nuTAB-BackSpace: Rewriting to Normalize Non-determinism in Post-silicon Debug Traces*

Flavio M. De Paula[1], Alan J. Hu[1], and Amir Nahir[2]

[1] Dept. of Comp. Sci., Univ. of British Columbia, Canada
{depaulfm,ajh}@cs.ubc.ca
[2] IBM Corp. Haifa, Israel
nahir@il.ibm.com

Abstract. A primary challenge in post-silicon debug is the lack of observability of on-chip signals. In 2008, we introduced BackSpace, a new paradigm that uses repeated silicon runs to automatically compute debug traces that lead to an observed buggy state. The original BackSpace, however, required excessive on-chip overhead, so we next developed TAB-BackSpace, which uses only pre-existing on-chip debug hardware to compute an abstract debug trace with very low probability of error. With TAB-BackSpace, we demonstrated root-causing a (previously known) bug on an IBM POWER7 processor, in actual silicon.

The problem with these BackSpace approaches, however, is the need to repeatedly trigger the bug via the exact same execution. In practice, non-determinism makes such exact repetition extremely unlikely. Instead, what typically arises is an intuitively "equivalent" trace that triggers the same bug, but isn't cycle-by-cycle identical. In this paper, we introduce nuTAB-BackSpace to exploit this observation. The user provides rewrite rules to specify which traces should be considered equivalent, and nuTAB-BackSpace uses these rules to make progress in trace computation even in the absence of exact trace matches. We prove that under reasonable assumptions about the rewrite rules, the abstract trace computed by nuTAB-BackSpace is concretizable — i.e., it corresponds to a possible, real chip execution (with the same low possibility of error as TAB-BackSpace). In simulation studies and in FPGA-emulation, nuTAB-BackSpace successfully computes error traces on substantial design examples, where TAB-BackSpace cannot.

1 Introduction

Post-silicon validation/debug is the problem of determining whether the fabricated chip of a new design is correct, and what is wrong if it behaves incorrectly. The problem lies between pre-silicon validation, which searches for design errors in models of the design before fabrication, and VLSI test, which searches for random manufacturing defects on each fabricated chip in high-volume production. Naturally, post-silicon validation/debug inherits characteristics from both,

* Supported in part the Natural Sciences and Engineering Research Council of Canada.

P. Madhusudan and S.A. Seshia (Eds.): CAV 2012, LNCS 7358, pp. 513–531, 2012.

but the differences necessitate novel solutions. Like pre-silicon validation, post-silicon validation focuses on *design errors*. The difference, however, is that the validation is of the actual silicon chip, which is roughly a billion times faster than simulation, can run the real software in the real system at full speed, and exhibits the true (not simulated) electrical and physical properties. Accordingly, post-silicon validation catches numerous bugs that escape pre-silicon validation, due to inadequate coverage, inaccurate models, approximate analyses, and misspecified properties and constraints. Unfortunately, like VLSI test, post-silicon validation shares the problems of limited controllability and observability, as the internal signals on-chip are essentially inaccessible. Test and debug structures can be (and are) added on-chip, but any increase of the chip's area, power, or pins is expensive. These issues make post-silicon debugging extraordinarily challenging. Post-silicon debug currently consumes more than half of the total verification schedule on typical large designs, and the problem is growing worse [1,10].

Post-silicon validation/debug is broad and multi-faceted. To provide context, we briefly survey the overall debug flow and cite some representative research. Note that the post-silicon debug process is iterative, just like any other kind of debugging: at all stages of the process, the debug engineer formulates hypotheses about what might be going wrong, develops a test for the hypotheses, and then formulates new hypotheses based on the results. Because the focus is design errors, debug engineers typically have deep knowledge of the design.

The validation/debug process starts with test planning and stimulus generation: how to thoroughly exercise the die? This is analogous to simulation testbenches in pre-silicon validation, except that controllability/observability are limited to the pins and the test stimuli must be generated *quickly*. Typical tests include booting the OS, running applications, random instruction [17,7], and focused test suites and exercisers for hard-to-verify parts of the design (e.g., [7,2]). To stress electrical bugs[1], these tests are run under a variety of system configurations and operating conditions (frequency, voltage, temperature, etc.).

When testing reveals the presence of a bug, the next step is to get a trace of what happened on the chip when the bug occurred. The challenge is the lack of observability, so the basic techniques are on-chip structures to improve observability, e.g., scan chains [29], trace buffers [27,4], and networks to access signals to record [23,1]. Typically, one can take a snapshot of many/most latches of the design at a single cycle (scan), or record tens or hundreds of signals over a few hundred or thousand cycles (trace buffers), but getting this data off-chip is extremely slow and completely disrupts the test. Accordingly, the debug engineer has to trigger recording at exactly the right moment, and anecdotally, many debug engineers describe this as one of the most time-consuming tasks in

[1] For bugs that create functional errors, it's useful to distinguish between *logical bugs*, which could be replicated pre-silicon in the RTL, and *electrical bugs*, which result from electrical effects such as noise coupling, voltage droop, and timing errors, as some methods apply to only one or the other. This paper handles both. There are also electrical and physical bugs detected post-silicon that are not functional errors, e.g., power consumption, yield, reliability, etc., which we do not consider.

post-silicon debug. Most research supporting this phase of the debug process has focused on selecting signals that provide the best observability (e.g., [20,18,22,6]) harkening back to earlier work on observability for test (e.g., [19]). There is also research on *computing* debug traces: For example, assuming a deterministic test that is short enough to be simulated in its entirety on a deterministic fault-free model of the chip, it is possible to focus the trace buffer only on cycles where electrical errors are likely, relying on simulation to fill-in the fault-free cycles [3,30]. Closer to our work, IFRA [21] eliminates the assumptions of short tests and determinism, allowing a trace to be computed, for example, for a processor booting an operating system. The method works even if the error occurs very rarely, but is only for electrical bugs and is processor-specific. Our work builds on the BackSpace framework [14,15] (described below), which also computes traces of the full-speed silicon running long tests. The framework handles non-determinism and both logical and electrical bugs, but requires bugs to be reasonably repeatable.

Only when bug traces are available can debugging proceed. In a manual debug flow, the debug engineer finally has some insight into what is happening on-chip and can start ruling out possibilities and forming new hypotheses. Research results to support this process include automatically simplifying the bug trace [11,16], and using the trace to localize possible explanations [28,31], and even make layout repairs [10]. All of these methods depend on having traces showing what is happening on-chip leading up to the bug.

This paper focuses on the central task of deriving such debug traces, showing on-chip signals for many cycles leading up to an observed bug or crash. Until the trace is obtained, further debugging is essentially impossible.

1.1 The BackSpace Framework

Our work builds on the BackSpace framework, a novel paradigm that uses repeated silicon runs to automatically compute debug traces that lead to an observed buggy state. The core assumption of BackSpace is that the bring-up tests can be run repeatedly, and the bug being targeted will be at least somewhat repeatable (e.g., with probability $1/n$ for reasonably small n). The methods rely on repetition, which is fast on silicon, to compensate for the lack of observability.

The original BackSpace [14] introduced the basic theory and a proof-of-concept implementation on a small design. The method relied on some on-chip hardware, pre-image computations, and repetition to compute a provably correct trace to the bug. In theory, it solved the problem of computing a trace perfectly: computing arbitrarily long sequences of all signals on-chip, leading up to the bug. However, this perfect solution came with impractical overhead: correctness relied on computing breakpoints, signatures, and pre-images over the entire concrete state of the chip. The hardware overhead was too high to be practical.

Most complex chips, however, already include some on-chip debug hardware. In TAB-BackSpace [15], we flipped the problem around: instead of adding excessive on-chip hardware for a perfect debug solution, we leveraged the BackSpace approach to get much more out of the *already existing* in-silicon debug logic

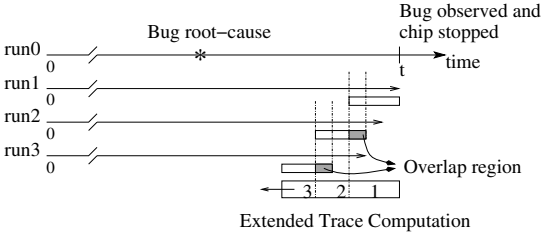

Fig. 1. TAB-BackSpacing. Once the bug is observed, we re-run the chip with trace arrays enabled, i.e., run1; we collect the information from the trace arrays and compute a new set of triggers for the subsequent run (run2); and we iterate these steps, extending the length of the computed trace beyond the trace arrays' depth.

(i.e., trace buffers). Thus, there is no additional hardware cost. TAB-BackSpace achieves the effect of extending the trace buffer arbitrarily far back in time (assuming no spurious traces — more on this below).

Fig. 1 gives an overview of TAB-BackSpace. We assume the trace buffer records until stopped by a trigger. TAB-BackSpace iterates the following:

1. Run the chip until it "crashes" (hits the bug or the programmed breakpoint).
2. Dump out the state of the trace buffer into a file.
3. Select an entry from the trace dump as the new trigger condition, configuring the breakpoint circuitry to stop the chip when it hits this breakpoint on the next run.

The trace-buffer dump of the next run will overlap the most recent trace-dump by some number of cycles f. If all states in the overlapping region agree, we join the new trace-dump to the previous trace-dump, extending the length of the computed trace; if not, we select another state to be the breakpoint and try again. If the length of each trace-dump is m, then after n iterations, we will have computed a trace approximately $n(m - f)$ cycles long (approximate because f may vary between runs). Using TAB-BackSpace, we demonstrated root-causing a (previously known) bug on an IBM POWER7 processor, in actual silicon.

In theory, the weakness of TAB-BackSpace is the possibility of spurious abstract traces. By practical necessity, a trace buffer can record only a tiny fraction of on-chip signals. Therefore, the trace computed is an abstract trace. When two abstract trace dumps agree on the overlap region, TAB-BackSpace joins the two into a longer abstract trace, implicitly assuming that the underlying concrete traces agree as well, which might not be true. Empirically, we showed that by using a reasonably sized overlap region, the possibility of spurious traces could be made very small.

In practice, the real weakness of TAB-BackSpace is the need to repeatedly trigger the bug via the same execution. Non-determinism in the hardware and bring-up environment makes such exact repetition unlikely. The result is that the new trace-dump doesn't completely agree with the previous trace over the entire overlap region, so TAB-BackSpace fails to make progress. Indeed, the

POWER7 result was achieved only by creating an environment that minimized non-determinism: running on bare metal, only one core enabled, and using a specialized post-silicon exerciser [15]. Creating an environment to minimize non-determinism while still triggering a bug is a difficult and time-consuming task.

We have observed, however, that although an *exact* match rarely occurs, what typically happens in practice is that the same bug is triggered by an intuitively "equivalent" trace, that isn't cycle-by-cycle identical. How can we formalize the debug engineer's informal notion of "equivalent"? And how do we extend TAB-BackSpace to correctly account for such user-specified equivalences?

This paper is an answer to those questions. The debug engineer provides rewrite rules to specify which traces should be considered equivalent, and our new algorithm uses those rules to make progress in trace computation even in the absence of exact trace matches. Under reasonable assumptions about the rewrite rules, and about the trace buffer length and signals, we prove that the abstract trace computed by nuTAB-BackSpace is concretizable — i.e., it corresponds to a possible, real chip execution. In simulation studies, we show that nuTAB-BackSpace can indeed compute correct error traces, even when non-determinism renders TAB-BackSpace infeasible. Finally, we demonstrate nuTAB-BackSpace successfully computing error traces on an industrial-size SoC in FPGA-emulation, where TAB-BackSpace cannot.

2 Background

2.1 Trace Buffers

A trace buffer is an on-chip structure for storing limited history of internal events that occur on-chip during full-speed execution. Because of the importance of post-silicon debug, most complex chips are now built with trace buffers.

A typical trace buffer consists of a memory array, organized as a FIFO, perhaps with some simple compression capabilities. A small number (typically tens to a few hundred) of important signals on the chip are routed to the FIFO. The signals routed to the trace buffer must be chosen before the chip is fabricated (although some limited reconfigurability is sometimes provided). The signals can be recorded in the FIFO in real-time as the chip runs, capturing typically a few hundred to a few thousand cycles of history. Control logic allows triggering the starting and stopping of this recording based on the signals that appear, cycle counters, watchdog timers, etc. There must also be some mechanism to read out ("dump") the contents of the trace buffer, for example, by putting the chip into debug mode. Dumping the trace buffer is slow and radically perturb the execution of the chip, so debug methodologies avoid trying to continue an execution after a trace buffer dump.

In this paper, we assume very minimal trace buffer capabilities. We assume the recording can run continuously (the array treated as a circular buffer), and that we can set a breakpoint to stop recording when a specified input signal reaches the trace buffer. The trace buffer can be dumped arbitrarily later. This

gives the effect recording the last m cycles of the trace buffer signals before the chip "stops" at the breakpoint, where m is the length of the trace buffer.

2.2 Abstraction

We will reason about both the signals recorded in the trace buffer as well as the underlying state of the full chip as it runs in actual silicon. Because the signals recorded in the trace buffer are a subset of the total signals on-chip, we can view a state in the trace buffer as an abstraction of the state of the chip.

Formally, we model the full chip on-silicon as a finite-state transition system with state space S_c and (possibly non-deterministic) transition relation $\delta_c \in S_c \times S_c$. This is the *concrete* system. As is typical in model checking [12], we abstract away the inputs and consider only signals on-chip as the state. A *concrete trace* is a finite sequence of concrete states $s_1, \ldots, s_n$ such that $\forall i.\ (s_i, s_{i+1}) \in \delta_c$.

The choice of signals to record in the trace buffer defines an abstraction function $\alpha : S_c \to S_a$ that projects away everything but the chosen signals. S_a is the abstract state space, and the abstract transition relation $\delta_a(s_a, t_a)$ is defined as usual (e.g., [13]): $\exists s_c, t_c\, [\delta_c(s_c, t_c) \wedge s_a = \alpha(s_c) \wedge t_a = \alpha(t_c)]$. An *abstract trace* is a finite sequence of abstract states $s_1, \ldots, s_n$ such that $\forall i.\ (s_i, s_{i+1}) \in \delta_a$.

We lift the abstraction function to traces by abstracting each state of the trace: given a concrete trace σ_c, we get a unique abstract trace $\alpha(\sigma_c)$. In the opposite direction, an abstract trace σ_a is said to be *concretizable* if there exists a concrete trace σ_c such that $\sigma_a = \alpha(\sigma_c)$. Because the abstract transition relation is conservative, not all abstract traces are concretizable; such traces are called *spurious*. In practice, concretizability is a crucial property: a spurious trace doesn't correspond to any possible execution of the real hardware, so it is not only wrong, but it misleads the debug engineer and wastes time.

2.3 Semi-Thue Systems

We will allow the debug engineer to specify intuitive notions of "equivalence" by providing rewrite rules. This provides ease-of-use, expressiveness, and a rich underlying theory that allows efficient checking of equivalent traces. In particular, we treat the debug trace and trace buffer dumps as strings whose alphabet is the abstract state space, and the user-provided rewrite rules produces a string rewriting system AKA a semi-Thue system. Semi-Thue systems have been extensively studied; our presentation is based on [9,5].

Definition 1. *A semi-Thue system is a tuple* $(\Sigma^*, \mathcal{R})$*, where*

- Σ is a finite alphabet,
- $\mathcal{R}$ is a relation on strings from Σ^*, i.e., $R \subseteq \Sigma^* \times \Sigma^*$.

Each element $(l, r) \in R$ is called a rewrite rule, notated as $l \to r$. Rewrite rules can be applied to arbitrary strings as follows: for any $u, v \in \Sigma^*$, $u \to v$ iff there exists an $(l, r) \in R$ such that for some $x, y \in \Sigma^*$, $u = xly$ and $v = xry$. The

notation $\not\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$. We denote the symmetric closure of $\not\rightarrow^*$ by $\not\leftrightarrow^*$, which is an equivalence relation on Σ^*.

The question we need to solve is whether two strings x and y are equivalent, i.e., whether $x \not\leftrightarrow^* y$. This is the standard "word problem" for semi-Thue systems. In general, the problem is undecidable, but under certain restrictions on the rewrite rules, the problem can be solved efficiently by reducing each of x and y to a unique normal form representing the equivalence class.

Definition 2. *A semi-Thue system is Noetherian (terminating) if there is no infinite chain $x_0, x_1, \ldots$ such that for all $i \geq 0$, $x_i \rightarrow x_{i+1}$.*

Noetherianness can be established by finding an ordering function (e.g., string length) that all rewrite rules obey. Under the assumption of Noetherianness, the two properties in the next definition are equivalent:

Definition 3. *A semi-Thue system is **confluent** if for all $w, x, y \in \Sigma^*$, the existence of reductions $w \not\rightarrow^* x$ and $w \not\rightarrow^* y$ implies there exists a $z \in \Sigma^*$ such that $x \not\rightarrow^* z$ and $y \not\rightarrow^* z$. A semi-Thue system is **locally confluent** if for all $w, x, y \in \Sigma^*$, the existence of reductions $w \rightarrow x$ and $w \rightarrow y$ implies there exists a $z \in \Sigma^*$ such that $x \not\rightarrow^* z$ and $y \not\rightarrow^* z$.*

A key result from rewriting theory is that for a rewriting system that is confluent and Noetherian, any object can be reduced to a unique normal form by applying rewrite rules arbitrarily until the object is irreducible (i.e., no rules apply). Furthermore, two objects are equivalent $x \not\leftrightarrow^* y$ iff their unique normal forms are the same. We will use the notation $N(x)$ to denote the unique normal form for any string x.

3 nuTAB-BackSpace

3.1 Formalizing the Intuition

Before describing the nuTAB-BackSpace algorithm, we first need to formalize our assumptions about the user-supplied abstraction and rewrite rules.

The fundamental principle underlying the BackSpace approaches is to use repetition to compensate for the lack of on-chip observability. The fundamental challenge, therefore, is how to determine when a new run of the chip is following "the same" execution as a previous one, so that information from the two physical runs can be combined.

The first technique is the breakpoint mechanism. We never try to combine traces unless the new trace breakpoints (i.e., the hardware reaches a specified state) on a state from the older traces. Because the two traces share an identical state, we are guaranteed that we can combine the two traces at that state and have a valid, longer trace — but the guarantee is only valid at the level of abstraction of the breakpoint state. In the original BackSpace, the breakpoint was concrete, guaranteeing that the algorithm constructed a valid, concrete trace leading to the bug. In TAB-BackSpace and nuTAB-BackSpace, the breakpoint

is only on a partial state, so the guarantee is only that the constructed trace is a legal, but possibly spurious (non-concretizable), abstract trace.

To reduce the possibility of spurious traces, and since a trace buffer provides multiple cycles of history anyway, we therefore insist that not only the breakpoint match, but every abstract state match in a multicycle overlap region between a new trace buffer dump and the previously computed trace. Intuitively, the longer the overlap region we require to match, the less likely that we compute spurious traces. We can formalize the intuition that a large enough overlap eliminates spurious traces as follows:

Definition 4. *Let l_{div} ("divergence length") be the smallest constant such that for all concrete traces $x_1y_1z_1$ and $x_2y_2z_2$ (where the xs, ys, and zs are strings of concrete states), if $\alpha(y_1) = \alpha(y_2)$ and the length $|\alpha(y_1)| > l_{div}$, then $x_1y_1z_2$ and $x_1y_2z_2$ are also valid concrete traces.*

In other words, if two concrete executions share a long enough period of abstracting to the same states, then the future concrete execution is oblivious to what happened before that period, and so the combined abstract trace is not spurious. Note that the divergence length is specific to the design and also to the chosen abstraction function.

Although l_{div} may not always exist (because, for example, the abstraction function might abstract away key information from the concrete traces), in theory, it is straightforward to check whether the length of the overlapping region is longer than l_{div}: let f be the length of the overlapping region. Do there exist two traces $\sigma_1{=}x_1y_1z_1$ and $\sigma_2 = x_2y_2z_2$ such that $|x_i|{=}|z_i|{=}1$, $|y_1|{=}|y_2|{=}f$, $\alpha(y_1){=}\alpha(y_2)$, and either $x_1y_1z_2$ or $x_1y_2z_2$ are not valid traces? If not, we know that $f > l_{div}$. Otherwise, $f \leq l_{div}$. Therefore, all we need is to unroll the design (as in bounded model checking [8]) up to $f + 2$ cycles and check for a witness.

In practice, it may be unrealistic to unroll the design for $f+2$ cycles. However, in [15] and in Section 4.1, we show that we can empirically limit the number of spurious traces. In particular, if we have trace dumps from different concrete executions that match on the overlap region, we dub this a "false match", which is a necessary (but not sufficient) condition for a spurious trace. Our experiments show that false matches are rare when the overlap region is reasonably long.

Indeed, as noted earlier, the problem in practice is not too many matches generating spurious traces, but the lack of exact matches preventing any progress in trace computation. Empirically, however, we have often observed intuitively "equivalent" traces that are not cycle-by-cycle matches, e.g., a trace with slightly different timing, with independent events reordered, etc. These are all differences that could be manipulated via rewriting, so we propose to allow the debug engineer to specify rewrite rules to define what "equivalent" means to them, on a particular design. nuTAB-Backspace will then match overlap regions if they are equivalent under the specified rewriting, rather than requiring an exact match.

Will this idea produce correct traces? Correctness depends on the rewrite rules respecting the semantics of the design. Accordingly, we impose a few restrictions on the rewrite rules. Not surprisingly, we require that the rules be Noetherian and confluent, which allows efficient equivalence checking via reduction to the unique

normal form. To capture the notion that the rewrite rules truly reflect equivalent traces of the underlying concrete chip, we define the concept of concretization preservation:

Definition 5. *Consider a rewrite rule $l \rightarrow r$ on strings of abstract states. The rewrite rule is* ***concretization preserving*** *if for all concrete states x_c and z_c, the concretizability of the abstract state sequence $\alpha(x_c)l\alpha(z_c)$ to a concrete sequence starting with x_c and ending with z_c implies the concretizability of the abstract state sequence $\alpha(x_c)r\alpha(z_c)$ to a concrete sequence starting with x_c and ending with z_c, i.e.:*

$$\forall \textit{concrete states } x_c, z_c \left[\begin{array}{c} (\exists \textit{concrete trace } x_c y_l z_c \ . \ \alpha(y_l) = l) \\ \Rightarrow \\ (\exists \textit{concrete trace } x_c y_r z_c \ . \ \alpha(y_r) = r) \end{array} \right]$$

Obviously, a rewrite rule should be rejected if it breaks concretizability altogether. This definition is slightly stronger in that it requires that a pre-existing concretization be preserved, *mutatis mutandis* the rewriting.

As with l_{div}, in theory, it is straightforward to check whether a rule is concretization preserving. There are a finite number of rewrite rules, $l \rightarrow r$, each of which is finite in length. Does there exist a concrete trace $x_c y_l z_c$ such that $\alpha(y_l) = l$, but where no string y_r exists such that $x_c y_r z_c$ is a concrete trace and $\alpha(y_r) = r$? One could, for example, use bounded model checking to enumerate all x_c and z_c that satisfy the antecedent of the definition, and then use bounded model checking to check that each satisfying x_c and z_c also satisfies the consequent.

In practice, depending on the design and abstraction, this check may also not be realistic. On the other hand, debug engineers have expert design knowledge, so they are capable of defining rewrite rules that are concretization preserving (or close enough for their purposes).

3.2 Algorithm

Algorithm 1 presents the nuTAB-BackSpace procedure: starting from a given crash state and its corresponding trace-buffer, it iteratively computes an arbitrarily long sequence of predecessor abstract states by going backwards in time. This procedure has 4 user-specified parameters: *steps_bound* specifies how many iterations back the algorithm should go; *retries_timeout* limits the amount of search for a new trace dump where the overlapping region with the trace computed so far is equivalent; the *time_bound* is a timeout for each chip-run and is a mechanism to tell whether a chip-run went on a path that does not reproduce the crash-state or buggy-state; and, *lbindex* is the trace buffer's smallest index, which defines a region either for the overlapping (TAB-BackSpace) or the normalization (nuTAB-BackSpace) of two consecutive trace buffers.

This procedure has 2 nested loops. The outer loop, lines 15 – 45, controls the three termination conditions for the algorithm: we reach the user-specified number of iterations; we reach the initial states; or the previous iteration was

unsuccessful. The outer loop is also responsible for joining the new trace buffer dump onto the successful trace computed so far (line 36), and then selecting a new state as the breakpoint for the next iteration. The inner loop, lines (20 – 35), is responsible for controlling the hardware while trying out different candidate-states, s_{cand}, given a *retries_timeout*. The procedure keeps track of time using the subroutine *ElapsedTime*() (passing *reset* as parameter resets the time counter, otherwise it counts the elapsed time since it was last reset). In each loop iteration, the procedure loads s_{cand} into the breakpoint-circuit (line 22), and runs the chip. The objective is to collect a new trace-buffer upon matching s_{cand} and match (after rewriting) it with the previous trace-buffer. If *ResetAndRun*() returns *TRUE* then the breakpoint circuitry matched s_{cand} and we have a new trace-buffer. Otherwise, the chip-run violates the *time_bound* parameter (line 24) because the current run took another path (caused by non-determinism). If the breakpoint occurs, we dump the contents of the trace-buffer for comparison with the trace computed so far. The *NormalizeAndCheck()* subroutine (line 28) computes the unique normal form of the overlapping region of the previously computed trace as well as the new trace dump, as described in Sec. 2.3, and then compares them to check equivalence. If the procedure neither breakpoints nor proves equivalence, *PickState*() (line 33) selects another candidate-state from the previous trace using a round-robin scheme while respecting *lbindex* and the inner loop iterates. The procedure exits the inner loop when either it successfully proves equivalence of the overlapping regions of the two trace-buffers, or this loop has iterated longer than the specified *retries_timeout*.

3.3 Correctness

The main correctness theorem proves that the trace computed by Algorithm 1 is as informative as one could hope: it concretizes to a trace that leads to the actual crash state, using reachable states.

Theorem 1 (Correctness of Trace Computation). *If the rewriting rules are Noetherian, confluent, and concretization preserving, and if the size of all unique normal forms used to prove equivalence of overlapping regions is greater than l_{div}, then the trace produced by Algorithm 1 is concretizable to the suffix of a concrete trace leading from the initial states Q_0 to the crash state s.*

Proof: The proof is by induction on the iteration count i at the bottom of the outer loop. The base case is trivial, as when $i = 0$, the trace is a single trace buffer dump that ends at the crash state. Since this trace dump is taken from the physical chip, it can be concretized to the specific physical execution that occurred on-chip.

In the inductive case, let uy represent the trace computed so far, and let xv represent the new trace dump t_i, with $N(v) = N(u)$. In other words, u and v are the overlap region that has been proven equivalent by rewriting. By construction, x and y are non-empty.

We know that xv is concretizable to a trace with all states reachable from the initial states, because it is taken directly from the hardware. Therefore,

Algorithm 1. Crash State History Computation

input Q_0 : set of initial states,
  (s, t) : crash-state and trace-buffer
  $steps\_bound \in \mathbb{N}^+$: user-specified bound on the number of iterations,
  $retries\_timeout \in \mathbb{N}^+$: user-specified time-bound on retrials,
  $time_{bound}$: user-specified time bound for any chip-run
  $lbindex$: user-specified lower-bound length of normal region;
output $trace$: equivalent sequence of abstract states;
$i := 0$;
// initialize breakpointable candidate-state and current trace-buffer
$i := 0$; $s_{cand} := s$; $t_i := t$;
$trace := (t_i)$; // i.e., initialize trace with current trace-buffer
// initialize variable nindex; nindex gets updated by $PickState()$
// nindex range is $[lbindex, |trace\text{-}buffer|]$
$nindex := lbindex$; $succ\_iteration := TRUE$
while $(i < steps\_bound)\ AND\ (s_{cand} \notin Q_0)\ AND\ (succ\_iteration = TRUE)$ **do**
  $equivalent := FALSE$;
  $matched := FALSE$;
  //Resets retrial elapsed time
  $ElapsedTime(reset)$
  while $(!equivalent)\ AND\ (ElapsedTime(go) \leq retries\_timeout)$ **do**
    // Program the hardware-breakpoint circuitry with s_{cand}
    $LoadHardwareBreakpoint(s_{cand})$;
    // (Re-)run *M'* at full-speed with timeout $time_{bound}$
    $matched := ResetAndRun(time_{bound})$;
    if $matched$ **then**
      // Dump trace-buffer contents t_i
      $t_i := ScanOut()$;
      $equivalent := NormalizeAndCheck(t_i, t_{i-1}, nindex)$;
    end if
    if $(!matched)\ OR\ (!equivalent)$ **then**
      // Pick another state following a round-robin scheme
      // and updates $nindex$
      $s_{cand} := PickState(nindex, t_{i-i})$;
    end if
  end while
  if $equivalent = TRUE$ **then**
    // Accumulate trace
    $OverlapConcatenate(t_i, trace)$;
    // Pick a candidate-state in t_i for the next iteration
    $s_{cand} := PickState(nindex, t_i)$;
    $i := i + 1$;
  else
    $succ\_iteration := FALSE$
  end if
end while
return $trace$;

$xN(v)$ has the same properties, by preservation of concretization. Similarly, uy is concretizable to a trace that leads to the crash state s, by the inductive hypothesis, and therefore, $N(u)y$ is, too, by preservation of concretization. Let $x_c v_c$ be a witness to the concretizability (with additional properties) of $xN(v)$, with $x = \alpha(x_c)$ and $N(v) = \alpha(v_c)$. Similarly, let $u_c y_c$ be a witness to the concretizability of $N(u)y$, with $N(u) = \alpha(u_c)$ and $y = \alpha(y_c)$.

From the hypotheses, $|N(u)| = |N(v)| > l_{div}$, so by the definition of l_{div}, both $x_c u_c y_c$ and $x_c v_c y_c$ are legal concrete traces. By construction, both start at reachable states, and therefore contain all reachable states. And both end at the crash state s. Therefore, either is a witness that the new trace computed by Algorithm 1, $xN(u)y$, is concretizable to the suffix of a concrete trace leading from the initial states to the crash state. ∎

4 Experiments

We present two experiments demonstrating the feasibility of nuTAB-BackSpace. In both, we compare our new method against TAB-BackSpace. We start with a simulation-based evaluation, where we have more controllability and can identify false matches. Then, we evaluate nuTAB-BackSpace on a hardware prototype.

4.1 Simulation-Based Evaluation

We use a router design (henceforth, the "Router"), which is an RTL implementation of a 4x4 routing switch. The Router is typically used by IBM for training new employees with IBM's tools. The Router is a non-trivial design, but also not too complex to be simulated in its entirety. The design has 9958 latches, which is larger than most open-source design examples (e.g., from [25]).

The Router implements a routing policy, which is programmed beforehand in configuration registers. The Router routes incoming packets from four distinct input ports into one of four output ports. The Router recognizes packets in a pre-defined format containing source and destination addresses, payload, and bit-parity. In addition to routing the packets, the Router also checks the validity of incoming packets and rejects bad packets.

To simulate the Router, we use a constrained-simulation environment developed by IBM, using Cadence's Incisive Simulator (with Specman Elite) v.09.20-s016. This proved very helpful when modeling environmental non-determinism.

We claim that when non-determinism cannot be extensively removed from the environment/design, TAB-BackSpace will either fail to produce a trace or will require an excessive number of re-trials. To validate this claim, we (1) set the TAB length to be 50 with no compression; (2) set the TAB width to 75 bits; (3) abstract three of the Router's design blocks onto these 75 bits by using our architectural insight; (4) set a goal of 20 iterations for each "crash" state (setting $steps\_bound = 20$ in Algorithm 1); (5) for each iteration, we set a timeout of 5 hours to allow for a large number of re-trials when necessary (typically, each simulation-run takes about 10min); (6) and, randomly choose 30 abstract-states as our "crash" states.

To make sure the traces produced from these "crash" states are independent, we first generate a lengthy "crash" trace via constrained-random simulation. Then, we randomly choose 30 abstract-states, a_i, from this trace, with the following properties (Q_0 is the initial set of states): $\forall i, j.a_i, a_j \notin Q_0 \wedge (a_i \neq a_j) \wedge |i\text{-}j| > 1000$. These properties guarantee that the "crash" states in this experiment are far enough apart so that the computed traces are distinct. In other words, since *steps_bound* = 20 and each trace dump has 50 cycles, even if the overlap between two consecutive trace-dumps were one single cycle, the total number of cycles for each trace would be 20*49+1 < 1000, which is smaller than the distance between two crash states.

In [15], we have empirically shown that, for the Router, an overlap of 30 cycles or more would most likely prevent false matches. Thus, in these experiments, we use 30 cycles as the lower bound for the overlapping region of two consecutive trace-buffers.

We use the constrained-simulation environment to simulate non-determinism. This environment provides many parameters to make each simulation run very different from one another. However, we want to control the non-determinism so that we have a fair comparison between TAB-BackSpace and nuTAB-BackSpace. Thus, we simulate non-determinism only affecting the delays on packet arrivals (a real scenario encountered in bring-up labs). We accomplish this behavior by changing the simulation environment such that it always uses a fixed random seed for everything except packet generation. For packet generation, we use an external and independent random generator to add different delays between packets in each run.

We need to provide a set of rules for normalizing the non-determinism during nuTAB-BackSpace simulations. In practice, defining rewrite rules will follow the same iterative process as debugging. In this case, we had worked with this design in [15] and had a good understanding of it. We had been unable to TAB-BackSpace the Router and this was due to non-determinism in the inter-packet delays, and so, in these experiments, we develop a set of rules to normalize such effects of non-determinism.

Recall that our abstract-model is based on 3 design blocks. In particular, three sets of signals in this abstract-model represent the same state-machine that is replicated across the 3 design blocks. In Fig. 2, we present one such machine. Notice that 6 states have self-loops, namely *idle, wait_buff, wait_data, wait_idle, wait_route, get_rest*. In [15], we observed that non-determinism in the inter-packet delays affects all these states with self-loop edges (e.g., a long delay might cause an input port to remain in *idle* or *wait_data* states for some number of cycles). The exception is the state *get_rest*. In this state, the Router processes incoming packets without interruption, that is, the Router does not accept partial packets. Thus, to normalize non-determinism, we define a rewriting system, $Router_{RS}(\Sigma^*, R)$.

Let $\text{Proj}(\cdot)_{sm}$ be a projection function that takes in an abstract-state, a, and projects it onto the set of bits representing the state machine from Fig. 2 and let $P = \{idle, wait\_buff, wait\_data, wait\_route, wait\_idle\}$. Now, we can define R as follows:

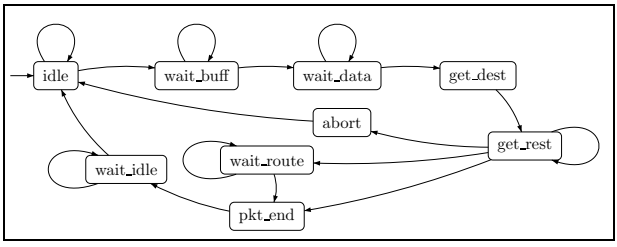

Fig. 2. Router's Internal Packet-Processing State-Machine

$$\forall a.Proj(a)_{sm} \in P \ . \ aa \rightarrow a \tag{1}$$

Thus, this rewriting rule creates an equivalence class of traces, treating traces with different numbers of repetitions of certain states as similar.

Before we can apply nuTAB-BackSpace, we need to show that the rewriting system, $Router_{RS}(\Sigma^*, R)$, is Noetherian, confluent, and concretization-preserving. First, note that $Router_{RS}(\Sigma^*, R)$ is a length-reducing rewriting system, and so it is Noetherian. Next, note that Eq. 1 contains 5 rules (or technically, rule schema), and no two rules have overlapping left-hand sides. The only possible critical pairs arise from rewriting a string of the form *aaa* into *aa* with two different applications of a single rewrite rule. Obviously, these are locally confluent. Thus, the entire rewrite system, $Router_{RS}(\Sigma^*, R)$, is confluent. For concretization preservation, we have only an informal argument. Based on our knowledge of the design, any execution of the system that goes through a state that projects to P can spend more or less time in that state, without impacting the rest of the execution. This is exactly the property that concretization preservation captures. In contrast, when the state machine is in the state *get_rest*, the underlying concrete state tracks the number of cycles for the packet, so a rule that changed the number of *get_rest* cycles would not be concretization-preserving.

We deem a TAB-BackSpace iteration successful when two consecutive trace-buffers agree cycle-by-cycle over all 30 cycles, i.e., a full-overlap match; and a nuTAB-BackSpace is successful when the normalization-region (30 cycles or more) from the consecutive trace-buffers are equivalent under $Router_{RS}(\Sigma^*, R)$.

The experiments are successful. In Table 1, we show that nuTAB-BackSpace computes, for all crash-states, longer traces than TAB-BackSpace. Moreover, TAB-BackSpace could not compute even one iteration for 1/3 of the cases. And, when TAB-BackSpace is comparable to nuTAB-BackSpace with respect to the number of successful iterations (e.g., crash states 19, 27-30), nuTAB-BackSpace requires, for the most cases, an order of magnitude smaller number of runs.

4.2 Case Study: The Leon3 SoC Hardware Prototype

To demonstrate that nuTAB-BackSpace is feasible in practice, we emulate on an FPGA board [26] a System-on-Chip (SoC) including software. We use a Leon3-based SoC [24] as our hardware-prototype. This prototype is a full-blown SoC,

Table 1. TAB-BackSpace vs nuTAB-BackSpace Experiments. We use the same crash states. "# of Successful Iterations" is the number of iterations before timing out or reaching the set limit of 20. The timeout per iteration was chosen to be 5h. Each simulation run averages 10 minutes. "# of Chip Runs" is the total number of iterations plus the number of retries. Because "# of Chip Runs" is an aggregate, when the number of iterations for TAB is smaller than the number of iterations for nuTAB, the number of nuTAB runs may be greater than TAB runs (e.g., 1, 13-15). Crash states with a † are states that nuTAB-BackSpace computed all 20 iterations, but somewhere during the computation it deviated from the "expected" trace (in simulation, we can determine if the run reached the specified "crash" state). Therefore, these might be spurious traces. We suspect that, at some iteration, the normalized region of two different traces was too small to discriminate them.

Crash State	# of Successful Iterations		# of Chip Runs	
	TAB	nuTAB	TAB	nuTAB
1	0	11	62	143
†2	0	20	339	21
3	0	20	67	52
4	0	20	77	75
5	0	20	79	24
6	0	20	93	27
7	0	20	283	28
†8	0	20	128	20
†9	0	20	58	23
10	0	20	342	20
11	1	20	173	57
12	2	7	204	137
13	3	15	399	536
14	3	20	134	144
15	4	19	270	661

Crash State	# of Successful Iterations		# of Chip Runs	
	TAB	nuTAB	TAB	nuTAB
†16	4	20	112	27
17	5	20	157	28
18	5	20	199	41
19	6	6	120	58
†20	6	20	60	26
21	6	20	181	24
22	6	20	788	24
†23	7	20	568	22
24	12	20	251	27
25	12	20	308	25
26	15	20	534	20
27	20	20	282	28
28	20	20	403	29
29	20	20	463	52
30	20	20	726	26

with a SPARC V8 compatible core, AMBA bus, video, DDR2, Ethernet, i2c, and keyboard and mouse controllers. This SoC also has built-in debug features that can be enabled. In particular, we enable the provided trace buffer, LOGAN, but with a minimal configuration. The LOGAN has no signal compression. The signals we monitor are a combination of AMBA bus signals and some signals of the SPARC V8's execution-pipeline-stage, totaling 134 signals.

Since one of the goals of demonstrating nuTAB-BackSpace on a hardware-prototype is to show that it works in a real (or as realistic as possible) debugging environment, we run non-trivial software on the Leon3. In our experiments, we are booting Linux (Linux Kernel 2.6.21).

Our debug scenario is as follows: while booting Linux, we want to derive the sequence of CPU and bus operations leading to the kernel's function *start_kernel*. Thus, *start_kernel* is our "crash" state. The boot sequence up to this "crash" state is more than 20 million cycles deep. Simulating it with a logic simulator is impractical given this depth. Similarly, model checking it is infeasible.

The first experiment is to try TAB-BackSpace. We follow the same steps as Algorithm 1. The main difference is that instead of normalizing the extracted trace, we try to find an exact match on the overlap region between the current and previous traces. We set an address within *start_kernel* function as our breakpoint and run the chip; when it breakpoints, we extract a trace. From that trace, we pick a trace-buffer entry as our new crash-state and repeat. In our experiments, we set 2 hours as our retry timeout limit. The result of these experiments is a total of 207 chip runs, all of which breakpoint successfully, but none overlap cycle-by-cycle. In other words, we cannot TAB-BackSpace at all because, at each run, non-determinism changes the path the chip takes and so the probability of an exact match is too low.

The next experiment is to try nuTAB-BackSpace using the same scenario as before. However, we need to define the rewrite rules first. In this case, the SoC was built entirely from third-party IP, so our learning process was from the documentation and trace buffer dumps from the actual system running. Studying the trace buffer dumps, we observed that sometimes entire trace-buffers might have not a single video-controller transaction. Also, we noted that nullified instructions, although they vary from run to run, do not affect overall functionality of a system run. Therefore, for this debug scenario, we hypothesize that traces may have video-controller activity occurring at essentially arbitrary times, and that nullified instructions can be ignored. From our understanding of the design, we can create rewrite-rules easily to formalize the hypotheses and test them. (If our hypotheses produced uninterested traces, we would start again with a new hypothesis, creating new rewrite rules to try.)

We define the rewrite rules using the same notation as we used for the Router. Let $\text{Proj}(\cdot)_{ahbm}$ and $\text{Proj}(\cdot)_{inst}$ be two projection functions that map abstract-states, a, onto the subset of AMBA signals, which identify the current bus-master and onto the subset of signals from the CPU that define whether an instruction has been nullified. We can define R as follows:

$$\forall a.Proj(a)_{ahbm} = 0x3\ .\ a \rightarrow \epsilon \tag{2}$$

$$\forall a.Proj(a)_{inst} = annul\ .\ a \rightarrow \epsilon \tag{3}$$

The rewrite rules ignore AMBA bus transactions from the video-controller and states where instructions have been nullified in the CPU's execution pipeline stage. (Note that the ignored cycles do not get deleted from the generated trace — the rewriting is solely to establish equivalence on the overlap region. The generated trace will always consist of actual states taken from trace buffer dumps.)

As in Subsection 4.1, we need to show that $Leon3_{RS}(\Sigma^*, R)$ is Noetherian, confluent, and concretization-preserving. As before, the system is length-reducing, and hence Noetherian. No two rules have an overlapping left-hand side. Consequently, there are no critical pairs, so $Leon3_{RS}(\Sigma^*, R)$ is locally confluent. The argument for concretization preservation is again based on insight into the design. The video controller bus transactions are irrelevant to the boot sequence

Table 2. nuTAB-BackSpace on Leon3. *Trace-Buffer Length* is the physical depth of the trace-buffer. Since we do not use compression, its depth is fixed. *Normalization-Region Length* is the number of cycles in the current trace-buffer that we normalize and use as a reference for the next trace-buffer. *New Cycles* is the number of new states present in the current trace-buffer.

Trace #	Trace-Buffer Length	Normalization Region Length	Normalized Length	New Cycles	Accumulated new cycles
1	1024	904	354	1024	1024
2	1024	519	137	384	1408
3	1024	781	133	241	1649
4	1024	680	168	514	2163
5	1024	709	168	348	2511
6	1024	892	141	45	2556
7	1024	–	–	398	2954

and can be arbitrarily ignored.[2] Similarly, nullified instructions have no effect on the (bus-level) debugging process, so they can be safely ignored as well. Any concrete execution trace which has these ignorable states corresponds to a concrete execution trace where those states have been deleted.

We show the results in Table 2. We iterated 7 times, resulting in a trace more than 2.5x the length of a single trace-buffer. Unlike TAB-BackSpace, the new technique handles the non-determinism, computing an abstract trace based on the trace-buffer signals.

5 Conclusion and Future Work

We have presented nuTAB-BackSpace, a novel technique to compute post-silicon debug traces in the presence of non-determinism. We exploit the observation that traces that are not cycle-by-cycle equal still share similarities from the debug engineer's point-of-view. We let the user provide rewrite rules, and under some reasonable assumptions, we prove that nuTAB-BackSpace computes an abstract trace that concretizes to a trace that is reachable and leads to the crash state. We have demonstrated the effectiveness of nuTAB-BackSpace both in simulation and in hardware, computing abstract traces even when TAB-BackSpace cannot.

Increasingly, complex chips have many clock-domains and even completely asynchronous domains. Capturing traces on these designs and reasoning about them is a major challenge. We believe nuTAB-BackSpace holds promise for this problem, and this is the direct line of future work.

[2] Technically, ignoring video controller transactions is not truly concretization preserving, since any real concrete trace *will* have the occasional video transaction, whose timing is determined by state hidden in the video controller and the external video hardware. What the rewrite rule is really specifying is that that hidden state is irrelevant for the current debugging scenario. If we were debugging some video controller timing interaction, we would use different rewrite rules.

References

1. Abramovici, M., Bradley, P., Dwarakanath, K., Levin, P., Memmi, G., Miller, D.: A Reconfigurable Design-for-Debug Infrastructure for SoCs. In: DAC. IEEE (2006)
2. Adir, A., Golubev, M., Landa, S., Nahir, A., Shurek, G., Sokhin, V., Ziv, A.: Threadmill: a post-silicon exerciser for multi-threaded processors. In: DAC. IEEE (2011)
3. Anis, E., Nicolici, N.: Low Cost Debug Architecture using Lossy Compression for Silicon Debug. In: DATE. IEEE (2007)
4. ARM. Embedded Trace Macrocell Architecture Specification. Trace and Debug. ARM (2007), Ref: IHI00140
5. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge Univ. Press (1998)
6. Basu, K., Mishra, P., Patra, P.: Efficient combination of trace and scan signals for post silicon validation and debug. In: International Test Conference (ITC 2011). IEEE (2011)
7. Bentley, B., Gray, R.: Validating the Intel Pentium 4 processor. Intel Technology Journal (Quarter 1, 2001)
8. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
9. Book, R.V., Otto, F.: String-Rewriting Systems. Springer (1993)
10. Chang, K.H., Markov, I.L., Bertacco, V.: Automating Post-Silicon Debugging and Repair. In: ICCAD (2007)
11. Chang, K.H., Bertacco, V., Markov, I.L.: Simulation-Based Bug Trace Minimization With BMC-Based Refinement. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 26(1), 152–165 (2007)
12. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM TOPLAS 8(2), 244–263 (1986)
13. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. In: POPL, pp. 343–354 (1992)
14. de Paula, F.M., Gort, M., Hu, A.J., Wilton, S.J.E., Yang, J.: BackSpace: formal analysis for post-silicon debug. In: FMCAD. IEEE (2008)
15. de Paula, F.M., Nahir, A., Nevo, Z., Orni, A., Hu, A.J.: TAB-BackSpace: Unlimited-length trace buffers with zero additional on-chip overhead. In: DAC. IEEE (2011)
16. Hong, T., Li, Y., Park, S.B., Mui, D., Lin, D., Kaleq, Z.A., Hakim, N., Naeimi, H., Gardner, D.S., Mitra, S.: QED: Quick Error Detection tests for effective post-silicon validation. In: International Test Conference (ITC). IEEE (2010)
17. Klug, H.P.: Microprocessor testing by instruction sequences derived from random patterns. In: International Test Conference (ITC). IEEE (1988)
18. Ko, H.F., Nicolici, N.: Algorithms for State Restoration and Trace-Signal Selection for Data Acquisition in Silicon Debug. IEEE TCAD 28(2), 285–297 (2009)
19. Lee, D.H., Reddy, S.M.: On Determining Scan Flip-Flops in Partial-Scan Designs. In: IEEE International Computer-Aided Design. Digest of Technical Papers, pp. 322–325. IEEE International (November 1990)
20. Park, S., Yang, S., Cho, S.: Optimal State Assignment Technique for Partial Scan Designs. Electronics Letters 36(18), 1527–1529 (2000)

21. Park, S.B., Mitra, S.: IFRA: Instruction Footprint Recording and Analysis for Post-Silicon Bug Localization in Processors. In: DAC. IEEE (2008)
22. Prabhakar, S., Hsiao, M.: Using Non-trivial Logic Implications for Trace Buffer-Based Silicon Debug. In: Asian Test Symposium. IEEE (2009)
23. Quinton, B.R., Wilton, S.J.E.: Concentrator Access Networks for Programmable Logic Cores on SoCs. In: Int'l. Symp. on Circuits and Systems. IEEE (2005)
24. Web Reference, `http://www.gaisler.com`
25. Web Reference, `http://www.opencores.org`
26. Web Reference, `http://www.xilinx.com/univ/xupv5-lx110t.html`
27. Riley, M., Chelstrom, N., Genden, M., Sawamura, S.: Debug of the CELL Processor: Moving the Lab into Silicon. In: International Test Conference. IEEE (2006)
28. Safarpour, S., Mangassarian, H., Veneris, A.G., Liffiton, M.H., Sakallah, K.A.: Improved Design Debugging Using Maximum Satisfiability. In: Formal Methods in Computer-Aided Design. IEEE (2007)
29. Williams, M.J.Y., Angell, J.B.: Enhancing Testability of Large-Scale Integrated Circuits via Test Points and Additional Logic. IEEE TC C-22(1), 46–60 (1973)
30. Yang, J.S., Touba, N.A.: Expanding Trace Buffer Observation Window for In-System Silicon Debug through Selective Capture. In: VLSI Test Symposium 2008. IEEE (2008)
31. Zhu, C.S., Weissenbacher, G., Malik, S.: Post-silicon fault localisation using maximum satisfiability and backbones. In: Formal Methods in Computer-Aided Design (FMCAD). IEEE (2011)