BackSpace: Formal Analysis for Post-Silicon Debug

by

Flavio Miana de Paula

M.Sc. in Computer Science, University of British Columbia, 2007
M.Sc. in Electrical Engineering, Universidade Federal de Minas Gerais, 1999
B.Sc. in Electrical Engineering, Universidade Federal de Minas Gerais, 1996

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

May, 2012

Abstract

IC technology continues to closely follow Moore's Law, while the ability to verify designs lags behind. The International Technology Roadmap for Semiconductors (ITRS) predicts production of chips using 16nm technology already by 2015, but the verification gap, i.e., advancements in verification technology not keeping up with advancements in design technology, seems to be also increasing at a fast pace. A recent study shows a drop of more than 10 percentage points in the number of 1^{st}-silicon success from 2002 through 2009. By 2007, more than two-thirds of chips had to be respun due to bugs. The increasing verification gap is to blame. Unfortunately, because more bugs are slipping into the fabricated chip, post-silicon debug is the only way to catch them.

Post-silicon debug 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 traditional VLSI test focuses on random manufacturing *defects* on each fabricated chip. Post-silicon debug currently consumes more than half of the total verification schedule on typical large designs, and the problem is growing worse.

The general problem of post-silicon debug is broad and multi-faceted, spurring a diverse variety of research. In this thesis, I focus on one of the most fundamental tasks: getting an execution trace of on-chip signals for many cycles leading up to an observed bug or crash. Until such a trace is obtained, further debugging is essentially impossible, as there is no way to know what happened on the chip. However, the ever-increasing chip complexity compounded with new features that add non-determinism makes computing accurate traces extremely difficult.

Thus, to address this issue, I present a novel post-silicon debug frame-

work, which I call BackSpace. From theory to practice, I have methodically developed this framework showing that BackSpace effectively computes accurate traces leading up to a crash state, has low cost (*zero*-additional hardware overhead), and handles non-determinism. To support my claims, I demonstrated BackSpace with several designs using simulation models, hardware prototypes, and on actual silicon.

Preface

I conducted the research presented in this thesis in collaboration with my PhD supervisor Alan Hu. The following papers resulted from my research and were originally published with the corresponding collaborators as co-authors:

1. Flavio M. de Paula, Marcel Gort, Alan J. Hu, Steven J. E. Wilton, Jin Yang. BackSpace: Formal Analysis for Post-Silicon Debug. *Formal Methods in Computer-Aided Design, FMCAD'08*, IEEE Press, 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, MTV'08*, pages 49-54, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

3. Flavio M. de Paula, Amir Nahir, Ziv Nevo, Avigail Orni, and Alan J. Hu. Tab-Backspace: Unlimited-Length Trace Buffers with Zero Additional On-Chip Overhead. *In Proceedings of the 48th Design Automation Conference, DAC'11*, pages 411-416, New York, NY, USA, 2011. ACM.

4. Flavio M. de Paula, Alan J. Hu, Amir Nahir. nuTAB-BackSpace: Rewriting to Normalize Non-Determinism in Post-Silicon Debug Traces. To appear *In Proceedings of the 24th International Conference on Computer-Aided Verification, CAV'12*, Berkeley, CA, USA, 2012. Springer-Verlag.

Since I expanded all these papers into chapters, some of the sentences and paragraphs were authored by Alan J. Hu. In addition, Section 3.4.2 is

mostly based on the text written by Avigail Orni (IBM-Israel), with whom I collaborated on item 3.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I acknowledge Prof. Alan J. Hu whose support and mentoring have been essential through these years. I would also like to acknowledge: Prof. Mark Greenstreet for his inspiring lectures in both theory of computation and parallel computation courses; my fellow colleagues in the Integrated Systems Design Laboratory, Zvonimir Rakamaric and Brad Bingham, whose diverse interests made this journey more enjoyable; the companies Intel Corporation and IBM Corporation which were an essential part of these doctoral years — in particular, Flemming Andersen and Jesse Bingham (Intel, US), and Moshe Levinger (IBM, Israel) who made my research internships possible; both the Semiconductor Research Corporation and the Natural Sciences and Engineering Research Council of Canada who have partially sponsored my research; and last, but definitely not least, I acknowledge Rose-Marie Brown, OFS, Cesare Stefanon, OFS, Father Hugo, MSpS., Sister Dorothea, OCD and Father Casey, OCSO, whose spiritual support have been and will always be a blessing.

Dedication

אֶהְיֶה אֲשֶׁר אֶהְיֶה

Chapter 1

Introduction

Philosophy begins in wonder.

Plato

1.1 Motivation and Philosophy

IC technology continues to closely follow Moore's Law, while the ability to verify designs lags behind. The International Technology Roadmap for Semiconductors [20] (ITRS) predicts production of chips using 16nm technology by 2015. The verification gap, i.e., advancements in verification technology not keeping up with advancements in design technology [19, 48], seems to be also increasing at a fast pace. A recent study referenced in [21] shows a drop of more than ten percentage points from 2002 through 2009. By 2007, more than two-thirds of chips had to be respun due to bugs. The increasing verification gap is to blame. Unfortunately, because more bugs are slipping into the fabricated chip, post-silicon debug is the only way to catch them.

Post-silicon debug 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 traditional VLSI test focuses on random manufacturing *defects* on each fabricated chip. Post-silicon debug currently consumes more than half of the total verification schedule on typical large designs, and the problem is growing worse [1, 13].

The general problem of post-silicon debug is broad and multi-faceted (e.g., lack of observability, bug localization, bug rectification) spurring a diverse variety of research. In this thesis, I focus on one of the most fundamental tasks: getting an execution trace of on-chip signals for many cycles leading up to an observed bug or crash. Until such a trace is obtained,

further debugging is essentially impossible, as there is no way to know what happened on the chip.

However, the ever-increasing chip complexity compounded with new features that add non-determinism to the design (e.g., multiple clocks, asynchronous communication) makes computing accurate traces extremely difficult. Consider, for example, the use of embedded trace-arrays [5], a technology that provides an improved, though still limited, internal multi-cycle visibility. When embedded trace-arrays are used in the presence of non-determinism, the biggest challenge is figuring out when to start/stop the trace-array's operation[1] so that it saves only relevant information. Therefore, it is not always the case that partial traces extracted with these trace-arrays are accurate enough to aid debugging.

Thus, to address this issue, I present a novel post-silicon debug framework: BackSpace. In a nutshell, the BackSpace approaches automatically compute valid and accurate sequences of states that lead the chip to a crash state. My philosophy is that computing a trace backwards in time is much less error-prone, and thus, promotes better debugging for debug-engineers. By working from the crash state backwards, the BackSpace approaches eliminate the time-consuming, expensive task of guessing when to start/stop collecting data from the chip. Also, with BackSpace, it is possible to methodically and correctly construct a complete trace, that is, from initial to crash states. Therefore, BackSpace is not bound by the available physical resources dedicated to hardware debug. Consequently, BackSpace dramatically improves observability.

1.2 Post-Silicon Debug: State of the Art

1.2.1 Preliminaries

In this sub-section, I cover relevant definitions used in this thesis. First, I present different types of bugs. Then, I describe the difference between validation and debug. Finally, I present a typical post-silicon debug flow.

[1] Personal communication w/ John Bishop (IBM-US), February 2010.

Silicon bugs are generally classified as either electrical or functional bugs. I will present a couple of commonly used, intuitive definitions. Josephson [28], for example, defines functional bugs as logically incorrect circuits that do not operate properly under any condition; and, electrical bugs are logically correct circuits that fail under some operational condition. For Park and Mitra [42], functional bugs are design errors, and electrical bugs are caused by interactions between the design and physical effects. The intuition on both of these definitions is that an electrical bug is a malfunction of a physical device under some operating condition, whereas a functional bug is simply a logic error. Examples of electrical bugs are setup/hold time violations and excessive current-leakage. Examples of functional bugs are unintended logic, e.g., the use of an *and* gate instead of a *nand* gate, implementing a wrong protocol, or a *request* signal followed by a late *acknowledge* signal. Basic BackSpace targets only functional bugs, while TAB-BackSpace and nuTAB-BackSpace target both functional and electrical bugs.

Debugging and validation are two distinct, although complimentary, activities. Validation is the process of checking the system for compliance to its specification[2]. I will illustrate with an example. Consider a request-acknowledge communication protocol. Let's assume an electrical specification which states that the setup time of each latch is 5ns. To check conformance with an electrical specification, validation would entail applying different test-vectors under different operating conditions and checking whether the setup times are properly met. Debugging is the process of finding and removing bugs that violate a specification. Finding the bugs (AKA bug-localization) is rather difficult. It entails deep analysis of the design and extensive human insight. Once the bug is found, then the correcting action would be to change the implementation. In the case of the electrical bug, a fix would be reducing the combinational logic so that, for example, the *acknowledge* signal could be held active for the duration of the 5ns setup time. The focus of this thesis is on debugging, that is, I assume the validation process has observed a system malfunction due to some existing bug (or multiple bugs).

[2]I am referring to validation in the sense commonly used in hardware development.

Generally, in a post-silicon debug flow, once a design violation is observed, one is expected to i) run an experiment, ii) collect data, iii) analyze data, and possibly iv) run a new experiment. More precisely, running an experiment can be as simple as power-on-reset or as complicated as booting an operating system; data collection entails access to the internal state and I/O of a design; and data analysis can be done with logic analyzers[3], wave-form viewers and/or more advanced analysis tools. The underlying assumptions in this flow are that the state of the design is readily available, and re-running experiments always produce the same results.

These underlying assumptions break down, however, as technology advances. First, the number of available pins on a chip today is extremely small compared to its size (e.g. number of latches). Second, the majority of today's complex chips are "non-deterministic" (discussed below), making repeatable results less likely. Furthermore, current technologies, e.g., multiple clock-domains and asynchronous communications, make reproducibility one of the key issues in post-silicon debug.

A formal model of the actual physical world is key to better reason about these reproducibility issues. In particular, non-deterministic and probabilistic models are appropriate for such modeling.[4] The main advantage of the former is that non-determinism very easily models the lack of knowledge (i.e., under-specification) of the system/environment. The main disadvantage is that non-determinism is a weak model, that is, very little can be said about the system's behavior. On the other hand, when modeling the system as probabilistic, the assumption is that some knowledge of the system's transition probabilities exists. The main advantage of this model is that it is possible to reason more precisely about the system's behavior (e.g., how often the system traverses the same execution path). Therefore, throughout

[3]A hardware debugging tool that captures and displays data from the system being debugged.

[4]To be more precise, a non-deterministic finite automaton is a 5-tuple: with Q as its set of states, Σ the input alphabet, $\delta : Q \times \Sigma \times Q$ the transition relation, Q_0 the initial set of states and F the final set of states. A probabilistic finite automaton is an automaton with probabilities assigned to each transition in δ such that each transition is chosen according to some probability distribution.

the thesis, when reasoning about repeatability, I rely on a probabilistic modeling of the system. In other cases, when there is no need for such stronger modeling, I refer to the system as non-deterministic.

Not surprisingly, research on post-silicon debug has focused on these two main areas: improving observability and managing reproducibility. In the next section, I survey the state-of-the-art research on post-silicon debug while emphasizing how each research result tackles these two areas.

1.2.2 Related Work

In the last five years, post-silicon debug has been receiving increasing attention. The body of published work exceeds one hundred papers over these five years alone. I will select and describe the most relevant papers in this section.

To make this exposition easier, I group post-silicon debug research publication onto two classes: research focused on improving observability; and research focused on improving reproducibility. However, because research on post-silicon debug is not always focused on a single problem, my classification is approximate. My classification serves only the purpose of grouping together works that have similar objectives so that it becomes easier to contrast the existing techniques with my thesis contributions.

Improving Observability

1) Scan-Based Data Acquisition

   This is the most basic mechanism to peek inside the chip. Scan-chains [53] are present on almost all chips to allow efficient manufacturing test. A chip with scan-chains can be configured into test mode, in which most or all of the latches on the design are connected together into a small number of very long shift registers. At any point in time, the chip can be stopped, and the values of the latches can be shifted in or out. With hold-scan latches, it is even possible to scan out a snapshot of the state of the chip at one point in time, while

allowing the chip to continue to execute during the scan-out process, at the cost of substantial on-chip overhead [32].

When chaining all latches is not possible, selecting which latches to add to the scan-chain becomes crucial. In [3, 34, 40], the authors explore a wide range of techniques aimed at reducing the number of latches in the scan-chain while not hindering its testability. The key insight is that values of some latches may be inferable from others due to feedback cycles. In particular, Agrawal et al. [3], given a fault-model, apply test generation patterns to remove unnecessary latches from the scan-chain. Lee and Reddy [34] reduce the problem of defining the set of scannable latches to a graph transformation problem in which the graph representing the circuit is to be transformed into an acyclic graph. Park et al. [40] propose a technique for finding state-encodings that reduce the required number of scannable latches by either eliminating or shortening feedback cycles.

Alternatively, two research groups [1, 45] propose reconfigurable architectures that provide more flexibility for monitoring selectable signals. Abramovici et al. [1] propose a technique in which a set of selectable signals are chosen via scan-chain configuration bits. Quinton and Wilton [46] also use configurability via scan-chains to select desired signals, but using *concentrators*, an architecture that allows for efficient access (and observation) of chosen signals.

In Section 2.3, I assume full-scan to be available. Later, however, I relax this assumption and investigate how to best handle partial-scans.

2) Trace-Based Data Acquisition

A very limited history of some number of signals can be recorded on-chip at full speed, and this history can be read out (very slowly), e.g., via the scan-chains. These techniques ("on-chip logic analyzers", "trace buffers") typically consist of a flexible mechanism to access desired signals on-chip and some way to store the signals for later read out (e.g., RAM or specialized cells [5]). The main trade-off is that

considerable die area overhead must be used for each cycle's worth of history for each signal monitored, severely limiting how much history can be stored. Three different research groups have recently attempted to address this issue by identifying the set of signals that should be captured in trace buffers [31, 35, 43]. Ho and Nicolici [31] aim at storing signals that enable larger restorability of missing states. They explore two different metrics to select signals: topology, i.e, the cone-of-influence of each signal; and structure, the type of logic gate driving each signal. Liu and Xu [35] improve upon [31] by defining a more precise metric for signal selection using conditional probabilities. Prabhakar and Hsiao [43] use logic-implication to discover the minimum set of signals that can restore the maximum number of missing signals.

In all three works, they use standard techniques (e.g. logic simulators, SAT engines) to propagate, both forward and backward, signal values and restore missing signals. Given a trace-buffer, a signal restoration algorithm propagates logic 1 or 0 when there is enough information available. For example, assume a 2-input AND gate has one of its inputs set to 0. Therefore, the logic 0 is propagated to the AND's gate output. On the other hand, if one of its inputs is logic 1, and the other is unknown, then logic X is propagated.

Out of these last three works, [31, 35, 43], Prabhakar and Hsiao [43] achieve the highest signal restorability results. Nevertheless, I am not convinced about the scalability of these proposed methods. The experiments are based on the ISCAS'89 benchmarks, which are very small, contrived designs. For example, on most of the experiments in [43], they achieve more than 90% signal restorability by using only a few test-vectors. This suggests these circuits have very high fan-in/fan-out logic, which is very unlikely in complex, industrial-size designs

In this thesis, I am not focusing on identifying which signals should be in the trace-buffers. Instead, I assume the monitored signals have been already selected by either some automatic mechanism such as the

ones mentioned above (assuming they scale), or by some architectural insight into the design.

3) Embedding Monitor Circuits

Several groups have proposed leveraging the intellectual investment into formal *specifications* during post-silicon debug, by compiling the formal specifications into on-chip monitor/assertion circuits (e.g., [9, 26, 37]). In particular, three groups propose synthesizing monitors based on different assertions languages. Nacif et al. [37] propose synthesizing OVL[5] assertions; Hu et al. [26] propose GSTE[6] assertions; and Boule et al. [9], a subset of PSL[7] assertions. Despite each group's usage of different assertion languages, all three share a common and expected result. The interesting assertions are usually more complex assertions, which, once synthesized, yield a high area overhead. The increased use of trace-buffers may also play against the deployment of synthesized assertions into silicon. The more interesting assertions require more latches, which are more expensive than RAM (used by trace-buffers). However, it is more advantageous to use assertions, if the monitors are also used for implementing fault-tolerant circuits [37].

Ray and Hunt [47] propose a technique to increase observability based on partially implemented (embedded) assertions. The assumption is that some assertions may be too expensive to be fully implemented in hardware. Therefore, the authors propose to partition pre-silicon assertions into two parts, a post-silicon monitor (off-chip) and an integrity unit (on-chip). Consider a sequence of events as a sequence of signal-value changes over multiple clocks. Informally, the integrity unit guarantees that the observed silicon events represent part of a longer sequence of events on a higher-level model of the silicon (e.g. behavioral model, RTL netlist). Thus, the post-silicon monitor can check that if an event sequence, on silicon, does not violate the post-silicon

[5] *Open Verification Library*[39].
[6] *Generalized Symbolic Trajectory Evaluation* [51].
[7] *Property Specification Language* [44].

monitor, then it must also not violate the original assertion. Although this work is preliminary (most of the given theorems are based on a memory system example), it targets an important issue in post-silicon debug, which is linking silicon traces (with limited observability) with logic-simulator traces (full observability).

4) Data Propagation of Diverging Latches

Caty et al. [12] attempt to isolate electrical failures by comparing multi-cycle scan-dumps[8] from good and bad runs. Given two operating conditions where in one of them the chip passes and in the other one the chip fails, the authors compare the values of the latches from both runs to find out diverging latches between good and bad runs. Then, they propagate the values of those diverging latches backward to improve observability and root-cause bugs. The authors assume the system to be deterministic and also that full-scan is available.

5) Model Checking Post-Silicon Bugs

Ahlschlager and Wilkins [4] describe their experience directly using a model checker for post-silicon debug: they write a formal property to describe the observed buggy behavior and ask the model checker to generate a trace. Such an approach is ideal when the model checker can verify the entire chip or when the debug engineer correctly maps the observed chip-level buggy behavior onto a block-level formal specification. In practice, however, this is not often the case.

6) Rebuilding Architectural State of a Processor by Off-Chip Analysis

Park and Mitra [41] propose a processor-specific technique using summaries of in-flight instructions to rebuild (off-line) the architectural state of the processor. Thus, the authors are able to improve observability of what is happening inside the chip. For example, assume a processor that is capable of storing in-flight instructions' summaries is running some test program. Each fetched instruction receives a unique

[8] Refers to dumping the chip's internal state using scan-chains (described in "Scan-Based Data Acquisition", page 5.

ID. As the instruction flows through the pipeline, each pipeline-stage records the instructions' ID (plus some other auxiliary information). Once the run is completed, the stored information is dumped from the chip. Then, the off-line analysis (e.g. data and control flow analysis, load/store analysis) links those summaries and the test program's binary code, thus restoring the architectural state of the processor. Any observed instruction-flow inconsistency (caused by an electrical bug, e.g., a bit-flip) could be linked to a pipeline-stage, and, thus, to a micro-architecture block.

Improving Reproducibility

1) Specialized Bring-up Hardware

   Some companies (e.g., [52]) have specialized hardware and a great deal of high-end test equipment that can be used to record and replay *all* I/O signals between the chip and its environment. This infrastructure allows for deterministic repeatability of stimuli that triggered a bug on-chip. The main drawbacks are the high cost of the test equipment, the extremely limited ratio of observable I/O pins and pads versus the internal state of the chip, the inability to debug internal IP blocks, and the ability to debug the chip only in the specialized bring-up system. Note that sometimes, a bug will manifest itself only in some OEM system but not in the original bring-up system. Furthermore, the logic analyzer traces alone do not allow for reproducing the bug in a logic simulator. First, we need the internal state of the chip to start the simulation and not having the chip's state prevents any attempt to reproduce the bug in a logic simulator; and second, since logic simulators are many orders of magnitude slower than the real hardware, the logic simulator can replay only some limited amount of data captured by the logic analyzer.

2) "Carbon-Copy" Executions

   For a system implemented on FPGAs, the problem of system-level non-determinism can be eliminated by duplicating the entire design [33].

One copy of the design runs in the system as usual; the second copy has all of its inputs delayed in a FIFO. When the first copy hits the bug, it triggers trace recording on the second copy. In this way, the second copy is, in fact, reproducing the exact behavior of the first copy, but now the execution is being recorded (i.e., "carbon copied"). For a non-FPGA design, it is obviously impractical to duplicate the design on-chip, but if two identical dies are available, both of which are fully deterministic in an identical manner, one could imagine building a specialized bring-up board that implements this solution[9]. However, as systems start adding more sources of non-determinism (e.g., multiple clocks, asynchronous communication, soft-errors) this technique becomes impractical. I illustrate this case with a processor. In one copy, due to a failing checksum caused by a non-deterministic fault, a processor may require an instruction re-issue, resulting in a pipeline stall while in the other copy, the instruction flows normally in its pipeline. Once this happens, the execution-flow of each system diverges from the other. Therefore, information recorded by the second copy may not be at all related to the first copy.

3) Periodic Sampling

In [10, 29], the chip can be stopped at regular intervals to scan out a snapshot of the internal state and then allowed to continue execution. With hold-scan latches, the chip need not even be stopped. When the crash occurs, the most recent snapshot and the logic analyzer traces of the I/O can be used to recreate the bug in simulation. An obvious problem is that stopping the chip disturbs system-level timing interactions, potentially changing the execution and hiding the bug. Furthermore, because the scan-out process is so slow, the interval between snapshots must be long, meaning that the most recent snapshot might be millions of cycles in the past, rendering it useless for the debugging. The main issue is that logic simulators are orders

[9]This idea was suggested to us by Igor Markov, June 30, 2008.

of magnitude slower than the real hardware. Therefore, given two consecutive, but far apart snapshots, it will be impossible to simulate a design in a timely manner, and reproduce the entire run from one snapshot to another.

4) One-shot Data Acquisition

In [41], the chip uses trace information to off-line reconstruct the chip run. The technique targets electrical bugs and is very processor-specific. The authors' insight is that there are some early signs the chip is about to crash and they tap into these early signs to start data acquisition. This idea, although very ingenuous, assumes that every early sign can be identified, which is clearly not possible for non-processor chips. This approach, if focused towards functional bugs, would be equivalent to having assertions that would trigger a data acquisition mechanism. Obviously, it is impossible to predict all possible bugs and add corresponding hardware monitors to trigger data acquisition. Furthermore, as explained on page 8 (Embedding Monitors Circuits), the more interesting assertions have high area-overhead.

5) Removing Non-Determinism with Scan-Dump Analysis

Dahlgren et al. [17] propose a technique to remove non-deterministic elements of a functional test environment via scan-dump analysis so that bugs can be more precisely root-caused. Although the idea is a very simple one, this is the first publication which describes a method for removing non-determinism from functional tests. The authors claim that even on a deterministic platform, functional tests would be vulnerable to non-determinism since parts of the design are uninitialized. Thus, applying the same test and at the same operating condition may yield different traces, making it harder to isolate the location of a bug. Consider the scenario in which one test under two operating conditions lead the chip to two different results: pass and fail. Assuming bugs to be consistently repeatable and being always observable in the exact same manner, the authors describe their method as follows: (1)

run the chip N times using a passing experiment and compare each run against each other, annotating each latch with respect to the cycle number and whether the latch diverges from other runs; (2) remove the latches that diverge in the previous step; (3) run the chip N times using a failing experiment and compare each run against each other, annotating each latch with respect to the cycle number and whether the latch diverges from other runs; (4) remove the diverging latches from the previous step. Since we have removed diverging latches from the passing and failing runs, what is left to do is to compare a passing run against a failing one. The latches that differ point to the root-cause of a bug. One of the problems with this method is that some tests may be too short to capture the latches that are vulnerable to non-determinism. Another problem is the assumption that bugs would not be caused by "non-deterministic" latches.

Like many classification schemes, mine has exceptions, most notably, works that fall under the broad term "rectification" (e.g, [50], [55], [13]). These works assume data acquisition has been completed and the complete state of the chip is available (in practice, they only require full-state information for specific time-frames). They focus on localizing bugs and proposing "fixes" to repair the gate-level netlist. Although, these works are important, they are complementary to my work.

1.2.3 Summary

I have presented a survey of the research focused on improving observability and reproducibility. Despite the encouraging number of new techniques in the past few years, many of these techniques work only under stringent simplifying assumptions. For example, Dahlgren et al. [17] assumes the system to be deterministic, Ahlshlager and Wilkins [4] assumes the whole chip can be model checked, and Prabhakar and Hsiao [43] rely on the fact that a "good" trace already exists. On the other hand, as today's systems are increasing the sources of non-determinism — many clock-regions; internal arbitration circuits; and asynchronous communication — these techniques

become impractical without other *ad-hoc* tricks. Instead, we need a general, but methodical, framework in which we can better use current/new debug techniques in the face of complex post-silicon bugs.

1.3 Scope of the Thesis

This research is focused on improving the state-of-the-art of post-silicon debug. To this effect, I take a fresh look at the problem of post-silicon debug of functional bugs and introduce new, systematic techniques to better handle it. This thesis targets both observability and reproducibility issues while uniquely bridging hardware debug techniques with formal analysis.

Ideally, a post-silicon debug method should account for both electrical and functional bugs. However, targeting either one of them alone will improve the state of the art of post-silicon debug. On the one hand, electrical bugs are an important issue and some even claim it is the most important problem [41], but there is also clear evidence that functional bugs are, at least, as important as electrical ones. Foster [21] refers to a study by *Far West Research* which shows that, in the period 2004-2007, while there was an increase from 75% to 77% of functional bugs requiring chip re-spins, the impact of other bugs decreased significantly. Clearly, both are important. Although, in this thesis, I primarily focus on functional bugs, I show later that the techniques presented here can be extended to electrical bugs.

We need a general, but methodical, framework in which we can better use current/new debug techniques to reason about complex post-silicon bugs. As I have shown in Section 1.2.2, most techniques are still being used *ad-hoc*, heavily dependent on guesses and simplifying assumptions (e.g. a system being deterministic and/or bugs being reproducible on a deterministic platform). One of the problems in post-silicon debug is that these techniques have not been able to keep up with the ever-increasingly complex problems we face. In particular, 10-to-15 years ago, chips did not have many sources of non-determinism; thus it used to be easy to decide when to stop the chip and start data-acquisition (e.g. chip single-stepping[10]).

[10]Single-stepping refers to repeated toggling of the clock signal and scan-dumps.

However, under the presence of non-determinism, guessing (predicting) when to start and stop the chip (or data-acquisition) becomes incredibly hard, time-consuming, and error-prone. Instead, reconstructing the past may be a better approach to post-silicon debug. Thus, I introduce my formal framework, which I call BackSpace because its name embeds the notion of going towards the past.

The BackSpace framework includes the original Basic BackSpace, TAB-BackSpace and nuTAB-BackSpace. I developed Basic BackSpace as a more theoretical approach. Subsequently, I introduced the other techniques to address the practical limitations I encountered with Basic BackSpace.

Intuitively, BackSpace's objective is to iteratively reconstruct the history of a chip run. To that end, I need a mechanism to stop the chip at certain points of the run (e.g., breakpoint circuitry) and a storage capability to save some history information about the run. BackSpace starts by letting the chip run until it crashes; then, dumps the state and history information; computes the crash's predecessor state; loads it into the breakpoint circuitry and runs the chip again. If the chip does not stop, then we know the chip-run took a different path due to non-determinism. In that case, BackSpace runs the chip again. I assume that bugs are repeatable with probability greater than some $\epsilon > 0$. Therefore, the chip will eventually encounter the breakpoint; at which point BackSpace dumps its state and history. This process iterates as many times as necessary until BackSpace computes a trace that is long enough for a designer to debug.

The Basic BackSpace technique computes predecessor states by formal analysis (pre-image computation), working perfectly in theory. However, this perfect solution comes with impractical overhead: correctness relies on computing breakpoints, signatures, and pre-images over the entire concrete state of the chip. The hardware overhead is too high to be practical.

Most complex chips, however, already include some on-chip debug hardware. In TAB-BackSpace, I flipped the problem around: instead of adding excessive on-chip hardware for a perfect debug solution, I leveraged the Basic BackSpace approach to get much more out of the *already existing* in-silicon debug logic (i.e., trace buffers). Thus, there is no additional hardware cost.

In particular, TAB-BackSpace does not use pre-image computations to compute predecessor states. Instead, upon a breakpoint match, TAB-BackSpace picks one entry of the trace buffer as the new "crash state"and run the chip again. TAB-BackSpace imposes the condition that the next, successive trace buffer must successfully overlap cycle-by-cycle with the previous trace. If they do successfully overlap, then TAB-BackSpace considers the (abstract) states in the trace buffers as valid predecessor states of the crash state and the trace computation iterates. Thus, TAB-BackSpace achieves the effect of extending the trace buffer arbitrarily far back in time (assuming no spurious traces).

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

In practice, the real weakness of TAB-BackSpace is the need to repeatedly trigger the bug via the same execution. Non-determinism in the hardware and in the bring-up environment make such exact repetition very unlikely. The result is that the newly computed trace-dump does not always completely agree with the previous trace over the entire overlap region, in which case TAB-BackSpace fails to make progress. Thus, TAB-BackSpace benefits from an environment which has better controllability (by extensively reducing non-determinism). However, creating such an environment while still triggering a bug might not be practical.

What I have observed, however, is that although an *exact* match rarely occurs, what typically happens in practice is that the same bug is triggered by an intuitively "equivalent" trace, that is not cycle-by-cycle identical. Thus, I present nuTAB-BackSpace. This technique is based on *rewriting*. I prove that, under reasonable assumptions, nuTAB-BackSpace computes concretizable abstract traces — i.e., traces corresponding to possible, real

chip executions.

1.4 Contributions of the Thesis

All the techniques presented here have a common goal to improve the state-of-the-art of post-silicon debug. More specifically, I addressed one fundamental issue, extracting an accurate trace from a buggy chip. In particular, these are the main contributions of this thesis:

- I have developed the theory of BackSpace. I have investigated issues such as: what is the proper image-computation framework (e.g., BDDs, SAT); the use of signatures, more specifically, I investigate the tradeoff between hardware cost and precision with different signatures (e.g., CRC, universal-hashing); and finally, the backspaceability of designs. I have demonstrated by both logic simulation and hardware prototyping that BackSpace works. However, when applying BackSpace in practice, I have found limitations (e.g., excessive chip area-overhead, large-constant time-cost due to the lack of reproducibility). From these experimental results I have created new hypotheses.

- To address the problem of excessive area-overhead, I have developed TAB-BackSpace. By leveraging only existing hardware debug-logic, I have shown that TAB-BackSpace has *zero-additional* area-overhead. Moreover, I have demonstrated its effectiveness on a real silicon, the IBM POWER7 processor.

- To address the reproducibility cost due to non-determinism, I have put forward a hypothesis that, from the debug-engineer's point-of-view, different traces might share some notion of equivalence. To that end, I have leveraged the theory of *string-rewriting* to normalize the non-determinism found in different traces. I call this technique nuTAB-BackSpace. I have demonstrated that, indeed, given some reasonable assumptions, different traces can be normalized into equivalence classes. More importantly, nuTAB-BackSpace can compute an

accurate trace even on an environment in which non-determinism is pervasive (e.g. many clock domains, asynchronous communication). Experiments on simulation and a complex SoC prototype show the success of nuTAB-BackSpace.

In summary, this thesis addresses the need for a methodical framework to post-Si debug. In particular, this thesis targets the problems of lack of observability and reproducibility by uniquely combining formal analysis and existing hardware debug techniques.

Chapter 2

Basic BackSpace

No problem can be solved from the same level of consciousness that created it.

Albert Einstein

2.1 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.

I 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, one 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, one could run the bring-up tests on the simulator, or via single-stepping[11]. 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 runtime 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 the bug *ab initio* in simulation is clearly not

[11] As defined in Section 1.3.

feasible. Similarly, trying to monitor externally the full execution trace of the chip running full-speed is electrically impossible.

In this chapter, I consider a few simplifying assumptions that I made as I was starting to develop my framework:

- 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 I am focusing on functional bugs, I will assume that manufacturing testing has eliminated manufacturing defects and that the chips has no electrical bugs. Therefore, I 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 (with some reasonably large probability).

In Chapter 3, I relax some of these assumptions. In particular, not all chips are full-scan capable, but have other mechanisms to capture some history of a chip run (e.g. trace buffers). I make use of this fact in later chapters.

My 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 set of predecessor-candidates (i.e., the pre-image) of the crash state. The signature must provide enough information so that the number of predecessor-candidates is reasonably small.

4. For each predecessor-candidate s, let s be the new breakpoint; re-run the chip; if the chip reaches the breakpoint, then s is a valid predecessor.

The iteration stops when it has 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 – going back one cycle. The approach exploits the capabilities of different analyses: formal analysis is very slow with limited capacity, but can go forward or backwards equally well; simulation is too slow to run in a real system with actual software, but the visibility of a simulation trace is user-friendly and well-accepted for design understanding and debugging; the actual silicon runs full-speed, rapidly hitting bugs that may have escaped pre-silicon validation, but offers very poor visibility and no way to go backwards to determine how the chip arrived in some state.

2.2 BackSpace Theory

In this section, I formalize BackSpace. I start by describing how I model the system and by introducing some definitions. Then, I formalize the (intuitive) algorithm I presented in Section 2.1.

Let $M = (Q, \Sigma, \Gamma, Q_0, \delta, \omega)$ be a finite state machine modeling a digital design D, where:

- $Q = 2^S$ is the set of states, where S is the set of latches in D;
- Σ is the input alphabet;
- Γ is the output alphabet;
- $Q_0 \subseteq Q$ is the set of initial states;
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation;
- $\omega \in Q \times \Sigma \mapsto \Gamma$ is the output function.

Notice that M is modeled as a non-deterministic finite state machine, so the formalism can handle randomness in the bring-up tests as well as transient errors, race conditions, etc.

Given a state machine M, I 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. Let $R = 2^T$ be the set of signatures. 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 $\delta \subseteq Q \times \Sigma \times Q$ as well as a $\delta' \subseteq Q \times R \times \Sigma \times R$. 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.

Definition 1 (k-Backspaceable State)**.** *A state (s', t') of augmented state machine M' is k-backspaceable if the size of its pre-image projected onto Q is bounded by $k \geq 1$, i.e.,*

$$Pre = \{(s,t) | s \in Q \wedge t \in R \wedge \exists i \in \Sigma \,.\, (((s,i,s') \in \delta) \wedge ((s,t,i,t') \in \delta'))\} \quad (2.1)$$

$$\Pi_Q(Pre) = \{s \mid (s,t) \in Pre\} \quad (2.2)$$

$$\| \Pi_Q(Pre) \| \leq k \quad (2.3)$$

Equations (2.1) and (2.2) are typical definitions of pre-image computation and projection. Equation (2.3) bounds the size of the projected pre-image by k.

Definition 2 (k-Backspaceable Machine)**.** *An augmented state machine M' is k-backspaceable iff all reachable states are k-backspaceable.*

In Algorithm 1 (pp. 24), I present the procedure that, starting from a given crash state, iteratively computes an arbitrarily long sequence of predecessor states by going backwards in time. The procedure has 2 nested loops. The outer loop, lines $10-28$, controls the two termination conditions for the algorithm: either the procedure has computed a trace of length j or the procedure has reached the set of initial states. The outer loop is also responsible for the pre-image and projection computations and continually prepending the trace with newly found predecessor states. The inner loop,

lines $14-21$, is responsible for controlling the hardware while trying out different predecessor-candidate states, s_{cand}. In each loop iteration, the procedure selects, using a round-robin scheme, a predecessor-candidate state (line 16), loads it into the breakpoint-circuit (line 18) and starts a run (line 20). The objective of the inner loop is to validate a predecessor-candidate state. If *ResetAndRun* returns *TRUE* then the breakpoint circuitry matched (hence, validated) s_{cand}, in which case, s_{cand} is prepended to the trace sequence, line 27. Otherwise, the run violates the $time_{bound}$ (line 20) parameter because either the current run took a "wrong path" (caused by non-determinism) or the current s_{cand} is not a valid predecessor. In this case, the inner loop is repeated, the procedure selects another predecessor-candidate and starts a new chip run.

Theorem 1 (Correctness of Trace Computation of Algorithm 1)**.** *Given a sufficiently large $time_{bound}$, the sequence of states returned by Algorithm 1 is the suffix of a valid execution of M.*

Proof: Let l be the length of the trace returned by Algorithm 1. For all states s_i in *trace* : $s_0, \cdots, s_{l-1}$, I must prove that two properties hold: (1) $\forall i \ . \ 1 \leq i \leq l-1$, the state s_{i-1} is a predecessor of s_i in M; (2) $\forall i \ . \ 0 \leq i \leq l-1$, the state s_i is reachable in M. These two properties combine to yield the desired result. To prove (1), notice that s_{i-1} and s_i in *trace* are related by the *Preimage* function call in line 12. Therefore, the set *pre* resulting from $Preimage(s_i, t_i)$ contained at least one state of the form (s_{i-1}, x) for some x. In other words, there exists x such that (s_{i-1}, x) is a predecessor of (s_i, t_i) in M'. By the definition of the augmented state machine, s_i cannot depend on x, therefore s_{i-1} must be a predecessor of s_i in M. That establishes the first property. For the second property, notice that there are only two possibilities for a state to be prepended to *trace*. In line 9, the state s_{l-1}, which is given as reachable, is prepended to *trace* ; the remaining states, $s_0, \cdots, s_{l-2}$ are prepended to *trace* in line 27. But, to execute line 27, *ResetAndRun* must have matched those states, which makes them reachable. Therefore, all states in *trace* are reachable. Thus, I have established the second property and conclude the proof. ∎

Algorithm 1 Crash State History Computation

input Q_0 : set of initial states,
        (s, t) : reachable state of a k-backspaceable augmented state machine M',
        $j \in \mathbb{N}^+$: user-specified bound on the trace length,
        $time_{bound}$: user-specified time bound for any run of M';
output $trace$: sequence of states of M;
$i := j - 1$;
$s_i := s$;
$t_i := t$;
$trace := (s_i)$; // i.e., initialize trace with target state (s, t) projected onto Q
while $i > 0$ **and** $s_i \notin Q_0$ **do**
    $matched := FALSE$;
    $pre := Preimage((s_i, t_i))$; // as defined in Eq. 2.1
    $proj := Projection_Q(pre)$; // as defined in Eq. 2.2
    repeat
        // Pick a candidate-state in *proj* following a *round-robin* selection scheme
        $s_{cand} := PickState(proj)$;
        // Program the hardware-breakpoint circuitry with s_{cand}
        $LoadHardwareBreakpoint(s_{cand})$;
        // (Re-)run M' at full-speed with timeout $time_{bound}$
        $matched := ResetAndRun(time_{bound})$;
    until $matched = TRUE$
    // This line is only reached if the hardware breakpoints at s_{cand}
    $i := i - 1$;
    $s_i := s_{cand}$;
    // Scan-out hardware signature t_i
    $t_i := ScanOut()$;
    $Prepend(s_i, trace)$;
end while
return $trace$;

The parameter $time_{bound}$ is a user-specified timeout value. I assume users can roughly measure the time it takes for the chip to crash. Thus, users can use this approximate measurement as the $time_{bound}$ value (plus some safety margin). Intuitively, $time_{bound}$ is an approximation of the length of a path leading to the bug. Therefore, even in the presence of randomness, there is a non-zero probability the chip will take such path again.

Given a $time_{bound}$ value, what happens with Algorithm 1 if the $time_{bound}$ value is too short (shorter than the shortest path to the crash state)? If the $time_{bound}$ value is too short, the algorithm will never leave the *repeat*-loop (lines 14-21) and Algorithm 1 will never terminate.

Even with a large enough $time_{bound}$ value, if we consider non-determinism (as opposed to randomness), the algorithm is not guaranteed to terminate. For example, assume that only *1* out of the k states in *proj*, I call it s_{pred}, is a valid predecessor of (s_i, t_i). It is possible that, every time *PickState* selects s_{pred}, the chip will take a different path not leading to this state. Thus, Algorithm 1 will never leave the *repeat*-loop and this algorithm will not terminate. Can we guarantee termination otherwise (i.e., under an assumption of randomness and with a sufficiently large $time_{bound}$ value)?

Theorem 2 (Probabilistic Termination of Algorithm 1)**.** *If $time_{bound}$ is sufficiently large to allow the chip to crash, and the executions σ' of M' are chosen randomly such that all valid transitions have probability greater than some $\epsilon > 0$, then termination occurs with probability 1.*

Proof: Algorithm 1 has two nested loops: the *repeat*-loop (lines 14-21) and the *while*-loop (lines 10-28). To prove termination, I have to show that this algorithm exits both loops. I will start the proof with the *repeat*-loop. Let s_i be a state in *trace* and its corresponding state in M' be (s_i, t_i). Since $s_i \in trace$, the state (s_i, t_i) must be a reachable state of M'. By the definition of k-backspaceable augmented state machine M', the set of predecessor-candidate states of (s_i, t_i) has at most k states when projected onto Q (computed by $Preimage$ and $Projection_Q$, lines 12 and 13, respectively). I must simultaneously satisfy two conditions to exit the *repeat*-loop: (1) choose a valid predecessor-candidate; and (2) choose a valid execution

path leading the chip to the chosen state in (1). In each iteration of the *repeat*-loop, *PickSate* chooses a state following a round-robin scheme. Since there is at least one valid predecessor-candidate state, say s_{pred}, this state will be chosen every k *repeat*-loop iterations. Because all valid transitions have probability greater than some $\epsilon > 0$ of being taken, every valid finite execution path also has a non-zero probability of being chosen. Therefore, since s_{pred} is a valid state, there exists a valid and finite path, σ', from an initial state to s_{pred}. Let l be the length of σ'. A lower-bound on the probability of taking this path is ϵ^l, which is greater than zero, and so, eventually, *ResetAndRun* will traverse such a path and return *TRUE* (line 20). Thus, the computation leaves the *repeat*-loop. To show that Algorithm 1 leaves the *while*-loop, notice that there are two conditions for which this loop terminates: $i \not> 0$ or $s_i \in Q_0$. Let's ignore the $s_i \in Q_0$ condition. Note that i is decremented by 1 in each iteration and so i must eventually be less than or equal to zero. Therefore, Algorithm 1 exits both loops and, thus, terminates.

∎

I draw the following fact from this proof to support a later theorem:

Corollary 1. *Every* (s_i, t_i) *is a reachable state in M'.*

Proof: As stated in proof of Theorem 2, Algorithm 1 starts with a reachable state. After that, the algorithm references state (s_i, t_i) in line 12 only after *ResetAndRun* has matched (reached) a candidate state s_{cand} and scanned-out signature t_i, lines 20 and 26, respectively.

∎

Note that it is not obvious what happens if the user-specified bound, j, is infinite. Besides the fact that Algorithm 1 may not terminate under the assumption of non-determinism, there is also the possibility that the algorithm will not make progress towards the initial states. For example, consider two states, (s_a, x) and (s_b, y), of k-backspaceable augmented machine *M'*. Furthermore, assume that setting the programmable breakpoint hardware to s_a will result in an execution σ_a that reaches (s_a, x) from state (s_b, y). In theory, it is conceivable that by reprogramming the breakpoint hardware to

target state s_b and re-running the chip, non-determinism might cause the chip to follow a different execution σ_b that reaches (s_b, y) from the state (s_a, x). In this case, Algorithm 1 will still compute a valid suffix execution of M, as indicated by the Theorem 1, but this execution will not 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 I can prove (analogously to Theorem 2) that Algorithm 1 makes progress towards the initial states with probability 1:

Theorem 3 (Probabilistic Progress of Algorithm 1)**.** *Given a sufficiently large $time_{bound}$, if we terminate Algorithm 1 only 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 probability greater than some $\epsilon > 0$, then Algorithm 1 reaches the set of initial states with probability 1.*

Proof: To prove this theorem, it suffices to show that Algorithm 1 performs a random walk on the reachable state space, R, of M', arriving at the initial set of states Q'_0. By Corollary 1, I know that every state (s_i, t_i) considered by Algorithm 1 is reachable. Therefore, the random walk always remains within the reachable set of states. Two cases are possible: (1) $(s_i, t_i) = (u, v)$ for some $(u, v) \in Q'_0$ and (2) $(s_i, t_i) \neq (u, v)$ for all $(u, v) \in Q'_0$. In (1), the algorithm is already in the initial state and, thus, the theorem holds. In (2), there is at least one execution path from some $(u, v) \in Q'_0$ to (s_i, t_i). Let's call this path σ' and let l be the length of σ'. Since each valid transition has probability greater than some $\epsilon > 0$, σ' has probability greater than ϵ^l of being taken. Note that, in Algorithm 1, if *ResetAndRun* is repeatedly called $k \times l$ times and that each time the algorithm follows the path σ' (an event that occurs with lower-bound probability ϵ^{kl^2}), then the algorithm reaches the initial set of states after l steps. Otherwise, because there may be multiple (valid) execution paths to (s_i, t_i), *ResetAndRun* may "choose" a valid path other than σ'; thus, the algorithm might make a step on this other path and the random walk continues thereon (the algorithm iterates). Therefore, the random walk will eventually end up at an initial state.

∎

Informally, the proof of Theorem 3 says that the walk that Algorithm 1 is making in the reachable state of M' is a Markov process. And, as long as each valid transition has a probability greater than some $\epsilon > 0$ of being taken, this algorithm will eventually hit the set of initial states which are the only absorbing states in this Markov process (arriving at any initial state causes the algorithm to terminate).

In my preliminary experiments (Section 2.3) and also in the hardware prototype I built (Section 2.3.5), I did not find these issues of non-determinism, randomness and termination to be a problem. The main difficulty with randomness is 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.

Precise measurement of the runtime cost of Algorithm 1 is complicated because some functions mix software and hardware operations (e.g., *ResetAndRun*). However, I can measure the expected number of function calls, and I will consider this measurement the expected runtime cost of Algorithm 1. I assume that non-determinism simply amounts to randomness. But even then, computing the expected runtime cost from Algorithm 1 is not straightforward. To simplify this analysis, I make some assumptions:

(i) I replace lines 15 through 20 with a function I call *Round()*, which returns a boolean value. In other words, the *repeat*-loop now has only one statement: $matched := Round()$ (Algorithm 2, pp. 29, depicts the *Round()* function);

(ii) the *Round()* function tries out all k-states in *proj* regardless of a successful match (e.g., in Algorithm 2, if *breakpoint* becomes *TRUE* in the first *for*-loop iteration, then it remains *TRUE* throughout the remaining $k - 1$ iterations);

(iii) I assume some repeatability of runs, i.e., a crash state is reproducible in 1 out r calls to *Round()*.

Under the aforementioned assumptions, I can compute the cost of each loop

separately. Given assumption (ii), the runtime cost of *Round()* is $\mathcal{O}(k)$; due to assumption (iii), the expected cost of the *repeat*-loop is

$$E[\text{time to match } s_{cand}] = \mathcal{O}(rk). \tag{2.4}$$

Finally, the expected runtime cost of Algorithm 1 depends on the length, j, of the trace being computed. Since each trace increment has expected runtime cost $\mathcal{O}(rk)$, the expected runtime of Algorithm 1 is

$$E[\text{time to compute trace of length j}] = \mathcal{O}(jrk). \tag{2.5}$$

Algorithm 2 Round()

```
1: breakpoint := FALSE;
2: for l in 0 .. k-1 do
3:     // Select candidate-state l in proj
4:     s_cand := SelectState(l, proj);
5:     // Program the hardware-breakpoint circuitry with s_cand
6:     LoadHardwareBreakpoint(s_cand);
7:     // (Re-)run M' at full-speed with timeout time_bound
8:     breakpoint := breakpoint | ResetAndRun(time_bound);
9: end for
10: return breakpoint;
```

In Algorithm 1, the best computational cost happens when $k = 1$ because I need to try out only one predecessor-candidate states. But, is it always possible to augment any state machine to make it *1*-backspaceable? The answer is yes. I can simply make $\| T \| = \| S \|$ and set up δ' to copy the values in the latches of S to the latches of T. In other words, I can always backspace to a unique predecessor state because I have stored that state.

Is it possible to do better, to make any state machine *1*-backspaceable using fewer than $\|S\|$ additional latches? Unfortunately, in the worst case, the answer is no. For a simple example, consider the state machine in Figure 2.1. This example is a simple n-bit counter, with a single input. If the input is low, the counter transitions to the 0 state; otherwise, it counts up. Almost all states have only a single predecessor, making them *1*-backspaceable with no additional signature. However, the 0 state has every

state as a predecessor. To make the machine *1*-backspaceable, I must add the full n additional state bits, just to handle one particularly bad state. I call such states "convergence states" because many incoming transitions converge on them.

Figure 2.1 shows that in the worst case, I can do no better than by storing a copy of all the state bits. However, it also suggests that I might be able to do much better for *most* states. Is it good enough if I make most states *1*-backspaceable; or even k-backspaceable?

Definition 3 (k-Backspace Coverage)**.** *Given state machine M augmented into M', the k-backspace coverage of M' for M is the fraction of the reachable states of M' that are k-backspaceable.*

Can I get good backspace coverage with much fewer than $\| S \|$ bits in the signature, or more to the point, can I backspace a long enough trace to be useful before hitting a convergence state? The convergence states are likely to be states that are easy to get to and easy to understand (like reset or idle states); backspacing to a convergence state may be sufficient for debugging purposes. In fact, as I show in the next sections, I am always able to backspace a substantial number of states, each of which with the pre-image-set size bounded by a reasonably small k.

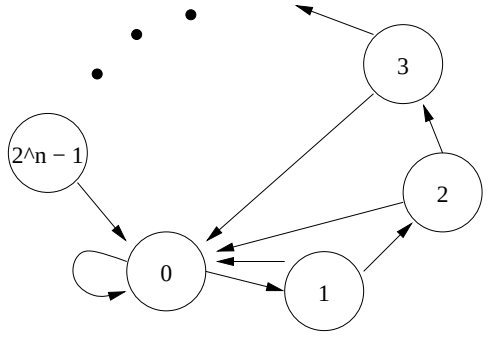

Figure 2.1: State Machine Requiring $\| S \|$ Extra State Bits to Be Backspaceable

2.3 Experimental Results

2.3.1 Experimental Setup

My goal in this section was to explore and evaluate the basic components of my framework, namely, signature functions and backspacing capability; thus, I chose to focus on two easy-to-learn design examples so that I could generate research hypotheses. The designs also had to be small enough so that repeated experiments were feasible, and so that the supporting algorithms and tools that are not germane to this research did not need to be highly optimized. On the other hand, the designs had to be realistic, to capture characteristics of real designs.

In particular, I picked two processors: a 68HC05 and an 8051. These are both open-source designs that are rebuilds from datasheets of the respective classic 8-bit microcontrollers from Motorola and Intel [38]. The 68HC05 is smaller, with 109 latches. The 8051 implementation has 702 latches. In both cases, I developed a simulation testbench based on the testbenches supplied with the designs: the 68HC05 ran real LED and LCD controller applications, and the 8051 ran some small software routines.

For these experiments, I treated the design running on a commercial logic simulator as if it were the actual chip running on silicon. I simulated the designs in a logic simulator for an arbitrary number of cycles and randomly selected 10 states each to serve as "crashed" states for our analysis. In addition, my testbench also recorded the immediate predecessor state before the crash state (which wouldn't be possible in silicon); this predecessor state is the correct answer that my analysis is trying to recover. Thus, my experiments generated 10 pairs of states per design to serve as testcases.

2.3.2 Signature Functions

As a first step, I needed to find some plausible signature functions. I concentrated on the 68HC05 and tried a variety of approaches. Fig. 2.2 summarizes the results of my experiments.

My first idea was to try a quick experimental upper bound on the size

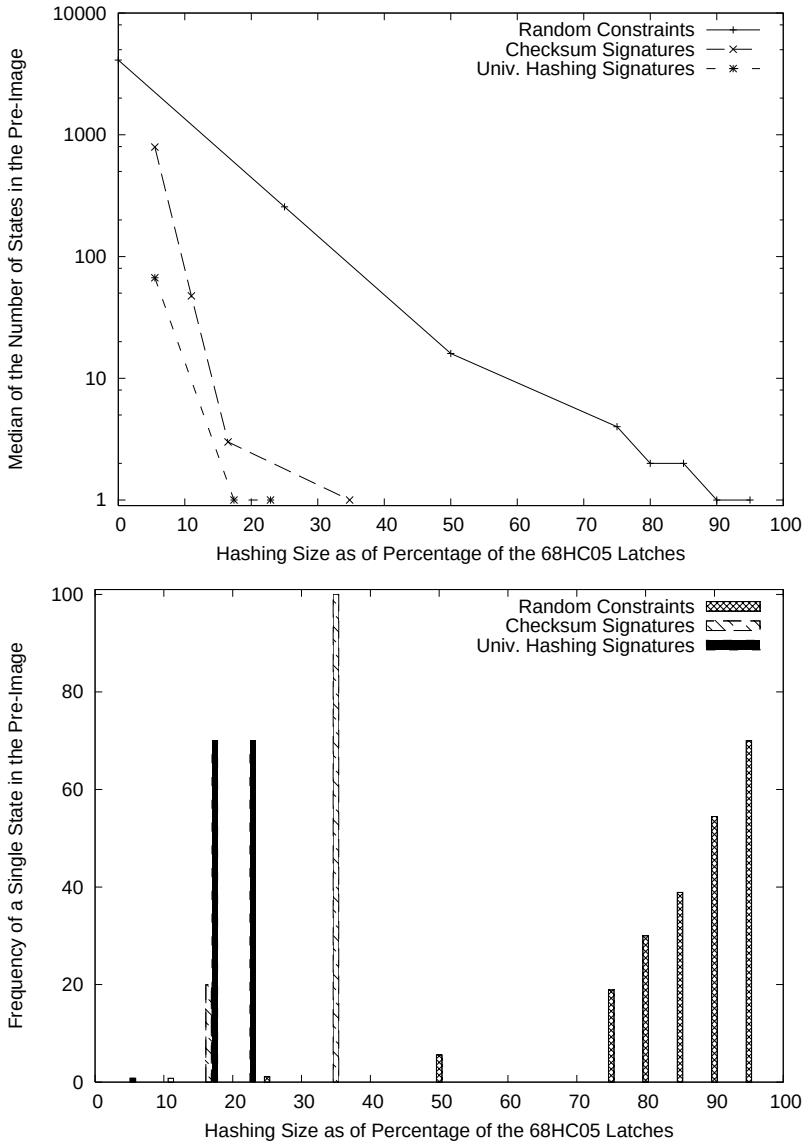

Figure 2.2: Results for Compressed Signatures Based on Architectural Insight

of the signature. I created the signature as a randomly selected subset of all state bits. Unfortunately, this approach fared poorly: assuming $k = 1$ for k-backspaceable machine, we observe that 90% of the state bits were needed in the signature before the median size of the pre-image was 1, and even for a small k, more than 50% of the state bits would be required in the signature.

In real life, the designers understand their design, and architectural insight might allow selecting a particularly good subset of the state bits to use as a signature. Based on a careful study of the 68HC05, I identified 38 latches (35% of the design) to use as the signature. This approach was very successful, yielding unique pre-image states for all 10 test cases.

Spurred by that success, I tried some simple checksums on those 38 bits, reducing the number of bits used to 6, then 12, and then 19. These results were not very successful at getting unique pre-image states, but the plot suggested that better compression would be promising.

Accordingly, I tried a perfect hash function — universal hashing [11] (essentially the same as X-Compact [36], which is easier to implement on-chip) to compress the 38 bits to 6, 19, and 25 bits. These results demonstrated the promise of universal hashing.

In all of these experiments, computations were fast, and the SAT solver had no problem computing pre-image states.

2.3.3 BackSpacing

With some promising ideas for signature functions, I proceeded to the real test: can I backspace for hundreds of cycles from the random crash states? I created an automatic framework to experiment and explore the BackSpace paradigm (Fig. 2.3). The components of the framework are the BackSpace Manager, a commercial logic simulator, and a SAT solver. The input to the framework is a synthesized design (gate-level netlist). The logic simulator plays the role of the silicon: I use it to run my testbench, exactly as the real silicon would run bring-up tests. The SAT solver is the engine to compute the required pre-image states. The core of the framework is the BackSpace

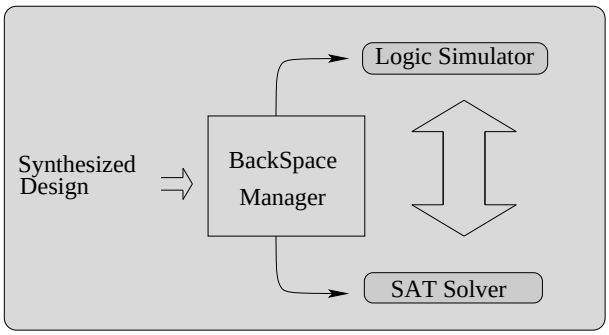

Figure 2.3: BackSpace Framework

Manager.

The BackSpace Manager coordinates the logic simulation and the SAT solving tasks by dispatching each task and processing their intermediate results (shown as the double-headed arrow in Fig. 2.3). For logic simulation, the BackSpace Manager automatically generates a testbench instance based on the synthesized design, dispatches the logic simulation, awaits its termination, and captures the crash state and signature. For SAT solving, given the crash state and the signature, the BackSpace Manager generates a SAT problem instance. When the SAT solver finds a solution, it means there is one (more) state in the pre-image of the crash state. 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. 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 reachable. The BackSpace Manager dispatches logic simulation, setting a candidate state as a simulation breakpoint. If simulation 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. If simulation does not reach the breakpoint, it means we need to try another candidate. For logic simulation, I used Synopsys VCS (version 7.2), and for our SAT solving, I used Minisat (version 2.0). Due to VCS licensing issues and GCC compatibility

problems, I had to run these tools on different machines: logic simulation was run on a Sun Fire V880 server (UltraSPARC III at 900Mhz); SAT solving was run on an Intel Xeon at 3.00GHz.

I ran experiments for both the 68HC05 and the 8051. For each, the goal was to see how far I could backspace before the pre-image set got too large or the computation blew-up. For the 68HC05, I reused the signature consisting of a hand-selected subset of 38 of the 109 total state bits, chosen based on my insight into the design. I also tried a 38-bit hash generated via universal hashing over the 109 state bits. For the 8051, I hand-selected a 281 bit subset of the 702 total state bits to be the "human architectural insight" signature. I also tried to use a 281 bit universal hash of the 702 state bits.

In these experiments, I used the k-backspaceable computation (i.e., pre-image sets are allowed to have up to k states), with k set to 300 states. To keep my experiments manageable, I also set an upper limit of 500 cycles of backspacing per test crash state.

Tables 2.1 and 2.2 show the results for the 68HC05. With the hand-chosen subset of bits, my framework hit the preset 500-cycle limit on 3 of the 10 test crash states. But on 4 of the 10, it cannot backspace more than a handful of cycles. With a universal hash of the same size, all 10 test crash states can be backspaced to our limit, and all of the pre-images are very small. In Section 2.3.4, we will see that a hand-chosen subset of bits is a very low-overhead signature, whereas universal hashing all bits of a large design appears to be prohibitively expensive. This suggests a trade-off between quality and cost.

Table 2.3 presents the results for the 8051 using the hand-chosen subset of the state bits as the signature. The results are excellent: my framework can backspace up to the preset 500 cycle limit in 9 out of the 10 test crash states. Initially, my framework was unable to complete results for the 8051 with a 281-bit universal hash. The SAT solver exhausted the system's main memory (1 hour timeout and 1GB memory limit) on all 10 test cases. The universal hash function is essentially a matrix-multiplication over GF(2), with a random matrix, so it is not surprising that large instances are chal-

Crash State	# of Cycles BackSpaced	Max States in PreImg	Sim Time	Sat Time	Manager Time
s1	54	4096	63.44	0.93	204.42
s2	1	65536	1.45	86.61	4.38
s3	37	4096	62.65	0.71	139.67
s4	7	4096	37.76	0.52	27.11
s5	53	4096	116.16	0.92	200.34
s6	500	1	1261.48	3.24	1884.31
s7	500	1	2384.29	3.15	1890.91
s8	500	1	4575.41	3.01	1893.89
s9	2	4096	22.93	0.51	22.93
s10	9	65536	2424.55	91.18	34.86

"Sim Time" is the time spent in the logic simulator. This time would be replaced by time running on the actual silicon. "Sat Time" is the time spent in the SAT solver. "Manager Time" is the time spent by the BackSpace Manager to supervise the framework and connect the various tools. Our BackSpace Manager implementation is very preliminary and can be optimized extensively.

Table 2.1: 68HC05 w/ 38-bit Subset Hand-Chosen Signature

Crash State	# of Cycles BackSpaced	Max States in PreImg	Sim Time	Sat Time	Manager Time
s1	500	2	1097.25	185.57	8524.15
s2	500	2	2011.04	187.21	8397.09
s3	500	2	2737.15	171.57	8335.45
s4	500	2	2988.38	242.89	8477.88
s5	500	2	3358.40	216.81	8398.14
s6	500	1	3176.94	31.89	8175.62
s7	500	1	6247.61	31.42	8280.93
s8	500	1	12207.49	38.58	8297.21
s9	500	2	15280.79	42.31	8173.19
s10	500	1	34084.53	36.63	8125.62

Table 2.2: 68HC05 w/ 38-bit Universal Hashing Signature

Crash State	# of Cycles BackSpaced	Max States in PreImg	Sim Time	Sat Time	Manager Time
t1	205	512	2841.07	4.58	6048.46
t2	500	256	21759.74	9.70	14720.71
t3	500	257	8326.66	10.84	14746.10
t4	500	257	10342.40	10.77	14772.03
t5	500	256	11587.21	11.26	14742.81
t6	500	256	11581.93	8.72	14735.07
t7	500	255	25767.40	8.54	14742.60
t8	500	256	13581.20	11.57	14759.73
t9	500	257	22493.04	10.62	14735.48
t10	500	257	24793.42	10.81	14759.77

Table 2.3: 8051 w/ 281-Bit Subset Hand-Chosen Signature

lenging for current SAT solvers. However, note that any full-rank matrix provides correct universal hashing, but a sparse matrix will be easier for the SAT solver, and also reduce area overhead. Thus, all that is needed is to generate a sparse matrix, which is done by increasing each matrix-element's probability of being zero, and then to check that the resulting matrix is full-rank. In these experiments, I generate the random hash matrix with a 0.985 probability of each entry being 0. The end result is that my framework can backspace up to our set limit for all 10 test crash states. Furthermore, the number of states in the pre-image is 2 orders of magnitude smaller for all crash states. Table 2.4[12] gives these results.

To summarize, Basic BackSpace works. It can compute hundreds of cycles of error trace backwards from a crash state. However, the area overhead is a potential limitation for this method and so it needs further investigation.

[12]Note that simulation time is an order of magnitude longer than the results shown in Table 2.3. Commercial logic simulators are optimized for speed. However, these optimizations are ineffective when testbenches probe signals deeper into the design through hierarchical references. In particular, the 8051 hash function is computed at the top level of the design, but all the 281 signals used in this computation are accessed through hierarchical references. Thus, the end result is a slow down of the simulation runs. In practice, because Basic BackSpace is to be used with a real chip, this simulation overhead can be ignored.

Crash State	# of Cycles BackSpaced	Max States in PreImg	Sim Time	Sat Time	Manager Time
t1	500	8	138616.15	1379.21	55389.29
t2	500	4	497905.92	1350.15	55104.32
t3	500	4	191655.42	1378.15	55462.20
t4	500	4	183283.27	1383.10	55642.82
t5	500	8	431057.79	1377.87	55039.00
t6	500	4	151950.65	1399.62	55601.11
t7	500	4	506787.53	1388.94	55639.58
t8	500	8	506229.79	1368.52	55512.44
t9	500	4	488157.90	1379.14	55049.31
t10	500	4	534870.14	1378.37	55448.52

Table 2.4: 8051 w/ 281-Bit Universal Hashing Signature

2.3.4 Initial Architectural Considerations

In [24], Marcel Gort describes in detail a post-silicon debug architecture for BackSpace, some possible variants and, their associated overheads. To make this proposal self-contained, I will describe the post-silicon debug architecture used during my preliminary experiments and comment on its implementation costs. I will present more advanced architecture considerations in Chapter 3.

In Figure 2.4, I depict a circuit under debug (CUD) augmented with our debug logic. More precisely, the basic debug architecture contains three major blocks: a breakpoint circuit (BRE); a signature creation circuit (SCR); and a signature collection circuit (SCO). The debug logic probes the CUD via two buses, N_{break} and N_{mon}. For my preliminary experiments, I assumed that the entire state of the chip could be probed. In other words, if N_{state} represents the CUD latches, then $\parallel N_{break} \parallel = \parallel N_{mon} \parallel = \parallel N_{state} \parallel$. In Chapter 3, I propose alternative architectures to relax this assumption. But, in general, if the set of N_{mon} signals cannot be determined at fabrication time, the selection of these signals can be made programmable at debug-time (e.g., use of tree-type multiplexer structures, concentrator access networks [45]). Such a network would programmably connect a subset

of the N_{mon} monitored signals for use in the signature.

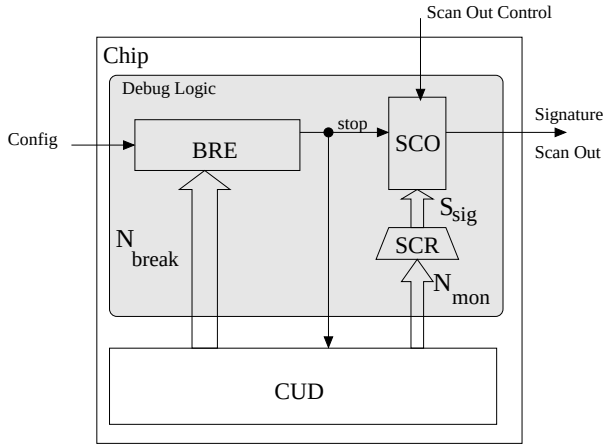

Figure 2.4: Debugging Architecture. BRE: breakpoint circuit; SCO: signature collection circuit; SCR: signature creation circuit; CUD: circuit under debug. During debug, while the chip runs, the BRE circuit compares the breakpoint value and the state of the CUD, via N_{break}, at every clock-cycle; the SCR monitors N_{mon} signals, generates signatures, S_{sig}, over those signals and stores them in the SCO. Upon a successful comparison, the BRE signals the SCO to stop the signature collection and also to stop the running chip. Typically, designs have some external controller (e.g., a single pin; or an interface to an external logic analyzer) that can signal to stop the chip. So, having BRE stopping the chip is not a design requirement, but this is how I implemented this debug logic for my preliminary experiments. Then, an external controller scans out the collected signatures.

The construction of the signature, SCR circuit, depends on what signature function the circuit implements (described in Section 2.3.2). Consider S_{sig} as a signature generated by SCR. If $\parallel N_{state} \parallel = \parallel S_{sig} \parallel$, then the history of all latches is stored in the SCO, and the circuit becomes trivially backspaceable. If $\parallel N_{state} \parallel > \parallel S_{sig} \parallel$, then missing bits must be reconstructed using off-chip analysis (described in Section 2.3.3).

The BRE is basically comprised of comparators and some number of configuration registers. I assume that a configuration, i.e., a breakpoint value, is always loaded into the BRE during the chip's initialization phase. Typically, designs have some external controller that can signal to stop the

chip/CUD at will (e.g., via a single pin; or via an interface to an external logic analyzer). In this basic debug logic, however, BRE's *stop* signal stops the running CUD when asserted, otherwise BRE does not interfere with the CUD.

The SCO is mainly a memory circuit, controlled by the BRE and some external controller that scans-out the SCO's internal memory. I assume the SCO's memory is arranged as a FIFO buffer. The depth of this FIFO buffer dictates how many consecutive states can be stored, i.e., when the FIFO's depth is larger than one, this circuit implements a so-called trace-array (AKA trace-buffer). I assume that the SCO is always running.

Clearly, the area overhead is a function of these three circuits. The breakpoint circuit comprises a set of latches and a comparator; SCR can be a variety of circuits. One example is a *universal hash function* (discussed in Section 2.3.3), which will incur an overhead due to extra logic-gates. Another example is a hard-wired signature, in which a pre-selected set of latches is chosen as a signature incurring no area overhead. The SCO overhead depends on the choice for the signature storage (e.g. latches, SRAM). Based on Marcel's architectural studies [24], I have drawn a couple of important conclusions regarding the area overhead of the presented debug logic: using concentrator networks to select observable signals yields an overhead between 1.5x and 40x compared to hard-wired signatures; and using the *universal hash function* to generate signatures yields an overhead between 10x and 20x compared to hard-wired signatures. These two points indicate that based on area overhead, using hard-wired signatures is the best alternative. (Obviously, these are impractical overheads. I will revisit them later.)

2.3.5 Results on a Hardware Prototype

Given the encouraging results of my preliminary experiments, my next step was 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. I did this work together with Marcel Gort, an M.Sc. student in the ECE department at UBC. We

selected a classic RISC processor with 32-bit datapaths and a 5-stage integer pipeline: the OpenRisc 1200 [38]. This core is non-trivial, open-source, and software-friendly (i.e., we can compile and simulate real applications). The only drawback is that the RTL implementation does not have a memory controller, which prevented us from working with more complex applications (e.g., the Linux operating system). 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 2.5 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.

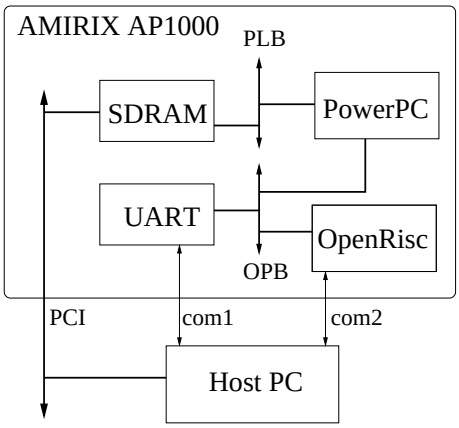

Figure 2.5: OpenRISC 1200 Implemented onto AMIRIX AP1000 Board

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.

My implementation of the BackSpace hardware was also fairly straightforward. The OpenRisc design did not have scan-chains. Thus, to avoid having to interface my framework with a commercial scan-chain insertion tool, Marcel augmented the design with hardware structures to mimic a full scan-chain circuit. For a signature, I hand-selected 1276 out of the 3007 latches, without any further compression. To provide a more flexible environment for experimentation, Marcel 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 (we only store the most recent, however). 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.

We were able to successfully run this prototype[13]. We can load simple software applications onto the OpenRisc, run, and then BackSpace at will. We report in Table 2.5 some results on using BackSpace on this system. We show results for two simple application programs: the Euclidean Algorithm for greatest common divisor; and the Sieve of Eratosthenes for computing prime numbers. For target/crash states, I 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, my framework was 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. Run-times 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 modeled by Basic BackSpace's theory, the framework can handle the non-determinism, but it produces a run-time slowdown,

[13]In 2008, we gave live demonstrations in two fairs: Canadian Microelectronics Corporation's TEXPO and Semiconductor Research Corporation's TECHCON.

because it must repeatedly try to hit each breakpoint.

Similarly, in my preliminary experiments, 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, *gcd*1 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, in Fig. 2.5, column 5, 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., *prime*1, 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.

Crash State	# of Cycles BackSpaced	Max States in PreImg (freq)	Most Frequent PreImg Size (freq)
gcd1	322	> 300(1)	17(321)
gcd2	442	> 300(1)	17(441)
gcd3	500	34(1)	17(499)
prime1	500	80(1)	1(498)
prime2	500	80(1)	64(100)
prime3	369	> 300(1)	64(100)

Crash State	# Retries per cycle			Run Time			Sat Time	Manager Time
	run1	run2	run3	run1	run2	run3		
gcd1	30.5	31.0	25.4	9,840	11,302	8,952	366.3	2,440
gcd2	27.1	35.0	30.1	11,982	17,134	14,460	496.3	3,330
gcd3	28.7	21.1	32.3	14,387	10,562	17,936	556.0	3,774
prime1	2.4	15.2	4.7	1,826	7,637	3,026	539.7	3,782
prime2	58.3	34.8	23.3	31,718	19,270	13,345	565.0	3,730
prime3	14.1	24.8	14.3	5,910	10,118	6,226	420.6	2,783

We used two programs, gcd and prime. The upper table shows number of cycles computed and pre-image statistics. The bottom table shows number of retries needed due to non-determinism and overall runtime statistics.
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. 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. For each of these states, we repeated the backspace computation 3 times (reported as "run1", "run2" and "run3" in the bottom table). In the bottom table, for each crash state and for each of the 3 runs, we report "# Retries per Cycle"— the average number of retries over the 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. Notice that the average varies from run to run, which 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 runs.

Table 2.5: Results for BackSpacing the OpenRisc 1200

2.4 Practical Limitations

In the preceeding sections, I presented the theory of BackSpace and explored its feasibility by means of simulation and hardware prototyping. Although the results are promising, I found two practical limitations, namely, excessive area overhead and excessive number of repetitions due to non-determinism and/or randomness. I detail these limitations and what to look for in the next chapters.

2.4.1 Area Overhead

The initial architectural studies (Section 2.3.4) point to an area overhead of 1.5x to 40x, which renders Basic BackSpace a theoretical work. However, other researchers such as Park and Mitra [41] and, more recently, Gort et al. [23], have shown that it is possible to achieve far less area overhead. In particular, Gort et al., whose research is based on Basic BackSpace, demonstrated that the area overhead can be cut to about 20%. Nevertheless, insisting on either replacing existing hardware debug logic with Basic BackSpace or requesting more overhead to accommodate this framework is not practical.

In Chapter 3, I will describe a technique that leverages existing hardware debug logic. In fact, the new technique has *no additional* area overhead, and thus, is much easier to adopt in practice.

2.4.2 Non-Determinism

In Section 2.3.5, it became clear that non-determinism can negatively affect the performance of Basic BackSpace. It appears that the only solution is "determinizing" the system. Unfortunately, as systems start adding more sources of non-determinism (e.g., multiple clocks, asynchronous communication), determinizing these systems become more difficult. Also, because bugs may be observable only in the original, non-deterministic design, determinization may hide bugs.

I will present a new technique in Chapter 4 that, instead of being con-

cerned with reducing non-determinism, can better cope with it and thus, be able to handle complex bugs in designs/environment where non-determinism is pervasive.

Chapter 3

TAB-BackSpace: Computing Traces with Zero-Additional Area Overhead

...the separation between past, present, and future is only an illusion, although a convincing one.
Albert Einstein

3.1 Introduction

In the previous chapter, I showed that BackSpace perfectly solves the trace computation problem by computing arbitrarily long sequences of all on-chip signals up to the bug. I also showed, however, that BackSpace requires an impractical amount of hardware overhead: correctness relies on computing breakpoints, signatures, and pre-images over the entire concrete state of the chip. In this chapter, I present a new technique, dubbed TAB-BackSpace (Trace-Array Buffer BackSpace), which lifts the BackSpace algorithm to an abstract setting, leverages *already existing* in-silicon debug logic (i.e., trace buffers), and therefore has *no additional* hardware cost.

Recall that, in post-silicon debug, little can be done until a trace leading to an observable bug or crash is available. Because of the critical importance of these traces, almost all chips have some debug logic to facilitate deriving them. For example, the same scan chains [53] for manufacturing test can be used to get a single-cycle snapshot of the state of many on-chip signals. However, this process is slow, so these snapshots can be taken only

rarely during a chip's execution. Furthermore, stopping the chip to take a scan dump disturbs the chip's interaction with its environment, potentially changing or obscuring buggy behavior. To compensate for the single-cycle and disruptive nature of scan dumps, complex chips often include "trace buffers" or "on-chip logic analyzers" (e.g.,[5, 49]): a limited number of the most important on-chip signals are routed to and recorded in a FIFO, with some mechanism to trigger starting and stopping of recording. These allow recording a multi-cycle trace of internal signals, while the chip is running at full speed. Unfortunately, because of the die area overhead of the trace buffer, the number of cycles of history that can be stored is small. In practice, considerable ingenuity, persistence, and luck are required to trigger scan dumps or trace buffer recordings at exactly the right times to observe the correct signals just before a bug manifests itself. The on-chip debug logic helps a lot, but obtaining debugging traces is still an exceedingly challenging problem.

Similarly to BackSpace, I show that TAB-BackSpace eliminates all this guess-work by computing a trace backwards in time (from the bug towards the initial state). Also, given that most complex chips already include some on-chip debug hardware, I treat this existing hardware as "free", and by making use of it, I flipped the problem around: instead of insisting on an excessive hardware overhead, could BackSpace successfully work with this existing on-chip debug logic? Indeed, I show in this chapter that TAB-BackSpace not only leverages existing in-silicon debug hardware, but also achieves the effect of extending the trace buffer arbitrarily far back in time, i.e., an effectively unlimited length trace buffer (assuming no spurious traces — See Section 3.3.2.).

I present TAB-BackSpace in the next two sections. In Section 3.2, I develop a theory of BackSpace with abstraction, which provides the framework for using partial information (available from trace buffers) and introduces the danger of spurious traces. Then, in Section 3.3, I introduce the TAB-BackSpace algorithm, including how to suppress spurious traces. In Section 3.4, I conduct both simulation and hardware experiments to assess and validate the overall TAB-BackSpace method.

3.2 Abstract BackSpace

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

Formally, I model the full chip on-silicon as a finite-state transition system with state space S_c and (possibly non-deterministic) transition relation $\delta_c \in S_c \times S_c$. This is the *concrete* system. As is typical in model checking [14], let's abstract away the inputs and consider only signals on-chip as the state. A *concrete trace* is a finite sequence of concrete states $s_1, \ldots, s_n$ such that $\forall i.\ (s_i, s_{i+1}) \in \delta_c$.

The choice of signals to record in the trace buffer defines an abstraction function $\alpha : S_c \to S_a$ that projects away everything but the chosen signals. S_a is the abstract state space, and the abstract transition relation $\delta_a(s_a, t_a)$ is defined as usual (e.g., [16]):

$$\exists s_c, t_c\, [\delta_c(s_c, t_c) \wedge s_a = \alpha(s_c) \wedge t_a = \alpha(t_c)] \tag{3.1}$$

An *abstract trace* is a finite sequence of abstract states $s_1, \ldots, s_n$ such that $\forall i.\ (s_i, s_{i+1}) \in \delta_a$. Informally, given a concrete crash-state, which when abstracted onto the trace-buffer signals yields an abstract crash-state, the Abstract BackSpace algorithm computes an arbitrarily long abstract trace following these steps:

1. Run the chip until it crashes or exhibits the bug. Like in the original BackSpace, this could be an actual crash or a programmed breakpoint.

2. Scan out the abstract crash-state, including the signature (e.g. the partial information in the trace-buffer).

3. Using formal analysis of the corresponding RTL, compute the abstract pre-image of the abstract crash-state (constrained by the signature as in BackSpace).

4. For each abstract-state s in the pre-image, let s be the new breakpoint;

re-run the chip; if the chip reaches the breakpoint, then s is a valid abstract predecessor-state.

If the algorithm successfully computes the abstract pre-image set in step 3, the algorithm proceeds with step 4 and iterates. In each iteration this algorithm adds one new abstract-state to the trace computed so far. Otherwise, it terminates (the pre-image computation is too large, exceeding memory's capacity).

The upside of this algorithm is that now the pre-image computation is done in the abstract state space of the transition system, thus reducing the chance of exceeding memory resources (compared to the pre-image computation over the entire concrete state space in BackSpace). The downside, however, is that Abstract BackSpace may compute "spurious traces", which I define next.

Given a concrete trace, σ_c, let the abstraction function, $\alpha(\cdot)$, be lifted to traces such that each concrete state in σ_c is pointwise abstracted. Thus, the result of the abstraction function is a unique abstract trace $\alpha(\sigma_c)$. In the opposite direction, an abstract trace σ_a is said to be *concretizable* if there exists a concrete trace σ_c such that $\sigma_a = \alpha(\sigma_c)$. Because the abstract transition relation is conservative, not all abstract traces are concretizable; such traces are called *spurious*. In practice, concretizability is a crucial property: a spurious trace does not correspond to any possible execution of the real hardware, so it is not only wrong, but it misleads the debug engineer and wastes time. There are two sources of spurious transitions in the abstract BackSpace algorithm:

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

2. Because the breakpoint is done on the abstract state, when I re-run the chip, it may breakpoint at the wrong time, or (in the presence of non-determinism) on a completely different trace, because a wrong concrete

state might map to the same abstract state as the correct concrete state. I call this situation a *false match*. For example, consider a 4-bit counter, where the high-order two bits are the abstract state. If the algorithm tries to compute a trace leading to state 1111 from state 0000, the abstract target will be state 11, which has abstract predecessor states of 11 (because, e.g., 1110 goes to 1111) and 10 (because 1011 goes to 1100). If I set 10 as the abstract breakpoint, the algorithm will hit it, and then try to compute abstract predecessors of 10, etc. Eventually, abstract BackSpace can compute a 4-cycle trace 00, 01, 10, 11, which is spurious. The shortest non-spurious abstract trace is 00, 00, 00, 00, 01, 01, 01, 01, 10, 10, 10, 10, 11, 11, 11, 11.

The problem of identifying spurious traces have been extensively studied. The most common approach is by *Counterexample Guided Abstraction Refinement*, CEGAR [15, 18, 25]. Fundamentally, the goal of abstraction refinement techniques is to prevent spurious counterexamples (non-concretizable traces) by creating new abstract models containing more information about the design. Unfortunately, refining the abstract model in post-silicon debug means that every time a spurious trace is found, a chip re-spin would be required, which is unrealistic.

Therefore, the challenge is to minimize the risk of spurious traces, which is addressed in the next section.

3.3 TAB-BackSpace

3.3.1 Intuition

For TAB-Back-Space, my goal is to leverage the underlying insights of BackSpace, but use existing on-chip debug hardware and *no pre-image computation*. Accordingly, I have to make some assumptions about what is available. I assume a trace buffer that records a set of signals. (In this thesis, I assume this set of signals have already been determined via some automatic framework, e.g., [31, 40], or using architectural insights from the chip designers.) This recording must be able to run continuously (e.g., trace

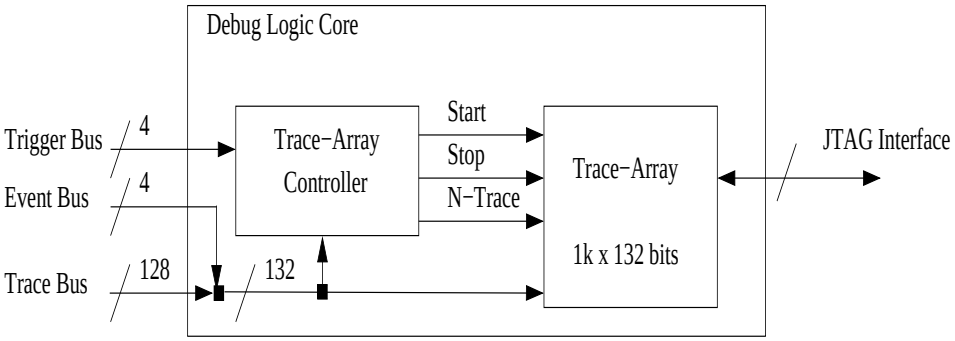

Figure 3.1: IBM's Cell Processor Debug Logic Core (DLC) High-Level Block Diagram. The DLC has 3 inputs: the trigger bus, carrying signals used to control the trace array; the event bus, carrying additional status signals that may be stored in the trace array; and the trace bus, carrying pre-selected signals to be stored in the trace array. The trace array controller (TAC) uses signals from both the trigger and the trace buses to control the recording of information (*start/stop* recording, and *N-trace* for recording N consecutive cycles).

buffers are usually implemented as circular buffers). And it must be possible to set a breakpoint to stop recording when the circuit reaches a specific value on the trace buffer input signals. These are minimum requirements; the method can be improved if some of these are better, e.g., if I can set multi-cycle breakpoints.

To make things more concrete, I base my experiments on a well documented post-silicon debug infrastructure: the debug architecture used on IBM's Cell processor[49]. Fig. 3.1 depicts the debug architecture. This architecture provides many debugging features, but for this thesis, I use only the minimum requirements: trace buffer recording, and the breakpoint (trigger) capability.

This architecture has been typically used by the lab engineer guessing when to start/stop recording information into the trace-array. However, finding the right time window to capture the chip's partial state information is one of the most time-consuming tasks faced in post-silicon debug[14]. With

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

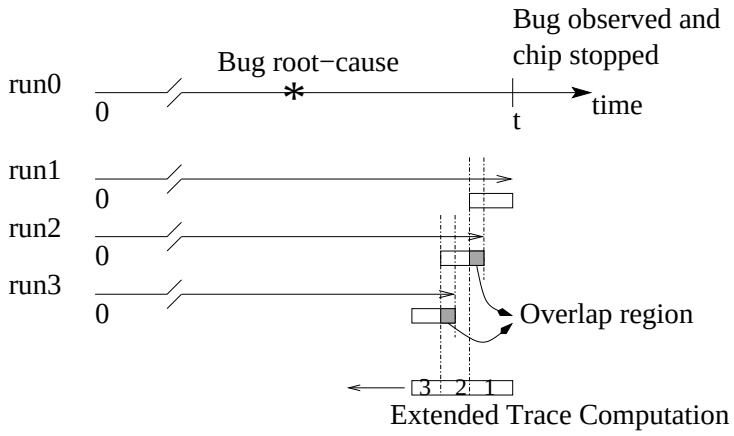

Figure 3.2: TAB-BackSpacing. Once the bug is observed, re-run the chip with trace arrays enabled, i.e., run1; collect the information from the trace arrays and compute a new set of triggers for the subsequent run (run2); and iterate these steps, extending the length of the computed trace beyond the trace arrays' depth.

TAB-BackSpace, I will eliminate this problem.

Fig. 3.2 gives an overview of TAB-BackSpace. I assume that the trace buffers are always recording until stopped by a trigger. TAB-BackSpace iterates the following steps:

1. Run the chip until it "crashes" (hits the bug or the programmed breakpoint).

2. Dump out the state of the trace buffer into a file.

3. Select an entry from the trace dump as the new trigger condition, configuring the breakpoint circuitry to stop the chip when it hits this breakpoint on the next run.

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

The next two subsections formalize this intuition.

3.3.2 Theory of TAB-BackSpace

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

The first technique is the breakpoint mechanism. I never try to combine traces unless the new trace, hits the breakpoint in the previous trace (i.e., the hardware reaches a specified state). Because the two traces share an identical state, I am guaranteed that by combining the two traces at that state I have computed a valid, longer trace — but the guarantee is only valid at the level of abstraction of the breakpoint state. In the original BackSpace, the breakpoint was concrete, guaranteeing that the algorithm constructed a valid, concrete trace leading to the bug. In TAB-BackSpace, the breakpoint is only on a partial state, so the guarantee is only that the constructed trace is a legal, but possibly spurious (non-concretizable), abstract trace.

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

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

In other words, if two concrete executions share a long enough period of abstracting to the same states, then the future concrete execution is oblivious to what happened before that period, and so the combined abstract trace

is not spurious. Note that, in practice, the divergence length is specific to the design and also to the chosen abstraction function. Although l_{div} may not always exist (because, for example, the abstraction function might abstract away key information from the concrete traces), in theory, it is straightforward to check whether the length of the overlapping region is longer than l_{div}: let f be the length of the overlapping region. Do there exist two traces $\sigma_1{=}x_1y_1z_1$ and $\sigma_2{=}x_2y_2z_2$ such that $|x_i|{=}|z_i|{=}1$, $|y_1|{=}|y_2|{=}f$, $\alpha(y_1){=}\alpha(y_2)$, and either $x_1y_1z_2$ or $x_1y_2z_2$ are not valid traces? If not, then $f > l_{div}$. Otherwise, $f < l_{div}$. Therefore, all that needs to be done is to unroll the design (as in bounded model checking [7]) up to $f{+}2$ cycles and check for a witness.

In practice, it may be unrealistic to unroll the design for $f{+}2$ cycles. However, I show that it is possible to empirically limit the number of experimentally observed spurious traces. In particular, if I have trace dumps from different concrete executions that match on the overlap region, I dub this a "false match", which is a necessary (but not sufficient) condition for a spurious trace. In Section 3.4.1, experiments show that false matches are rare when the overlap region is reasonably long.

Algorithm 3, shown on page 58, presents the TAB-BackSpace procedure: starting from a given crash state and its corresponding trace-buffer, it iteratively computes an arbitrarily long sequence of predecessor abstract states by going backwards in time. This procedure has 4 user-specified parameters: *steps_bound* specifies how many iterations back the algorithm should go; *retries_timeout* limits the amount of search for an equivalent overlapping region between the new trace dump and the trace computed so far; the *time_bound* (as in Algorithm 1, page 24) is a timeout for each chip-run and is a mechanism to tell whether a chip-run went on a path that does not reproduce the crash-state or buggy-state; and, *lbindex* is the trace buffer's smallest index, which defines a region for the overlapping of two consecutive trace buffers.

This procedure has 2 nested loops. The outer loop, lines 13 – 41, controls the three termination conditions for the algorithm: we reach the user-specified number of iterations; we reach the initial states; or the previous

iteration was unsuccessful. The outer loop is also responsible for joining the new trace buffer dump onto the successful trace computed so far (line 32), and then selecting a new state as the breakpoint for the next iteration. The inner loop, lines (17 – 31), is responsible for controlling the hardware while trying out different candidate-states, s_{cand}, given a *retries_timeout*. The procedure keeps track of time using the subroutine *ElapsedTime*() (passing *reset* as parameter resets the time counter, otherwise it counts the elapsed time since it was last reset). In each loop iteration, the procedure loads s_{cand} into the breakpoint-circuit (line 19), and runs the chip. The objective is to collect a new trace-buffer upon matching s_{cand} and match (after overlapping) it with the previous trace-buffer. If *ResetAndRun*() returns $TRUE$ then the breakpoint circuitry matched s_{cand} and we have a new trace-buffer. Otherwise, the chip-run violates the *time_bound* parameter (line 21) because the current run took another path (caused by non-determinism). If the breakpoint occurs, we dump the contents of the trace-buffer, for comparison with the trace computed so far. The *OverlapAndCheck()* subroutine (line 24) is responsible for matching the overlapping region of the previously computed trace with the new trace dump and checking for equality. If the procedure neither breakpoints nor proves equality, *PickState*() (line 29) selects another candidate-state from the previous trace using a round-robin scheme while respecting *lbindex*, and the inner loop iterates. The procedure exits the inner loop when either it successfully proves equality of the overlapping regions of the two trace-buffers, or this loop has iterated longer than the specified *retries_timeout*.

The main correctness theorem proves that the trace computed by Algorithm 3 is as informative as one could hope: it concretizes to a trace that leads to the actual crash state, using reachable states.

Theorem 4 (Correctness of Trace Computation)**.** *If the size of the overlapping region used to check equality is greater than l_{div}, then the trace produced by Algorithm 3 is concretizable to the suffix of a concrete trace leading from the initial states Q_0 to the crash state s.*

Proof: The proof is by induction on the iteration count i at line 37. The

Algorithm 3 Crash State History Computation

```
1: input Q_0 : set of initial states,
2:        (s, t) : crash-state and trace-buffer
3:         steps_bound ∈ ℕ^+ : user-specified bound on the number of iterations,
4:         retries_timeout ∈ ℕ^+ : user-specified time-bound on retrials,
5:         time_bound : user-specified time bound for any chip-run
6:         lbindex: user-specified lower-bound length of overlap region;
7: output trace : equivalent sequence of abstract states;
8: // init. breakpointable candidate-state, current trace-buffer and final trace
9: i := 0; s_cand := s; t_i := t; trace := (t_i);
10: // initialize variable nindex; nindex gets updated by PickState()
11: // nindex range is [lbindex, |trace-buffer|]
12: nindex := lbindex; succ_iteration := FALSE;
13: while (i < steps_bound) && (s_cand ∉ Q_0) && (succ_iteration = TRUE) do
14:       equivalent := FALSE; matched := FALSE;
15:       //Resets retrial elapsed time
16:       ElapsedTime(reset)
17:    while (!equivalent) && (ElapsedTime(go) ≤ retries_timeout) do
18:       // Program the hardware-breakpoint circuitry with s_cand
19:       LoadHardwareBreakpoint(s_cand);
20:       // (Re-)run the chip at full-speed with timeout time_bound
21:       matched := ResetAndRun(time_bound);
22:       if matched then
23:          t_i := ScanOut();// Dump trace-buffer contents t_i
24:          equivalent := OverlapAndCheck(t_i, t_{i-1}, nindex);
25:       end if
26:       if (!matched) || (!equivalent) then
27:          // Pick another state following a round-robin scheme
28:          // and updates nindex
29:          s_cand := PickState(nindex, t_{i-i});
30:       end if
31:    end while
32:    if equivalent = TRUE then
33:       // Accumulate trace
34:       OverlapConcatenate(t_i, trace);
35:       // Pick a candidate-state in t_i for the next iteration
36:       s_cand := PickState(nindex, t_i);
37:       i := i + 1;
38:    else
39:       succ_iteration := FALSE
40:    end if
41: end while
42: return trace;
```

base case is trivial, as when $i = 0$, the trace is a single trace buffer dump that ends at the crash state. Since this trace dump is taken from the physical chip, it can be concretized to the specific physical execution that occurred on-chip.

In the inductive case, let uy represent the trace computed so far, and let xv represent the new trace dump t_i, with $v = u$. In other words, u and v are the overlap region. By construction, x and y are non-empty.

We know that xv is concretizable to a trace with all states reachable from the initial states, because it is taken directly from the hardware. Let $x_c v_c$ be one such concretization of xv, with $x = \alpha(x_c)$ and $v = \alpha(v_c)$. Similarly, uy is concretizable to a trace that leads to the crash state s, by the inductive hypothesis. Let $u_c y_c$ be one such concretization of uy, with $u = \alpha(u_c)$ and $y = \alpha(y_c)$. From the hypotheses, $|u| = |v| > l_{div}$, so by the definition of l_{div}, both $x_c u_c y_c$ and $x_c v_c y_c$ are legal concrete traces. By construction, both start at reachable states, and therefore contain all reachable states. Furthermore, both end at the crash state s. Therefore, both $x_c u_c y_c$ and $x_c v_c y_c$ are witnesses that the new trace computed by Algorithm 3, xuy, is concretizable to the suffix of a concrete trace leading from the initial states to the crash state. ∎

When comparing TAB-BackSpace to both preceding algorithms (BackSpace and Abstract BackSpace), I point out the following:

- Like BackSpace, because it works backwards from the crash state, TAB-BackSpace eliminates manual guesswork about when to trigger state recording for the trace arrays. Completely automatically, it computes an arbitrarily long trace dump.

- Unlike BackSpace, because trace buffers typically have many cycles of history, TAB-BackSpace can compute many cycles back on each iteration, gradually gluing together entire trace buffer dumps, instead of individual states.

- Like BackSpace, TAB-BackSpace relies on repetition to compensate for lack of observability. Hence, it needs the same assumption that

the bug appears somewhat repeatably. Therefore, like BackSpace, TAB-BackSpace can handle some non-determinism/randomness in the system behavior, but not too much (see Section 3.4.1).

- Unlike BackSpace, there is no pre-image computation. Instead, TAB-BackSpace use the fact that the trace buffer records actual history of the chip. This completely eliminates a major computational bottleneck of BackSpace. Furthermore, this also eliminates the assumption that the silicon matches the RTL; it can still compute a trace.

- Because TAB-BackSpace is using only the small set of important signals that reach the trace buffer, it is computing an abstract trace. Therefore, TAB-BackSpace has low overhead thanks to abstraction. Furthermore, because we are using pre-existing debug hardware, whose overhead has been already accounted for, TAB-BackSpace has no additional on-chip overhead at all.

- Assuming an overlapping region longer than l_{div}, Theorem 4 proves it is always the case that TAB-BackSpace computes a concretizable trace leading up to the actual crash state, which is not always true for Abstract BackSpace. However, finding this appropriate length for the overlapping region may be hard in practice. Thus, there is a risk for spurious traces. Note that since there is no pre-image computation with TAB-BackSpace, one source of spurious transitions that exists in Abstract BackSpace is completely eliminated. The only danger is false matches.

How can false matches be suppressed in practice? I investigate this question in the next section.

3.4 Experimental Results

In this section, I validate TAB-BackSpace using both simulation and hardware experiments. I use simulation because some of the experiments, such

as investigating spurious traces, need full visibility to the design and so simulation is necessary. The real question is whether TAB-BackSpace works in real silicon. To answer this question, I validate TAB-BackSpace on an actual chip, the IBM POWER7[15].

3.4.1 Results on Simulation

Suppressing Spurious Traces

In the previous section, I showed that Algorithm 3 computes a concretizable trace leading up to the actual crash state. The fundamental assumption is that Algorithm 3 always use an overlapping region, f, that is longer than l_{div}. However, as noted earlier, determining the value of f may be hard or impossible in practice.

It is tempting to make analytical models of the probability of false matches to compute the appropriate value of f. Unfortunately, very little can be said, because the abstract states are not random states, but the result of an abstraction function. Given a really bad abstraction function (e.g., one that focuses on irrelevant bits whose values seldom change), there will be many false matches.

Instead, I show experimentally that the risk of false matches can be made zero or near zero in practice. In particular, I measure the frequency of undetected false matches that may generate a spurious transition: when the abstract breakpoint has a false match, but so does the entire overlap region.

Setup:

I chose to work with a design that is non-trivial, but also not too complex so that I can simulate it in its entirety, understand it and leverage any architectural insight in selecting signals to be probed using a trace-buffer. I chose to use a router design, which is an RTL implementation of a 4x4 routing switch (conceptual block depicted in Fig. 3.3). This router is typically used by IBM for training new employees with IBM's tools. The design has

[15]These experiments were conducted during my internship with IBM Corp. - IBM Haifa Research Lab, Israel.

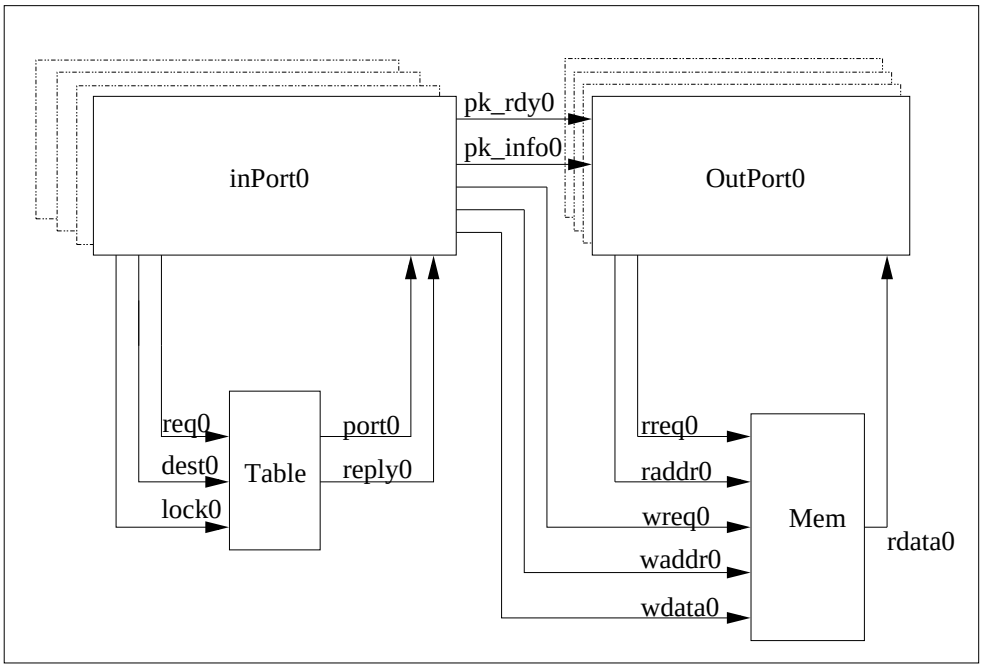

Figure 3.3: Router4x4 Conceptual Block. The router has 4 input/output packet processing blocks; a routing table; and a buffer for storing in-flight packets. For the sake of readability, I am presenting the basic signals used for communication among only four design blocks while abstracting away the input/output signals of the router top-level block.

9958 latches, which is larger than most open-source design examples (e.g., [38]), but not too large too run experiments, and collect and analyze results quickly.

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

I simulated the router on a feature-rich constrained-simulation environ-

ment developed by IBM, using Cadence's Incisive Simulator (with Specman Elite) version 09.20-s016. This proved very helpful when modeling environmental non-determinism.

Experiments:

I present two different experiments. The first experiment evaluates the probability of false matches versus amount of overlap. In Section. 3.3, I asserted that if the overlap region, f, is greater than l_{div}, then there can be no false matches. However, determining the value of f such that $f > l_{div}$ may be hard. Thus, I empirically evaluate what value for f is appropriate for the router. I set the simulation environment to be fully deterministic, i.e., using the same random seed will always generate the same simulation trace. I uniformly chose 100 abstract states, a_i, from this trace such that $\forall i.a_i \notin Q_0$ and $\forall i, j.(a_i \neq a_j) \wedge |(i-j)| > l$, where l is the TAB length. I set l to be 50, but unlike the CELL's debug logic, I am not relying on any cycle compression; and, the TAB width is set to 120 bits. I used my architectural insight of the design to select 120 different design signals. For each selected state a_i, I checked if there were earlier false matches, while incrementing the size of TAB overlap. Fig. 3.4 shows the results. These results substantiate my claims: even with a small overlap (25 cycles) the probability of a false match drops to about 1%. If I had set the length of the overlap to be 29 cycles, then I would have no false matches at all. Thus, choosing a value of $f \geq 30$ is likely to guarantee no false-match.

The second experiment concerns false matches and non-determinism/randomness. Intuitively, because complex chips have many sources of non-determinism, it is very unlikely that the exact same concrete state in one chip run, occurs in different, randomized runs. Does this phenomenon happen also with abstract states?

To test this hypothesis, I used the same constrained-simulation environment, but now with non-determinism enabled. This simulation environment provides many parameters to make each simulation run very different from the others. However, I control the non-determinism in the environment as much as possible so that I can better evaluate its impact on chip runs. More

specifically, I simulate non-determinism only affecting the delays on packet arrivals (a real scenario encountered in bring-up labs). I modified the original simulation environment so that it always uses a fixed random seed for everything except packet generation. For packet generation, I use an external and independent random generator to add different delays between packets in each run.

To simulate the scenario in which the desired "buggy trace" (original trace showing the bug) is repeatable with probability 1/6, I generated 5 additional random traces of similar length to the first trace (i.e., after a specified number of packets were sent). I uniformly selected 100 abstract states from the buggy trace in the same way as I did in the first experiment. For each selected abstract state, a_i, in the buggy trace, I checked for a false match on all other 5 traces. For all selected states, I found no false matches whatsoever, even with only one cycle overlap. This result shows that even with some non-determinism, traces from the same test-case can be very different, making this type of mismatch unlikely.

Computing Abstract Traces

The experiments in the previous subsection focused on showing empirically that choosing proper values for the overlap region greatly reduced the chances of a false match. In this section, my focus is on validating my claim that, indeed TAB-BackSpace can compute traces.

To this end, I use the Router design/environment from the previous subsection, but with following configuration:

1. TAB length set to 50 with no compression;
2. TAB width set to 75 bits with signals chosen out of 3 design blocks;
3. the overlap region set to $f = 30$;
4. total number of iterations is set to 20 (*steps_bound*=20 in Algorithm 4, page 81), that is, compute 20 trace buffers that overlap for f cycles;
5. timeout is set to 5 hours (*retries_timeout*=5 in Algorithm 4);

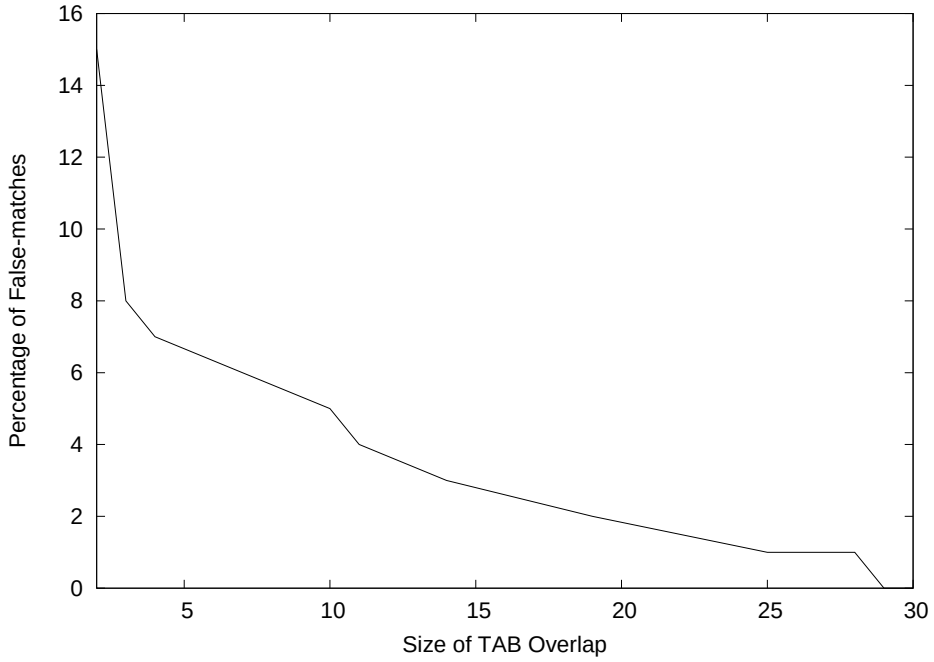

Figure 3.4: This is the percentage of false-matches for the Router design, assuming a TAB length of 50 entries, a TAB width of 120 bits and the maximum overlap of 30 consecutive cycles. We randomly selected 100 abstract states from a trace 13581-cycles long. This graph shows the percentage of those states that have false matches for a given overlap size. If I set the overlap size to 29, I eliminate all false matches.

6. 30 randomly chosen crash states, a_i, satisfying these two properties: $\forall i.a_i \notin Q_0$ and $\forall i, j.(a_i \neq a_j) \wedge |(i - j)| > 1000$ (These properties guarantee that the "crash" states in this experiment are far enough apart so that the computed traces are distinct. In other words, since $steps\_bound = 20$ and each trace dump has 50 cycles, even if the overlap between two consecutive trace-dumps were one single cycle, the total number of cycles for each trace would be 20*49+1 $<$ 1000, which is smaller than the distance between two crash states);

7. randomized inter-packet delays.

As item 7 suggests, non-determinism only affects the inter-packet delays.

CS	#Succ. Iter.	#Chip Runs	CS	#Succ. Iter.	#Chip Runs	CS	#Succ. Iter.	#Chip Runs
1	3	57	11	20	123	21	20	330
2	5	94	12	20	281	22	20	235
3	6	65	13	20	112	23	20	973
4	7	75	†14	20	115	24	20	360
5	8	65	15	20	210	25	20	390
6	8	306	16	20	311	26	20	345
7	8	234	17	20	137	27	20	193
8	20	321	18	20	246	28	20	87
9	20	324	19	20	271	29	20	383
10	20	233	20	20	365	30	20	531

Table 3.1: TAB-BackSpace Experiments. "CS" are the 30 crash states. "# Succ. Iter." is the number of iterations before timing out or reaching set limit of 20. The timeout per iteration was chosen to be 5 hours. Each simulation run averages 10 minutes. "# Chip Runs" is the total number of iterations plus the number of retries. Crash states with a † are states that nuTAB-BackSpace computed all 20 iterations, but somewhere during the computation it deviated from the "expected" trace. Therefore, these might be spurious traces. I suspect that, at some iteration, the normalized region of two different traces was too small to discriminate them.

These experiments simulate the scenario in which non-determinism has been extensively removed, i.e., the range of delays, that is, the number of cycles between packets, is small ($0 \leq \text{delay} \leq 5$).

Table 3.1 shows that TAB-BackSpace works successfully. More than 70% of the cases were able to complete the goal of 20 iterations. However, these results show evidence that non-determinism may negatively impact TAB-BackSpace. Consider only the cases which completed the 20 iterations. The average number of runs necessary to complete this goal is approximately 300 runs. This average is surprisingly high given that non-determinism is very limited in these experiments. In Chapter 4, I will address this issue. The real question now is whether TAB-BackSpace can repeat this success on silicon. This is what I investigate in the next section.

3.4.2 Results on Silicon

To demonstrate that TAB-BackSpace is feasible in practice, we used[16] an existing IBM processor, running in the post-silicon bring-up lab. Using the existing debug logic in the processor, we performed several iterations of TAB-BackSpace, extending our initial trace by nearly a thousand cycles.

Our experiments were conducted with the IBM POWER7 processor [30]. This processor has built-in hardware debug capabilities, whose architecture is similar to the debug features of the CELL processor [49](briefly discussed in Section 3.3.1).

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

At the time this bug was initially found, it was easily worked-around using existing logic on the processor. Our experiments were conducted after this fix, therefore we used a modified configuration of the processor, in which the workaround was disabled and the bug became active again. In addition, we created an environment in which we could deterministically re-run the processor and reach the crash caused by the bug. This involved running on bare metal, using the Threadmill post-silicon exerciser[2]. We configured the processor to use only one active core, since this is sufficient to reproduce the bug.

In each of the runs, we activate trace-arrays of the active core, to record signal values throughout the run (we activate a fixed subset of the trace-array signals, used in all our runs). When the processor stops, the contents of the trace-arrays reflect the values of the recorded signals, for some number

[16] I conducted the preliminary experiments to study the IBM POWER7 debug hardware and validate, in principle, that we could TAB-BackSpace the IBM POWER7 chip. Avigail Orni (IBM-Israel) conducted the final experiments with my remote guidance (since my internship had ended a few months earlier).

of cycles at the end of the run. The number of cycles represented in the trace-arrays may vary, since some compression is applied when values are repeated for consecutive cycles. After each run, the values of the trace-arrays are dumped to a file, and are processed in order to produce a decompressed.

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

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

Starting from this initial trace, we applied repeated iterations of TAB-Back-Space steps, creating a sequence of trace-array dumps. In each TAB-BackSpace step, we are assured that it will not continue to run past the breakpoint cycle. However, it is possible that the trigger pattern appears in an earlier cycle (an abstract *false match*, as described in Sec. 3.2), and thus the run will stop earlier than the breakpoint cycle. In a bare-metal lab run, we typically do not have a cycle counter, which could help us to detect whether we have stopped at the breakpoint cycle or earlier. We therefore use the overlap check. If the new trace and the current trace agree on all of the recorded signals, for the entire prefix of the current trace up to the breakpoint cycle, we consider the new trace to be a true trace leading up to the breakpoint cycle. To reduce the risk of *false matches*, we strive to make the overlap of two consecutive runs greater than (or equal to) half the length of the current trace. In addition, although we use only the 176 trigger signals for defining the trigger pattern, we perform the overlap check on all 352 recorded signals, which gives additional confidence we stopped at the correct breakpoint cycle.

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

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

In our experiments, we executed 10 TAB-BackSpace iterations, producing 10 new traces in addition to the initial trace. The results of these iterations are shown in Table 3.2. The first row represents the initial trace, while the following rows represent the traces generated by the TAB-BackSpace iterations. These traces are all 256 cycles long. The *New cycles* column shows the number of new cycles added in the current iteration. The *Accumulated new cycles* column shows the accumulated number of new cycles in the trace in all iterations up to and including the current one. The final accumulated number, at the bottom of this column, shows the total backward extension that we have achieved in these 10 iterations, which amounts to 988 cycles. For this particular bug, this extension was sufficient to reveal the root cause of the bug.

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

Table 3.2: TAB-BackSpace on POWER7

Chapter 4

nuTAB-BackSpace: Normalizing Non-Deterministic Traces into Equivalence Classes

Very few things happen at the right time, and the rest do not happen at all.

HERODOTUS

4.1 Introduction

In the previous chapter, I have shown that TAB-BackSpace works successfully in practice. The simulation results, however, show that non-determinism leads to an excessive number of retries (chip re-runs) per TAB-BackSpace iteration. The reason is the lack of reproducibility of the exact same chip run. Is it possible to do better with fewer retries? The answer to this question is yes, but, as I point out in the next paragraphs, not with conventional post-silicon debug methods.

In practice, engineers make great efforts to "determinize" as much as possible the system to improve reproducibility. This effort usually requires building highly specialized systems (e.g. [33, 52]) to improve controllability. In some cases, parts of the design may run at slower clock or simply turned off completely. In others, it may require confining the debug instances to a

single core (as we did with the IBM POWER7 processor in Section 3.4.2). This determinization effort works great only if bugs are reproducible in the determinized environment. Thus, there is still a risk that determinization might not help. For example, determinizing the system may not help debug problems that happen in the field (OEM boards), since the in-house, determinized systems are very different from what is in the field.

On the other hand, I observed that, in many cases, different traces share "similar views" of what is going on in the design. For example, a processor waiting for a grant to access a bus may sit idle for a different number of cycles from run to run. However, if the bus stalls the processor execution until it grants access, then the number of cycles the processor waits does not affect its internal state. Thus, in reality, these different executions represent essentially equivalent behaviors.

To capitalize on this insight, I extend TAB-BackSpace to account for these equivalent behaviors. I call this new technique nuTAB-BackSpace (Normalized TAB-BackSpace) since it normalizes traces into equivalent classes. More specifically, the goal of nuTAB-BackSpace is to allow the debug engineer to specify intuitive notions of "equivalence" by providing rewrite rules.[17] This provides ease-of-use, considerable expressiveness, and a rich underlying theory that allows efficient checking of equivalent traces. In particular, I treat the debug trace and trace buffer dumps as strings whose alphabet is the abstract state space of the design being debugged, and the user-provided rewrite rules produces a string rewriting system (also known as a semi-Thue system).

In the next section, I review definitions of semi-Thue systems that are relevant to this thesis. In Section 4.3, I formally present nuTAB-BackSpace. Finally, I conclude this chapter with experimental results.

[17]Recall that the debug process is iterative. Debug engineers formulate hypotheses about what might be going wrong, develop a test for the hypotheses, and then formulate new hypotheses based on the results. Because the focus is design errors, debug engineers have deep understanding of the design. Therefore, defining rewrite rules nicely follows the same debug flow of formulating/testing hypotheses.

4.2 Semi-Thue Systems

As pointed out in the previous section, my interest in semi-Thue systems lies in the fact that traces might form equivalence classes. Thus, instead of trying to exactly reproduce traces, it suffices to find equivalent ones.

In this section, I present an overview of semi-Thue systems[18] (STS). This subject is very broad, and so I focus on concepts such as normal forms, termination and confluence, which are fundamental for computing equivalence classes.

Definition 5. *A semi-Thue system, S, is a tuple* $(\Sigma^*, \mathcal{R})$*, where*

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

Each element $(l, r) \in \mathcal{R}$ is called a rewrite rule. A rewrite rule (l, r) may be notated as $l \to r$. The symbol $\to$ is called a reduction. Rewrite rules can be applied to arbitrary strings as follows: for any $u, v \in \Sigma^*$, $u \to v$ iff there exists an $(l, r) \in \mathcal{R}$ such that for some $x, y \in \Sigma^*$, $u = xly$ and $v = xry$. In Table 4.1, I recursively define the notation used for reductions $\to$.

The definition of $\overset{*}{\leftrightarrow}$ and its relationship with normal forms are of particular interest in this thesis. First, the relation $\overset{*}{\leftrightarrow}$ is the least equivalence relation on Σ^* containing $\to$. And second, computing semi-Thue systems' equivalence classes depends heavily on the existence of normal forms. Let's now define normal forms.

Definition 6. *Let S be an STS with alphabet* Σ*.*

a) Denote the set of descendants of $u \in \Sigma^*$ *as* $\Delta(u)^* = \{v | u \overset{*}{\to} v\}$ *and the set of proper descendants as* $\Delta(u)^+ = \{v | u \overset{+}{\to} v\}$*.*

b) A string $u \in \Sigma^*$ *is called irreducible if* $\Delta(u)^+ = \emptyset$*.*

[18] The name Thue comes from Axel Thue, who developed the string rewriting calculus. Semi-Thue systems have been extensively studied; the presentation in this section is based on [6, 8, 27].

$\overset{0}{\rightarrow} \equiv \{(x,x) \mid x \in \Sigma^*\}$	identity
$\rightarrow \equiv \{(x,y) \mid (x,y) \in \mathcal{R}\}$	simple reduction
$\overset{i+1}{\rightarrow} \equiv \overset{i}{\rightarrow} \circ \rightarrow$	(i+1)-fold composition $i \geq 0$
$\overset{+}{\rightarrow} \equiv \bigcup_{i>0} \overset{i}{\rightarrow}$	transitive closure
$\overset{*}{\rightarrow} \equiv \overset{0}{\rightarrow} \cup \overset{+}{\rightarrow}$	reflexive transitive closure
$\overset{-1}{\rightarrow} \equiv \{(y,x) \mid (x,y) \in \mathcal{R}\}$	inverse
$\leftrightarrow \equiv \rightarrow \cup \overset{-1}{\rightarrow}$	symmetric closure
$\overset{+}{\leftrightarrow} \equiv (\leftrightarrow)^+$	transitive symmetric closure
$\overset{*}{\leftrightarrow} \equiv (\leftrightarrow)^*$	reflexive transitive symmetric closure

Table 4.1: Reduction Notations and Descriptions. The symbol $\circ$ represents the standard definition of relational composition.

c) If $u \overset{*}{\leftrightarrow} v$ and $\Delta(v)^+ = \emptyset$, then v is called a normal form of u and $[v] = \{u | u \overset{*}{\leftrightarrow} v\}$ denotes the entire equivalence class of v.

For example, consider the STS S_1 in Fig. 4.1. Let $u = abcdeffff$. The set of all descendants of u are $\Delta(u) = \{abcdeef, abbdeffff, abbdeef, abbeef\}$; the string $abbeef$ is irreducible because $\Delta(abbeef) = \emptyset$; $abbeef$ is a normal form of u since by definition, $\overset{*}{\leftrightarrow} \equiv (\rightarrow \cup \overset{-1}{\rightarrow})^*$ and so $abcdeffff \overset{*}{\leftrightarrow} abbeef$; and, the equivalence class is $[abbeef] = \{abcdeffff, abcdeef, abbdeffff, abbdeef, abbeef\}$. Furthermore, $abbeef$ is the unique normal form of u.

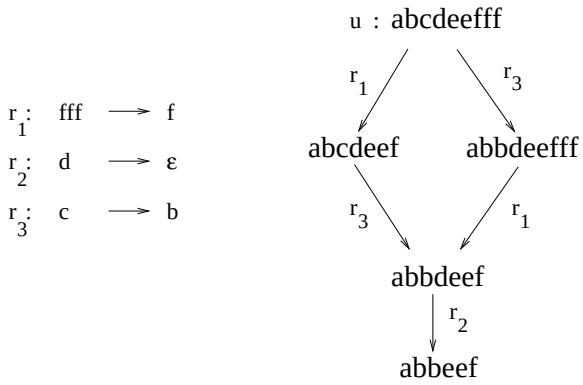

Figure 4.1: Given $\Sigma = \{a, b, c, d, e, f\}$ and $u \in \Sigma^*$, let $S_1 = (\Sigma^*, \mathcal{R})$ and $\mathcal{R} = \{r_1, r_2, r_3\}$.

Given an STS, computing the set of descendants and the set of irreducible strings is undecidable in general. However, if every element in STS has a

unique normal form, then these problems become solvable. To determine whether an STS has a unique normal form we use the concepts of termination and confluence.

Definition 7. *A semi-Thue system is Noetherian (terminating) if there is no infinite chain $x_0, x_1, \ldots$ such that for all $i \geq 0$, $x_i \rightarrow x_{i+1}$.*

Assuming a Noetherian STS, the two properties in the next definition are equivalent:

Definition 8. *A semi-Thue system is **confluent** if for all $w, x, y \in \Sigma^*$, the existence of reductions $w \twoheadrightarrow x$ and $w \twoheadrightarrow y$ implies there exists a $z \in \Sigma^*$ such that $x \twoheadrightarrow z$ and $y \twoheadrightarrow z$. A semi-Thue system is **locally confluent** if for all $w, x, y \in \Sigma^*$, the existence of reductions $w \rightarrow x$ and $w \rightarrow y$ implies there exists a $z \in \Sigma^*$ such that $x \twoheadrightarrow z$ and $y \twoheadrightarrow z$.*

Theorem 5 (Unique Normal Form)**.** *If $S = (\Sigma^*, \rightarrow)$ is an STS that is Noetherian and confluent, then for every $x \in \Sigma^*$, $[x]$ has a unique normal form.*

Proof: Described in Theorem 1.1.12 [8], pp. 13. ∎

Thus, a key result from rewriting theory is that for a rewriting system that is confluent and Noetherian, any object can be reduced to a unique normal form by applying rewrite rules arbitrarily until the object is irreducible. Furthermore, two objects u and v are equivalent under $\overset{*}{\leftrightarrow}$ iff their unique normal forms are the same. In this thesis, the notation $N(u)$ denotes the unique normal form for any string u.

To show that a unique normal form exists, all that is needed is a proof that the string rewrite system is Noetherian and locally confluent. Proving noetherianess is also undecidable in general. However, by defining a strict partial ordering function (e.g., string length) which all rewrite rules obey, it can be established that a string rewrite system is terminating. Now, if the system is terminating, then it is possible to test for local confluence. Consequently, many algorithms for testing an STS for local confluence have been developed (e.g. computing critical pairs, Knuth-Bendix Completion). Put

simply, these algorithms rely on testing whether critical pairs (the results of applying two rules whose left-hand side overlap) have a common descendant — a necessary and sufficient condition for local confluence (see Theorem 6.2.4 in [6]). For a detailed treatment of these algorithms, I refer the reader to Chapters 1, 2 and 3 in [8], Chapters 1, 5, 6 and 7 in [6], and Chapters 1 and 2 in [27].

4.3 Trace Computation Modulo Confluence

4.3.1 Formalizing the Intuition

The fundamental principle underlying both BackSpace and TAB-BackSpace approaches is to use repetition to compensate for the lack of on-chip observability. However, as noted in Section 4.1, non-determinism can make the execution of "the same" trace very unlikely.

Indeed, the experiments in Section 3.4 show that the problem in practice is not too many matches generating spurious traces, but the lack of exact matches preventing any progress in trace computation. Empirically, however, I have often observed intuitively "equivalent" traces that are not cycle-by-cycle matches, e.g., a trace with slightly different timing, with independent events reordered, etc. Consider, for example, sample traces from two different simulation runs of the Router after they have breakpointed (timing-diagram shown in Fig. 4.2). Let a_i be the abstract state defined by the following

$$STATE \times \mathbb{Z}^+ \times \mathbb{Z}^+ \times \mathbb{Z}^+ \tag{4.1}$$

where $STATE = \{idle,\ wait\_buff,\ wait\_data,\ get\_dest,\ get\_rest,\ wait\_route,\ wait\_idle,\ pkt\_end,\ abort\}$, $\mathbb{Z}^+$ the set of positive integers[19]. Both traces breakpoint at the same "crash state", but, at first glance, these traces are very different. For example, consider the marking "S0". This marking represents the valuation $(idle, 2, 59, 1B)$ in the top diagram and in the bottom diagram. Notice the number of cycles that the Router stays in "S0" is very

[19]The 2^{nd}, 3^{rd} and 4^{th} numbers in this mapping are actually bounded, positive integers representing different FIFO pointers.

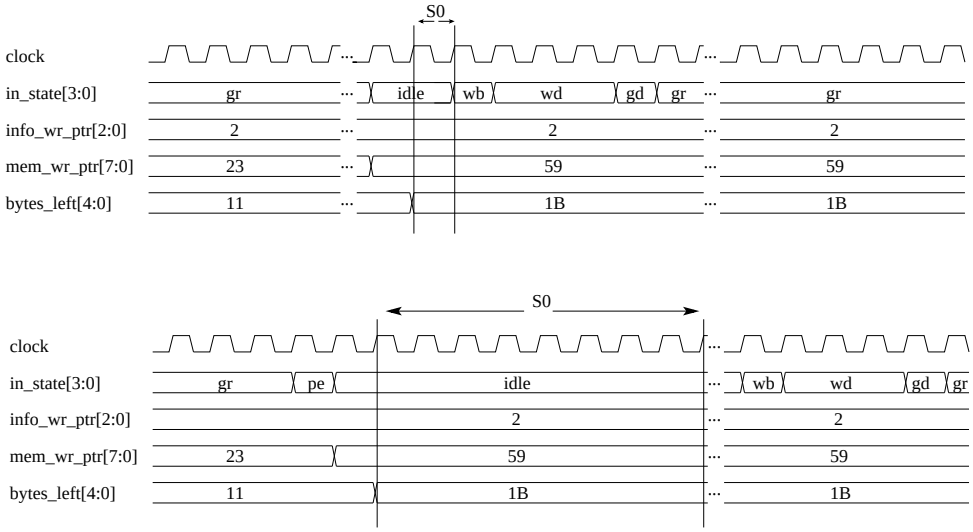

Figure 4.2: Two trace-buffers of length 50 from different runs that breakpointed at the same abstract state. The aliases "gr", "pe", "wb", "wd" and"gd" stands for *get_rest*, *pkt_end*, *wait_buffer*, *wait_data* and *get_dest*, respectively.

different from one run to the next. Nevertheless, it is possible these traces share some notion of equivalence. In fact, after further inspection and with the help of the automata representing these two traces, depicted in Fig. 4.3, notice that there are only 8 distinct states. Thus, the differences in these two runs are only the timings in each state.

These are all differences that could be manipulated via rewriting. But, because it is not obvious, *a priori*, what is the correct notion of equivalence, I propose to allow the debug engineer to specify rewrite rules to define what "equivalent" means to them, on a particular design. nuTAB-Backspace will then match overlap regions if they are equivalent under the specified rewriting, rather than requiring an exact match.

Will this idea produce correct traces? Correctness depends on the rewrite rules respecting the semantics of the design. Accordingly, nuTAB-BackSpace imposes a few restrictions on the rewrite rules. Not surprisingly, it requires that the rules be Noetherian and confluent, which allows efficient equivalence

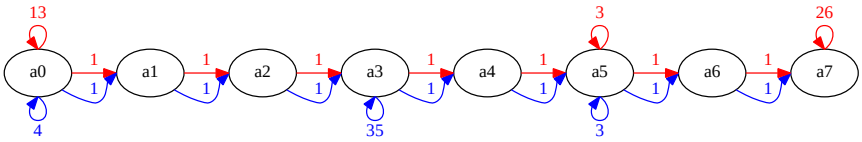

Figure 4.3: This automata represents the state-machine from the timing diagrams of Fig 4.2. The edges on top match the delays of the top timing diagram. Similarly, the edges on the bottom match the delays of the bottom diagram.

checking via reduction to the unique normal form. The following definition captures the notion that the rewrite rules truly reflect equivalent traces of the underlying concrete chip:

Definition 9. *Consider a rewrite rule $l \rightarrow r$ on strings of abstract states. The rewrite rule is* ***concretization preserving*** *if for all concrete states x_c and z_c, the concretizability of the abstract state sequence $\alpha(x_c)l\alpha(z_c)$ to a concrete sequence starting with x_c and ending with z_c implies the concretizability of the abstract state sequence $\alpha(x_c)r\alpha(z_c)$ to a concrete sequence starting with x_c and ending with z_c, i.e.:*

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

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

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

also satisfies the consequent.

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

4.3.2 Algorithm

Algorithm 4 presents the nuTAB-BackSpace procedure, which is exactly the same as Algorithm 3 (page 58) except for the *OverlapAndCheck()* subroutine, here replaced with *NormalizeAndCheck()*. Because Algorithm 4 works with normalization instead of overlapping regions, the correctness argument is also different. To make this exposition self-contained, I explain Algorithm 4: starting from a given crash state and its corresponding trace-buffer, it iteratively computes an arbitrarily long sequence of predecessor abstract states by going backwards in time. This procedure has 4 user-specified parameters: *steps_bound* specifies how many iterations back the algorithm should go; *retries_timeout* limits the amount of search for an equivalent overlapping region between the new trace dump and the trace computed so far; the *time_bound* is a timeout for each chip-run and is a mechanism to tell whether a chip-run went on a path that does not reproduce the crash-state or buggy-state; and, *lbindex* is the trace buffer's smallest index, which defines a region either for the overlapping (TAB-BackSpace) or the normalization (nuTAB-BackSpace) of two consecutive trace buffers.

The procedure has 2 nested loops. The outer loop, lines 13 – 41, controls the three termination conditions for the algorithm: we reach the user-specified number of iterations; we reach the initial states; or the previous iteration was unsuccessful. The outer loop is also responsible for joining the new trace buffer dump onto the successful trace computed so far (line 32), and then selecting a new state as the breakpoint for the next iteration. The inner loop, lines (17 – 31), is responsible for controlling the hardware while trying out different candidate-states, s_{cand}, given a *retries_timeout*. The procedure keeps track of time using the subroutine *ElapsedTime*() (passing

reset as parameter resets the time counter, otherwise it counts the elapsed time since it was last reset). In each loop iteration, the procedure loads s_{cand} into the breakpoint-circuit (line 19), and runs the chip. The objective is to collect a new trace-buffer upon matching s_{cand} and match (after rewriting) it with the previous trace-buffer. If *ResetAndRun*() returns $TRUE$ then the breakpoint circuitry matched s_{cand} and we have a new trace-buffer. Otherwise, the chip-run violates the *time_bound* parameter (line 21) because the current run took another path (caused by non-determinism). If the breakpoint occurs, we dump the contents of the trace-buffer, for comparison with the trace computed so far. The *NormalizeAndCheck()* subroutine (line 24) computes the unique normal form of the overlapping region of the previously computed trace as well as the new trace dump, as described in Sec. 4.2 and then compares them to check equivalence. If the procedure neither breakpoints nor proves equivalence, *PickState*() (line 29) selects another candidate-state from the previous trace using a round-robin scheme while respecting *lbindex*, and then the inner loop iterates. The procedure exits the inner loop when either it successfully proves equivalence of the overlapping regions of the two trace-buffers, or this loop has iterated longer than the specified *retries_timeout*.

4.3.3 Correctness

The main correctness theorem proves that the trace computed by Algorithm 4 is as informative as one could hope: it concretizes to a trace that leads to the actual crash state, using reachable states.

Theorem 6 (Correctness of Trace Computation)**.** *If the rewriting rules are Noetherian, confluent, and concretization preserving, and if the size of all unique normal forms used to prove equivalence of overlapping regions is greater than l_{div}, then the trace produced by Algorithm 4 is concretizable to the suffix of a concrete trace leading from the initial states Q_0 to the crash state s.*

Proof: The proof is by induction on the iteration count i at the bottom of the outer loop. The base case is trivial, as when $i = 0$, the trace is a single

Algorithm 4 Crash State History Computation

1: **input** Q_0 : set of initial states,
2: (s, t) : crash-state and trace-buffer
3: $steps\_bound \in \mathbb{N}^+$: user-specified bound on the number of iterations,
4: $retries\_timeout \in \mathbb{N}^+$: user-specified time-bound on retrials,
5: $time_{bound}$: user-specified time bound for any chip-run
6: $lbindex$: user-specified lower-bound length of normal region;
7: **output** $trace$: equivalent sequence of abstract states;
8: // init. breakpointable candidate-state, current trace-buffer and final trace
9: $i := 0$; $s_{cand} := s$; $t_i := t$; $trace := (t_i)$;
10: // initialize variable nindex; nindex gets updated by $PickState()$
11: // nindex range is $[lbindex, |trace\text{-}buffer|]$
12: $nindex := lbindex$; $succ\_iteration := FALSE$;
13: **while** $(i < steps\_bound)$ && $(s_{cand} \notin Q_0)$ && $(succ\_iteration = TRUE)$ **do**
14: $equivalent := FALSE$; $matched := FALSE$;
15: //Resets retrial elapsed time
16: $ElapsedTime(reset)$
17: **while** (!$equivalent$) && $(ElapsedTime(go) \leq retries\_timeout)$ **do**
18: // Program the hardware-breakpoint circuitry with s_{cand}
19: $LoadHardwareBreakpoint(s_{cand})$;
20: // (Re-)run the chip at full-speed with timeout $time_{bound}$
21: $matched := ResetAndRun(time_{bound})$;
22: **if** $matched$ **then**
23: $t_i := ScanOut()$;// Dump trace-buffer contents t_i
24: $equivalent := NormalizeAndCheck(t_i, t_{i-1}, nindex)$;
25: **end if**
26: **if** (!$matched$) || (!$equivalent$) **then**
27: // Pick another state following a round-robin scheme
28: // and updates $nindex$
29: $s_{cand} := PickState(nindex, t_{i-i})$;
30: **end if**
31: **end while**
32: **if** $equivalent = TRUE$ **then**
33: // Accumulate trace
34: $OverlapConcatenate(t_i, trace)$;
35: // Pick a candidate-state in t_i for the next iteration
36: $s_{cand} := PickState(nindex, t_i)$;
37: $i := i + 1$;
38: **else**
39: $succ\_iteration := FALSE$
40: **end if**
41: **end while**
42: **return** $trace$;

trace buffer dump that ends at the crash state. Since this trace dump is taken from the physical chip, it can be concretized to the specific physical execution that occurred on-chip.

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

We know that xv is concretizable to a trace with all states reachable from the initial states, because it is taken directly from the hardware. Therefore, $xN(v)$ has the same properties, by preservation of concretization. Similarly, uy is concretizable to a trace that leads to the crash state s, by the inductive hypothesis, and therefore, $N(u)y$ is, too, by preservation of concretization. Let $x_c v_c$ be a witness to the concretizability (with additional properties) of $xN(v)$, with $x = \alpha(x_c)$ and $N(v) = \alpha(v_c)$. Similarly, let $u_c y_c$ be a witness to the concretizability of $N(u)y$, with $N(u) = \alpha(u_c)$ and $y = \alpha(y_c)$.

From the hypotheses, $|N(u)| = |N(v)| > l_{div}$, so by the definition of l_{div}, both $x_c u_c y_c$ and $x_c v_c y_c$ are legal concrete traces. By construction, both start at reachable states, and therefore contain all reachable states. And both end at the crash state s. Therefore, either is a witness that the new trace computed by Algorithm 4, $xN(u)y$, is concretizable to the suffix of a concrete trace leading from the initial states to the crash state. ∎

4.4 Experimental Results

In this section, I demonstrate the feasibility of nuTAB-BackSpace with experiments on a simulation environment and a hardware prototype. First, I use a simulation-based evaluation since it offers full visibility of the design, and therefore it is possible to identify false matches. Then, I evaluate nuTAB-BackSpace on actual hardware. In both experiments, I compare nuTAB-BackSpace against TAB-BackSpace.

4.4.1 Results on Simulation

Setup:

Similarly to Section 3.4.1, I chose to use the same IBM Router design. This is a non-trivial design (9958 latches), but not too complex to be simulated in its entirety. Also, I used the same simulation environment as before (Cadence's Incisive Simulator with Specman Elite v.09.20-s016).

Experiment:

In Section 3.4.1, I have shown that TAB-BackSpace works successfully when non-determinism in the environment/design has been extensively reduced. My claim is that when non-determinism cannot be extensively removed from the environment/design, nuTAB-BackSpace will either succeed when TAB-BackSpace fails or it will require far less effort to compute a trace than TAB-BackSpace (explained later in this section). To validate this claim, I use the same configuration as in Section 3.4.1, that is:

1. number of iterations is set to 20 (*steps_bound*=20 in Algorithm 4, page 81);
2. timeout is set to 5 hours (*retries_timeout*=5 in Algorithm 4);
3. the TAB length is set to 50 with no compression;
4. the TAB width is set to 75 bits with the same signals chosen out of 3 design blocks;
5. the same 30 randomly chosen crash states, a_i, satisfying these two properties: $\forall i.a_i \notin Q_0$ and $\forall i, j.(a_i \neq a_j) \wedge |(i - j)| > 1000$;
6. randomized inter-packet delays.

As item 6 suggests, non-determinism only affects the inter-packet delays as in the experiments in Section 3.4.1. However, unlike those experiments, I chose a wider range of values for the inter-packet delays, thus simulating the case in which non-determinism has not been extensively reduced from the design/environment.

Section 3.4.1 empirically shows that, for the Router, an overlap of 30 cycles or more would most likely prevent false matches. Thus, in these experiments, I use 30 cycles as the lower bound for either the overlapping region (TAB-BackSpace) or the normalization region (nuTAB-BackSpace) of two consecutive trace-buffers.

To be able to normalize the non-determinism during nuTAB-BackSpace simulations, I need to provide a set of rewrite rules. In practice, defining such rewrite rules would follow the same iterative process as debugging. In this case, however, I had worked with this design in Chapter 3 and had a good understanding of it.

A subset of the 75 signals being probed describes three identical, but independent state-machines from three of the Router's design-blocks. Fig. 4.4 presents one such machine. Notice that 6 states have self-loops, namely *idle, wait_buff, wait_data, wait_idle, wait_route, get_rest*. Non-determinism in the inter-packet delays affects all these states with self-loop edges (e.g., a long delay might cause an input port to remain in *idle* or *wait_data* states for some number of cycles). The exception is the state *get_rest*. In this state, the Router processes incoming packets without interruption, that is, the Router does not accept partial packets. Thus, to normalize non-determinism, I define a rewrite system, $Router_{RS}(\Sigma^*, R)$:

> Let $\text{Proj}(\cdot)_{sm}$ be a projection function that takes in an abstract-state, a, and projects it onto the set of bits representing the state machine from Fig. 4.4 and let $P = \{idle, wait\_buff, wait\_data, wait\_route, wait\_idle\}$. Now, define R as follows:

$$\forall a.Proj(a)_{sm} \in P \;.\; aa \rightarrow a \tag{4.2}$$

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

How can it be shown that the rewrite system, $Router_{RS}(\Sigma^*, R)$, is Noetherian, confluent, and concretization-preserving? Let's consider each in turn. First, note that $Router_{RS}(\Sigma^*, R)$ is a length-reducing rewrite system, and so it is Noetherian. Next, note that Eq. 4.2 contains 5 rules (or technically,

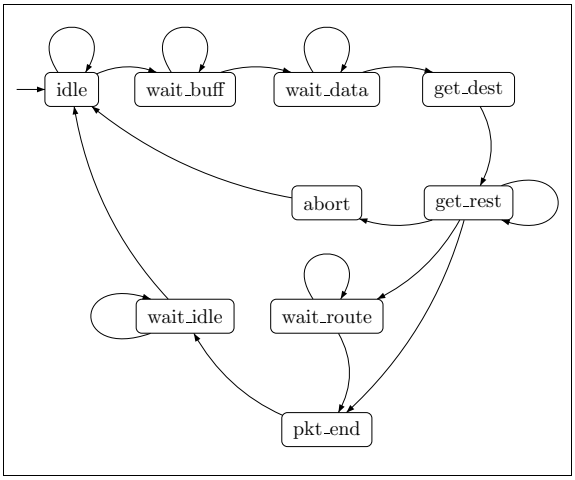

Figure 4.4: Router's Internal Packet-Processing State-Machine.

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

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

The results are as expected. Table 4.2 shows that nuTAB-BackSpace

CS	# of Succ. Iterations		# of Chip Runs	
	TAB	nuTAB	TAB	nuTAB
1	0	11	62	143
†2	0	20	339	21
3	0	20	67	52
4	0	20	77	75
5	0	20	79	24
6	0	20	93	27
7	0	20	283	28
†8	0	20	128	20
†9	0	20	58	23
10	0	20	342	20
11	1	20	173	57
12	2	7	204	137
13	3	15	399	536
14	3	20	134	144
15	4	19	270	661
†16	4	20	112	27
17	5	20	157	28
18	5	20	199	41
19	6	6	120	58
†20	6	20	60	26
21	6	20	181	24
22	6	20	788	24
†23	7	20	568	22
24	12	20	251	27
25	12	20	308	25
26	15	20	534	20
27	20	20	282	28
28	20	20	403	29
29	20	20	463	52
30	20	20	726	26

Table 4.2: TAB-BackSpace vs nuTAB-BackSpace Experiments. "CS"is the index of each crash state. The same crash states are used for each TAB- and nuTAB- experiment. "# of Successful Iterations" is the number of iterations before timing out or reaching the set limit of 20. The timeout per iteration was chosen to be 5 hours. Each simulation run averages 10 minutes. "# of Chip Runs" is the total number of iterations plus the number of retries. Because "# of Chip Runs" is an aggregate, when the number of iterations for TAB is smaller than the number of iterations for nuTAB, the number of nuTAB runs may be greater than TAB runs (e.g., crash states 1, 13-15). Crash states with a † are states that nuTAB-BackSpace computed all 20 iterations, but somewhere during the computation it deviated from the "expected" trace (in simulation, we can determine if the run reached the specified "crash" state). Therefore, these might be spurious traces. I suspect that, at some iteration, the normalized region of two different traces was too small to discriminate them.

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

4.4.2 Results on a Hardware Prototype

To demonstrate that nuTAB-BackSpace is feasible in practice, I conducted experiments with a hardware prototype running on an FPGA emulation board [54]. The hardware prototype is a Leon3-based [22] SoC. This prototype is a full-blown SoC featuring a SPARC V8 compatible core, AMBA bus, video, DDR2, Ethernet, i2c, keyboard and mouse controllers (depicted in Fig. 4.5). The SoC also has some built-in debug features that can be enabled. In particular, I enable the provided on-chip logic-analyzer, LOGAN, but with minimal configuration. Note that, in contrast to the CELL debug logic, LOGAN has no signal compression. Using this debug logic in the SoC, nuTAB-BackSpace iterated several times, more than doubling the initial trace-buffer contents, more importantly, it shows that the same could not be done with TAB-BackSpace. The signals monitored are a combination of AMBA bus signals and some signals of the SPARC V8's execution-pipeline-stage, totaling 134 signals.

One of the goals of demonstrating nuTAB-BackSpace on a hardware-prototype is to show that it works in a real (or as realistic as possible) debugging environment.Thus, in these experiments, the SoC is booting Linux (Linux Kernel 2.6.21).

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

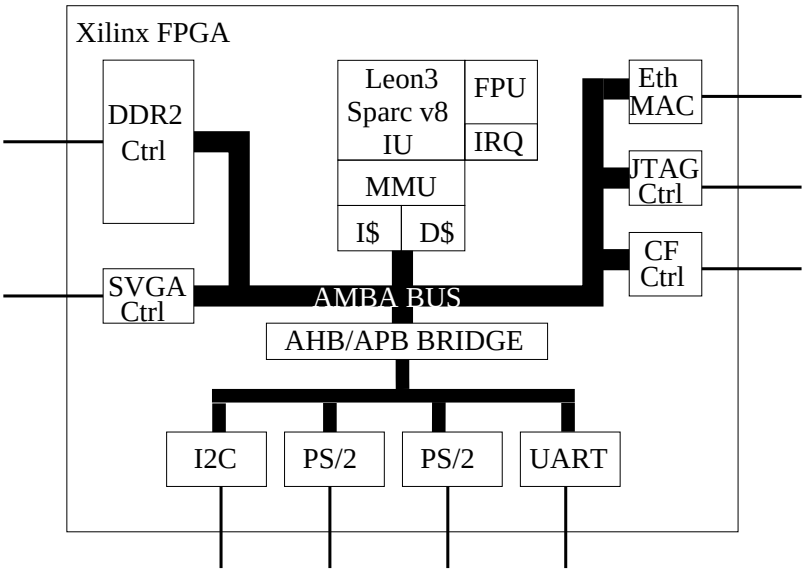

Figure 4.5: Leon3 SoC Block Diagram.

The first experiment is to try to TAB-BackSpace. I follow the same steps as Algorithm 4. The main difference is that instead of normalizing the extracted trace, I try to find an exact match on the overlap region between the current and previous traces. I set an address within *start_kernel* function as the breakpoint and run the chip; when it breakpoints, the tool dumps the contents of the trace-buffer into a file. From that trace-buffer, I pick a trace-buffer entry as the new crash-state and repeat. In these experiments, I set 2 hours as the retry timeout limit. The result of these experiments is a total of 207 chip runs, all of which breakpoint successfully, but none overlap cycle-by-cycle. In other words, the SoC cannot TAB-BackSpaced at all because, at each run, non-determinism changes the path the chip takes and so the probability of an exact match is too low.

The next experiment is to try nuTAB-BackSpace using the same scenario as before. However, I need to define the rewrite rules first. In this case, the SoC was built entirely from third-party IP, so my learning process was from the documentation and trace buffer dumps from the actual system

running. Studying the trace buffer dumps, I observed that sometimes entire trace-buffers might not have a single video-controller transaction. Also, I noted that nullified instructions, although they vary from run to run, do not affect overall functionality of a system run. Therefore, for this debug scenario, my hypotheses are that traces may have video-controller activity occurring at essentially arbitrary times, and that nullified instructions can be ignored. From this understanding of the design, I can create rewrite-rules easily to formalize the hypotheses and test them. (If these hypotheses produced uninteresting traces, I would start again with a new hypothesis, creating new rewrite rules to try.)

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

$$\forall a.Proj(a)_{ahbm} = 0x3 \;.\; a \rightarrow \epsilon \tag{4.3}$$

$$\forall a.Proj(a)_{inst} = annul \;.\; a \rightarrow \epsilon \tag{4.4}$$

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

As in Section 4.4.1, I need to show that $Leon3_{RS}(\Sigma^*, R)$ is Noetherian, confluent, and concretization-preserving. As before, the system is length-reducing, and hence Noetherian. No two rules have an overlapping left-hand side. Consequently, there are no critical-pairs, so $Leon3_{RS}(\Sigma^*, R)$ is locally confluent. The argument for concretization preservation is again based on insight into the design. The video controller bus transactions are irrelevant to the boot sequence and can be arbitrarily ignored.[20] Similarly, nullified

[20]Technically, ignoring video controller transactions is not truly concretization preserv-

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

Table 4.3: nuTAB-BackSpace on Leon3. *Trace-Buffer Length* is the physical depth of the trace-buffer. Since I do not use compression, its depth is fixed. *Normalization-Region Length* is the number of cycles in the current trace-buffer that nuTAB-BackSpace normalizes and use as a reference for the next trace-buffer. *New Cycles* is the number of new states present in the current trace-buffer.

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

Table 4.3 shows the results. nuTAB-BackSpace iterated 7 times, resulting in a trace more than 2.5x the length of a single trace-buffer. Unlike TAB-BackSpace, the new technique handles the non-determinism, computing an abstract trace based on the trace-buffer signals.

ing, since any real concrete trace *will* have the occasional video transaction, whose timing is determined by state hidden in the video controller and the external video hardware. What the rewrite rule is really specifying is that that hidden state is irrelevant for the current debugging scenario. If we were debugging some video controller timing interaction, we would use different rewrite rules.

Chapter 5

Conclusion and Future Work

It is better to be wrong than to be vague.

Freeman Dyson

5.1 Conclusions

In this thesis, I have presented BackSpace, a novel post-silicon debug framework. From theory to practice, I have methodically developed this framework showing that BackSpace effectively computes accurate traces leading up to a crash state, has low cost (*zero*-additional hardware overhead), and handles non-determinism. I have successfully demonstrated BackSpace with several industrial designs using simulation models, hardware prototypes, and on actual silicon.

Because of BackSpace's success, I believe this framework holds promise for more complex designs. In particular, designs with multiple clocks and even designs with globally asynchronous, locally synchronous clocks (GALS) are suitable areas for future research since nuTAB-BackSpace provides the formalism necessary to handle non-determinism.

5.2 Future Work

5.2.1 Backspacing Multi-Clock Designs

Most of today's designs have multiple clock domains. In my experiments, I either assumed single-clock designs or confined debugging to a single-clock domain (e.g., we restricted debugging to one core of the IBM POWER7,

while turning off the other parts of the design). The next step is to consider the impact of multiple clocks on BackSpace.

One of the fundamental problems of debugging a design with multiple clocks is that it is not clear how to correlate traces from the different clock domains (assuming a design with debug logic such as trace-buffers). In the case of nuTAB-BackSpace, debug engineers could analyze signals crossing different clock-domain and define rewrite-rules to normalize traces with different delays (due to clock-domain crossing). The key is that nuTAB-BackSpace provides the formalism (*rewriting* systems) for such debugging. The problematic debug scenario, however, is when the bug is actually in the clock-crossing logic (e.g., a rewrite-rule that unintentionally "hides" the bug). Clearly, then, the rewrite-rules should be defined carefully. Nevertheless, I believe that nuTAB-BackSpace already offers a solid foundation for debugging multi-clock designs.

5.2.2 Protocol-Based BackSpace

In the previous section, I presented some of the problems of multi-clock designs. However, the debugging problem is exacerbated when designs, like some SoCs today, have a GALS architecture. For example, a multi-clock breakpoint may not be feasible in a design with independent clocks. Thus, I propose to investigate a technique to abstract the underlying GALS architecture and to BackSpace a high-level model. This idea is analogous to protocol-level modeling (or transaction-level modeling), which abstracts the low-level implementation details.

Intuitively, the main idea is that, if carefully crafted, the protocol-level model (PLM) would guide the debug-engineer to the source of a bug. Typically, PLM states are much smaller than chip states, and, thus, they could be captured with trace-buffers. In this way, if a bug is observed in the PLM, then, using a technique similar to nuTAB-Backspace, I could BackSpace the PLM. Two scenarios are possible. First, if the bug is fully captured in the PLM I am done. Second, if the bug is not fully captured in the PLM, then I would need to concretize a "suspect" bad PLM state and check if the bug ex-

ists in the actual (concrete) model. Fortunately, nuTAB-BackSpace always computes a concretizable trace (assuming an overlapping region greater than $f > l_{div}$).

Bibliography

[1] Miron Abramovici, Paul Bradley, Kumar Dwarakanath, Peter Levin, Gerard Memmi, and Dave Miller. A Reconfigurable Design-for-Debug Infrastructure for SoCs. In *DAC '06: Proceedings of the 43rd Annual Design Automation Conference*, pages 7–12, New York, NY, USA, 2006. ACM.

[2] A. Adir, M. Golubev, S. Landa, A. Nahir, G. Shurek, V. Sokhin, and A. Ziv. Threadmill: A Post-Silicon Exerciser for Multi-Threaded Processors. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 860 –865, june 2011.

[3] Vishiwani D. Agrawal, Kwang-Ting Cheng, Daniel D. Johnson, and Tony Sheng Lin. Designing Circuits with Partial Scan. *IEEE Design and Test*, 5(2):8–15, 1988.

[4] Catherine Ahlschlager and David Wilkins. Using Magellan to Diagnose Post-Silicon Bugs. *Synopsys Verification Avenue Technical Bulletin*, 4(3):1–5, September 2004.

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

[6] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998.

[7] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction*

and Analysis of Systems, TACAS '99, pages 193–207, London, UK, 1999. Springer-Verlag.

[8] Ronald V. Book and Friedrich Otto. *String-Rewriting Systems*. Texts and Monographs in Computer Science. Springer, 1993.

[9] Marc Boule and Zeljko Zilic. Incorporating Efficient Assertion Checkers into Hardware Emulation. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 221–228, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Adrian Carbine and Derek Feltham. Pentium Pro Processor Design for Test and Debug. *IEEE Design and Test*, 15(3):77–82, 1998.

[11] J. Lawrence Carter and Mark N. Wegman. Universal Classes of Hash Functions (Extended Abstract). In *STOC '77: Proceedings of the Ninth Annual ACM Symposium on Theory of computing*, pages 106–112, New York, NY, USA, 1977. ACM.

[12] O. Caty, P. Dahlgren, and I. Bayraktaroglu. Microprocessor Silicon Debug Based on Failure Propagation Tracing. In *International Test Conference*, pages 293–302. IEEE International, Nov. 2005.

[13] Kai-Hui Chang, Igor L. Markov, and Valeria Bertacco. Automating Post-Silicon Debugging and Repair. In *International Conference on Computer-Aided Design*, pages 91–98. IEEE/ACM, 2007.

[14] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 8:244–263, April 1986.

[15] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *J. ACM*, 50:752–794, September 2003.

[16] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. In *Symposium on Principles of Programming Languages*, pages 343–354. ACM, 1992.

[17] P. Dahlgren, P. Dickinson, and I. Parulkar. Latch Divergency in Microprocessor Failure Analysis. In *International Test Conference*, pages 755–763. IEEE International, 2003.

[18] Satyaki Das and David L. Dill. Successive Approximation of Abstract Transition Relations. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 51–, Washington, DC, USA, 2001. IEEE Computer Society.

[19] The International Roadmap for Semiconductors. Design. *2009 Edition*, page 7. Downloaded on February, 2009 (http://www.itrs.net/reports.html).

[20] The International Roadmap for Semiconductors. Overall Technology Roadmap Characters - Tables. *2009 Edition*. Downloaded on February, 2009 (http://www.itrs.net/reports.html).

[21] Harry Foster. Assertion-Based Verification: Industry Myths to Realities (Invited Tutorial). In *CAV '08: Proceedings of the 20th International Conference on Computer Aided Verification*, pages 5–10, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] Gaisler. http://www.gaisler.com.

[23] M. Gort, F. M. De Paula, J. J. W. Kuan, T. M. Aamodt, A. J. Hu, S. J. E. Wilton, and J. Yang. Formal-Analysis-Based Trace Computation for Post-Silicon Debug. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, PP(99):1 –14, 2012.

[24] Marcel Gort. Practical Considerations for Post-Silicon Debug using BackSpace. Master's thesis, Electrical and Computer Engineering Department - University of British Columbia, 2009.

[25] Shankar G. Govindaraju and David L. Dill. Counterexample-Guided Choice of Projections in Approximate Symbolic Model Checking. In *Proceedings of the 2000 IEEE/ACM International Conference on*

Computer-Aided Design, ICCAD '00, pages 115–119, Piscataway, NJ, USA, 2000. IEEE Press.

[26] Alan J. Hu, Jeremy Casas, and Jin Yang. Efficient Generation of Monitor Circuits for GSTE Assertion Graphs. In *International Conference on Computer-Aided Design*, pages 154–159. IEEE/ACM, 2003.

[27] M. Jantzen, editor. *Confluent String Rewriting*. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[28] Don Douglas Josephson. The Manic Depression of Microprocessor Debug. In *International Test Conference*, pages 657–663, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[29] Don Douglas. Josephson, Steve Poehhnan, and Vincent Govan. Debug Methodology for the McKinley Processor. In *International Test Conference*, pages 451 –460. IEEE International, 2001.

[30] R. Kalla, B. Sinharoy, W.J. Starke, and M. Floyd. Power7: IBM's Next-Generation Server Processor. *Micro, IEEE*, 30(2):7 –15, March-April 2010.

[31] Ho Fai Ko and N. Nicolici. Algorithms for State Restoration and Trace-Signal Selection for Data Acquisition in Silicon Debug. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(2):285 –297, Feb. 2009.

[32] Ravishankar Kuppuswamy, Peter DesRosier, Derek Feltham, Rehan Sheikh, and Paul Thadikaran. Full Hold-Scan Systems in Microprocessors: Cost/Benefit Analysis. *Intel Technology Journal*, 8(1):63–72, February 2004.

[33] Mario Larouche. *Infusing Speed and Visibility into ASIC Verification*. Synopsys's Synplicity Business Group. Downloaded on Jan, 2007. www.synplicity.com/literature/whitepapers/pdf/totalrecall_wp_1206.pdf.

[34] D.H. Lee and S.M. Reddy. On Determining Scan Flip-Flops in Partial-Scan Designs. In *IEEE International Computer-Aided Design. Digest of Technical Papers*, pages 322–325. IEEE International, Nov 1990.

[35] Xiao Liu and Qiang Xu. Trace Signal Selection for Visibility Enhancement in Post-Silicon Validation. In *Design, Automation Test in Europe Conference Exhibition*, pages 1338 –1343. IEEE Computer Society, April 2009.

[36] Subhasish Mitra and Kee Sup Kim. X-Compact: An Efficient Response Compaction Technique for Test Cost Reduction. In *International Test Conference*, pages 311–320. IEEE, 2002.

[37] José Augusto M. Nacif, Flavio M. de Paula, Claudionor N. Coelho, Jr., Fernando C. Sica, Harry Foster, Antônio O. Fernandes, and Diógenes C. da Silva. The Chip is Ready, Am I done? On-Chip Verification using Assertion Processors. In *International Conference on Very Large Scale Integration of System-on-Chip (VLSI-SoC)*, pages 111–116. IFIP WG 10.5, 2003.

[38] OpenCores. http://www.opencores.org.

[39] Accellera Standards Organization. http://www.accellera.org/activities/committes/ovl.

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

[41] Sung-Boem Park and Subhasish Mitra. IFRA: Instruction Footprint Recording and Analysis for Post-Silicon Bug Localization in Processors. In *45th Design Automation Conference*, pages 373–378. ACM/IEEE, 2008.

[42] Sung-Boem Park and Subhasish Mitra. Post-silicon Bug Localization for Processors using IFRA. *ACM Communications*, 53(2):106–113, 2010.

[43] S. Prabhakar and M. Hsiao. Using Non-trivial Logic Implications for Trace Buffer-Based Silicon Debug. In *Asian Test Symposium*, pages 131 –136. IEEE Computer Society, Nov. 2009.

[44] IEEE P1850 PSL. http://www.eda.org/ieee-1850/.

[45] B. R. Quinton and S .J. E. Wilton. Concentrator Access Networks for Programmable Logic Cores on SoCs. In *IEEE International Symposium on Circuits and Systems*, pages 45–48, 2005.

[46] B .R. Quinton and S. J. E. Wilton. Programmable Logic Core Based Post-Silicon Debug For SoCs. In *4th IEEE Silicon Debug and Diagnosis Workshop*, Germany, May 2007.

[47] Sandip Ray and Warren A. Hunt Jr. Connecting Pre-Silicon and Post-Silicon Verification. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 160–163, 2009.

[48] Collett International Research. IC/ASIC Functional Verification Study. *Industry Report*, page 34, 2004.

[49] Mack Riley, Nathan Chelstrom, Mike Genden, and Shoji Sawamura. Debug of the CELL Processor: Moving the Lab into Silicon. In *International Test Conference*, pages 1–9. IEEE International, Oct. 2006.

[50] Sean Safarpour, Hratch Mangassarian, Andreas Veneris, Mark H. Liffiton, and Karem A. Sakallah. Improved Design Debugging Using Maximum Satisfiability. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 13–19. IEEE, 2007.

[51] Carl-Johan H. Seger and Randal E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. volume 6, pages 147–189, Hingham, MA, USA, 1995. Kluwer Academic Publishers.

[52] Ronak Singhal, K. S. Venkatraman, Evan R. Cohn, John G. Holm, David A. Koufaty, Meng-Jang Lin, Mahesh J. Madhav, Markus

Mattwandel, Nidhi Nidhi, Johathan D. Pearce, and Madhusudana Seshadri. Performance Analysis and Validation of the Intel Pentium 4 Processor on 90nm Technology. *Intel Technology Journal*, 8(1):34–42, February 2004.

[53] Michael J. Y. Williams and James B. Angell. Enhancing Testability of Large-Scale Integrated Circuits via Test Points and Additional Logic. *IEEE Transactions on Computers*, C-22(1):46–60, January 1973.

[54] Xilinx. http://www.xilinx.com.

[55] Y. Yang, N. Nicolici, and A. Veneris. Automated Data Analysis Solutions to Silicon Debug. In *Design, Automation Test in Europe Conference Exhibition*, pages 982–987. IEEE Computer Society, April 2009.