

BackSpace: Moving Towards Reality

Flavio M. De Paula¹, Marcel Gort², Alan J. Hu¹, Steven J. E. Wilton²

¹ Dept. of Computer Science, University of British Columbia, {depaulfm, ajh}@cs.ubc.ca

² Dept. of Electrical and Computer Engineering, University of British Columbia, {mgort, stevew}@ece.ubc.ca

Abstract—In recent work, we proposed *BackSpace*, a new paradigm for using formal analysis, augmented with some on-chip hardware, to support post-silicon debugging. *BackSpace* allows the chip to run at full speed, but then provides the effect of being able to run backwards from a crash or observed bug, computing a trace of exactly what led up to the problem. In the original paper, we presented the theoretical framework and some preliminary simulation results. This paper recapitulates the basics of the theory and then presents our results moving *BackSpace* to a more realistic design: an OpenRISC 1200 processor implemented in hardware (FPGA). The result is successful, we can run simple software on the processor, at full speed, but then stop the chip at arbitrary states and back up for hundreds of cycles.

I. INTRODUCTION

Post-silicon debug (AKA post-silicon validation, silicon debug, silicon validation) is the problem of determining what’s wrong when the fabricated chip of a new design behaves incorrectly. The focus of post-silicon debug is *design errors*, whereas VLSI test focuses on random *defects* introduced during the manufacturing process of each chip. Post-silicon debug currently consumes a large fraction (e.g., more than half¹) of the total verification schedule on typical large designs, and the problem is growing worse. Even worse, the schedule variability is greatest post-silicon, creating unacceptable uncertainty in time-to-market.

In recent work [1], we proposed *BackSpace*, a new paradigm for using formal analysis, augmented with some on-chip hardware, to support post-silicon debugging. *BackSpace* focuses on the initial task of post-silicon debug: simply finding out what was happening on-chip just (say, the last few thousand cycles) before a crash, observed error, or other behavior of interest. A typical scenario provides motivation for the problem being solved:

After a long and methodical development process, first silicon (or second, or third...) comes back from the fab. Yield is mediocre, but several “good” die pass the manufacturing tests and make it to the bring-up lab. Power-on reset and basic functionality tests work. But 30 seconds into booting the OS, the chip crashes. Worse yet, every single “good die” crashes in the same way.

Scanning out the state of the crashed chips show an internal data structure (e.g., a routing table or a coherence directory) has been corrupted in an inexplicable manner. The logic analyzer dumps from

just before the crash show routine traffic to/from memory and I/O devices. The stimulus to generate the crash (i.e., booting the OS for 30 seconds) is far too deep to replay in simulation or by single-stepping the die, and trying to hit the crash state with full-chip formal verification exceeds the capacity of the formal tools. Increasingly many engineers, from many different teams (pre-silicon verification, design, test, architects, OS, drivers,...) get sucked into the debugging effort. Yet debugging proceeds painfully slowly, because everyone is flying blind, trying to guess what happened.

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

More specifically, the problem is the lack of a certain kind of observability: we can’t observe backwards in time — when the error becomes apparent, it’s impossible to back up and capture the preceding states, which tell us what happened. Although existing test and debug access mechanisms let us start/stop the chip and scan out internal state, we don’t know *when* to capture the state. In the *BackSpace* paradigm, we use techniques from formal verification to compute states backwards in time. In a nutshell, the basic framework consists of adding on-chip circuitry to monitor key operating points and record a signature of the monitored information. During debug, this signature, as well as the crash state, can be scanned out using existing on-chip test access mechanisms. By itself, the signature provides insufficient information (lossy) and would be meaningless to a human, but we can combine this information with formal analysis of what is possible in the design (by computing pre-images) to determine the unique, or nearly unique, predecessor states that led to the crash state. Such an approach can backspace only a limited number of cycles, so we also add circuitry to the chip to allow programmable breakpoints. By setting a predecessor state as a breakpoint, we can re-run the failing test on-chip (e.g., booting the OS), but crash some number of cycles earlier, then scan out the new crash state and signature, and iterate the entire this process to compute arbitrarily long back traces that demonstrate exactly how the bug occurs. The approach exploits the capabilities of different techniques: the actual silicon runs full-speed, rapidly hitting

¹Andreas Kuehlmann, personal communication, September 21, 2006.

bugs that may have escaped pre-silicon validation, but offers very poor visibility and no way to backspace to see how the chip arrived in some state; test-access mechanisms give excellent visibility of the on-chip state, but only for a snapshot in time; formal analysis is slow with poor capacity, but can go forward or backwards equally well; and simulation is too slow to run the real software from reset, but the visibility of a simulation trace is user-friendly and well-accepted for design understanding and debugging.

In the original paper [1], we presented the theory of BackSpace, a sample on-chip architecture with studies of the area impact, and preliminary simulation results on two tiny designs (a few hundred latches), where we showed that we could fully automatically BackSpace several hundred cycles. This paper briefly recapitulates the basic BackSpace theory and then presents our new results in scaling BackSpace to more realistic designs. In particular, we have implemented a multi-thousand latch RISC processor core on FPGA, added our BackSpace capability, and demonstrated the ability to run software applications at full-speed on this processor, yet stop and back-up for hundreds of cycles from arbitrary states.

II. BACKSPACE THEORY

To make this paper self-contained, we present in this section the highlights of the basic BackSpace theoretical framework. Additional details are in the previous paper [1].

A. Intuition and Assumptions

The basic problem is that we have observed the chip in some buggy state, and we have no idea how that could have happened. The goal is to explain the inexplicable buggy state, by creating a “backspace” capability — iteratively computing predecessor states in an execution that leads to the bug. The resulting trace can be viewed like a simulation waveform, except it shows what actually happened just before the bug/crash on the real silicon.²

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

²Many chips contain trace buffers, which appear to provide exactly this capability. However, to keep overhead down, trace buffers are necessarily incomplete, providing only a very narrow record of a few signals in the design. One can view BackSpace as using formal analysis to provide the capability of a perfect trace buffer that records *all* signals.

the bug *ab initio* in simulation is clearly not feasible. Similarly, trying to monitor externally the full execution trace of the chip running full-speed is electrically impossible.

We start with a few simplifying assumptions:

- It must be possible to recover the state of the chip when an error has occurred. For example, this could be done with the chip in test mode, via the scan chain.
- The key assumption is that since we are focusing on *design errors*, we will assume that manufacturing testing has screened out manufacturing defects. Therefore, we assume that the silicon implements the RTL (or gate-level or layout or any other model of the design that can be analyzed via formal tools).
- The bring-up tests can be run repeatedly and the bug being targeted will be at least somewhat repeatable (one out of every n tries, for a reasonably small value of n).

(The original paper [1] discusses relaxing these assumptions, but we do not consider that here.)

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

- 1) Run the chip until it crashes or exhibits the bug. This could be an actual crash or a programmed breakpoint.
- 2) Scan out the full crash state, including the signature.
- 3) Using formal analysis of the corresponding RTL (or other model), compute the possible predecessors of the crash state. The signature must provide enough information to keep this pre-image set reasonably small.
- 4) For each state in the pre-image, try setting it as the new breakpoint and re-run the chip. When the running chip hits one of these states, we have found a predecessor state that leads to the crash, allowing us to repeat this process and go back another cycle.

until we have computed enough of a history trace to debug the design (or Step 3 fails). Each iteration of the loop is like hitting “backspace” on the design – we go back one cycle.

B. Formalization

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

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

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

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

- 1) Let P be the pre-image of (s_i, t_i) , projected onto the original state bits of M , i.e.,:

$$P = \{x \in S \mid \exists i, t [((x, i, s_i) \in \delta) \wedge ((x, t, i, t_i) \in \delta')]\}$$

- 2) Run M' (possibly repeatedly) until it reaches a state (x, y) with $x \in P$. Define $s_{i+1} = x$ and $t_{i+1} = y$.

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

In step 2, we may have to re-run the chip several times before reaching a state in the pre-image P , due to two reasons: non-determinism in the behavior of the chip, and the fact that we can set only one state at a time as a breakpoint, so we must try each state in P individually. Fortunately, there is no combinatorial blow-up — run time grows linearly in the amount of randomness (factor of n slowdown if bug appears in 1 out of n runs) and the size of P . Nevertheless, it’s important to choose a signature function that keeps P small, and in our implementation, we set a bound k and terminate the algorithm if $|P| > k$.

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

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

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

Algorithm 1 performs repeated pre-image computation, which can be expensive. We encountered problems in our

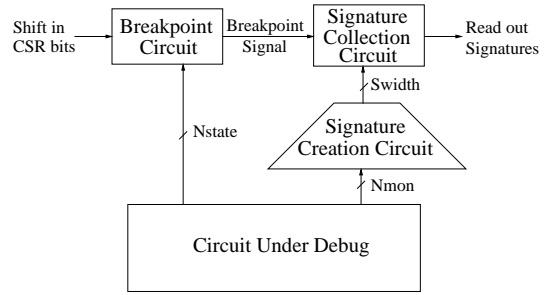


Fig. 1. Debugging Architecture

initial experiments with BDDs and All-SAT. However, due to the small bound k that we impose on the size of the pre-images, we can use a state-of-the-art, off-the-shelf SAT solver to find pre-image states one at a time, greatly alleviating this computational bottleneck in the algorithm.

III. BACKSPACE ARCHITECTURE

The BackSpace framework includes two interacting components: a hardware component and a software component. The hardware component is responsible for collecting signature data from the design, and also for supporting programmable breakpoints. The software component is responsible for two tasks: (1) the “BackSpace Manager” supervises the hardware component and implements the BackSpace algorithm: setting breakpoints, triggering the chip to run its tests, and scanning out states and signatures for formal analysis; and (2) the formal analysis, which given the design’s description and information collected by the hardware component, uses SAT solving to compute possible predecessor states.

A. Hardware Component

A wide variety of architectures for the hardware component are conceivable, but we have chosen a straightforward implementation for our studies. Figure 1 shows how we augment the original Circuit Under Debug (CUD) with a breakpoint circuit, a signature creation circuit, and a signature collection circuit.

During debugging, as the circuit operates, the signature creation circuit monitors N_{mon} of the N_{state} state bits in the CUD. Each cycle, the signature creation circuit uses these state bits to construct a signature of size S_{width} .

The signature is then stored in an SRAM trace buffer within the signature collection circuit. In general, this trace buffer can store many consecutive cycles of signature, allowing multiple cycles of backspacing per run of the chip. For simplicity, however, in this paper, we assume only one cycle of signature.

Meanwhile, the breakpoint circuit also monitors the state bits. When the state bits match a predetermined state (the target state), a signal is sent to stop the collection of signatures. The signature(s) stored in the buffer can then be read out and processed.

The heart of the architecture is the signature collection circuit. The simplest way to construct a signature is to simply use a subset of the state bits directly. If the set of N_{mon} signals

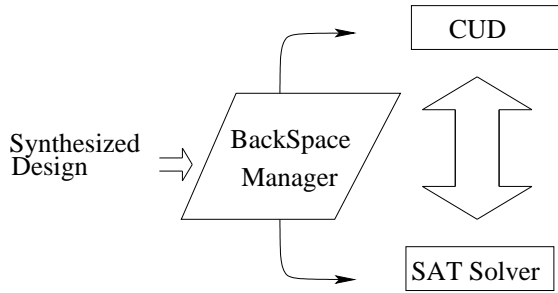


Fig. 2. BackSpace Manager

cannot be determined at fabrication time, the selection of these signals could be made programmable at debug-time using, e.g., a concentrator network [3]. One could also use various hashing techniques to compress the signature.

B. Software Component

1) *BackSpace Manager*: The BackSpace Manager supervises the interactions between the CUD and the formal analysis, automating the computation of the backspace trace.

The BackSpace Manager coordinates the CUD runs and the SAT solving tasks by dispatching each task and processing their intermediate results (shown as the double-headed arrow in Fig. 2). The BackSpace Manager controls the CUD by a set of API commands, and the CUD implements the corresponding logic in hardware to translate such commands into actions, e.g., reset the chip, run it, load the breakpoint circuit, and dump the state and signature.

2) *Formal Analysis*: Given the crash state and the signature, the BackSpace Manager generates a SAT problem instance. When a solution is found, the BackSpace Manager generates a blocking clause based on this solution and asks the SAT solver for another solution. If another solution is found, this process repeats until there are no more solutions, or a pre-specified upper limit is reached. Each solution that the SAT solver finds is a state in the pre-image of the crash state. At that point, a single state or a set of states is available as candidate states prior to the crash state. The task now is to find which candidate state is the actual one. The BackSpace Manager loads a candidate state in the breakpoint circuitry, and re-runs the CUD. If the CUD does not reach the breakpoint, the BackSpace Manager tries another candidate. If the CUD reaches the breakpoint, it means we have a new crash state and a signature. This process continues until we have backspaced some pre-determined number of cycles.

IV. MOVING TOWARDS REALITY: OPENRISC 1200

In our earlier work [1], we tested the BackSpace idea using simulation: we implemented the BackSpace Manager, with pre-image computation and API interface to the CUD, but instead of a real chip on silicon, we simulated the design running on a commercial logic simulator. We added the BackSpace logic to two small open-source microcontrollers, with 109 and 702 latches, respectively. For signatures, we tried two points in the design space: a hand-chosen subset of the state bits

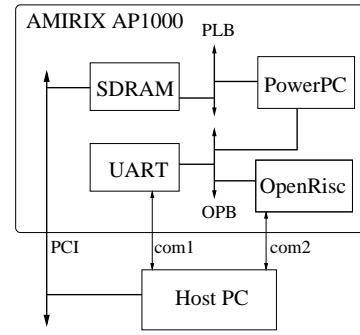


Fig. 3. OpenRISC 1200 Implemented onto AMIRIX AP1000 Board

(38 out of 109 latches, and 281 out of 702), or a universal hash of the same size of all of the state bits. The hand-chosen subset is an easy signature to implement, but doesn't capture much information, whereas universal hashing captures information from the entire state, but a naive implementation uses a prohibitive amount of area. We simulated the designs for an arbitrary number of cycles and randomly selected 10 states each to serve as "crashed" states for our analysis, with the goal of seeing whether we could use the BackSpace technique to compute backwards from the "crash" state a sequence of states that led up to the crash. The results were very encouraging: for 80% of the target states, we could easily backspace up to the 500-cycle limit we had set for our experiments. (Details of these results are reported in [1].)

Our current work is to move BackSpace closer towards reality, by stepping up the size, complexity, and realism from the preceding two microcontrollers, and by implementing BackSpace on a design in actual hardware. We have selected a classic RISC processor with 32-bit datapaths and a 5-stage integer pipeline: the OpenRisc 1200 from opencores.org. The configuration we are using has 3007 latches in the processor core plus one UART for I/O.

Our hardware implementation is on an AMIRIX AP1000 FPGA development board. Figure 3 shows the overall architecture. The development system consists of a PC workstation (the "host PC"), with the AMIRIX board mounted in one of its PCI slots. The AMIRIX board has a Virtex-II Pro FPGA, containing a PowerPC and a PCI bridge hard core in addition to the programmable logic, as well as an SDRAM memory subsystem and two UART ports.

We use the FPGA's programmable logic to implement the OpenRisc, a UART for I/O, and the hardware components for BackSpace (breakpoint and signature circuits). So, the programmable logic corresponds to the CUD on silicon in a real debug scenario. We have programmed the PowerPC to act as a middleman between the host PC and the OpenRisc, as well as serving as the memory controller. Thus, the PowerPC and SDRAM emulate the rest of the bring-up system for the OpenRisc. The host PC functions as the test/debug controller, where the software components of BackSpace (the BackSpace Manager and formal analysis) run.

The basic implementation of the OpenRisc on the FPGA

was fairly standard, although it did require some ingenuity. We started with a different design provided by CMC Microsystems (who also supplied the prototyping system), in which the PowerPC communicates with on-board SDRAM through the processor local bus (PLB) and provides serial communication to the host through a UART sitting on the on-chip peripheral bus (OPB). We added our OpenRisc processor implementation to the OPB bus. The OpenRisc also includes a UART, which was connected directly to the second serial port on the AMIRIX AP1000 board. This allows running software applications on the OpenRisc, using this UART port for console I/O. The host PC and the PowerPC communicate via a “mailbox” mechanism on the SDRAM.

Our implementation of the BackSpace hardware was also fairly straightforward. The OpenRisc design did not have scan, so we added full scan manually. (We did scan insertion manually to avoid having to interface our BackSpace software to a commercial scan insertion tool.) For a signature, we hand-selected 1276 out of the 3007 latches, without any further compression. To provide a more flexible environment for experimentation, we implemented a breakpoint circuit that matched all state bits, but also allowed partial matches by masking off bits. This capability proved very helpful when we were initially debugging our design.

On the software side, the BackSpace Manager and SAT solver ported to the host PC with minimal changes. The primary tasks were implementing the API interface for the BackSpace Manager to communicate with the PowerPC via the mailbox mechanism, and for the PowerPC to control the OpenRisc. For example, to begin backspacing, the host PC writes a *reset* command to the mailbox; the PowerPC then resets the OpenRisc. Next, the host PC writes a *run* command, and the PowerPC starts the OpenRisc. For these experiments, we specify some number of cycles for the OpenRisc to run before crashing. As the OpenRisc is running, signatures are generated and collected at every clock cycle. The PowerPC stops the OpenRisc when the number of cycles hits the target, after which the OpenRisc’s current state and signature are written into memory. The host PC reads them and computes the set of candidate predecessor-states (the pre-image). For each candidate, the host PC writes a *load* command to load it into the breakpoint circuitry and requests the OpenRisc to run until a state matches the breakpoint. When the breakpoint is hit, it means we have a new state and a signature to work with. Otherwise, the BackSpace manager keeps trying until eventually finding the right candidate. Altogether, the BackSpace Manager will automatically and accurately compute a backward trace of arbitrary length from the crash state.

The entire system is now running. We can load simple software applications onto the OpenRisc, run, and then BackSpace at will.

We report in Table I some results on using BackSpace on this system. We show results for two simple application programs: the Euclidean Algorithm for greatest common divisor, the Sieve of Eratosthenes for computing prime numbers. For target/crash states, we picked 3 states during the run of each

program. For the GCD program, these states were tens of thousands of cycles deep after reset; for the prime number program, they were roughly 300,000 cycles deep.

The results are excellent. Even with the simple signature, we were able to backspace for hundreds of cycles from all crash states, and hitting our self-imposed experimental limit of 500 cycles in 3 of the 6 cases. Runtimes were typically a few hours, with most of the time spent on communication overhead between the CUD and the debug manager.

As in larger, real-life designs, our implementation has non-determinism. In this system, the source of the non-determinism is the variable memory access time. Exactly as predicted by our theory, we can handle this non-determinism, but it produces a run-time slowdown, because we must repeatedly try to hit each breakpoint.

Similarly, in our earlier work, we noted the need for good signatures that constrain the number of states in the pre-image. The current implementation results confirm that need, as the average size of the pre-image set also produces a slowdown, and these two slowdown factors combine. For example, *gcd1* has only 17 candidate states in most of the pre-image computations. However, the pre-image size impact over runtime gets compounded with non-determinism. For example, the 30.5 average retries per cycle during one BackSpace run means that the BackSpace Manager had to request almost two chip runs, on average, for each of the 17 candidate states. On the other hand, when the pre-image size is 1 for most of the pre-image computations, e.g., *prime1*, the run time improves considerably. Curiously, we see a wide range on the number of retries over all crash states. The intuition here is that some of the state bits and/or some segments of the running application may be more susceptible than others to non-determinism. We believe that architectural insight (choosing to mask out some state bits/inputs) might help reduce this problem.

CONCLUSION AND FUTURE WORK

In brief, it works: We have a backspaceable processor implemented in FPGA, capable of running software at full speed, stopping at arbitrary states, and backing up for hundreds of cycles.

The main direction for future work is to reduce the hardware overhead. We are exploring several concrete ideas (e.g., constraining the inputs during pre-image computation, partial matching of breakpoints) and are very hopeful we can greatly reduce the overhead.

ACKNOWLEDGMENTS

This work was funded by the Semiconductor Research Corporation (SRC) Task ID: 1586.001. The AMIRIX prototyping board, the workstation supporting it, and access to commercial CAD software were supplied by the Canadian Microelectronics Corporation (CMC Microsystems).

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*. IEEE/ACM, 2008.

Crash State	# of Cycles BackSpaced	Max States in PreImg (freq)	Most Frequent PreImg Size (freq)	# Retries per cycle	Run Time	Sat Time	Manager Time
gcd1	322	> 300(1)	17(321)	(30.5; 31.0; 25.4)	(9,840; 11,302; 8,952)	366.3	2,440.0
gcd2	442	> 300(1)	17(441)	(27.1; 35.0; 30.1)	(11,982; 17,134; 14,460)	496.3	3,330.0
gcd3	500	34(1)	17(499)	(28.7; 21.1; 32.34)	(14,387; 10,562; 17,936)	556.0	3,774.0
prime1	500	80(1)	1(498)	(2.4; 15.2; 4.7)	(1,826; 7,637; 3,026)	539.7	3,781.7
prime2	500	80(1)	64(100)	(58.3; 34.8; 23.3)	(31,718; 19,270; 13,345)	565.0	3,729.7
prime3	369	> 300(1)	64(100)	(14.1; 24.8; 14.3)	(5,910; 10,118; 6,226)	420.6	2,783.3

There are two programs, gcd and prime. For each program, we selected 3 crash states from which to attempt to backspace as far as possible, up to a pre-set limit of 500 cycles. For each of these states, we repeated the backspace computation 3 times. We set an upper-bound of 300 for the size of a pre-image set; if a pre-image exceeded that size, we terminated that run. The third and fourth columns give statistics on the pre-image sizes. For each crash state run, we report “# Retries per Cycle”, an average of retries over the number of backspaced cycles and “Run Time”, the total elapsed time spent between the time the BackSpace Manager issued a run command and the time the new ‘crash’ state is available in memory. Because we ran each crash state 3 times, we report 3 numbers in each column. The variation is due to the non-determinism in the hardware. “Sat Time” is the total elapsed time spent in the SAT solver. “Manager Time” is the total time spent by the BackSpace Manager to supervise the framework and connect the various tools. “Sat Time” and “Manager Time” had minimal variance, so we report the averages over the 3 crash state runs.

TABLE I
RESULTS FOR BACKSPACING THE OPENRISC 1200

- [2] N. Eén and N. Sörensson, “An extensible SAT solver,” in *SAT '03: Proc. of the 6th International Conference on theory and Applications of Satisfiability Testing*, ser. LNCS, vol. 2919. Springer, 2003, pp. 502–518.
- [3] B. Quinton and S. Wilton, “Concentrator Access Networks for Programmable Logic Cores on SoCs,” in *IEEE International Symposium on Circuits and Systems*, 2005, pp. 45–48.