis for post-silicon debug," in *Proc. FMCAD*, 2008.

[8] B. Keng, S. Safarpour, and A. Veneris, "Bounded model debugging," *IEEE Trans. CAD of ICs and Systems*, vol. 29, no. 11, 2010.

[9] S. Safarpour and A. Veneris, "Automated design debugging with abstraction and refinement," *IEEE Trans. CAD of ICs and Systems*, vol. 28, no. 10, 2009.

[10] Y.-S. Yang, N. Nicolici, and A. Veneris, "Automated data analysis solutions to silicon debug," in *Proc. DATE*, 2009.

[11] J. Gao, Y. Han, and X. Li, "A new post-silicon debug approach based on suspect window," in *Proc. VTS*, 2009.

[12] W. Li, A. Forin, and S. A. Seshia, "Scalable specification mining for verification and diagnosis," in *Proc. DAC*, 2010.

[13] J.-S. Yang and N. Touba, "Expanding trace buffer observation window for in-system silicon debug through selective capture," in *Proc. VLSI Test Symposium*, 2008.

[14] S. Jha, W. Li, and S. Seshia, "Localizing transient faults using dynamic bayesian networks," in *Proc. HLDVT*, 2009.

[15] B. R. Liblit, "Cooperative bug isolation," Ph.D. dissertation, Berkeley, CA, USA, 2004, aAI3183833.

[16] P. Dahlgren, P. Dickinson, and I. Parulkar, "Latch divergency in microprocessor failure analysis," in *Proc. ITC*, 2003.

[17] R. A. Frohwerk, "Signature analysis: A new digital field service method," *Hewlett-Packard Journal*, May 1977.

[18] J. Savir, "Syndrome-testable design of combinational circuits," *IEEE Trans. Computers*, vol. C-29, no. 6, 1980.

[19] A. DeOrio, I. Wagner, and V. Bertacco, "DACOTA: Post-silicon validation of the memory subsystem in multi-core designs," in *Proc. HPCA*, 2009.

[20] R. McLaughlin, S. Venkataraman, and C. Lim, "Automated debug of speed path failures using functional tests," in *Proc. VTS*, 2009.

[21] S.-B. Park, A. Bracy, H. Wang, and S. Mitra, "BLoG: Post-silicon bug localization in processors using bug localization graphs," in *Proc. DAC*, 2010.

[22] S. Baartmans and B. White, *U.S. Patent no. 6438664: Customizable event creation logic for hardware monitoring*, Intel Corp., 2007.

[23] B. Quinton and S. Wilton, "Programmable logic core based post-silicon debug for SoCs," in *IEEE Silicon Debug and Diagnosis Workshop*, 2007.

[24] "Sun microsystems OpenSPARC," http://opensparc.net/.

[25] X. Dong and Y. Xie, "System-level cost analysis and design exploration for three-dimensional integrated circuits," in *Proc. ASPDAC*, 2009.