TAB-BackSpace: Unlimited-Length Trace Buffers with Zero Additional On-Chip Overhead

Flavio M. de Paula[†], Amir Nahir[‡], Ziv Nevo[‡], Avigail Orni[‡], Alan J. Hu[†]
[†]Dept. of Computer Science, Univ. of British Columbia, Canada
[‡] IBM Haifa Research Lab, Haifa, Israel
{depaulfm,ajh}@cs.ubc.ca, {nahir,nevo,ornia}@il.ibm.com

ABSTRACT

This paper presents TAB-BackSpace, our novel scheme to provide the effect of an unlimited-length trace buffer with no on-chip overhead beyond the existing debug logic. We present the theoretical foundation of our work, simulation studies on how we reduce the possibility of computing an erroneous trace, and results from the bring-up lab on real silicon of an IBM POWER7 processor, where TAB-BackSpace computes almost a thousand additional cycles of trace buffer information without any additional on-chip overhead.

Categories and Subject Descriptors

R.12.8 [**Testing**]: Silicon debug and diagnosis, post-silicon design validation.; B.7.2 [**Design Aids**]: Verification.

General Terms

Design, Theory, Verification.

Keywords

Post-silicon debug, validation, design for debug.

1. INTRODUCTION

Post-silicon debug is the problem of determining what's wrong — specifically, finding *design errors* rather than random manufacturing defects — when the fabricated chip of a new design behaves incorrectly. With increasing design complexity and integration, the problem has become a bottleneck in the product development cycle: greater complexity means more bugs escape into silicon, and greater integration means decreased observability, making debugging much more difficult.

The general problem of post-silicon debug is broad and multi-faceted, spurring a diverse variety of research. For example, consider three different aspects of post-silicon debug: lack of observability, bug localization, and bug rectification. Quinton and Wilton [11] and Abramovici et al. [1] address the lack of observability with reconfigurable architectures, so that monitoring on-chip signals is more flexible. Simply embedding a large reconfigurable logic-block would render the entire design unroutable, so they develop novel reconfigurable architectures, distributed across the chip. An example of bug localization is Park and Mitra's IFRA [10], which is a processor-specific technique for electrical bugs. They discover that there are early signs a chip is about to crash, and so they tap into these early signs to start recording some selected signals. The recorded data is then used to rebuild (off-line) the architectural state of the processor facilitating the localization of bugs. Chang et al. [3] provide an example of bug rectification. Their algorithms are layout-aware to automatically repair electrical bugs while preserving the functional correctness of the circuit being debugged.

In this paper, we focus on yet another, fundamental task: getting an execution trace of on-chip signals for many cycles leading up to an observed bug or crash. Until such a trace is obtained, further debugging is essentially impossible, as there is no way to know what happened on the chip.

Because of the critical importance of these traces, almost all chips have some debug logic to facilitate deriving them. For example, the same scan chains [13] for manufacturing test can be used to get a single-cycle snapshot of the state of many on-chip signals. However, this process is slow, so these snapshots can be taken only rarely during a chip's execution. Furthermore, stopping the chip to take a scan dump disturbs the chip's interaction with its environment, potentially changing or obscuring buggy behavior. To compensate for the single-cycle and disruptive nature of scan dumps, complex chips often include "trace buffers" or "on-chip logic analyzers" (e.g.,[2, 12]): a limited number of the most important on-chip signals are routed to and recorded in a FIFO, with some mechanism to trigger starting and stopping of recording. These allow recording a multi-cycle trace of internal signals, while the chip is running at full speed. Unfortunately, because of the die area overhead of the trace buffer, the number of cycles of history that can be stored is small. In practice, considerable ingenuity, persistence, and luck are required to trigger scan dumps or trace buffer recordings at exactly the right times to observe the correct signals just before a bug manifests itself. The on-chip debug logic helps a lot, but obtaining debugging traces is still an exceedingly challenging problem.

De Paula et al. [5] proposed BackSpace, a novel theoretical framework for attacking this problem. In theory, BackSpace solves the problem perfectly: it can compute arbitrarily long sequences of all signals on-chip, leading up to the bug. However, this perfection comes at a terrible price: BackSpace requires formal pre-image computations that can blow-up for complex chips, and the on-chip overhead for the BackSpace debug hardware is prohibitive.

This paper flips BackSpace around: instead of adding excessive on-chip hardware for a perfect debug solution, we leverage the BackSpace insights to get much more out of the *already existing* in-silicon debug logic (e.g., trace buffers). Thus, there is *no additional* hardware cost. What we achieve is the effect of extending the trace buffer arbitrarily far back in time, i.e., an effectively unlimited length trace buffer (assuming no spurious traces — See Sec. 3.).

In the next section, we present our new approach, which we call TAB-BackSpace (for Trace Array Buffer BackSpace). The central difficulty is the risk of computing an incorrect trace, due to the partial information in the trace buffer. In Section 3, we use simulation studies to assess the effectiveness of our techniques to reduce this risk. Finally, in Section 4, we present our results applying TAB-BackSpace on real silicon: an IBM POWER7 chip in the bring-up lab. Our results show that the technique really does work — we compute almost a thousand cycles of extra trace buffer information, using only the existing debug logic.

2. ALGORITHM

We build up to our TAB-BackSpace algorithm in steps. First, we review the original BackSpace algorithm. Then, we develop a theory of BackSpace with abstraction, which provides the framework for the using the partial information from trace buffers and introduces the danger of spurious traces. Finally, we introduce TAB-BackSpace, including how we suppress spurious traces.

2.1 BackSpace Review

BackSpace [5] considers the entire chip-under-debug as a single state machine. A *state* of the system, therefore, consists of every bit on this chip: all latches, register files, arrays, etc.

The original BackSpace assumes the following:

- It is possible to recover the state of the chip when an error has occurred (e.g. using scan chains).
- Since the focus is on *design errors*, the silicon is assumed to implement the RTL correctly.
- The test that stimulates the bug in silicon can be run repeatedly, and the bug being targeted will re-occur reasonably often. Thus, BackSpace can handle nondeterminism/randomness (e.g., multi-clock domains, input delays), as long as the bug isn't extremely rare.
- Related to the preceding point, because BackSpace will repeatedly re-run the test, each run can't take too long (on the physical chip), e.g., several minutes per run is fine; several days per run is too long. Given the speed of the physical silicon, allowable tests can be tens or hundreds of billions of cycles long.

The BackSpace framework adds additional debug logic to the chip: a signature that saves some history information but otherwise has no functional effect on the chip's behavior, and a programmable breakpoint mechanism that allows for "crashing" the chip when it reaches a specified state. Given these, the approach repeats the following steps

1. Run the chip (on the actual silicon) until it crashes or exhibits the bug. This could be an actual crash or hitting the programmed breakpoint.
2. Scan out the full crash state, including the signature.
3. Using formal analysis of the corresponding RTL (i.e., off-chip), compute the pre-image (possible predecessors) of the crash state. The signature helps reduce the size of the pre-image set.
4. Repeatedly try setting each of the states in the pre-image as the new breakpoint, and re-running the same test on the silicon, until we have a run that hits the chosen breakpoint. Add that state to the history trace we are computing.

until enough of a history trace to debug the design have been computed (or Step 3 fails). Each iteration of the loop goes back one cycle. The main BackSpace result is that the computed trace is provably a valid and correct trace leading the chip to the crash state. Clearly, this is a major advantage over existing trial-and-error methods of trying to guess when to take a scan or trace buffer snapshot. The key insights are to work backwards from the targeted crash state, re-run the failing test repeatedly under automatic control, and use a breakpoint mechanism to inductively extend the computed trace back in time.

BackSpace has two major flaws, however. First is the reliance on a formal verification engine to compute pre-images, which does not scale to large designs: BackSpace requires computing a pre-image over the *entire design*. The other major flaw is the excessive hardware overhead. For example, in their experiments, the BackSpace authors typically used 30%-40% of the total design state as the signature. A full programmable breakpoint circuit would add another 100% overhead of the total design state. Later optimizations reduced the breakpoint overhead to around 2% [6], but by any analysis, the overall overhead (say, 30% plus 2% plus interconnect) is still unacceptably huge.

2.2 Abstract BackSpace

The root cause of BackSpace's excessive overhead is the requirement that the entire state of the chip be included in the analysis. What if we make a more realistic assumption that only a small subset of all on-chip state can be monitored, recorded, and used for breakpoints?

Considering only a fraction of the state bits shifts the problem into the standard theoretical framework of formal verification with *abstraction* (e.g., [4]). All of the state bits on the chip are considered to be either invisible or visible. (This use of "visible" is from the theory of abstraction, and does not imply that the signal can be observed off-chip. It is simply a name for a small subset of the state bits.) The original state machine is called the *concrete* system, and the concrete state includes all state bits, both visible and invisible (as in the original BackSpace). By projecting away the invisible state bits, we create the *abstract* state machine, whose state consists of only the visible state bits. This is formalized by an abstraction function α that maps from concrete to abstract states. Because there are many fewer state bits, the abstract state machine is much smaller and easier to analyze.

To eliminate the risk of missing bugs, the abstraction is generally done conservatively: an abstract state a_1 is allowed to transition to abstract state a_2 if there exist any two concrete states c_1 and c_2 such that c_1 transitions to c_2 and $a_1 = \alpha(c_1)$ and $a_2 = \alpha(c_2)$. Accordingly, any trace of the concrete system corresponds to a trace of the abstract system, so if there exists a concrete execution that leads to a crash, we would ideally like to compute the corresponding abstract trace. (Of course, ideally, we would like to compute the full concrete trace, as was done in the original BackSpace, but if only the abstract state is being monitored and analyzed, we can't hope to do so.)

Let us now consider how the BackSpace algorithm can be lifted to apply to an abstract state machine. A small subset of the state bits are designated as "visible": these are state bits of the abstract machine. They are the only ones that need to be scannable and breakpointable, vastly reducing the hardware overhead. The pre-image computation is done on the abstract transitions, which reduces the computational cost, yet guarantees that we won't lose any possible concrete behavior. The same correctness invariant for full BackSpace still applies — the trace computed is provably a valid and correct trace..., but of the *abstract* state machine.

Therein lies the downside of abstraction (for BackSpace and for formal analysis in general). Every concrete trace maps to an abstract trace, and that abstract trace is what we seek. However, there can be abstract traces that do not correspond to any concrete traces — these are called *spurious* traces. There are two sources of spurious transitions in the abstract BackSpace algorithm:

1. Because the abstract transition relation is conservative, the abstract pre-image can include states that do not correspond to any concrete transition to the crash state. If the chip reaches any of those states, we add a spurious abstract transition to the abstract trace.

2. Because the breakpoint is done on the abstract state, when we re-run the chip, we may breakpoint at the wrong time, or (in the presence of non-determinism) on a completely different trace, because a wrong concrete state might map to the same abstract state as the correct concrete state. We call this situation a *false match*.

The challenge is to minimize the risk of spurious traces.

2.3 TAB-BackSpace

For TAB-BackSpace, our goal was to leverage the underlying insights of BackSpace, but use existing on-chip debug hardware and no pre-image computation. Accordingly, we have to make some assumptions about what is available. We assume a trace buffer that records a set of signals (In this paper, we assume this set of signals have already been determined via some automatic framework, e.g., [9, 7], or using architectural insights from the chip designers). This recording must be able to run continuously (e.g., trace buffers are usually implemented as circular buffers). And it must be possible to set a breakpoint to stop recording when the circuit reaches a specific value on the trace buffer input signals. These are minimum requirements; our method can be improved if some of these are better, e.g., if we can set multi-cycle breakpoints.

To make things more concrete, we base our experiments on a well documented post-silicon debug infrastructure: the debug architecture used on IBM's Cell processor[12]. We depict the debug architecture in Fig. 1. This architecture provides many debugging features, but for our purposes, we use only the minimum requirements: trace buffer recording, and the breakpoint (trigger) capability.

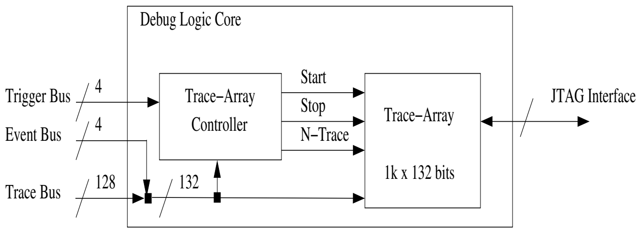

Figure 1: IBM's Cell Processor Debug Logic Core (DLC) High-Level Block Diagram. The DLC has 3 inputs: the trigger bus, carrying signals used to control the trace array; the event bus, carrying additional status signals that may be stored in the trace array; and the trace bus, carrying pre-selected signals to be stored in the trace array. The trace array controller (TAC) uses signals from both the trigger and the trace buses to control the recording of information (*start/stop* recording, and *N-trace* for recording N consecutive cycles).

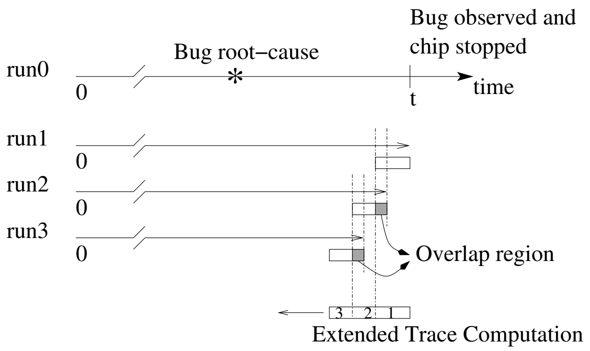

Figure 2: TAB-BackSpacing. Once the bug is observed, we re-run the chip with trace arrays enabled, i.e., run1; we collect the information from the trace arrays and compute a new set of triggers for the subsequent run (run2); and we iterate these steps, extending the length of the computed trace beyond the trace arrays' depth.

This architecture has been typically used by the lab engineer guessing when to start/stop recording information into the trace-array. However, finding the right time window to capture the chip's partial state information is one of the most time-consuming tasks faced in post-silicon debug[1]. With TAB-BackSpace, we will eliminate this problem.

Fig. 2 gives an overview of TAB-BackSpace. We assume the trace buffers are always recording until stopped by a trigger. TAB-BackSpace iterates the following steps:

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.

[1]Personal communication with Jim Bishop (IBM-US), February 2010.

Depending on how the trigger condition is configured, the trace-dump of the next run will overlap the most recent trace-dump by some number of cycles f. If the length of each trace-dump is m, then, after the n^{th} run we have computed a trace of length approximately $n(m-f)$ cycles long. (This is approximate because of non-determinism, and because in practice, f may vary from run to run.)

When comparing TAB-BackSpace to the preceding algorithms, we note the following:

- Like BackSpace, because it works backwards from the crash state, TAB-BackSpace eliminates manual guesswork about when to trigger state recording for the trace arrays. Completely automatically, it computes an arbitrarily long trace dump.
- Unlike BackSpace, because trace buffers typically have many cycles of history, we can BackSpace many cycles on each iteration, gradually gluing together entire trace buffer dumps, instead of individual states.
- Like BackSpace, we rely on repetition to compensate for lack of observability. Hence, we need the same assumption that the bug appears somewhat repeatably. Therefore, like BackSpace, we can handle some non-determinism/randomness in the system behavior.
- Unlike BackSpace, there is no pre-image computation. Instead, we use the fact that the trace buffer records actual history of the chip. This completely eliminates a major computational bottleneck of BackSpace. Furthermore, this also eliminates the assumption that the silicon matches the RTL; we can still compute a trace.
- Because we are using only the small set of important signals that reach the trace buffer, TAB-BackSpace is computing an abstract trace. We therefore have the low overhead of an abstraction-based computation. In fact, because we are using pre-existing debug hardware, there is no (additional) on-chip overhead at all.
- However, because we are computing an abstract trace, there is the risk of spurious traces. Note that since there is no pre-image computation, one source of spurious transitions in Abstract BackSpace is completely eliminated. The only danger is false matches.

How do we suppress false matches? The key is that since a trace buffer stores *multiple cycles* of history, we insist on matching not only the breakpoint, but multiple cycles of overlap between successive trace dumps. If we choose several cycles of overlap f, and insist that all f cycles must match, the risk of a false match ought to be very low. In particular, there are two ways for the algorithm to fail:

1. The abstract breakpoint has a false match, but the overlap region does not match. In other words, the algorithm knows that it has breakpointed at the wrong place. This can be solved by choosing a different abstract state from the previous trace dump as a breakpoint, or, if the debug hardware allows it, by programming the breakpoint circuitry to skip some number of matches before breakpointing (as is done, e.g., in [6]).

2. The abstract breakpoint has a false match, but so does the entire overlap region. This is the dangerous case that must be suppressed as much as possible.

The correctness of TAB-BackSpace relies on forcing the second risk to be zero or very near zero. In the next section, we investigate this issue in detail.

3. SUPPRESSING SPURIOUS TRACES

It's tempting to make analytical models of the probability of false matches. Unfortunately, very little can be said, because the abstract states are not random states, but the result of an abstraction function. Given a really bad abstraction function (e.g., one that focuses on irrelevant bits whose values seldom change), there will be many false matches. The only analytical claims that are solid are that (1) increasing the number of cycles of overlap does not increase the risk of a false match, and (2) if the abstract states are chosen so that that any two concrete states c_1 and c_1' that map to the same abstract states always diverge within k cycles to two concrete states c_2 and c_2' that do not map to the same abstract states, then choosing an overlap $f > k$ guarantees no false matches.

Instead, we show experimentally that the risk of false matches can be made zero or near zero in practice. In particular, we measure the frequency of undetected false matches that will generate a spurious transition: when the abstract breakpoint has a false match, but so does the entire overlap region. Evaluating on real hardware is impossible because of the lack of observability — we cannot tell if a match is false or not. Thus, we need a simulation-based evaluation[2].

Setup:

To make this study meaningful, we need a design that is non-trivial, but also not too complex so that we can understand it and leverage any architectural insight in selecting signals to be probed using a trace-buffer. We chose to use a router design, which is an RTL implementation of a 4x4 routing switch. This router is typically used by IBM for training new employees with IBM's tools. The design has 9958 latches, which is larger than other open-source design examples (e.g., OpenCores.org), but not too large too run experiments, and collect and analyze results quickly.

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 destinations addresses, payload, and bit-parity. In addition to routing the packets, the router also checks the validity of incoming packets and rejects bad packets.

We simulated the router on a feature-rich constrained-simulation environment developed by IBM, using Cadence's Incisive Simulator (with Specman Elite) version 09.20-s016. This proved very helpful when modeling environmental non-determinism.

Experiments:

We have two different experiments. The first experiment evaluates the probability of false matches versus amount of overlap. In Sec. 2.3, we asserted that the probability of false matches can be made very low by increasing the number of cycles f of overlap required to match between successive trace-buffer dumps. To isolate this claim, we set the simulation environment to be fully deterministic, i.e., using the same random seed will always generate the same simulation trace. We then uniformly chose 100 abstract states, a_i, from this trace such that $\forall i. a_i \notin Q_0$ and $\forall i,j.(a_i \neq a_j) \wedge |(i-j)| > l$, where l is the TAB length. We

[2]These simulation studies and the TAB-BackSpace framework's source-code are available at the web-site: www.cs.ubc.ca/∼depaulfm/BackSpace.

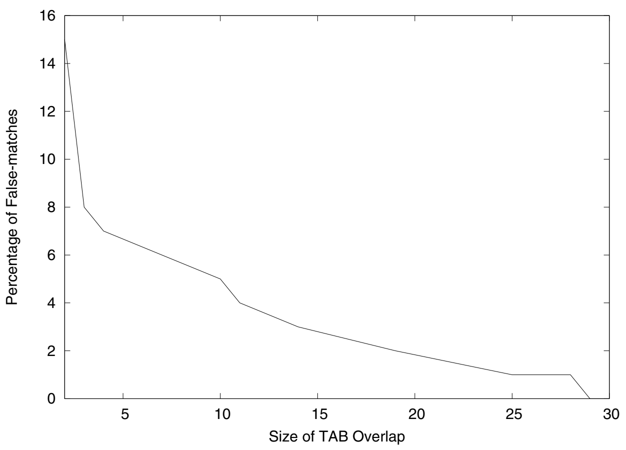

Figure 3: This is the percentage of false-matches for the Router design, assuming a TAB length of 50 entries, a TAB width of 120 bits and the maximum overlap of 30 consecutive cycles. We randomly selected 100 abstract states from a trace 13581-cycles long. This graph shows the percentage of those states that has false matches for a given overlap size. If we set the overlap size to 29, we eliminate all false matches.

set l to be 50, but unlike the CELL's debug logic, we are not doing any cycle compression; and, we set the TAB width to 120 bits. We used our architectural insight of the design to select 120 different design signals. For each selected state a_i, we checked if there were earlier false matches, while incrementing the size of TAB overlap. Fig. 3 shows the results. These results substantiate our claims, and even with a small overlap (25 cycles) the probability of a false match drops to about 1%. If we had set the length of the overlap to be 29 cycles, then we would have no false matches at all.

The second experiment concerns false matches and non-determinism/randomness. We used the same constrained-simulation environment, but enabled non-determinism. This environment provides many parameters to make each simulation run very different from each other. As in real life, however, we always want to reduce the amount of non-determinism as much as possible. Thus, we simulate non-determinism only affecting the delays on packet arrivals (a real scenario encountered in bring-up labs). We modified the original simulation environment so 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. To simulate the scenario in which we are able to repeat the "correct" trace with probability 1/6, we generated 5 additional random traces of similar length to the first trace (i.e., after a specified number of packets were sent). We uniformly selected 100 abstract states from the "correct" trace in the same way as we did in the first experiment. For each selected abstract state, a_i, in the "correct" trace, we checked for a false match on all other 5 traces. For all selected states, we found no false matches whatsoever, even with only one cycle overlap. This result shows that even with some non-determinism, same test-case traces can be very different, making this type of mismatch unlikely.

4. RESULTS ON SILICON

To demonstrate that TAB-BackSpace is feasible in practice, we conducted experiments with an existing IBM processor, running in the post-silicon bring-up lab. Using the existing debug logic in the processor, we performed several iterations of TAB-BackSpace, extending our initial trace by nearly a thousand cycles.

Our experiments were conducted with the IBM POWER7 processor. This processor has built-in hardware debug capabilities, whose architecture is similar to the debug features of the CELL processor.

The basis for our experiments is an example of a real machine bug, which was found during early stages of POWER7 bring-up in the lab. This bug is related to a problem in pipeline bypassing, which appears when floating point instructions are executed out-of-order. In order to discover the root cause of the bug, designers needed to trace backward from the point of the crash, to find all the participating instructions that caused the illegal situation. This was done by conventional trial-and-error methods.

At the time, this bug was easily worked-around using existing logic on the processor. Our experiments were conducted after this fix, therefore we used a modified configuration of the processor, in which the workaround was disabled and the bug became active again. In addition, we created an environment in which we could deterministically rerun the processor and reach the crash caused by the bug. This involved running on bare metal, using the Threadmill post-silicon exerciser[8]. We configured the processor to use only one active core, since this is sufficient reproduce the bug.

In each of our runs, we activate trace-arrays of the active core, to record signal values throughout the run (we activate a fixed subset of the trace-array signals, used in all our runs). When the processor stops, the contents of the trace-arrays reflect the values of the recorded signals, for some number of cycles at the end of the run. The number of cycles represented in the trace-arrays may vary, since some compression is applied when values are repeated for consecutive cycles. After each run, the values of the trace-arrays are dumped to a file, and are processed in order to produce a decompressed, non-cyclical trace.

The *recorded signals* are the subset of the trace-arrays activated during our runs. This set contains 352 signals. In addition, we selected a subset of 176 recorded signals, to use as *trigger signals* (this set is also fixed for all runs). An assignment of values to the trigger signals is a *trigger pattern*, and the pattern-matching mechanism of the debug logic can be configured to halt the processor when the trigger pattern is identified on the trigger signals.

Our initial trace is the trace produced by running the processor in the bug reproduction environment, until the crash is reached. This trace provides us with a window of 958 cycles leading up to the bug. In practice, this isolated window is too small for debugging purposes, since it does not include the root cause of the bug.

Starting from this initial trace, we applied repeated iterations of TAB-BackSpace steps, creating a sequence of trace-array dumps. In each TAB-BackSpace step, we are assured that it will not continue to run past the breakpoint cycle. However, it is possible that the trigger pattern appears in an earlier cycle (an abstract *false match*, as described in Sec. 2.2), and thus the run will stop earlier than the breakpoint cycle. In a bare-metal lab run, we typically do not have a cycle counter, which could help us to detect whether we have stopped at the breakpoint cycle or earlier. We therefore use the overlap check. If the new trace and the current trace agree on all of the recorded signals, for the entire prefix of the current trace up to the breakpoint cycle, we consider the new trace to be a true trace leading up to the breakpoint cycle. To reduce the risk of *false matches*, we strive to make the overlap of two consecutive runs greater than (or equal to) half the length of the current trace. In addition, although we use only the 176 trigger signals for defining the trigger pattern, we perform the overlap check on all 352 recorded signals, which gives additional confidence we stopped at the correct breakpoint cycle.

In practice, some of the runs do stop at a cycle that is too early, and therefore fail the overlap check. In this case, we select a new breakpoint cycle from the current trace, with a different trigger value, and repeat the run with this value. In practice, we found that 3 attempts were always sufficient, in any given iteration, for generating a trace with an overlap. Overall, for all the runs executed in all iterations, 86% of the runs were successful, i.e., produced a new trace that overlapped with the current trace.

Obviously, there is a tradeoff between the size of the overlap and the size of the backspace, i.e., the number of new cycles recorded in this iteration. If we aim for a large overlap, in order to increase our confidence in the correctness of the new trace, then the number of new cycles is reduced, and more iterations will be needed in order to extend the trace to a given length.

In our experiments, we executed 10 TAB-BackSpace iterations, producing 10 new traces in addition to the initial trace. The results of these iterations are shown in Table 1. The first row represents the initial trace, while the following rows represent the traces generated by the iterations. These traces are all 256 cycles long. The *New cycles* column shows the number of new cycles traced in the current iteration. The *Accumulated new cycles* column shows the accumulated number of new cycles traced, in all iterations up to and including the current one. The final accumulated number, at the bottom of this column, shows the total backward extension that we have achieved in these 10 iterations, which amounts to 988 cycles. For our example bug, this extension was sufficient to reveal the root cause of the bug.

Trace #	Length	Overlap with prev. trace	New cycles	Accumulated new cycles
0	958			
1	256	64	192	192
2	256	130	126	318
3	256	146	110	428
4	256	168	88	516
6	256	186	70	586
7	256	116	140	726
8	256	188	68	794
9	256	150	106	900
10	256	168	88	988

Table 1: TAB-BackSpace on POWER7

5. CONCLUSION AND FUTURE WORK

Given our successful experiments, the next step is to build a robust implementation of TAB-BackSpace, tailored to specific, next-generation products. Since the TAB-BackSpace framework provides enhanced on-chip observability, completely automatically, with zero on-chip overhead, there is no reason not to use it. Future work includes optimizing the debug logic under the assumption that TAB-BackSpace will be used. For example, we might reduce the on-chip physical trace buffer size, since TAB-BackSpace can compute the missing entries. In that case, TAB-BackSpace would effectively have *negative* area overhead.

6. ACKNOWLEDGMENTS

We thank Jim Bishop, Todd Venton, and Markus Kaltenbach whose technical contributions enabled this work. Authors De Paula and Hu were supported by the Semiconductor Research Corporation Task ID: 1586.001, and by the Natural Sciences and Engineering Research Council of Canada.

7. REFERENCES

[1] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs", *DAC*, 2006, pp. 7–12.

[2] ARM. "Embedded Trace Macrocell Architecture Specification", volume 20. July 2007. Ref: IHI0014O.

[3] K.-H. Chang, I. L. Markov, and V. Bertacco. "Automating Post-Silicon Debugging and Repair," *ICCAD*, 2007, pp. 91–98.

[4] E. M. Clarke, O. Grumberg, and D. E. Long. "Model Checking and Abstraction," *POPL*, 1992, pp. 343–354.

[5] F. M. de Paula, M. Gort, A. J. Hu, S. J. E. Wilton, and J. Yang. "BackSpace: Formal Analysis for Post-Silicon Debug," *FMCAD*, 2008, pp. 35–44.

[6] M. Gort. "Practical Considerations for Post-Silicon Debug using BackSpace", MASc Thesis, Elec. and Comp. Eng. Dept., Univ. of British Columbia, 2009.

[7] H. F. Ko and N. Nicolici. "Algorithms for State Restoration and Trace-Signal Selection for Data Acquisition in Silicon Debug," *IEEE TCAD*, 28(2):285–297, Feb 2009.

[8] M. Levinger. "Building a bridge: from pre-silicon verification to post-silicon validation", On-Line: http://es.fbk.eu/events/fmcad08/presentations/tutorial_moshe_levinger.pdf.

[9] S. Park, S. Yang, and S. Cho. "Optimal State Assignment Technique for Partial Scan Designs," *Electronics Letters*, 36(18):1527–1529, Aug 2000.

[10] S.-B. Park and S. Mitra. "IFRA: Instruction Footprint Recording and Analysis for Post-Silicon Bug Localization in Processors," *DAC*, 2008, pp. 373–378.

[11] B. Quinton and S. Wilton, "Concentrator Access Networks for Programmable Logic Cores on SoCs," *IEEE ISCAS*, 2005, pp. 45–48.

[12] M. Riley, N. Chelstrom, M. Genden, and S. Sawamura, "Debug of the CELL Processor: Moving the Lab into Silicon," *ITC*, 2006, pp. 1–9.

[13] M. J. Y. Williams and J. B. Angell, "Enhancing Testability of Large-Scale Integrated Circuits via Test Points and Additional Logic," *IEEE TC*, C-22(1):46–60, January 1973.