Progressive-BackSpace: Efficient Predecessor Computation for Post-Silicon Debug

Johnny J.W. Kuan and Tor M. Aamodt
Department of Electrical and Computer Engineering
University of British Columbia, Vancouver, BC
Email: jkuan@ece.ubc.ca, aamodt@ece.ubc.ca

***Abstract*—As microprocessors become more complex, finding errors in their design becomes more difficult. Most design errors are caught before the chip is fabricated, however, some make it into the fabricated design. One challenge in determining what is wrong with a new design after fabrication is the lack of observability into the state of the fabricated chip. To address this challenge, BackSpace [1], [2] proposes generating a trace of the states that lead up to an erroneous state. To add one state to the trace, BackSpace first generates a set of possible predecessor states (the pre-image), then tests them one at a time to find one that is reached during execution. In this paper, we propose an improved algorithm called Progressive-BackSpace. It does not enumerate every state in the pre-image. Instead, it first finds a reachable candidate state, and then determines if it is a predecessor state. This results in a practical implementation of BackSpace by greatly reducing the time needed to find predecessor states. The hardware overhead is also reduced by 94.4% relative to a recently proposed implementation of BackSpace [3]. These algorithms were implemented and evaluated on a RTL model of an out-of-order processor, that models non-deterministic effects.**

I. Introduction

It is common for errors to be found in the design of modern processors after they are released [4]. A well-known example of a design error is the Pentium floating point division bug that is estimated to have directly cost Intel $475 million in replacements and write-offs [5]. These design errors exist despite the tremendous effort put into finding and correcting them. This effort starts before the chip is fabricated, during pre-silicon validation, and continues afterwards during post-silicon debug (also known as silicon validation). Ideally, all design errors would be found and corrected before the chip is manufactured and post-silicon debug would not be required. However, this is not possible, due in part to the complexity of modern chip designs and the slow speed of simulation. For example during pre-silicon validation of the Pentium 4 the total amount of simulation performed was equivalent to less than two minutes of execution on a 1 GHz chip [6].

The process of finding and correcting errors continues after the chip is fabricated. The fabricated chip may, in addition to design errors, have electrical errors caused by the manufacturing process. In this paper we assume that a separate process, known as manufacture testing, has already selected a set of chips that are free of electrical errors for use in post-silicon debug. This paper will focus on enhancing the process of determining what is wrong when the design of a fabricated chip is incorrect.

Lack of observability into chips is one of the major challenges in post-silicon debug [7], [8]. This is in contrast with pre-silicon validation, where the entire internal state of the design is observable at all times. To improve observability post-silicon engineers currently use tools like scan chains. Scan chains allow the state of the chip to be read out once it is stopped provided special flip-flops are used on the chip. However, a snapshot of the state of the chip at one cycle may be insufficient to find the root cause of a bug. Trace buffers [9] provide a view of many cycles, but for a limited set of signals. What is needed is a way to expand observability in terms of both the number of internal signals in the chip that can be seen by the debug engineer and the length of time over which they can be observed.

One proposal to increase observability into a chip post-silicon is BackSpace [1], [2]. The goal of BackSpace is to produce a trace of on-chip events leading up to a bug that can be fed into a waveform viewer. Each iteration of BackSpace requires computing the set of all possible predecessor states (the pre-image), which can be very large or require substantial hardware overhead. Then, the states in the pre-image must be tested one a time by repeatedly running the chip, which can be time-consuming. We instead run the chip to find a candidate state first, and then determine whether the candidate state is a predecessor *without enumerating the pre-image* by using information obtained by running the chip forwards. This allows us to utilize the speed of the fabricated chips forward state *progression* in an efficient manner

In this paper, we propose a new post-silicon trace algorithm, Progressive-BackSpace, that (a) generates traces that satisfy the same properties as those generated by BackSpace, (b) requires 94.4% less on-chip storage than a recently proposed implementation of BackSpace and (c) requires dramatically fewer chip runs than practical implementations of BackSpace. We evaluate the performance of BackSpace and Progressive-BackSpace using a RTL model of an complex, non-deterministic, out-of-order processor.

We first discuss the baseline BackSpace algorithm and its limitations in Sec. II. A series of optimizations are then presented in Sec. III that culminate in the Progressive-BackSpace algorithm. A hardware implementation of this algorithm is discussed in Sec. IV. Our methodology is outlined in Sec. V and results are presented in Sec. VI. Related work is discussed in Sec. VII, and Sec. VIII concludes.

II. Background

De Paula *et al.* proposed a methodology to improve post-silicon observability by creating a complete trace of events on-chip using a technique they called BackSpace [1], [2]. A post-silicon trace can be fed into a waveform viewer and used by an engineer in the same manner as a trace generated pre-silicon by a simulator. This trace is created by running the chip multiple times; ideally adding one state to the trace every time the chip is run.

A. BackSpace Algorithm

The BackSpace algorithm begins by assuming that the chip is stopped at a state that we would like to generate a trace to. This state will be referred to as the crash state s_0. To extend the trace earlier into execution, the state that precedes the crash state is needed. Additional hardware is used to store a portion of the previous state s_1 which we call the signature $Sig(s_1)$. From the crash state, the design of the chip (in the form of a gate-level description) and the signature, the set of possible states that could have led to s_0 is computed. This set is the pre-image P. One of the states in P is the state that actually occurred before s_0, so we know that this state can be reached by the chip. The other states in P might not be reachable.

In the next step, we find a state in the pre-image that is reached during execution. This is done using a breakpoint circuit that checks, every cycle, if the current state matches a target state. One candidate state in the pre-image is loaded into the breakpoint circuit and the chip is run until either the candidate state is reached and the chip stops, or a timeout is reached. If the chip times out then another state in P is loaded into the breakpoint circuit. Eventually a state in the pre-image P will be reached. This state s_1 is known to occur during execution and also is a possible previous state to the earliest state in the trace so it can be added to the trace. This process is repeated to generate a trace of arbitrary length.

B. Pre-Image Computation

We model the circuit as a finite state machine with S latches, J inputs, initial states $Init \subseteq 2^S$, and transition relation $\delta \subseteq 2^S \times 2^J \times 2^S$ as in [1]. The pre-image of a state s_i is then $P_{sig} = \{s \mid \exists j[(s, j, s_i) \in \delta] \wedge Sig(s) = Sig(s_{i+1})\}$, where s_{i+1} is the state preceding s_i. It is computed by first constructing a boolean formula such that variable assignments that satisfy the formula correspond to a possible previous state. Some of the clauses in this formula describe the combinational logic of the chip, while others encode the current state of the chip and the signature of the previous state. All the solutions to this formula are then found. Please refer to [1] for details on the pre-image computation.

C. Partial Breakpoints

Gort [3] described a optimization that reduces the hardware overhead of BackSpace. Instead of comparing every bit in the state s with a stored target state t, only part of the state $\mathrm{part}(s)$ is compared with a partial breakpoint $\mathrm{part}(t)$. There are two problems, temporal mismatches and spatial mismatches, that can occur [3]. Temporal mismatches occur when states earlier in execution also match $\mathrm{part}(t)$. We resolve this problem by loading $\mathrm{part}(t)$ into a counting circuit that counts how many times $\mathrm{part}(t)$ matches before execution reaches the earliest state in the trace. After collecting this information we run the breakpoint circuit again and stop the circuit when $\mathrm{part}(t)$ is matched the recorded number of times. The other problem, a spatial mismatch, is that $\mathrm{part}(t)$ and the count matches during a run that does not lead to the correct state. Let s'_{i+1} be the state that matches $\mathrm{part}(t)$ and the count. Spatial mismatches can be resolved by scanning out s'_{i+1}, then checking if it is the state t we are looking for.

TABLE I
THE NUMBER OF STATE BITS THAT CAN BE OVERWRITTEN.

Register File Name	Total Bits	Number	Size
Renamed Registers	5120	80	64
Store Queue Addresses	1024	16	64
Load Queue Addresses	1024	16	64
Architectural RAT	224	32	7
Speculative RAT	224	32	7
Fetch PC (program counter)	64	1	64
Flush PC (program counter)	64	1	64
Total	**7744**		

D. BackSpace Limitations

One of the concerns with BackSpace is the number of states in the pre-image in the worst case. De Paula *et al.* [1] have shown that the pre-image can include all possible states if the current state is a reset. Even if we are not interested in generating a trace that goes past resets, a very similar situation occurs whenever a register gets overwritten. Suppose the 64-bit register X gets overwritten in the transition from state A to state B. Assume the chip is currently stopped at state B and the signature includes all the state bits except for register X. If we have no additional information about the value of register X in state A then there will be 2^{64} states in the pre-image. A conservative estimate of the number of state bits in registers that can be overwritten in the Illinois Verilog Model (IVM) processor introduced in Sec. V is shown in Table I.

To prevent this type of pre-image state explosion the contents of the overwritten register should be recorded in the signature. If one of these registers are not included in the signature, the pre-image may become very large when a register is overwritten. This problem highlights two related challenges of using BackSpace: the number of times the chip must be run and the hardware overhead required. Using a larger signature, at the expense of more hardware overhead, means that the pre-image will, on average, be smaller and so the chip needs to run fewer times. Using a smaller signature reduces the hardware overhead, but increases the size of the pre-image and the expected number of times the chip must be run. We address both of these challenges in the next section.

III. Optimizations to BackSpace

As noted, BackSpace has problems recovering the value of overwritten registers. This is because BackSpace explicitly enumerates all possible previous states, and then finds one which is reachable. We avoid this problem by first finding a reachable state, and then determining whether this state is a possible previous state without explicitly enumerating the pre-image. The algorithm proposed below builds on the partial breakpoint BackSpace algorithm [3] summarized in Sec. II-C. This algorithm produces a trace with the same properties as a trace produced by BackSpace more quickly and with less hardware overhead. These properties [1] are: every state added to the trace (after the crash state) is a predecessor of the last state added and is reachable. To show that this algorithm produces a trace that satisfies these properties we will present it as a series of optimizations to the baseline BackSpace algorithm. We first describe a method to check all the states in the pre-image simultaneously. Then we show how to check if a state is in the pre-image without computing it. Next, we eliminate the need to compute the pre-image before running the breakpoint circuit. Finally, we pipeline the two breakpoint steps to obtain the Progressive-BackSpace algorithm.

A. Check All States in Pre-image Simultaneously

We gave an example in Sec. II-D where there are 2^{64} states in the pre-image P_{sig}. It would be much faster to check all the states in P_{sig} for reachability simultaneously, rather than one at a time. We can do this by loading a set of states containing P_{sig} into the breakpoint circuit. First, we compute the subset of bits that are the same across all the states in P_{sig}. We refer to this subset of bits as $\text{known}(P_{sig})$. We then use the partial breakpoint technique described in Sec. II-C, with $\text{known}(P_{sig})$ as the partial breakpoint. The main difference is that instead of checking if the scanned out state s'_{i+1} is equal to $\text{part}(t)$, we check if s'_{i+1} is in P_{sig} and running the chip again if it is not. By loading the entire pre-image into the breakpoint circuit the number of runs to find a predecessor state is no longer dependent on the size of the pre-image. However, s'_{i+1} must be compared with every state in P_{sig}.

B. Transition Check

Comparing s'_{i+1} with every state in P_{sig} can be time-consuming if P_{sig} is large. Fortunately, we can determine if a state is in P_{sig} without these comparisons. First, consider the pre-image with no bits in the signature. This is the set of all predecessor states $P_{all} = \{s \mid \exists j[(s, j, s') \in \delta]\}$. Next, consider the set of all states that match the signature $S_{sig} = \{s \mid Sig(s) = Sig(s_{i+1}\}$. Recall that $P_{sig} = \{s \mid \exists j[(s, j, s_i) \in \delta] \wedge Sig(s) = Sig(s_{i+1})\}$. This means that instead of checking if s'_{i+1} is in P_{sig} we can check if it is in P_{all} and in S_{sig}. $s'_{i+1} \in S_{sig}$ if $Sig(s'_{i+1})$ is equal to the signature of the previous state. Checking if $s'_{i+1} \in P_{all}$ is equivalent to checking if $\exists j[(s'_{i+1}, j, s_i) \in \delta]$.

To check if a state s'_{i+1} is in P_{all} we construct a satisfiability problem (SAT) instance that checks for a transition from s'_{i+1} to s_i. It is similar to the SAT instance used to generate one state in the pre-image. The clauses that correspond to the combinational logic and the earliest state in the trace s_i are identical. Each bit in s'_{i+1} is also converted to a clause. A SAT solver [10] is then used to determine if this boolean formula has a solution. If it does, then there exists a transition between

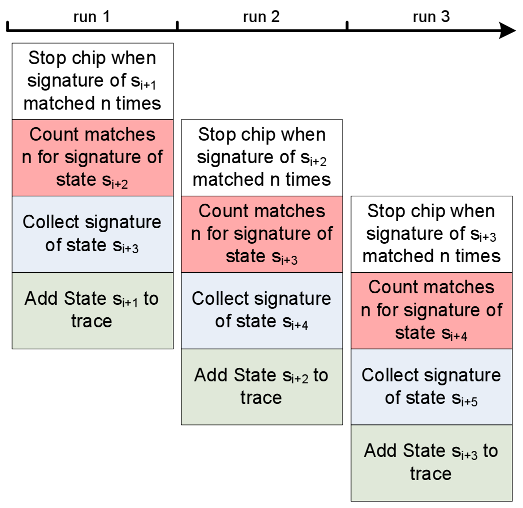

run 1
run 2
run 3
Stop chip when signature of s_{i+1} matched n times
Count matches n for signature of state s_{i+2}
Collect signature of state s_{i+3}
Add State s_{i+1} to trace
Stop chip when signature of s_{i+2} matched n times
Count matches n for signature of state s_{i+3}
Collect signature of state s_{i+4}
Add State s_{i+2} to trace
Stop chip when signature of s_{i+3} matched n times
Count matches n for signature of state s_{i+4}
Collect signature of state s_{i+5}
Add State s_{i+3} to trace

Fig. 1. Timeline of the steps to add four states to the trace.

s'_{i+1} and s_i. Although we *do not* compute P_{sig}, it may be useful to think of our approach as a "lazy evaluation" of P_{sig}.

C. Breakpoint and Signature Selection

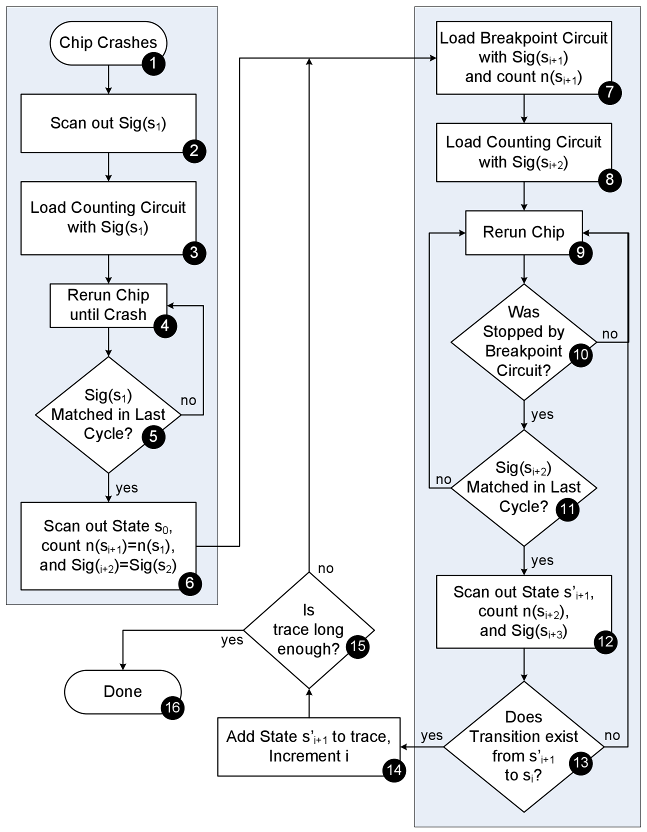
We would also like to eliminate the need to compute the pre-image before determining what bits to load into the partial breakpoint circuit. One way this can be accomplished is to approximate $\text{known}(P_{sig})$ with the signature $Sig(s_{i+1})$. This works because all the states in $\text{known}(P_{sig})$ must have the same signature $Sig(s_{i+1})$. Utilizing a larger partial breakpoint requires computing $\text{known}(P_{sig})$ or a better approximation to it, so in the rest of this paper the state bits used in the partial breakpoint will be the same as the bits use in the signature. Also, using a signature larger than the partial breakpoint does not change the operation of the breakpoint circuit. Since we use the signature $Sig(s_{i+1})$ as the partial breakpoint, there is no need to compute the pre-image P_{sig}.

D. Pipelining

The partial breakpoint technique described in Sec. III-A requires two processor runs to add a state to the trace. A signature is obtained in the first run, and the count for this signature is obtained in the second run. The number of runs needed can be reduced by recording a signature and counting the number of matches of a signature in the same run as shown in Fig. 1. We can record $Sig(s_{i+3})$ while we record the number of matches for its successor state s_{i+2}.

E. Progressive-BackSpace

After applying the above optimizations, we obtain the Progressive-BackSpace algorithm. Before each iteration, the signatures of two states preceding the earliest state in the trace s_i are known, as well as a count of the number of times $Sig(s_{i+1})$ was seen. After running the chip, the state s'_{i+1}, a count of the times $Sig(s_{i+2})$ was seen, and $Sig(s_{i+3})$ are also known. Next, we verify that a transition exists between s'_{i+1} and s_i. If it does, we add s_{i+1} to the trace. We now have the necessary information to start another iteration.



Chip Crashes 1
Scan out Sig(s1) 2
Load Counting Circuit with Sig(s1) 3
Rerun Chip until Crash 4
Sig(s1) Matched in Last Cycle? 5
no
yes
Scan out State s0, count n(si+1)=n(s1), and Sig(i+2)=Sig(s2) 6
Load Breakpoint Circuit with Sig(si+1) and count n(si+1) 7
Load Counting Circuit with Sig(si+2) 8
Rerun Chip 9
Was Stopped by Breakpoint Circuit? 10
no
yes
Sig(si+2) Matched in Last Cycle? 11
no
yes
Scan out State s'i+1, count n(si+2), and Sig(si+3) 12
Does Transition exist from s'i+1 to si? 13
yes
no
Add State s'i+1 to trace, Increment i 14
Is trace long enough? 15
no
yes
Done 16

Fig. 2. Flowchart describing the operation of Progressive-BackSpace.

The flowchart shown in Fig. 2. details the operation of Progressive-BackSpace. The shaded block on the left shows the steps to obtain the crash state s_0, the signature for the state before it $Sig(s_1)$, the number of times $Sig(s_1)$ is matched $n(s_1)$, and $Sig(s_2)$. The algorithm begins when the chip crashes (❶). The signature of the state before the crash is read (❷) and loaded into the partial breakpoint counter circuit (❸). The chip is then rerun (❹) until $Sig(s_1)$ is matched in the last cycle (❺). We then scan out the crash state s_0 (❻). The count for $Sig(s_1)$ and $Sig(s_2)$ are also read. With this information the algorithm can enter its main loop, which is shown in the shaded block on the right of Fig. 2.

We start each iteration of the main loop with the signature for a state $Sig(s_{i+1})$, its match count $n(s_{i+1})$, and the signature for the previous state $Sig(s_{i+2})$. We begin by loading the breakpoint circuit (❼) with the signature $Sig(s_{i+1})$ and its count $n(s_{i+1})$. The partial breakpoint counting circuit is loaded with $Sig(s_{i+2}$ (❽). The chip is then rerun (❾) until it is stopped by the breakpoint circuit (❿). If the chip is stopped for another reason, for example a timeout or a crash, the chip is rerun (❾) again. Also, if $Sig(s_{i+2})$ was not matched in the previous cycle (⓫) then $Sig(s_{i+2})$ is not the signature of the state before the one where the chip is stopped, so the chip is also rerun (❾) again. If the chip was stopped by the breakpoint circuit and $Sig(s_{i+2})$ matched in the previous cycle, then the state of the chip is scanned out (⓬). This state is a candidate to be s_{i+1} and is referred to as s'_{i+1}. The count for the signature of s_{i+2}, $n(s_{i+2})$, and $Sig(s_{i+3})$ are also read. The existence of a transition from s'_{i+1} to s_i is determined (⓭) as described in Sec. III-B. If a transition does not exist, the chip is rerun (❾). Otherwise, if a transition does exist, the state s'_{i+1} is added to the trace (⓮) as s_{i+1}. Also, i is incremented so that s_i is still the earliest state in the trace. If the trace is long enough (⓯) the algorithm is complete (⓰). Otherwise, we have the information needed to start another iteration.

IV. Hardware

In this section, the hardware needed to implement the Progressive-BackSpace algorithm will be described and the storage overhead required will be discussed.

A. Progressive-BackSpace Hardware

We will start by describing the three sub-circuits common to Progressive-BackSpace and BackSpace with partial breakpoints. Next, we will show how to use these components to implement Progressive-BackSpace.

Partial Breakpoint Circuit: generates a breakpoint signal that stops the chip so that the current state of the chip can be scanned out. Before the chip is run, the target partial breakpoint $Sig(s_{i+1})$ and target counter value $n(s_{i+1})$ are loaded into two registers. After the target $Sig(s_{i+1})$ matches the signature of the current state $Sig(s_c)$, the desired number of times $n(s_{i+1})$, the breakpoint signal is asserted.

Counting Circuit: obtains the target counter value used by the partial breakpoint circuit. Before the chip is run, the target partial breakpoint $Sig(s_{i+2})$ that we wish to obtain the count for is loaded into a register. As the chip runs, the number of states s_c where $Sig(s_{i+2}) = Sig(s_c)$ is counted. A match that occurs on the cycle the breakpoint triggers is not counted. Progressive-BackSpace requires an output that indicates whether the previous cycle matched the target.

Signature Creation Circuit: creates the signature. The signature could in general be an arbitrary function of the state, but in our implementations selects a subset of the state bits.

The hardware needed to implement Progressive-BackSpace is shown in Fig. 3. The signature, created by the signature creation circuit, is connected to the inputs for the partial breakpoint circuit, counting circuit, and the signature collection circuit. The signature collection circuit in Fig. 3 stores the signature for the two previous states. In Fig. 2 (❼) the Load signal is asserted which causes the partial breakpoint circuit to load in the new target signature and count. In Fig. 2 (❽) the same Load signal is asserted which causes the counting circuit to load in the new signature to count.

B. Hardware Overhead

W now consider the hardware overhead needed to implement the various post-silicon trace generation techniques discussed so far. The hardware described in Sec. IV-A was not synthesized so the area overhead will not be discussed. We will instead focus on the storage required on-chip.

All the post-silicon trace generation techniques discussed so far require storing additional information in hardware. In

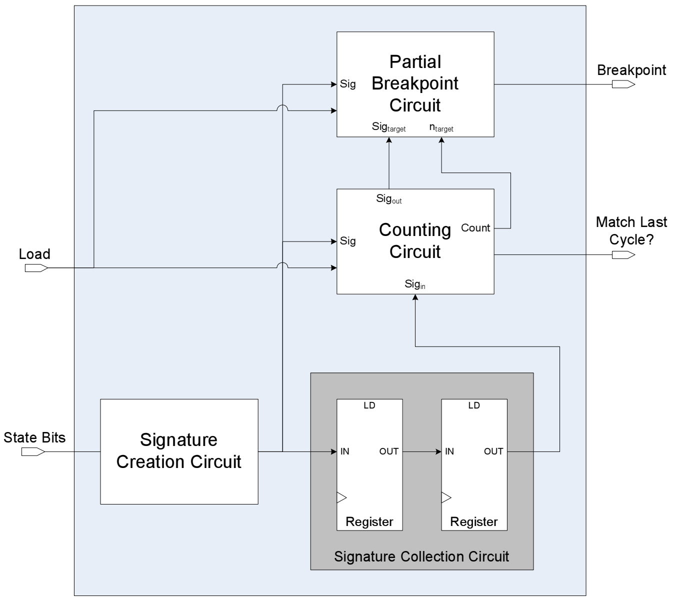


Fig. 3. Hardware schematic for Progressive-BackSpace.

TABLE II
STORAGE USED BY THE DIFFERENT TRACE GENERATION ALGORITHMS.

Component		BS	BS-part	Prog-BS
Breakpoint Circuit	Target State	88796	64	64
	Counter	n/a	64	64
	Target Count	n/a	64	64
Counting Circuit	Target State	n/a	64	64
	Counter	n/a	64	64
Signatures		7744	7744	128
Total		**96540**	**8064**	**448**

the breakpoint circuit, a full state (or a part of it) must be stored in a register to compare with the current state. This overhead can be reduced using the partial breakpoint technique discussed in Sec. II-C, but then requires a circuit to count matches. A signature of the previous state must be stored for each algorithm. Progressive-BackSpace also requires storing the signature of the state two cycles before the current one. For BackSpace, either with full breakpoints or partial breakpoints, the signature is chosen to consist of the registers that can be overwritten as discussed in Sec. II-D. This signature is used since it is a lower bound on what a practical signature would contain. In the counting circuit, 64 bit counters are used to be conservative. With a 10 GHz chip, if every state matched the target partial breakpoint, a 64 bit counter could count the matches for over 50 years.

The storage required for the hardware components of each of the post-silicon trace generation algorithms is shown in Table II. BS is the baseline BackSpace algorithm with a signature containing only registers that can be overwritten as described in Sec. II-D. BS-part is BackSpace using the partial breakpoints described in Sec. II-C. Prog-BS is the Progressive-BackSpace algorithm described in Sec. III-E. Our proposed algorithm requires 99.5% fewer bits of storage than the original BackSpace algorithm and 94.4% fewer bits than BackSpace with partial breakpoints.

TABLE III
AVERAGE NUMBER OF RUNS NEEDED TO ADD ONE STATE TO THE TRACE.

Program	BS-ideal	Prog-BS
`arith-add`	10.64167	10.68333
`fib-20`	72.44083	55.92547

V. METHODOLOGY

A superscalar out-of-order processor running a subset of the Alpha instruction set, the IVM processor [11], was used. This processor was synthesized using the BackSpace technology library [12] which contains logic gates and flip-flops. The synthesized processor was used to generate the SAT clauses that correspond to the circuit and to build the simulator. Non-determinism was introduced in the form of variable memory latency.

We compare the performance of Progressive-BackSpace against an unrealistic implementation of BackSpace that contains the entire previous state (88796 bits) in its signature. This implementation, BackSpace-ideal, provides an upper bound on the performance of BackSpace. Note that this was not the signature size used to determine the hardware overhead of BackSpace (BS) or BackSpace with partial breakpoints (BS-part) in the previous section. The 64 bit Fetch PC was used as the signature for Progressive-BackSpace. The intuition is that the PC, coupled with the number of times it occurs, serves as indicator of the chip's progress through a program.

VI. RESULTS

The performance of the post-silicon trace generation algorithms discussed in this paper are studied in this section. The average (mean) number of runs to add one state to the trace is the metric used to measure performance. To evaluate the performance of the different algorithms, traces from various programs were collected. These traces all end at (i.e., has as initial crash state) the end of program execution. Trace collection ends when a trace of 25 states has been collected, or the algorithm times out after 48 hours. For each algorithm, 100 traces were collected for each program. The average number of processor runs needed to add one state to the trace is shown in Table III. With deterministic execution, both these algorithms would only require one run to add one state to the trace. We can see that non-determinism significantly reduces the performance of both algorithms. The distribution of runs per state added is also interesting. Histograms showing the distribution of average runs needed to add a state to a trace of the `arith-add` program are shown in Fig. 4. There are 100 data points for each algorithm. We can see that the non-determinism has a similar effect on both distributions. Note that these distributions are only similar because this implementation of BackSpace uses the entire previous state as its signature. This means that there will only be one state in the pre-image, and multiple runs of the processor are required only to handle non-determinism.

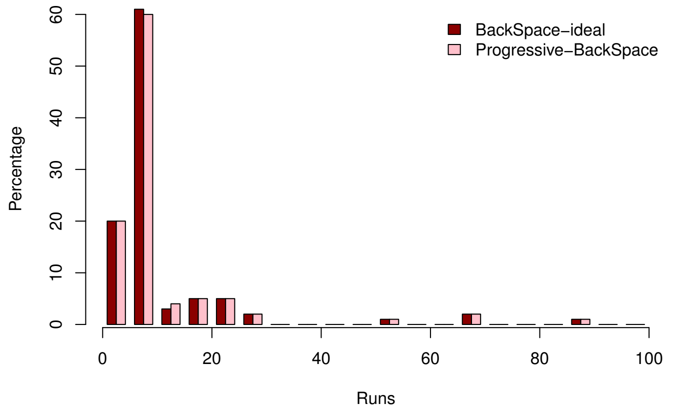


Fig. 4. Histograms of the number of runs to add one state to the trace of `arith-add`.

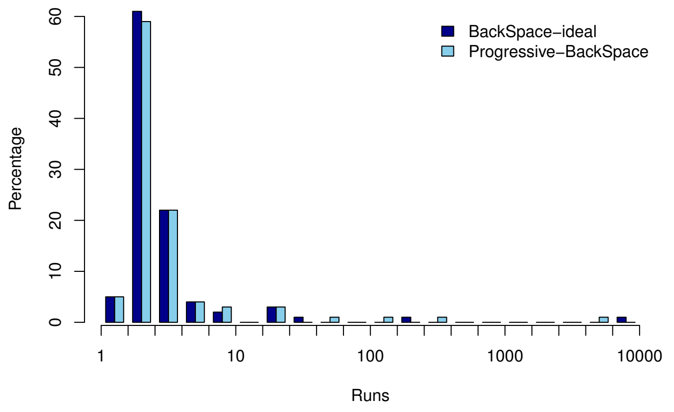


Fig. 5. Histograms with a logarithmic scale of the number of runs to add one state to the trace of `fib-20`.

The distributions for the average number of runs to add a state to a trace of the `fib-20` program are much wider, so a logarithmic scale is used for the average number of runs in Fig. 5. This figure shows the average run distributions for the `fib-20` program using the BackSpace-ideal and Progressive-BackSpace algorithms.

VII. Related Work

There are techniques that use scan chains and trace buffers directly to aid in post-silicon debug. For example, Ko and Nicolici [13] combine the information collected from scan chains and trace buffers, then apply state restoration techniques to create a trace that contains more state information than either scan chains or trace buffers alone could provide. However, the length of the trace is limited by the amount of storage used. There has also been a proposal to extend the length of traces generated by a trace buffer by running the chip multiple times and stitching together the traces [14]. Note that, unlike Progressive-BackSpace, neither of these proposals produce traces of complete states. However, the breakpoint technique used in [14] may be combined with our transition check technique and scan chains to produce traces with the same formal properties as ours.

One aspect of post-silicon debug is locating where in hardware and in which cycle a detected bug occurred. BLoG [15] records the data and instruction flows through certain design blocks during execution, and then analyzes these flows after execution in conjunction with the program binary to localize any errors. BLoG was evaluated by modelling electrical bugs as single bit-flips in a micro-architectural simulator.

VIII. Conclusions

We began by considering the limitations of the proposed post-silicon trace generation technique BackSpace. In particular, the number of states in the pre-image that must be enumerated and tested grows exponentially with the number of state bits that are overwritten in an unrecoverable way. We presented a series of optimizations to BackSpace that culminate in an algorithm that does not need to compute the pre-image and only requires as many processor runs as an unrealistic implementation of BackSpace which stores the entire previous state in its signature. Our proposed algorithm, Progressive-BackSpace, also requires storing 94.4% fewer bits than a recently proposed implementation of BackSpace.

It is clear that non-determinism significantly impacts the performance of Progressive-BackSpace. In the future, we would like to study the non-determinism present in real systems and investigate ways to reduce the impact of non-determinism on the performance of Progressive-BackSpace.

References

[1] F. M. de Paula, M. Gort, A. J. Hu, S. J. E. Wilton, and J. Yang, "BackSpace: Formal analysis for post-silicon debug," in *Formal Methods in Computer Aided Design*, 2008.

[2] F. M. de Paula, M. Gort, A. J. Hu, and S. J. E. Wilton, "BackSpace: Moving towards reality," in *International Workshop on Microprocessor Test and Verification*, December 2008, pp. 49–54.

[3] M. Gort, "Practical considerations for post-silicon debug using BackSpace," Master's thesis, University of British Columbia, 2009.

[4] "Revision guide for AMD family 10h processors," June 2010.

[5] Intel Corporation, Annual Report, 1994.

[6] B. Bentley, "Validating the Intel® Pentium® 4 microprocessor," in *Design Automation Conference, 2001. Proceedings*, 2001, pp. 244–248.

[7] T. Bojan, M. Aguilar Arreola, E. Shlomo, and T. Shachar, "Functional coverage measurements and results in post-silicon validation of Core^{TM} 2 Duo family," in *International High Level Design Validation and Test Workshop*. IEEE Computer Society, 2007, pp. 145–150.

[8] S. Mitra, S. A. Seshia, and N. Nicolici, "Post-silicon validation opportunities, challenges and recent advances," in *Proceedings of the 47th Design Automation Conference*, 2010, pp. 12–17.

[9] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs," in *43rd ACM/IEEE Design Automation Conference*, 2006, pp. 7–12.

[10] N. Eén and N. Sörensson, "An extensible SAT-solver," in *6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, May 2003, pp. 502–518.

[11] N. J. Wang and S. J. Patel, "ReStore: Symptom based soft error detection in microprocessors," in *International Conference on Dependable Systems and Networks (DSN)*, June 2005, pp. 30–39.

[12] F. M. de Paula, "Backspace toolkit," 2010. [Online]. Available: http://www.cs.ubc.ca/~depaulfm/BackSpace

[13] H. F. Ko and N. Nicolici, "Combining scan and trace buffers for enhancing real-time observability in post-silicon debugging," in *European Test Symposium (ETS), 15th IEEE*, May 2010, pp. 62–67.

[14] F. M. de Paula, A. Nahir, Z. Nevo, A. Orni, and A. J. Hu, "TAB-BackSpace: Unlimited-length trace buffers with zero additional on-chip overhead," in *48th Design Automation Conference*, June 2011.

[15] S.-B. Park, A. Bracy, H. Wang, and S. Mitra, "BLoG: Post-silicon bug localization in processors using bug localization graphs," in *Design Automation Conference, 47th ACM/IEEE*, June 2010, pp. 368 –373.