Accelerating Trace Computation in Post-Silicon Debug

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

Abstract— Post-silicon debug comprises a significant and highly variable fraction of the total development time for large chip designs. To accelerate post-silicon debug, BackSpace [1, 2] employs on-chip monitoring circuitry and off-chip formal analysis to provide a trace of states that lead up to a crash state. BackSpace employs repeated runs of the integrated circuit being debugged, which can be time consuming. This paper shows that correlation information characterizing the application running on the hardware up to the crash state can reduce the number of runs of the chip by up to 51%.

1. Introduction

Post-silicon debug (also known as silicon validation) is the process of determining what is wrong when the fabricated chip of a new design behaves incorrectly. The focus of post-silicon debug is finding *design errors*. Ideally all design errors would be found before fabrication using simulation, however, the billion-to-one slowdown between actual silicon and simulation means that it is impossible to simulate a design in all scenarios in which it will be used. Thus, regardless of how much effort is spent before fabrication, some design errors will escape into silicon. Finding these errors quickly is a vital, but difficult and expensive, process. Abramovici et al. [3] show that the percentage of total chip development time spent on post-silicon debug is over 35% and growing. Even worse, the schedule variability is greatest post-silicon, creating unacceptable uncertainty in time-to-market.

It is important to distinguish between post-silicon debug, in which the focus is finding *design errors*, and manufacture testing, in which the focus is finding *manufacturing defects*. In this paper, we are focusing on the former.

The primary challenge during post-silicon debug is the lack of observability into the chip. Scan chains, which likely already exist on the chip to support manufacture test, can be used to read out the state during debug once an incorrect behavior is observed. This requires stopping the chip, however, which may make it difficult to trace the execution leading up to an error. Designers often include small amounts of circuitry on-chip to enhance visibility during debug, however, this requires predicting during design which parts of the chip will need the extra visibility during debug.

A structured approach to post-silicon debug, called BackSpace, was presented in [1, 2]. Through a combination of on-chip monitoring circuitry and off-chip analysis using formal methods, BackSpace produces a trace of states that leads up to a known crash state. This trace of states can give a debugging engineer valuable information about the cause of the bug, increasing his or her chances of finding the design error that led to the incorrect behavior.

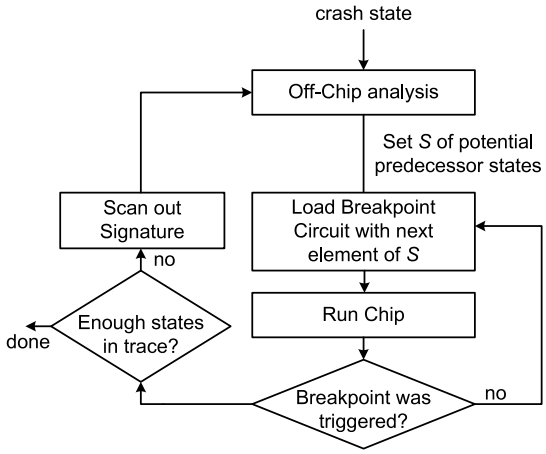

Fig. 1: BackSpace Debug Flow

The primary challenge with BackSpace is the time needed to construct this trace. BackSpace uses an iterative algorithm which requires running the chip many times. In this paper, we show that by gathering additional information about a circuit as it computes the trace leading up to a crash, and using this information to guide the BackSpace search algorithm, we can obtain significant run-time improvements. For the examples we evaluate, the number of runs of the chip was reduced by between 12% and 51% (leading to significant reduction in overall run-time) without affecting the accuracy of the resulting trace. This is important, since the usefulness of BackSpace relies on the debugging engineer being able to quickly and accurately obtain such traces.

This paper is organized as follows. Section 2 summarizes the BackSpace algorithm, and Section 3 compares this approach to previous work. Section 4 then presents our enhancements. Section 5 describes our methodology, and Section 6 presents experimental results. Section 7 concludes.

2. BackSpace

The overall flow of BackSpace is shown in Figure 1. When the chip crashes, the crash state is scanned out through the scan chain. A state of the chip is defined as the values of all flip-flops in the design, including all datapath registers, pipelining registers, and state machine flip-flops. This crash state is then sent to an off-chip analysis algorithm. This algorithm uses

formal verification techniques to compute a set S of potential *pre-image states*. The set S contains *all* states s_i that *could have* occurred one cycle prior to the crash state. Since there is often more than one way to get to a given state, there will likely be more than one pre-image state in S. In essence, this off-chip algorithm "works backwards" through the state machine.

Each pre-image state in S is then considered individually. The first state, s_0, is loaded into an on-chip breakpoint circuit, and the chip is re-run. If the chip passes through state s_0, a breakpoint is triggered. This indicates that s_0 is the correct predecessor state, so it is added to the trace. If the chip does not pass through s_0 (indicated by a time-out circuit), then the next potential predecessor state, s_1, is tried, and the process repeated. Ideally, one of the states in S will trigger a breakpoint. This predecessor state can then be used as input to the off-chip analysis algorithm, and the entire process repeated.

In practice, external events and non-determinism in the execution flow may mean that the execution times out for every state in S, even when the correct predecessor state is loaded into the breakpoint circuit. In this case, the chip is re-run until a breakpoint is triggered on one of the predecessor states.

For this to be feasible, the number of possible predecessor states in S must be as small as possible. To reduce the size of S, de Paula et al. [1] proposes gathering additional information, called a *signature*, during each run. This information can be used to "prune out" states in S which are not possible. There are several ways to construct a signature; in our implementation of BackSpace, we hand-select a subset of the state bits, and maintain a one-cycle history of these bits. This one-cycle history can then be used to rule out many of the states in S.

Even with a signature, however, the number of states in S may be large. For example, in our implementation of an Intel 8051 processor which contains 702 flip-flops, if we select 281 of them as signature bits, there are, on average, 31 elements in S. Since we try each of these individually until a match is found, we must try, on average, half of the 31 elements in S.

Clearly, the order that we try the elements of S is important. In the original implementation of BackSpace, the elements are ordered arbitrarily. In this paper, our key contribution is that we use additional information gathered for the circuit to *re-order the elements of S so states in S that are more likely are tried first*. This results in a reduction in the number of runs required to construct a trace.

3. Related Work

BackSpace helps with the observability problem after a bug has been detected. There has been some complementary work on detecting bugs. Online assertion checking [3] can be used to speed up the BackSpace process by initiating debugging when an assertion is violated rather than waiting until the problem becomes observable at the outputs. DIVA [4] is a proposal that suggests re-executing each instruction a processor commits using a simpler processor more likely to be free of errors. If an error is detected, it is fixed by using the result from the simpler processor and flushing subsequent instructions from the pipeline. DIVA could be employed to stop a processor earlier so that BackSpace starts running closer to the source of the problem causing a crash.

For systems that are completely deterministic a system much simpler than BackSpace could be used to provide backwards observability. Checkpointing with deterministic replay [5, 6] attempts to remove all sources of non-determinism using methods such as logging all external connections. Some subtle sources of non-determinism like that arising from correctable soft errors *during replay* are not addressed. An alternative method of removing system level non-determinism is to run two copies of the design with the same inputs but the second copy's inputs are delayed by n cycles using a buffer. When the first copy encounters an observable problem the second copy is also stopped and it's state is examined. If both chips implemented the same design, were completely deterministic, and were supplied with the same inputs (with a delay for the second copy) then the state of the second copy would be the state of the first copy n cycles in the past. This approach has been proposed for use with FPGAs [7].

Examples of other approaches to post-silicon debug include DACOTA and IFRA. DACOTA [8] is a proposed method to find coherence and consistency bugs in the memory subsystem of multi-core microprocessor designs. It leverages the execution speed and cache space of fabricated processors to provide trace buffering. However, it is limited to detecting memory bugs that involve more than one core. Park and Mitra [9] record data and control flows during execution and use the information in an analysis phase after a problem is detected to help localize a bug. They evaluated their IFRA technique by injecting soft errors in an architectural simulator.

4. Algorithm

The key idea behind our algorithm is to gather information about the *correlation* of bits in the design, and use this information to re-order the elements of S so that states that are more likely to be the correct predecessor state are tried first. In the following, we first describe the correlation information itself, and then show how this information is used to guide BackSpace. Finally, we describe two techniques by which the correlation information can be obtained.

4.1. Correlation Information

Since our goal is to determine the most likely predecessor state for a given crash state, an obvious solution would be to find the correlation of each state in cycle i with every state in cycle $i + 1$. The correlation between two states would be high if the first state was likely to occur exactly one cycle before the second state, and low if this was unlikely. This information could be gathered by running chip for a long period of time, and recording state transitions. Then, during debugging, when the set S is obtained from the off-chip analysis algorithm, we could sort S based on these correlation numbers. While this technique would likely give good results, it is not feasible, even for medium-sized designs. A circuit with n flip-flops (and hence n state bits) could have up to

2^n possible states, meaning we would need to compute and maintain 2^{2n} correlation values.

Instead, we compute the correlation between *individual bits within a state*. For any two state bits x and y, the correlation between these bits, $\rho_{x,y}$ indicates how likely these two bits are to be either positively or negatively correlated. More precisely,

$$\rho_{x,y} = \frac{1}{T}\sum_{i=1}^{T}(2x_i - 1)(2y_i - 1) \quad (1)$$

where x_i is the value of bit x in cycle i, and y_i is the value of bit y in cycle i. T is the number of cycles of correlation information that has been gathered so far. Section 4.3 discusses different methods of gathering this information. The value of $\rho_{i,j}$ is -1 if the value of bit i is always the inverse of the value of bit j and 1 if the value of bit i is always the same as the value of bit j. The value of $\rho_{i,j}$ is 0 if knowing the value of one bit provides no information about the value of the other bit. Thus, we can use the absolute value of $\rho_{i,j}$ to determine how correlated two bits are. Note that, unlike the first solution outlined above, this only requires computing and storing n^2 correlation values. Although this correlation information does not provide as much information as the first solution, we will show that it provides sufficient information to produce an intelligent ordering of the states in S.

4.2. Using Correlation Information

Consider a set of potential pre-image states S that are potential predecessors of a crash state c. We wish to sort the elements in S such that the states that are *most likely* to be the predecessor of c are first in the list. In this subsection, we show how we do this, given the correlation information described in the previous subsection.

The individual states in S differ by some number of bits. If we can predict the most likely value of these bits, then we can predict which of the states in S is the most likely to have occurred immediately before c. We do this in two steps: pre-image merging, and pre-image ranking. Both are described below, and the overall algorithm is summarized in Algorithm 1.

During pre-image merging, we find a word w which summarizes the set of elements in S. Bit i of w is determined as follows. If the corresponding bit in *all* elements in S is the same then bit i of w is set to this value. If bit i differs across the elements of S, then bit i of w is set to X.

Consider the example in Figure 2. The set S contains three potential predecessor states s_0, s_1, and s_2. In all predecessor states, bit 3 is a 0 and bit 4 is a 1. Thus, bit 3 of w is set to 0, and bit 4 of w is set to 1. Bits 0 to 2 differ across the set of predecessor states. Thus, we set bits 0 to 2 of w to X.

Next, we perform pre-image ranking. To do this, we first convert the word w into a prediction p of the predecessor state. For each bit i in w that is either 0 or 1, bit i of p is set to this value. For the bits in w that are X, the following is performed. The correlation information described earlier is first used to determine which other bit j in the design is most correlated to this bit (i.e. has the highest value of $|\rho|$, ties broken by minimizing $|i - j|$, further ties broken by smallest

	b_0	b_1	b_2	b_3	b_4
s_0	0	0	1	0	1
s_1	0	1	0	0	1
s_2	1	0	0	0	1
merged	X	X	X	0	1

Fig. 2: An example of merging the pre-image set S.

j). If this other bit's value is known, then either that value or it's complement is used in the corresponding location in p. If $\rho_{i,j}$ is positive or zero, the value is used, while if it is negative, it's complement is used. If the bit value is unknown, the corresponding location of p is again marked with an X.

It is important to note that the prediction p constructed in this way may not correspond to an actual (or even possible) state. It is actually an amalgamation of the predictions for each individual bit. Thus, the final step is to go through each element s_i in S and find the Hamming Distance between s_i and p (if the bit in p is an X, then this bit contributes 0 to the Hamming Distance). The set S is then sorted based on this Hamming Distance (lowest Hamming distance comes first).

In this way, the set of S is ordered in a manner such that the states that are "closest" to the predicted previous state are tried first. This ordered set is then used in the BackSpace flow as described in Section 2.

4.3. Gathering Correlation Information

Subsection 4.2 described the correlation information, but it did not indicate how this information can be gathered from the design. We consider two approaches to gather this information. The first approach is to gather the information statically, before the chip is debugged. Using a "training testcase", the design can be simulated (or the fabricated chip can be run) and individual states extracted. For a processor design, a training testcase can be a program executing on the processor; for a non-processor design, the training testcase can be a list of input values over time. From these extracted states, the correlation information for each pair of bits can be calculated. The summation limits in Equation 1 correspond to the start and end of the training testcase.

Experimentally, we have found that the static technique does not provide adequate information to guide BackSpace. The reason is that the state transition behavior of a design depends very much on the inputs or the program executing. It is unlikely that a short training testcase will be representative of the program or inputs occurring in the actual chip during debug.

Instead, we use a dynamic approach, in which we gather the correlation *during debugging*. As described in Section II, BackSpace is iterative; for each state added to the trace, the chip is run one or more times. In our dynamic solution, we compute the correlation information each time as states are scanned out of the chip and added to the trace history. Initially, BackSpace proceeds without any correlation information, however, later iterations are able to use the correlation information gathered during previous iterations. As we will show, this technique provides sufficient information to accelerate the BackSpace flow.

Input: Unsorted pre-image list S
Number of state bits n
Correlation matrix $\rho_{i,j} \quad \forall\ 0 \leq i, j < n$
Output: Sorted list S'
foreach *bit* i, $0 \leq i < n$ **do**
  if *bit* i *of* s_j *is the same* $\forall s_j \in S$ **then**
    bit i of w = bit i of s_0;
  else
    bit i of w = X;
  end
end
foreach *bit* i, $0 \leq i < n$ **do**
  if *bit* i *of* w *is not X* **then**
    bit i of p = bit i of w;
  else
    $r = j$ such that $|\rho_{i,j}|$ is maximum over all $0 \leq j < n,\ j \neq i$;
    if *bit* r of w *is X* **then**
      bit i of p = X;
    else if $\rho_{i,r} \geq 0$ **then**
      bit i of p = bit r of w;
    else
      bit i of p = $\overline{\text{bit } r \text{ of } w}$;
    end
  end
end
foreach $s_j \in S$ **do**
  k_j = HammingDistance (s_j, p);
end
S' = Sort S by key k_j;

Algorithm 1: Overall Algorithm for producing sorted pre-image list

5. Methodology

We use an 8051 simulator and benchmarks from the BackSpace Toolkit version 0.1 [10]. To evaluate our algorithm for sorting the states in S we collected and analyzed traces of the 8051 running the three programs shown in Table I. The 8051 used is an open-source implementation of an 8-bit Intel micro-controller available from opencores.org. Data was not taken from the first few hundred cycles because there are uninitialized flip flops in the design. The first cycle in the trace is referred to as the start cycle. After the program finishes execution the 8051 enters a 12 cycle loop. After one iteration of this loop we stop the trace and label that the end cycle of the trace. The start and end cycles for the benchmarks used are shown in Table I.

As described in Section 4.3 the correlation information is obtained as BackSpace is running. For the first 300 cycles of a BackSpace computation we do not apply the algorithm described in Section 4 and BackSpace proceeds as the baseline described in Section 2. After BackSpace computes a trace of 300 cycles leading up to the crash we compute the correlation ρ defined in Equation 1 for all $\frac{(702)(701)}{2}$ pairs of unique bits in the same state over the 300 cycles available. We can now use this information to sort the potential pre-image sets obtained for the next 300 cycles. After a trace of 600 cycles is obtained

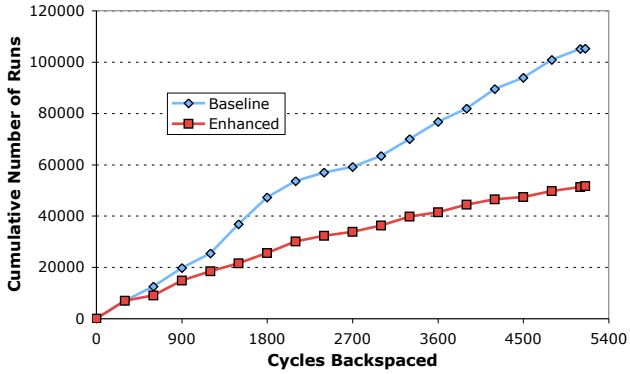

Fig. 3: Cumulative processor runs for the 8051 running `sqroot`. Baseline is the original BackSpace algorithm. Enhanced is the BackSpace algorithm using the algorithm proposed in Section 4 of this paper.

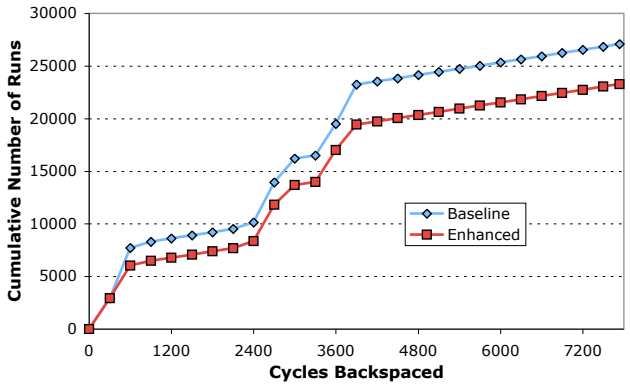

Fig. 4: Cumulative processor runs for the 8051 running `sort`.

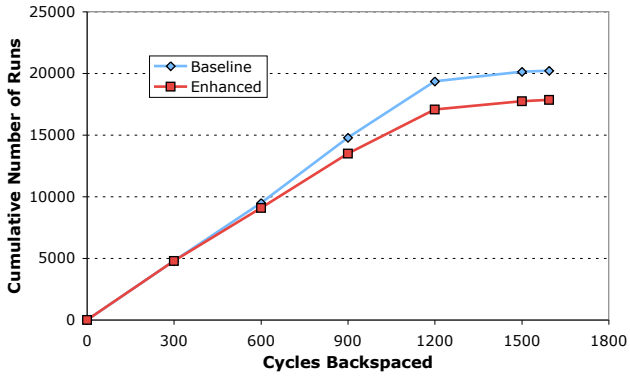

Fig. 5: Cumulative processor runs for the 8051 running `fib`.

we re-calculate the correlation values and use the updated values for the next 300 cycles. BackSpace continues in this fashion, updating the correlation values every 300 cycles, until the beginning of the trace is reached.

For the evaluation in this paper, the simulated 8051 processor was not run for every candidate state in S, as described in Section 2, to determine if it is the correct state. Instead, the sorted candidate states are compared with the previous state in the trace to determine how many runs it would take before the correct pre-image state is tried.

TABLE I: Programs that were run on the 8051.

Program Name	Start Cycle	End Cycle
fib	630	2224
sort	630	8368
sqroot	660	5813

6. Results

The results for running `sqroot`, `sort`, and `fib` can be seen in Figures 3, 4, and 5 respectively. The right-most point on each of these graphs shows the total number of times the chip must be run to BackSpace the entire trace. In Figure 3 the ratio between the cumulative number of processor runs for the enhanced algorithm and the baseline is decreasing. By the time BackSpace traces through all states in the `sqroot` program there is a 51% reduction in the number of times the processor must be run relative to the baseline. Figure 4 shows the result for `sort`. After backspacing 3900 cycles the lines for our enhanced algorithm and the baseline are parallel. This is because there is only one candidate predecessor state for those cycles and the baseline algorithm cannot be improved upon. This can be seen more clearly in Figure 7. Upon completion of this trace, there is a cumulative reduction of 14% in the total number of processor runs required. Figure 5 shows the result for `fib`. Although BackSpace was only run for 1594 cycles for this program, the results show that as BackSpace is run, our enhanced algorithm requires a smaller fraction of the number of processor runs required by the baseline. The overall improvement for this program is 12%.

The average number of runs needed over each 300 cycle epoch for `sqroot`, `sort`, and `fib` can be seen in Figures 6, 7, and 8 respectively. Each point shows the number of runs needed to reach the correct predecessor state averaged over 300 cycles. For each of these results the first point, at 300 cycles backspaced, is the same for both our enhanced algorithm and the baseline. This is because there has been no correlation data gathered for that program yet. The next point, at 600 cycles backspaced, shows the impact of using 300 cycles of correlation data to sort the candidate pre-images. In Figure 6 we see that our enhanced algorithm reduces the number of runs needed over a 300 cycle period by up to 77% (between 4201 and 4500 cycles backspaced) for the `sqroot` program. Over the entire trace the reduction in the number of times the chip would need to be run is 51%. With the `sort` program shown in Figure 7, we only see a 14% reduction in the number of times the chip needs to be run. This is partly due to the many states that only have one candidate pre-image. Figure 8 shows the result for the `fib` program, where the number of chip runs is only reduced by 12% overall. One of the reasons the overall improvement is lower for this program is because it is short. For about the first 20% of the cycles backspaced there is no correlation to sort the candidate pre-images with.

6.1. Alternative Algorithms

To gain further insight, we tried several other algorithms. First we investigated using the state the processor is currently stopped at as the prediction p. Intuitively this would work

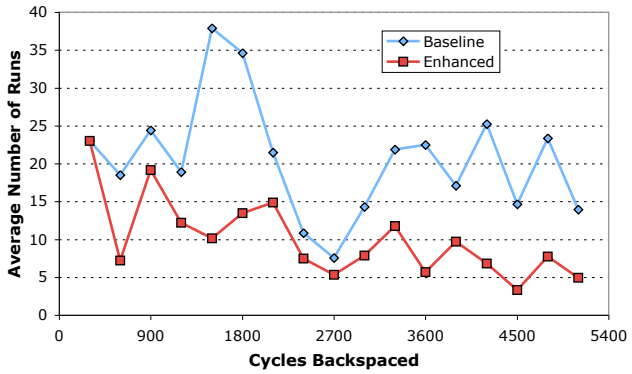

Fig. 6: Average processor runs per correlation sample interval for the 8051 running `sqroot`. Enhanced is the BackSpace algorithm using the algorithm proposed in Section 4 of this paper.

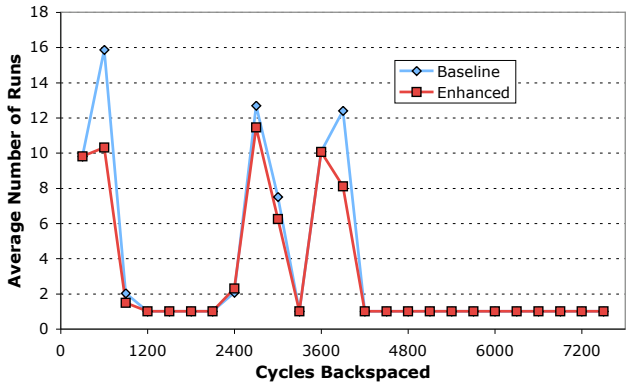

Fig. 7: Average processor runs per correlation sample interval for the 8051 running `sort`.

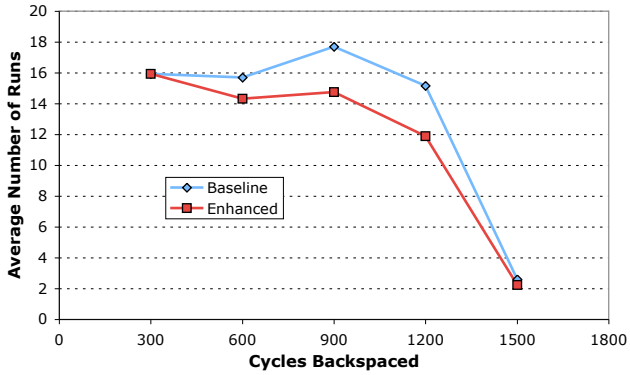

Fig. 8: Average processor runs per correlation sample interval for the 8051 running `fib`.

if the unknown bits of the state do not change very much from cycle to cycle. This algorithm (referred to as "Guess No Change" in Table II) was able to predict the correct state and reach the correct candidate pre-image on the first run more often than the baseline, however, the overall performance was 6% worse than the baseline. For the circuit we used, it was found that the state bits have a zero bias, that is there were more 0 bits in the trace than 1 bits. Guessing 0 for all the unknown bits, and using the baseline ordering to resolve ties, results in approximately the same performance as the baseline. Randomizing the order before sorting based on a

TABLE II: The relative number of runs required to BackSpace the different programs relative to the baseline.

	sqroot	sort	fib
Enhanced	0.49	0.86	0.88
Randomized Guess 0	1.02	1.09	1.07
Guess No Change	1.07	1.09	0.94
Baseline	1.00	1.00	1.00
Random	1.25	2.11	2.06

guess of 0 (referred to as "Randomized Guess 0" in Table II) for all the unknown bits results in performance 4% worse than the baseline, but still requires 34% fewer runs than a random ordering. These results may suggest that a portion of the performance improvement of the order tried by the baseline BackSpace algorithm over a random ordering results from the tendency of the baseline BackSpace algorithm to order states with more 0 bits first. The algorithm we propose in this paper ("Enhanced") performs better than the above simpler algorithms.

7. Conclusion

BackSpace generates a trace showing the state of the chip leading up to a crash. Each cycle added to the trace may require running the chip multiple times in order to reach the previous state. We proposed a technique where we use the trace generated so far to compute the correlation between state bits. Using this correlation information, we then make a prediction about which potential previous states to try first. Our results showed between 12% and 51% reduction in the number of times the chip must be run over an entire trace computation, which translates into a significant reduction in runtime. In the future, we would like to investigate how our algorithm scales with larger chip designs.

Acknowledgments

We would like to thank Wilson Fung, Ali Bakhoda, and the anonymous reviewers for their valuable comments. This work was funded by the Semiconductor Research Corporation(SRC) TaskID: 1586.001.

References

[1] Flavio M. De Paula, Marcel Gort, Alan J. Hu, Steven J. E. Wilton, and Jin Yang, "BackSpace: Formal Analysis for Post-Silicon Debug," in *Formal Methods in Computer Aided Design*, 2008.

[2] Flavio M. De Paula, Marcel Gort, Alan J. Hu, and Steven J. E. Wilton, "BackSpace: Moving Towards Reality," in *International Workshop on Microprocessor Test and Verification*, December 2008, pp. 49–54.

[3] 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.

[4] T.M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," in *32nd ACM/IEEE International Symposium on Microarchitecture (MICRO-32)*, 1999, pp. 196–207.

[5] S.R. Sarangi, B. Greskamp, and J. Torrellas, "CADRE: Cycle-Accurate Deterministic Replay for Hardware Debugging," in *International Conference on Dependable Systems and Networks (DSN 2006)*, June 2006, pp. 301–312.

[6] Isic Silas, Igor Frumkin, Eilon Hazan, Ehud Mor, and Genadiy Zobin, "System-Level Validation of the Intel Pentium M Processor," Tech. Rep., May 2003.

[7] Mario Larouche, "Infusing Speed and Visibility into ASIC Verification," Tech. Rep., January 2007.

[8] A. DeOrio, I. Wagner, and V. Bertacco, "Dacota: Post-silicon validation of the memory subsystem in multi-core designs," in *15th IEEE International Symposium on High Performance Computer Architecture (HPCA 2009)*, Feb. 2009, pp. 405–416.

[9] Sung-Boem Park and S. Mitra, "IFRA: Instruction Footprint Recording and Analysis for post-silicon bug localization in processors," in *45th ACM/IEEE Design Automation Conference*, June 2008, pp. 373–378.

[10] Flavio M. De Paula, "Backspace," http://www.cs.ubc.ca/ depaulfm/BackSpace, 2009.