Improving the Performance of Post-Silicon Trace Generation

by

Johnny Jone Wai Kuan

B.A.Sc., University of British Columbia, 2007

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

Master of Applied Science

in

THE FACULTY OF GRADUATE STUDIES
(Electrical and Computer Engineering)

The University Of British Columbia
(Vancouver)

August 2011

© Johnny Jone Wai Kuan, 2011
Abstract

As microprocessor designs become more complex, the task of finding errors in the design becomes more difficult. Most design errors will be caught before the chip is fabricated, however, there may be some that make it into the fabricated design. The process of determining what is wrong when the fabricated chip of a new design behaves incorrectly is called post-silicon debug (also known as silicon validation). One of the challenges of post-silicon debug is the lack of observability into the state of a fabricated chip. BackSpace [13, 14] is a proposal for tackling the observability challenge. It does this by generating a post-silicon trace using a combination of on-chip monitoring hardware and off-chip formal analysis. To add one state to the trace, BackSpace generates a set of possible predecessor states by analysing the design which are then tested one at a time. The testing is performed by loading a candidate state into a circuit that compares it with the current state of the chip, and running the chip. If the state is reached during execution, then it is added to the trace. The process of testing states one at a time is time consuming. This thesis shows that correlation information characterizing the application running on the chip can reduce the number of runs of the chip by up to 51%. New post-silicon trace generation algorithms, BackSpace-2bp and BackSpace-3bp, are also proposed that do not need to generate the set of possible predecessor states. This leads to a speedup relative to practical implementations of BackSpace and also reduces the hardware overhead needed to generate post-silicon traces by 94.4% relative to the state of the art implementation of BackSpace. To evaluate these new algorithms BackSpace, and the new algorithms, were implemented on a superscalar out-of-order processor using the Alpha instruction set. Non-determinism was introduced in the form of variable memory latency. The effects of non-determinism on the processor and the trace generation algorithms are also evaluated.
Preface

A version of Chapter 3 has been published. Johnny J.W. Kuan, Steven J.E. Wilton, and Tor M. Aamodt. Accelerating Trace Computation in Post-Silicon Debug. In The 11th International Symposium on Quality Electronic Design (ISQED), pages 244-249, March 2010. I designed the research program with guidance from my supervisor Dr. Tor Aamodt. I implemented the proposed algorithms and analyzed the experimental results. I prepared the manuscript with the assistance of Dr. Steve Wilton and Dr. Tor Aamodt. The idea for using correlation to sort the pre-image was first proposed in my EECE 527 course project [28].
Table of Contents

Abstract ................................................................. ii
Preface ................................................................. iii
Table of Contents ...................................................... iv
List of Tables ............................................................ viii
List of Figures ........................................................... ix
Glossary ................................................................. xii
Acknowledgments ......................................................... xiv

1 Introduction ......................................................... 1
  1.1 Background ......................................................... 2
      1.1.1 Product Definition .......................................... 2
      1.1.2 Architectural Exploration .................................. 3
      1.1.3 Architectural Refinement .................................. 4
      1.1.4 Circuit Design .............................................. 6
      1.1.5 Circuit Verification ........................................ 7
      1.1.6 Manufacture Testing ....................................... 8
      1.1.7 Post Silicon Debug ........................................ 8
  1.2 BackSpace ......................................................... 9
      1.2.1 BackSpace Algorithm ..................................... 9
4.1 Out-of-Order Alpha ISA Processor ........................................ 32
  4.1.1 BackSpace Implementation ......................................... 33
4.2 Non-deterministic Memory Access ................................. 34
  4.2.1 Implementation .................................................. 35
  4.2.2 Other Forms of Non-determinism ................................ 35
4.3 Effects of Non-determinism .......................................... 35
  4.3.1 Methodology ..................................................... 36
  4.3.2 Results .......................................................... 37
  4.3.3 Analysis .......................................................... 39

5 Performance Optimizations to BackSpace ............................ 42
  5.1 Performance Limitations of BackSpace .......................... 42
    5.1.1 Overwritten Registers ....................................... 42
    5.1.2 Non-determinism .............................................. 44
  5.2 Hardware Overhead Reduction ..................................... 45
    5.2.1 Check All States in Pre-image Simultaneously ............... 45
    5.2.2 Transition Check ............................................ 46
    5.2.3 Fixed Partial Breakpoint ................................... 49
    5.2.4 Optimized Signature ....................................... 49
    5.2.5 BackSpace-2bp ............................................. 50
  5.3 Reducing Processor Runs ........................................... 54
    5.3.1 Pipelining .................................................... 54
    5.3.2 BackSpace-3bp ............................................. 57
  5.4 Hardware .......................................................... 60
    5.4.1 Partial Breakpoint Circuit ................................ 60
    5.4.2 Counting Circuit ............................................ 61
    5.4.3 BackSpace-2bp Hardware ................................... 62
    5.4.4 BackSpace-3bp Hardware ................................... 63
List of Tables

Table 3.1  Start and end cycles for the programs run on the 8051. ......................... 25
Table 3.2  The relative number of runs required to BackSpace the different programs rela-
           tive to the baseline. .............................................................. 31
Table 4.1  IVM processor microarchitectural parameters. ..................................... 33
Table 4.2  Bits of storage in each component of the IVM processor. ....................... 39
Table 5.1  The number of state bits that can be overwritten. ............................... 44
Table 5.2  The storage overhead required for different post-silicon trace generation algo-
           rithms. ...................................................................................... 65
Table 5.3  Average number of runs needed to add one state to the trace. .............. 65
List of Figures

Figure 1.1  Flowchart showing the baseline BackSpace algorithm. ........................ 10
Figure 3.1  The BackSpace flow using sorted pre-images. .............................. 20
Figure 3.2  An example of merging the pre-image set \( P \). .............................. 22
Figure 3.3  Cumulative processor runs for the 8051 running \( \text{sqrt} \). .................. 26
Figure 3.4  Cumulative processor runs for the 8051 running \( \text{sort} \). ................... 27
Figure 3.5  Cumulative processor runs for the 8051 running \( \text{fib} \). ..................... 27
Figure 3.6  Average processor runs per correlation sample interval for the 8051 running \( \text{sqrt} \). ................................................................. 28
Figure 3.7  Average processor runs per correlation sample interval for the 8051 running \( \text{sort} \). ................................................................. 29
Figure 3.8  Average processor runs per correlation sample interval for the 8051 running \( \text{fib} \). 29
Figure 4.1  Simulation setup for running BackSpace on the IVM processor. .......... 34
Figure 4.2  An example where comparing two traces cycle by cycle will overestimate the difference between them. ................................. 36
Figure 4.3  Number of unique states at each branch commit point over 2000 runs. .... 37
Figure 4.4  Plot showing how many bits are different in at least one of 2000 runs at branch commit points. ....................................................... 38
Figure 4.5  A breakdown the bits that vary at each branch commit by component. .... 40
Figure 4.6  Plot showing how many bits are different on average between different runs of the processor at branch commit points. ............................ 41
Figure 5.1 This figure shows an example of a state with many possible predecessors.  
Figure 5.2 The relationship between $P_{all}$, $S_{partial}$, and $P_{sig}$.  
Figure 5.3 The trace is built up by iteratively collecting information about earlier states using BackSpace-2bp.  
Figure 5.4 Flowchart showing the BackSpace-2bp algorithm.  
Figure 5.5 A situation where the signature obtained in the first run of BackSpace-2bp may need to be discarded.  
Figure 5.6 This figure shows the timeline for the steps to add two states to the trace using BackSpace-2bp. Initially, state $s_i$ is the earliest state in the trace.  
Figure 5.7 This figure shows the timeline for the steps to add four states to the trace using BackSpace-3bp. Initially, state $s_i$ is the earliest state in the trace.  
Figure 5.8 The trace is built up by iteratively collecting information about earlier states using BackSpace-3bp.  
Figure 5.9 Flowchart describing the operation of BackSpace-3bp. State $s_i$ is the earliest state in the trace.  
Figure 5.10 Partial Breakpoint Circuit.  
Figure 5.11 Partial Breakpoint Counter Circuit.  
Figure 5.12 Hardware schematic for BackSpace-2bp.  
Figure 5.13 Hardware schematic for BackSpace-3bp.  
Figure 5.14 Histogram of the number of runs to add one state to the trace of $arith-add$ using BS-ideal.  
Figure 5.15 Histogram of the number of runs to add one state to the trace of $arith-add$ using BackSpace-2bp.  
Figure 5.16 Histogram of the number of runs to add one state to the trace of $arith-add$ using BackSpace-3bp.  
Figure 5.17 Histogram of the number of runs to add one state to the trace of $fib-20$ using BS-ideal.
Figure 5.18  Histogram with a logarithmic scale of the number of runs to add one state to the trace of fib-20 using BS-ideal.  

Figure 5.19  Histogram with a logarithmic scale of the number of runs to add one state to the trace of fib-20 using BackSpace-2bp.  

Figure 5.20  Histogram with a logarithmic scale of the number of runs to add one state to the trace of fib-20 using BackSpace-3bp.  

Figure 6.1  This graph shows the possible execution paths a circuit may take to reach one of the four crash states. The circuit is non-deterministic i.e., at certain states the next state is randomly chosen from two or more states.  

Figure 6.2  Graph showing an example of different execution paths converging. The transition probabilities for each of the next states for the states $S_i$ are $\frac{1}{2}$.  

Figure 6.3  Average number of runs needed versus the number of times the crash state is seen before trace collection is started.
Glossary

ALU arithmetic logic unit.

ATPG automatic test pattern generation.

BIST built-in self test.

CAD computer-aided design.

CADRE Cycle-Accurate Deterministic Replay.

CNF conjunctive normal form.

Dacota Data-coloring for Consistency Testing and Analysis.

DIVA Dynamic Implementation Verification Architecture.

ECC Error Correcting Code.

EDA electronic design automation.

FPGA Field Programmable Gate Array.

HLM high level model.

IFRA Instruction Footprint Recording and Analysis.

ISA instruction set architecture.
IVM  Illinois Verilog Model.

LVS  layout versus schematic.

PC   program counter.

PLL  phase-locked loop.

POR  plan of record.

RIT  random instruction testing.

RTL  register transfer level.

SAT  satisfiability problem.

SRAM static random access memory.

STA  static timing analysis.
Acknowledgments

I owe my deepest gratitude to Professor Tor Aamodt, my research supervisor, for his advice and guidance throughout my degree. This thesis would not have been possible without his support.

I would like to thank Professor Alan Hu and Professor Steve Wilton for being on my thesis committee. I am also very grateful to Professor Steve Wilton for his help and advice as a co-author on my first paper.

I would like to thank my colleagues Wilson Fung, Ali Bakhoda, Arun Ramamurthy, Andrew Turner, Tim Rogers, Jimmy Kwa, Rimon Tadros, George Yuan, Xi Chen, Marcel Gort, and Flavio de Paula for the stimulating conversation and helpful advice.

I would also like to thank the Semiconductor Research Corporation for their financial support.

I am forever indebted to my parents for their unwavering support throughout my studies.
Chapter 1

Introduction

It is very common for errors to be found in the design of modern processors after they are released [2, 20]. A well-known example of a design error is the Pentium floating point division bug that is estimated to have directly cost Intel $475 million (1994 USD) in replacements and write-offs [19]. In addition to the financial cost, Intel’s reputation was also affected [10]. These design errors exist despite the tremendous effort put into finding and correcting them. This effort starts before the chip is fabricated, during pre-silicon validation and continues afterwards during post-silicon debug (also known as post-silicon validation). Ideally, all the design errors would be found and corrected before the chip is manufactured, so that post-silicon debug would not be required. However, this is not possible, due in part to the complexity of modern chip designs and the slow speed of chip simulation. For example, during the pre-silicon validation of the Pentium 4 processor the total amount of simulation performed was equivalent to less than two minutes of execution on a 1 GHz chip [6].

Hence, the process of finding and correcting errors continues after the chip is fabricated. The fabricated chip may, in addition to design errors, have electrical errors caused by the manufacturing process. In this thesis we assume that a separate process, which we refer to as manufacture testing [36], has selected a set of chips that are free of electrical errors for use in post-silicon debug. Instead, this thesis will focus on enhancing the process of post-silicon debug, which is the process of determining what is wrong when a fabricated chip behaves incorrectly.
Lack of observability into chips is one of the major challenges in post-silicon debug [8, 36]. This is in contrast with pre-silicon validation where the entire internal state of the design is observable at all times. To improve observability, post-silicon engineers currently use tools like scan chains [31]. Scan chains allow the state of the chip to be read out once it is stopped provided special flip-flops are used on the chip. However, a snapshot of the state of the chip at one cycle may be insufficient to find the root cause of a bug.

One proposal to increase observability into a chip post-silicon is BackSpace [13, 14]. The goal of BackSpace is to produce a trace of what events on the chip leading up to a bug that can be fed into a waveform viewer. To achieve this, BackSpace uses scan chains, additional on-chip circuitry, and off-line analysis. This thesis improves on BackSpace by proposing modifications to the algorithm that reduce the number of times the chip must be run. New algorithms, BackSpace-2bp and BackSpace-3bp, are also proposed that generate traces with the same properties as those generated by BackSpace but requiring much less on-chip storage.

1.1 Background

The focus of this thesis is post-silicon debug. This is only one of many steps needed to design a high-performance microprocessor. To appreciate the challenges of post-silicon debug, it is useful to be familiar with some of the major steps in the design process. This section provides a brief overview of the steps a microprocessor design team takes to produce a successful microprocessor. The details provided here are based on Colwell [11] and Reilly [41]. The design process of each may be unique because of differences in microprocessor design projects and the teams working on them. Readers familiar with the process of large scale chip design may wish to skip to Section 1.2.

1.1.1 Product Definition

The project of designing a processor starts by determining what should be designed. Examples of what important parameters are attaining a certain level of performance on an important set of software applications, hardware features, cost, power consumption, and market timing.

An important factor determining what should be designed is what can be manufactured. The
group that will eventually manufacture the processor provides estimates of future semiconductor fabrication technology characteristics. This group may be part of the company designing the processor or another company that specializes in manufacturing integrated circuits (e.g., TSMC\(^1\)). Public information about process technology projections (e.g., [21]) may also be consulted [41].

Performance of the processor is very important. However, measuring the performance of potential designs is very difficult. The following section discusses ways to estimate the performance of potential designs using architectural simulators. One difficulty comes from not knowing what software applications will be important when the processor is finally on sale. To make an educated guess at what workloads will look like in the future, employees may be sent to attend conferences, seminars, and trade shows. Engineers may also visit companies that make the software that will run on the processor (e.g., the operating system).

The processor being designed will have competitors. If the performance and/or feature sets of future competing processors have been published in a roadmap (e.g., [18]) then that information can be used to determine a competitive feature set and performance targets for a planned processor. The planned release date may also be influenced by when competing products are anticipated to be released.

Once the requirements have been identified the next step is to explore how these requirements can be met.

### 1.1.2 Architectural Exploration

There may be many designs that meet the requirements determined in the previous step. In order to find one (or more) viable designs the architecture team tries to answer questions such as: How many levels of cache should there be? How fast and how big should each cache be? For each level of cache, should it contain both instructions and data? How many microarchitectural registers should there be? How many cores should there be on the chip? How many integer arithmetic logic units (ALUs) should there be per core? How many floating point ALUs should there be per core? What branch prediction algorithm should be used? How much storage should the branch predictor

\(^1\)http://www.tsmc.com
have? How many pipeline stages should the processor have? For superscalar designs, how many instructions should it be able to fetch, issue, or commit in one cycle?

In order to answer such questions, architects would like to be able to estimate the performance impact of different design decisions. Colwell [11] describes a progression of increasingly complex pre-silicon performance tools. For relatively simple in-order processors, such as the Pentium, a spreadsheet that combines information about the microarchitectural parameters and characteristics of the software applications of interest may suffice. To evaluate the out-of-order design of the Pentium Pro (P6) a data flow analyzer was created. This tool looks at the execution trace of a program, and computes the amount of (instruction level) parallelism available. Restrictions are placed on the amount of parallelism that can be extracted based on the design of the processor. For even more complex processors, such as the Willamette (Pentium 4), a detailed simulator that models the microarchitecture becomes necessary. These simulators may be very accurate, for example the simulator for the Pentium 4 had an average correlation of 97% with the register transfer level (RTL) model [45]. A very detailed simulator (e.g., Zesto [35]) may only be able to simulate $\sim 10^5$ instructions per second.

Reilly and Edmondson [42] provide more information on how performance simulators are used. Architectural simulators are also used in academia. Some examples are SimpleScalar [3], Zesto [35], and GPGPU-Sim [5].

The architectural exploration phase may find many ways of meeting the requirements. Eventually, one of them will be formalized as the plan of record (POR). The POR reflects the compromises made between the different project goals and will likely undergo many changes before the design is complete.

### 1.1.3 Architectural Refinement

For designs as recent as the Intel Pentium (P5), the architecture defined in the POR was broken down into blocks and the blocks were implemented directly in RTL logic [11]. For the Pentium Pro (P6) the team decided to first describe the architecture at a higher level of abstraction. They used the behavioral mode of Intel’s hardware design language (iHDL). Later Intel processor designs also
tried to use iHDL as a high level model (HLM) with limited success [27]. Other possible languages
to use for a HLM are C, C++, SystemC, and Bluespec.

There are many benefits to describing the architecture at a high level of abstraction. The first is
that it forces the architects to make concrete the abstract ideas from the previous step. There may
be a simple solution to problem X, and an elegant solution to problem Y, but these solutions may
not be compatible. The P6 team spent 18 months writing behavioral iHDL and resolving issues
that arose in converting the architectural ideas into a model of the architecture. Another benefit of
creating a HLM is that it makes it easier to make changes. As we will describe later, the high-level
description may change as a result of how blocks are implemented at the transistor level. A final
reason is that a higher level description is more likely to be correct. Algorithms have been proposed
to prove that the HLM is equivalent to the structural RTL description [26]. However, use of such
an algorithm may impose constraints on the HLM used. There are also algorithms to prove that the
structural RTL description is equivalent to the transistor level description [22], and that the transistor
level description is equivalent to the layout [34]. If a methodology that verifies that each model is
equivalent to the previous one is used, errors in the design are more likely to made in the higher
level descriptions of the architecture.

Regardless of whether a high level model of the processor is created, a structural RTL descrip-
tion must be written. The structural RTL description specifies every register and logic block. To
manage the complexity, the design is broken into “boxes” and the interfaces between these boxes
are defined. Each box is then designed by a specific team.

To validate the performance of the structural RTL description, we would like to run programs
on it. However, due to the slow RTL simulation speed, we may not want to start the simulation from
reset [45]. The structural RTL model’s caches, buffers, and branch predictors are loaded with values
from the architectural simulator. The structural RTL model is then run and its output compared with
that of the architectural simulator. Differences between the performance simulator and the structural
RTL model are resolved by changing the performance simulator if it did not model some detail, or
by changing the structural RTL if it is not correct [45].

Once the structural RTL description is complete and its performance is close to that predicted
by the architectural simulator the high-level design of the processor is done and the long task of converting this design into a layout can begin.

1.1.4 Circuit Design

The teams that will convert the design created by the architects into the masks used in production do not sit idle until the design is finalized. Parts of the design that appear unlikely to change can be designed before the rest of it is finished. New tools may need to be written in order to design the processor proposed by the architecture team. The computer-aided design (CAD) or electronic design automation (EDA) tools needed by the design team include tools that perform logic synthesis, routing, placement, design rule checking, and timing analysis. Although these tools are sold by companies such as Synopsys, Inc\(^2\) and Cadence Design Systems\(^3\) designers of high-performance circuits may create their own\[^{[41]}\].

Once the structural RTL description of the processor is complete it needs to be converted into a lower level description suitable for use by the layout team. The tools developed earlier may be able to convert the structural RTL description into an equivalent switch (transistor) level design, but they may not be able to meet performance goals, power limits, or other constraints. In these cases, designers will custom design parts of the processor. To verify that the design of these complex circuits is correct the simulations that were performed with the structural RTL description could be redone. However, the simulation of the switch level design is much slower. Another way to verify that the manually designed switch level design is correct is to formally prove that it is equivalent to the structural RTL description. For example, Motorola has a switch level verification tool\[^{[22]}\] that starts with a switch level design, converts it to an equivalent RTL design, and then applies equivalence checking\[^{[30]}\] between these RTL descriptions. If the structural RTL description differs from the switch level description a decision must be made regarding which one should be changed. If the switch level description of a block performs its function correctly, but its behaviour (e.g., timing) differs from the high-level description, the high-level description may be changed to match its operation\[^{[41]}\]. We will refer to this switch level description as the schematic.
The final step in the design process is to convert the schematic into a set of lithographic masks used to produce the chip. For some parts of the design automatic layout tools may suffice. But high performance blocks may require the layout to be done manually by skilled layout designers. The job of these designers is to convert each transistor in the schematic into a set of polygons. The wires that connect the transistors also need to be routed. To ensure that the layout is equivalent to the switch level description, layout versus schematic (LVS) verification software can be used. A LVS tool (e.g., [34]) converts the connectivity of the manually created layout into a switch level description. This description is compared with the schematic created in the previous step.

At this point, the logical design is complete and masks could be generated. However, we have only discussed the logical correctness of the layout so far. Real chips need to deal with circuit level issues such as timing, crosstalk, and leakage. These issues are discussed in the next section.

1.1.5 Circuit Verification

In practice, due to schedule pressure, circuit verification does not wait until the layout is complete before starting [41]. Effects that need to be modelled include transistor leakage, wire delays, crosstalk between wires, noise, and electromigration.

To verify the circuit layout, the connections between terminals (wires) are modelled with a capacitance matrix and an admittance (resistance) matrix [37]. These RC values are used to compute the delays used in static timing analysis (STA). The usage of STA is to verify that the setup and hold times of flip-flops are satisfied. This analysis can be performed as soon as the design is converted into a gate (or transistor) level description. At that point, STA may be useful in identifying slow paths, but only when the entire circuit including clock trees are laid out can the timing of the actual circuit be verified [7].

Once the layout is verified by modelling all the effects that need to be modelled, the layout can be sent to the fabrication facility. Some engineers on the team can relax until the first chips arrive, but others will be busy writing test vectors for use in the next phase.
1.1.6 Manufacture Testing

The exact process by which millions of transistors and the wires connecting them are created in a multi-layer structure is beyond the scope of this work. We will note that many chips are created at once from a circular wafer of silicon, and that some of the transistors or wires may not have been manufactured correctly. To avoid the cost of packaging a defective chip, each chip is tested using test patterns before packaging. A test pattern specifies the correct output after the inputs are driven in a particular pattern. Chips that pass these tests are then packaged so that they can be tested in real systems.

A process called manufacture testing detects physical faults in the manufactured chips. In the rest of this thesis, we will assume that manufacture testing has selected chips that correctly implement the design to be used in post-silicon debug. Two methods that can be used in manufacture testing are automatic test pattern generation (ATPG) and built-in self test (BIST) [23]. Passing the test patterns generated by ATPG may, for example, show that a chip does not have any bits that are stuck at 0 or 1. These stuck-at faults are useful as an abstraction of physical faults [40]. Tests may also be done on-chip using BIST to generate test patterns (possibly in a pseudo-random fashion), apply them, and verify that the response is correct. Once working chips have been identified, the process of post-silicon debug begins.

1.1.7 Post Silicon Debug

The process of post-silicon debug is the focus of this thesis and begins after chips free of electrical defects have been identified. There are many approaches to obtaining functional validation coverage that are useful [9]. The first is random instruction testing (RIT). The results of short random programs are obtained from an instruction set simulator and compared with the output of the chip. The next is directed testing which focuses on testing important corner cases which may not be reached by RIT. Another approach is to run the chip with real software and connected to a real system.

Design errors uncovered during this stage of testing are difficult to debug. An overview of different techniques that can be used here is presented in [Chapter 2]. After the errors are found, the design is changed to fix the error. This change may need to be made at the HLM level, then in the
RTL description, then in the schematics and layout. Then a new set of masks need to be made at considerable expense before an updated chip is available for further testing.

There are many challenges faced by engineers during post-silicon debug. One of them is schedule pressure. Customers, salespeople, and managers would like to know when the chip is going to be ready. Expensive fabrication facilities may be sitting idle waiting for the last bug to be resolved before mass production can begin. But it is difficult to debug the chip due to the lack of visibility into what is actually happening on-chip [8, 36]. In the next section, a method of improving on-chip visibility called BackSpace [13, 14] is introduced.

1.2 BackSpace

De Paula et al. proposed a methodology to improve post-silicon observability by creating a complete trace of arbitrary length of what actually occurred on-chip called BackSpace [13, 14]. This thesis builds on this work. A post-silicon trace could potentially be fed into a waveform viewer and used by an engineer in the same manner as a trace generated pre-silicon by a simulator. This trace is created by running the chip multiple times; ideally adding one state to the trace every time the chip is run.

1.2.1 BackSpace Algorithm

The BackSpace algorithm begins by assuming that the chip is stopped at a state that we would like to generate a trace to. This state will be referred to as the crash state. In order to extend the trace earlier into execution, the state that preceded the crash state is needed. From the crash state and the design of the chip (in the form of a gate-level description) the set of possible states that could have led to the crash state is computed. This set of states is called the pre-image $P$. The computation of $P$ is explained in Section 1.2.2. One of the states in $P$ will be the state that actually occurred before the crash state. It is known that this state can be reached by the chip. The other states in $P$ may not be reachable. In the next step, we find a state in the pre-image that is reached during execution. This is done using a breakpoint circuit that checks if the current state matches a target state every cycle. If the two states match then the chip will stop and allow the next iteration to
Figure 1.1: Flowchart showing the baseline BackSpace algorithm.

proceed. One candidate state in the pre-image is loaded into the breakpoint circuit and the chip is run until either the candidate state is hit and the chip stops, or a timeout is reached. If the chip times out, then another state in the pre-image is loaded into the breakpoint circuit. Eventually a state in the pre-image will be reached. This state is known to occur during execution and also is a possible previous state to the earliest state in the trace so it can be added to the trace as the new earliest state. A flowchart describing this algorithm is shown in Figure 1.1.

1.2.2 Pre-Image Computation

The pre-image is the set of possible previous states. It is computed by first constructing a boolean formula such that all variable assignments that satisfy the formula correspond to a possible previous state. A satisfiability problem (SAT) solver [16] is then used to find an assignment that satisfies the boolean formula. The variables that represent bits in the previous state are extracted and the potential previous state is added to the pre-image. A clause is created that this state does not satisfy and it is added to the boolean formula. This new formula is run through the SAT solver, another state is added to the pre-image, and a new clause is added to the boolean formula. This process repeats
until the boolean formula becomes unsatisfiable which indicates that all the states in the pre-image have been found.

In the worst case, every state in the pre-image needs to be tested to find the next state in the trace. One method of reducing the number of times the circuit must be run is to reduce the size of (number of states in) the pre-image. This can be achieved by recording the value of some bits in the previous state and adding these values to the boolean formula as clauses:\footnote{If the entire previous state was recorded then the pre-image would contain one state.} The information stored about the previous state is called the signature of the previous state.

1.2.3 Hardware Support

The hardware components needed for BackSpace are the breakpoint circuit, signature generation circuit, and signature collection circuit. The breakpoint circuit compares the current state of the chip with a stored target state and stops the circuit when they match. The signature contains some information about the previous state of the chip to reduce the number of possible predecessor states. In general the signature can be an arbitrary function of the bits in the previous state. De Paula et al. \cite{14} evaluated using universal hashes of the previous state as the signature. Although using universal hashes reduced the size of the pre-image by two orders of magnitude, the hardware overhead was very high. The estimated area overhead for implementing universal hashing on a LEON3 processor was over 150%. The only signatures we consider in this thesis are subsets of the state bits so the signature generation circuit, which generates the signature from the current state of the chip, will not be discussed further. The signature collection circuit stores the signature created by the signature generation circuit for at least one cycle. The baseline BackSpace algorithm stores the signature for one previous state.

1.2.4 Partial Breakpoints

Gort\cite{17} described a modified BackSpace framework that reduces the hardware overhead by reducing the number of bit comparisons the breakpoint circuit needs to do. Instead of comparing every bit in the state with a stored target state, only part of the state is compared with a partial breakpoint stored in the breakpoint circuit. Once a candidate predecessor state is known the subset of state bits
that are used in the partial breakpoint circuit are loaded into a counting circuit. The chip is run to determine how many times $n$ this partial breakpoint is matched before the circuit reaches the earliest state already in the trace. Once this count is obtained the partial breakpoint is loaded into the breakpoint circuit along with the count $n$, and when the partial breakpoint is matched $n$ times the chip is stopped. The possible downsides to this technique are the possibility of not reaching the correct state even when the partial breakpoint matches the correct number of times due to non-determinism and the requirement of having two partial breakpoint circuits. Note that this technique could also be implemented with one partial breakpoint circuit and a dedicated counting circuit as described in the original proposal for the counting circuit by Kuan [28].

1.2.5 Challenges

Two major challenges of using BackSpace are the number of times the chip must be run and the large hardware overhead required. These two challenges are related. Using a larger signature at the expense of more hardware overhead means that the pre-image will, on average, be smaller and so the chip needs to run fewer times. Using a smaller signature reduces the hardware overhead, but increases the size of the pre-image and the expected number of times the chip must be run. Chapter 3 will address the number of runs needed to add one state to the trace, and in Chapter 5 both of these challenges will be addressed.

1.3 Contributions

The main contributions of this thesis are proposing a method of intelligently sorting the pre-image, implementing BackSpace on a non-deterministic superscalar out-of-order processor, and proposing new post-silicon trace generation algorithms that require much less storage overhead than existing work. These contributions are described below.

1.3.1 Pre-image Sorting

Since the states in the pre-image are tried one at a time, it would be advantageous to try the states in some intelligent order. An algorithm is proposed that collects information about the correlation between state bits dynamically as a trace is being generated and uses that information to sort the
states in the pre-image. This leads to a reduction in the number of times the chip must be run by up to 51% [29].

1.3.2 Non-deterministic Superscalar Out-of-order Processor

Previous work on BackSpace has used relatively simple in-order processors. In this thesis, we describe an implementation of BackSpace on a superscalar out-of-order processor running the Alpha instruction set. Non-determinism is also introduced in the form of variable memory timing and the effects of this non-determinism on the processor is studied.

1.3.3 New Post-Silicon Trace Generation Algorithms

Overwriting values in registers can lead to extremely large pre-images if the overwritten value is not computable from the signature and next state. In order to reduce the signature size, and the storage overhead required, a new post-silicon trace generation algorithm called BackSpace-2bp is proposed that eliminates the need to compute the pre-image. BackSpace-2bp requires less than 1% of the storage of BackSpace. However, BackSpace-2bp may require twice as many runs as BackSpace with a signature that includes the entire previous state and therefore has only one state in the pre-image. By pipelining the steps of BackSpace-2bp, this performance penalty can be removed. This algorithm, BackSpace-3bp, requires storing 99.5% fewer bits than the baseline BackSpace algorithm. Relative to a state of the art algorithm that uses partial breakpoints, BackSpace-3bp stores 94.4% fewer bits. BackSpace-3bp requires the same number of runs as BackSpace with a signature that contains all the bits of the previous state. A practical implementation of BackSpace will not be able use that large a signature, so BackSpace-3bp will also require fewer runs of the processor.

1.4 Thesis Organization

Chapter 2 will provide an overview of existing work related to post-silicon debug besides the work on BackSpace described above. In Chapter 3 a technique that improves the performance of BackSpace, by intelligently sorting the pre-image, will be introduced. The evaluation in Chapter 3 uses an existing implementation of BackSpace [12] on the OpenCores 8051 processor. In Chapter 4
the implementation of BackSpace on a superscalar out-of-order processor using the Alpha instruction set with non-determinism in the form of variable memory latency will be described. Two new post-silicon trace generation algorithms, BackSpace-2bp and BackSpace-3bp, will be presented in Chapter 5. The main benefit of these algorithms is that they do not require the computation of pre-images or trying candidate predecessor states one at a time. The evaluation of these new algorithms will use the infrastructure introduced in Chapter 4. In Chapter 6 the idea of selecting “good” crash states to generate a trace to is explored. Chapter 7 will summarize this thesis and present ideas for future work.
Chapter 2

Related Work

This chapter presents some existing work in the field of post-silicon debug. Post-silicon debug is an increasingly important problem and there have been many proposed techniques to assist in this process. These include leveraging the execution speed of the fabricated chip to accelerate validation algorithms [15], using existing features for manufacturing test [25], applying techniques from formal verification [47], bug detection [1, 4], removing non-determinism [33, 43], and error localization [39]. Each of these approaches are summarized below in turn. They are also compared with the approach to post-silicon debug this thesis focuses on, BackSpace, which was introduced in Section 1.2.

2.1 Hardware Acceleration

Data-coloring for Consistency Testing and Analysis (Dacota) [15] is a proposed system that validates the memory system of a multiprocessor system post-silicon. It does this by recording memory accesses in a specially configured section of the cache and then periodically running an algorithm that checks that, for each cache line, all the cores have seen the same order of write operations. By leveraging the execution speed of the system under validation Dacota can verify the consistency of the memory system with minimal hardware overhead. However, Dacota is limited to detecting memory errors that result from the interaction between multiple cores.
2.2 Existing Observability

Complex modern circuits contain scan chains and trace buffers to aid in the debugging of electrical and design errors. Scan chains allow the state of the circuit to be read out. In this thesis, it is assumed that the entire state of the circuit is available to be scanned out, however in real designs this may not be the case. The state may be scanned out when the circuit is stopped or when it is still running [31]. For the techniques proposed in this thesis it is not required to scan out the state as the circuit is running. While scan chains only provide (part of) a single state, trace buffers [1] provide many cycles of information on a smaller subset of the state bits.

There are techniques that use scan chains and trace buffers directly to aid in post-silicon debug. For example, Ko and Nicolici [25] combine the information collected from scan chains and trace buffers and apply state restoration techniques [24] to create a trace that contains more state information than either scan chains or trace buffers alone could provide. However, even after applying state restoration algorithms the trace created may not contain complete states. Also, the length of the trace is limited by the amount of storage used for the trace buffer.

2.3 Formal Verification

There have been proposals that use techniques from formal verification to assist in post-silicon debug. Formal verification is an alternative to post-silicon debug that ensures the design is correct before it is fabricated. Properties about the design can be proved or disproved. The main problem with formal verification is its inability to deal with large designs. Yang et al. [47] consider a technique to determine which signals should be monitored using the trace buffers of an erroneous circuit during post-silicon debug. The circuit is considered erroneous because its output differs from a reference output. They do this by constructing a boolean formula in conjunctive normal form (CNF) from the inputs, circuit design, and a reference output. The circuit is generally sequential and must be unrolled into a combinational one in order to construct the boolean formula. This formula is not satisfiable because there is an error in the design. A subset of clauses that are also not satisfiable (an UNSAT core) of this boolean formula correspond to possible locations for the error and this information can be used to pick which bits should be traced using the trace buffer. One limitation
of this technique is the computational effort needed to solve SAT instances for this unrolled circuit.

2.4 Bug Detection

There has been previous work on how to detect when an error has occurred at runtime after a chip has been fabricated. Austin [4] has proposed an architecture called Dynamic Implementation Verification Architecture (DIVA) where all instructions are executed twice, once by a fast processor core that may contain errors and again by a DIVA checker that is much more likely to be error-free. Assertions can also be used to detect when the chip has encountered an error [1]. The debugging process is not over when an error is detected. The root cause of the error must still be found. These techniques for detecting bugs can be used in conjunction with the trace generation techniques discussed in this thesis to build a trace that ends closer to the source of the error instead of the point where the error becomes visible at the outputs.

2.5 Removing Non-determinism

If all forms of non-determinism could be removed then the process of post-silicon debug becomes much easier. Cycle-Accurate Deterministic Replay (CADRE) [43] is a proposal to log all the information about non-deterministic events necessary to replay them.

TotalRecall [33] is a proposed system for Field Programmable Gate Arrays (FPGAs) that removes system-level non-determinism by feeding the same inputs into two identical circuits, but one set of inputs is delayed by \( n \) cycles. When an error is encountered in the circuit running ahead the circuit running behind can be stopped and the state of the circuit \( n \) cycles before the error can be read. However, this approach only removes non-determinism that is external to the circuit. If the circuit itself had uninitialized state or contains multiple clock domains that lead to non-determinism then the state of the circuit running behind may not be the state of the circuit running ahead \( n \) cycles ago.

During bring-up in the lab, specialized equipment may be used to remove non-determinism that comes from I/O devices. There is little publicly available information on this equipment.
2.6 Error Localization

One aspect of post-silicon debug is locating where in hardware and in which cycle a detected bug occurred. Instruction Footprint Recording and Analysis (IFRA) [39] records the data and instruction flows through certain design blocks. When a bug is detected these flows are analyzed in conjunction with the program binary to localize the error. Later work [38] demonstrated how to automate the analysis. One important benefit of this approach is that it does not require reproduction of the bug. IFRA was evaluated by modelling electrical bugs as single bit-flips in a micro-architectural simulator.
Chapter 3

Sorting the Pre-image\textsuperscript{1}

The BackSpace algorithm was described in \textsection{1.2}. In order for this algorithm to be feasible, the number of possible predecessor states in the pre-image \(P\) must be as small as possible. The signature helps in reducing the size of the pre-image. Even with a signature, however, the number of states in \(P\) may be large. For example, in the implementation of an Intel 8051 processor we used in our evaluation which contains 702 flip-flops, if we select 281 of them as signature bits, there are, on average, 31 elements in \(P\). Since we try each of these individually until a match is found, we expect to try, on average, half of the 31 elements in \(P\).

Clearly, the order that we try the elements of \(P\) is important. In the original implementation of BackSpace, the elements are ordered arbitrarily. In this chapter we use additional information gathered for the circuit to re-order the elements of \(P\) so states in \(P\) that are more likely are tried first. This results in a reduction in the number of runs required to construct a trace. For the examples we evaluate, the number of runs of the chip was reduced by between 12\% and 51\% (leading to significant reduction in overall run-time) without affecting the accuracy of the resulting trace. This is important, since the usefulness of BackSpace relies on the debugging engineer being able to quickly and accurately obtain such traces.

\textsuperscript{1}A version of this chapter has been published. Johnny J.W. Kuan, Steven J.E. Wilton, and Tor M. Aamodt. Accelerating Trace Computation in Post-Silicon Debug. In \textit{11th International Symposium on Quality Electronic Design (ISQED)}, pages 244-249, March 2010.
3.1 Algorithm

The key idea behind our algorithm is to gather information about the correlation of bits in the design, and use this information to re-order the elements of $P$ so that states that are more likely to be the correct predecessor state are tried first. In the following, we first describe the correlation information itself, and then show how this information is used to guide BackSpace. Finally, we describe two techniques by which the correlation information can be obtained. The changes to the BackSpace flow needed to implement this algorithm are shown in Figure 3.1.

3.1.1 Correlation Information

Since our goal is to determine the most likely predecessor state for a given state, we first consider correlations between states. Ideally, any two states that could have a transition between them would have a non-zero correlation value, and any two states that could not have a transition would have zero correlation. Also, the correlation between two states would be high if the first state was likely to occur exactly one cycle before the second state, and low if this was unlikely. A direct solution would
be to find the correlation of each state in cycle $i$ with every state in cycle $i + 1$. This information could be gathered by running the chip for a long period of time, and recording state transitions. Then, during debugging, when the set $P$ is obtained from the off-chip analysis algorithm, we could sort $P$ based on these correlation numbers and the scanned out state. While this technique would likely give good results, it is not feasible, even for medium-sized designs. A circuit with $n$ flip-flops (and hence $n$ state bits) could have up to $2^n$ possible states, meaning we would need to compute and maintain $2^n \cdot 2^n = 2^{2n}$ correlation values.

Instead, we compute the $\frac{n(n-1)}{2}$ correlations between individual bits within a single state. Note that this solution does not measure the correlation between different states like the first solution. Instead it attempts to encapsulate the heuristic that certain states are more likely than others. For any two state bits $x$ and $y$, the correlation between these bits, $\rho_{x,y}$ indicates how likely these two bits are to be the same or different. The correlation used is the sample probability that the two bits are the same scaled to the range $[-1, 1]$. Note that this may be different from the Pearson product-moment correlation coefficient. More precisely,

$$
\rho_{x,y} = 2 \cdot P(x = y) - 1
$$

$$
= \frac{2}{T} \sum_{i=1}^{T} \left(1 - (x_i - y_i)^2\right) - 1
$$

$$
= \frac{1}{T} \sum_{i=1}^{T} \left(1 - 2(x_i - y_i)^2\right)
$$

$$
= \frac{1}{T} \sum_{i=1}^{T} \left(4x_i y_i - 2x_i^2 - 2y_i^2 + 1\right)
$$

$$
= \frac{1}{T} \sum_{i=1}^{T} \left(4x_i y_i - 2x_i - 2y_i + 1\right)
$$

$$
= \frac{1}{T} \sum_{i=1}^{T} \left(2x_i - 1\right)\left(2y_i - 1\right)
$$

(3.1)

where $x_i$ is the value of bit $x$ in cycle $i$, and $y_i$ is the value of bit $y$ in cycle $i$. Note that $x_i^2 = x_i$ because $x_i$ is binary. $T$ is the number of cycles of correlation information that has been gathered so far. Section 3.1.3 discusses different methods of gathering this information. The value of $\rho_{i,j}$ is -1 if the value of bit $i$ is always the inverse of the value of bit $j$ and 1 if the value of bit $i$ is always the same
as the value of bit $j$. The value of $\rho_{i,j}$ is 0 if knowing the value of one bit provides no information about the value of the other bit. Thus, we can use the absolute value of $\rho_{i,j}$ to determine how correlated two bits are. Although this correlation information does not provide as much information as the first solution, we will show that it provides sufficient information to produce an intelligent ordering of the states in $P$.

### 3.1.2 Using Correlation Information

Consider a set of potential pre-image states $P$ that are potential predecessors of a crash state $s_0$. We wish to sort the elements in $P$ such that the states that are most likely to be the predecessor of $s_0$ are first in the list. In this subsection, we show how we do this, given the correlation information described in the previous subsection.

The individual states in $P$ differ by some number of bits. If we can predict the most likely value of these bits, then we can predict which of the states in $P$ is the most likely to have occurred immediately before $s_0$. We do this in two steps: pre-image merging, and pre-image ranking. Both are described below, and the overall algorithm is summarized in Algorithm 1.

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

Consider the example in Figure 3.2. The set $P$ contains three potential predecessor states $p_0$, $p_1$, and $p_2$. In all predecessor states, bit 3 is a 0 and bit 4 is a 1. Thus, bit 3 of $w$ is set to 0, and bit 4 of $w$ is set to 1. Bits 0 to 2 differ across the set of predecessor states. Thus, we set bits 0 to 2 of $w$ to $X$.

Next, we perform pre-image ranking. To do this, we first convert the word $w$ into a prediction $r$ of the predecessor state. For each bit $i$ in $w$ that is either 0 or 1, bit $i$ of $r$ is set to this value. For the
**Input:** Unsorted pre-image list $P$
Number of state bits $n$
Correlation matrix $\rho_{i,j} \forall 0 \leq i, j < n$

**Output:** Sorted list $P'$

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

Let $w$ be a guess containing $n$ bits in $w$ that are $X$, the following is performed. The correlation information described earlier is first used to determine which other bit $j$ in the design is most correlated to this bit (i.e. has the highest value of $|\rho|$, ties broken by minimizing $|i - j|$, further ties broken by smallest $j$). If the value of this other bit is known, then either that value or its complement is used in the corresponding location in $r$. If $\rho_{i,j}$ is positive or zero, the value is used, while if it is negative, the complement of the value is used. If the bit value is unknown, the corresponding location of $r$ is again marked with an $X$.

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

In this way, the set of \( P \) is ordered in a manner such that the states that are “closest” to the predicted previous state are tried first. This ordered set is then used in the BackSpace flow as described in Section 1.2 and shown in Figure 3.1.

### 3.1.3 Gathering Correlation Information

Section 3.1.2 described the correlation information, but it did not indicate how this information can be gathered from the design. We consider two approaches to gather this information. The first approach is to gather the information statically, before the chip is debugged. Using a “training testcase”, the design can be simulated (or the fabricated chip can be run) and individual states extracted. For a processor design, a training testcase can be a program executing on the processor; for a non-processor design, the training testcase can be a list of input values over time. From these extracted states, the correlation information for each pair of bits can be calculated. The summation limits in Equation 3.1 correspond to the start and end of the training testcase.

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

Instead, we use a dynamic approach, in which we gather the correlation during debugging. As described in Section 1.2, BackSpace is iterative; for each state added to the trace, the chip is run one or more times. In our dynamic solution, we compute the correlation information as states are scanned out of the chip and added to the trace history. Initially, BackSpace proceeds without any correlation information, however, later iterations are able to use the correlation information gathered during previous iterations. As we will show, this technique provides sufficient information to accelerate the BackSpace flow.
Table 3.1: Start and end cycles for the programs run on the 8051.

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Start Cycle</th>
<th>End Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib</td>
<td>630</td>
<td>2224</td>
</tr>
<tr>
<td>sort</td>
<td>630</td>
<td>8368</td>
</tr>
<tr>
<td>sqroot</td>
<td>660</td>
<td>5813</td>
</tr>
</tbody>
</table>

3.2 Methodology

We use an 8051 simulator and benchmarks from the BackSpace Toolkit version 0.1 [12]. To evaluate our algorithm for sorting the states in $P$ we collected and analyzed traces of the 8051 running the three programs shown in Table 3.1. The 8051 used is an open-source implementation of an 8-bit Intel micro-controller available from OpenCores\(^2\). Data was not taken from the first few hundred cycles because there are uninitialized flip flops in the design. The first cycle in the trace is referred to as the start cycle. After the program finishes execution the 8051 enters a 12 cycle loop. After one iteration of this loop we stop the trace and label that the end cycle of the trace. The start and end cycles for the benchmarks used are shown in Table 3.1.

As described in Section 3.1.3 the correlation information is obtained as BackSpace is running. For the first 300 cycles of a BackSpace computation we do not apply the algorithm described in Section 3.1 and BackSpace proceeds as the baseline described in Section 1.2. After BackSpace computes a trace of 300 cycles leading up to the crash we compute the correlation $\rho$ defined in Equation 3.1 for all $\binom{702}{2} \binom{701}{2}$ pairs of unique bits in the same state over the 300 cycles available. We can now use this information to sort the potential pre-image sets obtained for the next 300 cycles. After a trace of 600 cycles is obtained we re-calculate the correlation values and use the updated values for the next 300 cycles. BackSpace continues in this fashion, updating the correlation values every 300 cycles, until the beginning of the trace is reached.

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

\(^2\)http://opencores.org
3.3 Results

The results for running \texttt{sqroot}, \texttt{sort}, and \texttt{fib} can be seen in Figures 3.3, 3.4, and 3.5 respectively. The cumulative number of runs needed to generate traces of varying length are shown for both the baseline BackSpace algorithm and our enhanced algorithm. In particular, the right-most point on each of these graphs shows the total number of times the chip must be run to BackSpace the entire trace. Figure 3.3 shows that the difference between the cumulative number of processor runs for the enhanced algorithm and the baseline increases as more states are added to the trace. By the time BackSpace traces through all states in the \texttt{sqroot} program there is a 51% reduction in the number of times the processor must be run relative to the baseline. Figure 3.4 shows the result for \texttt{sort}. After backspacing 3900 cycles the lines for our enhanced algorithm and the baseline are parallel. This is because there is only one candidate predecessor state for those cycles and the baseline algorithm cannot be improved upon. This can be seen more clearly in Figure 3.7 Upon completion of this trace, there is a cumulative reduction of 14% in the total number of processor runs required. The baseline algorithm needed only 3.32 runs on average to add one state to the trace.
Figure 3.4: Cumulative processor runs for the 8051 running sort. Baseline is the original BackSpace algorithm. Enhanced is the BackSpace algorithm using the algorithm proposed in Section 3.1.

Figure 3.5: Cumulative processor runs for the 8051 running fib. Baseline is the original BackSpace algorithm. Enhanced is the BackSpace algorithm using the algorithm proposed in Section 3.1.
Figure 3.6: Average processor runs per correlation sample interval for the 8051 running \texttt{sqroot}. Enhanced is the BackSpace algorithm using the algorithm proposed in Section 3.1.

For \texttt{sqroot}, Figure 3.5 shows the result for \texttt{fib}. Although BackSpace was only run for 1594 cycles for this program, the results show that as BackSpace is run, our enhanced algorithm requires a smaller fraction of the number of processor runs required by the baseline. The overall improvement for this program is 12%.

The average number of runs needed over each 300 cycle epoch for \texttt{sqroot}, \texttt{sort}, and \texttt{fib} can be seen in Figures 3.6, 3.7, and 3.8 respectively. Each point shows the number of runs needed to reach the correct predecessor state averaged over 300 cycles. For each of these results the first point, at 300 cycles backspaced, is the same for both our enhanced algorithm and the baseline. This is because there has been no correlation data gathered for that program yet. The next point, at 600 cycles backspaced, shows the impact of using 300 cycles of correlation data to sort the candidate pre-images. In Figure 3.6 we see that our enhanced algorithm reduces the number of runs needed over a 300 cycle period by up to 77% (between 4201 and 4500 cycles backspaced) for the \texttt{sqroot} program. Over the entire trace the reduction in the number of times the chip would need to be run is 51%. With the \texttt{sort} program shown in Figure 3.7, we only see a 14% reduction in the number
Figure 3.7: Average processor runs per correlation sample interval for the 8051 running `sort`. Enhanced is the BackSpace algorithm using the algorithm proposed in Section 3.1.

Figure 3.8: Average processor runs per correlation sample interval for the 8051 running `fib`. Enhanced is the BackSpace algorithm using the algorithm proposed in Section 3.1.
of times the chip needs to be run. This is partly due to the many states that only have one candidate pre-image. Figure 3.8 shows the result for the fib program, where the number of chip runs is only reduced by 12% overall. One of the reasons the overall improvement is lower for this program is because it is short. For about the first 20% of the cycles backspaced there is no correlation to sort the candidate pre-images with.

3.3.1 Alternative Algorithms

To gain further insight, we tried several other algorithms. First we investigated using the state the processor is currently stopped at as the prediction \( r \). Intuitively this would work if the unknown bits of the state do not change very much from cycle to cycle. This algorithm (referred to as “Guess No Change” in Table 3.2) was able to predict the correct state and reach the correct candidate pre-image on the first run more often than the baseline. This means that the Hamming distance between the prediction \( r \) and the actual predecessor state was often minimized. However, the average number of runs to find the predecessor state was 6% greater than the baseline. For the circuit we used, it was found that the state bits have a zero bias, that is there were more 0 bits in the trace than 1 bits. Guessing 0 for all the unknown bits, and using the baseline ordering to resolve ties, results in approximately the same performance as the baseline. Randomizing the order before sorting based on a guess of 0 (referred to as “Randomized Guess 0” in Table 3.2) for all the unknown bits results in performance 4% worse than the baseline, but still requires 34% fewer runs than a random ordering. These results may suggest that a portion of the performance improvement of the order tried by the baseline BackSpace algorithm over a random ordering results from the tendency of the baseline BackSpace algorithm to order states with more 0 bits first. The algorithm we propose in this chapter (“Enhanced”) performs better than the above simpler algorithms. On average, 20.4% of the states in the pre-image \( P \) need to be tried before the correct state is found using the “Enhanced” algorithm.

3.4 Summary

BackSpace generates a trace showing the state of the chip leading up to a crash. Each cycle added to the trace may require running the chip multiple times in order to reach the previous state. We
Table 3.2: The relative number of runs required to BackSpace the different programs relative to the baseline.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>sqroot</th>
<th>sort</th>
<th>fib</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enhanced</td>
<td>0.49</td>
<td>0.86</td>
<td>0.88</td>
</tr>
<tr>
<td>Randomized Guess 0</td>
<td>1.02</td>
<td>1.09</td>
<td>1.07</td>
</tr>
<tr>
<td>Guess No Change</td>
<td>1.07</td>
<td>1.09</td>
<td>0.94</td>
</tr>
<tr>
<td>Baseline</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Random</td>
<td>1.25</td>
<td>2.11</td>
<td>2.06</td>
</tr>
</tbody>
</table>

proposed a technique where we use the trace generated so far to compute the correlation between state bits. Using this correlation information, we then make a prediction about which potential previous states to try first. Our results showed between 12% and 51% reduction in the number of times the chip must be run over an entire trace computation, which translates into a significant reduction in runtime.

Although the technique proposed in this chapter is able to greatly reduce the number of times the chip must be run to generate a trace, we will show in Chapter 5 that the number of runs required may still be too great. We also have not yet addressed the area overhead needed to generate the post-silicon trace or considered the impact of non-determinism. In the following chapters we will address all these challenges.
Chapter 4

A Complex Non-deterministic Processor

In the previous chapter, a very simple processor was used to evaluate the technique of sorting states in the pre-image. For the evaluations in the rest of this thesis, we will be using a more complex processor with non-deterministic memory access timing. In the first section of this chapter, a superscalar out-of-order Alpha ISA processor will be introduced, and the simulation setup used to run BackSpace on this processor will be described. Next, the implementation of non-deterministic memory access timing will be described. Also, other forms of non-determinism that may exist in real systems will be discussed. Finally, we will evaluate the effects of the non-deterministic memory access timing on the Alpha ISA processor.

4.1 Out-of-Order Alpha ISA Processor

In this section, we describe the processor used in the rest of this thesis. One of our goals is to study the effects that non-determinism has on the performance of BackSpace. The simple in-order processor used in Chapter 3 may not capture all the effects that non-determinism can have on a complex modern processor. We therefore use a complex out-of-order processor, the Illinois Verilog Model (IVM) [46], originally used to model the effects of transient faults. Some important microarchitectural parameters of this processor are shown in Table 4.1. Two versions of the IVM processor are available[1] IVM–1.0 is capable of running Alpha programs, and IVM–1.1 is syn-

Table 4.1: IVM processor microarchitectural parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Width</td>
<td>8</td>
</tr>
<tr>
<td>Scheduler Width</td>
<td>6</td>
</tr>
<tr>
<td>Commit Width</td>
<td>8</td>
</tr>
<tr>
<td>Reorder Buffer Size</td>
<td>64</td>
</tr>
<tr>
<td>Scheduler Window Size</td>
<td>32</td>
</tr>
<tr>
<td>Physical Registers</td>
<td>80</td>
</tr>
<tr>
<td>Load Queue Entries</td>
<td>16</td>
</tr>
<tr>
<td>Store Queue Entries</td>
<td>16</td>
</tr>
</tbody>
</table>

The sizable. Verilog code from both versions was combined, and modified to resolve issues arising from uninitialized state and differences in the synthesis tools used.

The IVM processor includes advanced features such as branch prediction and memory dependence prediction. The branch predictor used is a combination of local and global predictors. A choice predictor selects either the local or global prediction using a 32 entry table of 2-bit saturating counters indexed by the lower 5 bits of the program counter (PC). The global predictor (gshare) uses a 32 entry table of 2-bit saturating counters indexed by the bitwise XOR of the lower 5 bits of the PC and 5 bits of global branch history. The local predictor stores 5 bits of local branch history which is used to index into a 32 entry pattern history table containing 3-bit saturating counters.

These performance enhancing features affect the way the processor reacts to the non-deterministic memory timings described in Section 4.2. We will study the effects of the introduced non-determinism on the IVM processor in Section 4.3.

4.1.1 BackSpace Implementation

In order to run BackSpace on the IVM processor, the verilog design needed to be synthesized into a flat RTL file. This RTL file is then used to both generate the SAT clauses that correspond to the circuit and to build the simulator. The synthesis was done using Synopsys Design Compiler version Z-2007.03-SP5 for amd64 and the BackSpace library [12]. The BackSpace library contains basic logic gates and flip-flops, and the synthesized processor is constructed from these components. The synthesized result is shown as the IVM pipeline in Figure 4.1. The data
memory and instruction cache were not synthesized. These components, the synthesized IVM pipeline, and the behavioural verilog code that implements breakpoints, signature collection, and scan were compiled using Synopsys vcs Y-2006.6. The vcs compiled simulator reads the program and places it directly into the instruction cache.

4.2 Non-deterministic Memory Access

In this section, we will discuss the non-deterministic behaviour added to the simulation environment described in Section 4.1 in order to study its effect on the IVM processor and BackSpace. In Section 2.5 we discussed a proposal, CADRE [43], to remove non-determinism from a computer system. One of the sources of non-determinism they discussed was source-synchronous buses that cross clock domains. An example of this is the Pentium 4 processor which has two phase-locked
loops (PLLs), one for the core clocks and one for the I/O clocks. Because of the multiple PLLs, synchronization is required at the clock domain crossing [32]. Although Sarangi et al. [43] show a method of removing this non-determinism by delaying messages after they have been transmitted until the latest possible time the message could have arrived, it may not be practical to enforce completely deterministic buses as a first-order design constraint. There is also extra memory latency introduced by their solution.

4.2.1 Implementation

Non-deterministic memory access was implemented in the data memory block shown in Figure 4.1. The data memory was not synthesized, and it models memory latency using a queue of read requests from the data cache. To model a clock domain crossing, each request has a probability of being stalled when it reaches the head of the queue. To approximate an unknown phase relationship between the two clock domains, half of the requests are stalled at random. Note that store operations that miss in the cache will also experience variable memory latency because the cache line containing the address to be written to will be loaded into the cache before the write occurs (i.e., a write allocate date cache is used).

4.2.2 Other Forms of Non-determinism

There are other forms of non-determinism that could potentially be modelled. Examples include uninitialized state, which could be removed by setting all state bits into a known state at reset. This may involve routing extra reset signals to arbiters that would otherwise not need to be reset. Non-determinism is also caused by the I/O devices connected to the processor. For example, the response time of a hard drive depends on what position the actuator arm is in. This non-determinism could be removed by logging and replaying the communication with I/O devices.

4.3 Effects of Non-determinism

In this section we will evaluate the effect that the non-determinism introduced has on the processor being modelled. One reason this was done was to ensure that the non-determinism introduced is sufficient to impact the execution of the processor. Another reason to study non-determinism is to
Figure 4.2: An example where comparing two traces cycle by cycle will overestimate the difference between them. In (a) the circuit transitions to $s_8$ after some external event occurs. In (b) this event is delayed by one cycle, so the transition to state $s_8$ happens one cycle later. This causes the state to be different at cycle $m$ for all $m \geq 2$.

gain insight into how different parts of the processor are impacted and how non-determinism can be tolerated in future post-silicon debug methods.

The methods used to quantify the non-determinism introduced are explained in the section below. Results for three different metrics are then presented and analyzed.

### 4.3.1 Methodology

The simplest method to evaluate the non-determinism is to run the same program on the same processor multiple times but with potentially different non-deterministic events and compare the states at a particular cycle $m$. However, using this method traces that are very similar but differ only in timing by a single cycle would be measured as being very different. An example of this is shown in Figure 4.2. A better method would be to compare states that correspond to each other. For example, we could compare the states where the $i$th instruction is committed. Although the cycles where no instruction is committed would not have a corresponding state in another trace, this method should provide a reasonable metric for processors that can only commit one instruction per cycle.

The superscalar processor we are using, however, can commit more than one instruction per cycle. This means that there may not be a corresponding state to an instruction (or set of instructions) being committed. Fortunately, the processor we are using can only commit one branch instruction per cycle. That means that we can use branch commits to identify states that correspond to the same
point in program execution with different non-deterministic inputs.

### 4.3.2 Results

Three metrics were used to measure the effect of non-determinism on the IVM processor: the number of unique states, the number of bits that vary, and the average hamming distance. The results for each metric are explained below.

For the first metric the number of unique states seen across all traces at a branch commit was measured and plotted against the number of branches committed since the program started. Data was collected for 2000 runs of a program computing Fibonacci numbers and the results are shown in Figure 4.3. We see that as the program runs the number of unique states approaches the number of traces collected.

The second metric is how many bits differed (i.e., was a 1 in at least one run and a 0 in at least one run) across all 2000 traces at each branch commit. This is equivalent to the total number of
state bits (88796) minus the number of bits that stayed the same across all the traces. This metric is shown in Figure 4.4. The horizontal line at 16987 bits shows the number of bits that changed at some point during the execution of the program in at least one of the traces. It can be thought of as an upper bound to how many bits can be different between any two states. The average number of bits that differed is 3999.75 across the 40 branch commits. To gain further insight into which parts of the processor are affected by the non-determinism the number of bits that differed in each component of the processor was determined. The components of the IVM processor and their sizes are listed in Table 4.2. How many bits of each component differed at branch commit is shown in Figure 4.5. We see that between branches 13 and 14, the non-determinism reaches every part of the processor. Investigating further, it was found that the processor is stalled on some runs between branches 14 and 39. Comparing two branch commit states, one that stalled and one that did not,
Table 4.2: Bits of storage in each component of the IVM processor.

<table>
<thead>
<tr>
<th>Component</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>15842</td>
</tr>
<tr>
<td>Decode</td>
<td>728</td>
</tr>
<tr>
<td>Rename</td>
<td>1791</td>
</tr>
<tr>
<td>Issue</td>
<td>9508</td>
</tr>
<tr>
<td>RegRead</td>
<td>7884</td>
</tr>
<tr>
<td>Execute</td>
<td>5302</td>
</tr>
<tr>
<td>Memory</td>
<td>4547</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>27227</td>
</tr>
<tr>
<td>Retire</td>
<td>15967</td>
</tr>
</tbody>
</table>

the processor state that did not stall should be ahead of the one that did stall at every component so it should not be surprising that there are bits that differ across the entire processor. Note that since the architectural state is the same at two corresponding branch commits the differences are in the microarchitectural state.

For the third metric, a trace is arbitrarily picked as the reference and at each branch commit the Hamming distance between the branch commit state in all the other traces and the reference trace is computed. Five references were picked arbitrarily and their average distance to the other 1999 traces is shown in Figure 4.6.

4.3.3 Analysis

The number of unique states in Figure 4.3 suggests that we are very unlikely to see the same state again at a particular branch soon after a program begins. Even if there was only one non-unique state, we can see that the amount of non-determinism introduced will slow down BackSpace substantially.

However, from Figure 4.4 the number of bits that are both 1 and 0 at the same branch commit says that over 90% of the state remains the same at each branch commit. If we take into account the total number of bits that change state (among all branch commits and across all 2000 runs) we still see that less than half of the bits vary at any one branch commit.

If the specific bits that vary in the branch commit states between runs were random, then the average difference between an arbitrary reference and the rest of the states would be half of the bits
that vary. We see that the average distance for 5 references is 46% of what would be expected if it were random. From this data, we can conclude that the states (even though many are unique) are “close” to each other in Hamming distance.

The effects of non-determinism on the performance of BackSpace will be evaluated in Chapter 5. New post-silicon trace generation algorithms are also proposed, and their performance under non-deterministic conditions is also evaluated.

Figure 4.5: A breakdown the bits that vary at each branch commit by component.
Figure 4.6: Plot showing how many bits are different on average between different runs of the processor at branch commit points. The average Hamming distance between five different reference runs and the 1999 other runs is shown.
Chapter 5

Performance Optimizations to BackSpace

This chapter introduces some performance optimizations to the BackSpace algorithm. We start by discussing the factors that limit the performance of BackSpace. Then trace generation algorithms that reduce the effect of these limitations are described.

5.1 Performance Limitations of BackSpace

In this section we will discuss some factors that limit the performance of BackSpace.

5.1.1 Overwritten Registers

One of the concerns with BackSpace is the worst case number of states in the pre-image with a given signature. De Paula et al. [14] pointed out that the pre-image can include all possible states if the current state is a reset. Even if we are not interested in generating a trace that goes past resets, a very similar situation occurs whenever a register gets overwritten. Suppose register X contains the value $0xFFFFFFFFFFFFFFFD$ in state A and $0x000000DEADBEEF$ in state B. Register X could be a processor register (instruction set architecture (ISA) defined or a renamed register), buffer, status register, or any other type of hardware register. Assume we are currently stopped at state B and that the signature includes all the state bits except for register X. With the baseline BackSpace algorithm
we would then compute all the possible predecessors of B one at a time. If we have no additional information about the value of register X in state A, for example from state B or the known bits of state A, then there will be $2^{64}$ states in the pre-image as shown in Figure 5.1. Each state corresponds to a possible value of register X. The number of state bits in registers that can be overwritten in the IVM processor introduced in Chapter 4 is shown in Table 5.1. These numbers underestimate the number of bits that could potentially be overwritten in a single cycle. For example, parts of the pipeline registers and re-order buffer may be overwritten.

In order to prevent this type of pre-image state explosion the contents of the overwritten register must be recorded. It would be impractical to record the contents of every register in a complex circuit such as an out-of-order processor. An alternative is possible if the registers are organized into register files that are implemented as a static random access memory (SRAM) block with a limited number of write ports. As a new value is written into a register the old value can be stored as part of the signature. The main disadvantage of writing the old value is the difference in timing
Table 5.1: The number of state bits that can be overwritten.

<table>
<thead>
<tr>
<th>Register File Name</th>
<th>Total Bits</th>
<th># of Registers</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Processor Registers</td>
<td>5120</td>
<td>80</td>
<td>64</td>
</tr>
<tr>
<td>Store Queue Address Registers</td>
<td>1024</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>Load Queue Address Registers</td>
<td>1024</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>Architectural RAT</td>
<td>224</td>
<td>32</td>
<td>7</td>
</tr>
<tr>
<td>Speculative RAT</td>
<td>224</td>
<td>32</td>
<td>7</td>
</tr>
<tr>
<td>Fetch PC</td>
<td>64</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>Flush PC</td>
<td>64</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>7744</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

caused by the latency added to a register write. If we require the timing of register writes to remain the same when constructing the trace, then another solution is required.

5.1.2 Non-determinism

One of the difficulties faced by any post-silicon debug method is non-determinism. The non-determinism introduced into the simulation environment used in this chapter was discussed in Section 4.2. Under deterministic conditions, BackSpace would never need to load the same pre-image state into the breakpoint circuit more than once. However, if the execution of the chip is non-deterministic, the same state may need to be used as the breakpoint multiple times. This means that the expected number of times the chip must be run is proportional to the product of pre-image size and expected number of runs to reach the earliest state in the trace. Another consequence is that sorting the pre-image as proposed in Chapter 3 is of limited benefit unless it is known exactly how likely each state in the pre-image will occur. Without this information, only knowing which states are relatively more likely, each state in the pre-image should be loaded as the breakpoint \( n \) times before the most likely previous state is loaded for the \((n + 1)\)th time. Note that with deterministic execution the same pre-image state will never need to be loaded more than once and sorting the pre-image will reduce the number of processor runs needed by a fraction of the pre-image size (which is the maximum possible number of runs needed). But with the non-determinism introduced in Chapter 4 the maximum possible number of runs needed is much higher while the reduction gained

\footnote{Even if we added an extra read port, it would still take time to read out the old value before we could write the new value into the same register}
from sorting the pre-image is still a fraction of the pre-image size.

There are two main limitations to the performance of BackSpace: the requirement of loading states from the pre-image into the breakpoint circuit one at a time and non-deterministic execution. In the next section, we describe how the first limitation may be overcome.

5.2 Hardware Overhead Reduction

In Section 5.1.1 we discussed the problems that BackSpace has recovering the value of overwritten registers. We propose a trace generation algorithm that avoids this problem by removing the requirement to compute the previous state, or more generally, the set of possible previous states. The algorithm proposed in this section builds on the partial breakpoint BackSpace algorithm described in [17] and summarized in Section 1.2.4.

The goal of this algorithm is to produce a trace with the same properties as a trace produced by BackSpace more quickly and with less hardware overhead. These properties [14] are: every state added to the trace (after the crash state) is reachable and is a predecessor of the last state added. To show that this algorithm produces a trace that satisfies these properties we will present it as a series of optimizations to the baseline BackSpace algorithm described in Section 1.2. We first describe in Section 5.2.1 a method to check all the states in the pre-image simultaneously. Then we show how to check if a state is in the pre-image without computing it in Section 5.2.2. We then apply an optimization in Section 5.2.3 to reduce the hardware overhead of the breakpoint circuit. Next, in Section 5.2.4 we reduce the size of the signature. The result of these optimizations, the BackSpace-2bp algorithm, is described in Section 5.2.5.

5.2.1 Check All States in Pre-image Simultaneously

If we have a situation like that described in Section 5.1.1 where there are $2^{64}$ states in the pre-image it would not be feasible to check each state in the pre-image one at a time by loading them one at a time into the breakpoint circuit. However, we could load a set of states containing the pre-image into the breakpoint circuit. This can be done by computing the subset of bits that are the same across all the states in the pre-image, loading that subset into the breakpoint circuit, and setting the other
bits to “don’t care”. We will refer to the subset of bits that are known as known\((P_{\text{sig}})\).

There are two problems that can occur. Gort [17] refers to these problems as spatial and temporal mismatches. Temporal mismatches occur when states earlier in execution also match known\((P_{\text{sig}})\). We resolve this problem by loading known\((P_{\text{sig}})\) into a circuit that counts how many times known\((P_{\text{sig}})\) matches before execution reaches the earliest state in the trace. After collecting this information we run the breakpoint circuit again and stop the circuit when known\((P_{\text{sig}})\) is matched the recorded number of times. This technique was first proposed by Kuan [28]. The other problem, a spatial mismatch, is that known\((P_{\text{sig}})\) and the count matches during a run that does not lead to the correct state. Let \(s_{i+1}'\) be the state that matches known\((P_{\text{sig}})\) and the count. Spatial mismatches can be resolved by scanning out \(s_{i+1}'\), then checking if it is in \(P_{\text{sig}}\) and running the chip again if it does not.

In Chapter 3 we saw that, even after intelligently sorting the pre-image, on average 21% of the pre-image must be tried before finding one that will trigger the breakpoint. By loading the entire pre-image into the breakpoint circuit the number of times the circuit must be run to find a predecessor state is no longer dependent on the size of the pre-image. However, \(s_{i+1}'\) must still be compared with every state in \(P_{\text{sig}}\).

### 5.2.2 Transition Check

Comparing \(s_{i+1}'\) with every state in \(P_{\text{sig}}\) can be time-consuming if \(P_{\text{sig}}\) is large. Fortunately, we can determine if a state is in \(P_{\text{sig}}\) without doing these comparisons. To see this is the case, first consider the pre-image with no bits in the signature. This is the set of all predecessor states \(P_{\text{all}}\). Next, consider the set of all states that match the signature \(S_{\text{sig}}\). Lemma 1 shows that \(P_{\text{sig}} = P_{\text{all}} \cap S_{\text{sig}}\). This means that instead of checking if \(s_{i+1}'\) is in \(P_{\text{sig}}\) we can check if it is in \(P_{\text{all}}\) and \(S_{\text{sig}}\). A state \(s_{i+1}'\) is in \(S_{\text{sig}}\) if its signature \(\text{Sig}(s_{i+1}')\) is equal to the signature of the previous state. Checking if a state is in \(P_{\text{all}}\) is equivalent to checking if that state is a predecessor of the current state.

To check if a state \(s_{i+1}'\) is in \(P_{\text{all}}\) we construct a SAT instance that checks for a transition from \(s_{i+1}'\) to \(s_i\). It is similar to the SAT instance used to generate the pre-image. The clauses that correspond to the combinational logic and the earliest state in the trace \(s_i\) are identical. Each bit in \(s_{i+1}'\) is also
The relationship between $P_{\text{all}}$, $S_{\text{partial}}$, and $P_{\text{sig}}$ when the signature is chosen to be the same as the partial breakpoint.

A SAT solver \cite{16} is then used to determine if this boolean formula has a solution. If it does, then there exists a transition between $s_{i+1}^\prime$ and $s_i$.

The relationship between $P_{\text{all}}$, $S_{\text{partial}}$, and $P_{\text{sig}}$ is shown in Figure 5.2. Note that there cannot be a state in $P_{\text{all}}$ and $S_{\text{partial}}$ that is not in $P_{\text{sig}}$.

**Lemma 1.** The pre-image with a signature is identical to the intersection of the set of all possible predecessor states and the states that match the signature.

$$P_{\text{sig}} = P_{\text{all}} \cap S_{\text{sig}}.$$  \hspace{1cm} (5.1)

**Proof.** We will define $P_{\text{sig}}$, $P_{\text{all}}$, and $S_{\text{sig}}$ in terms of the clauses that make up a SAT instance and then derive (5.1).

We first define the notation that $c_i(x) = \text{TRUE}$ if $x$ is a state that satisfies the boolean expression $c_i$. The set of solutions to $c_i$ then is $\text{sol}(c_i) = \{x | c_i(x) = \text{TRUE}\}$.

We will use the property that the set of solutions to the conjunction of $n$ clauses, $c_1, c_2, \ldots, c_n$, is equal to the intersection of the solutions to the individual clauses. This is derived by starting with
the definition of $\text{sol}(c_1 \land c_2 \land \cdots \land c_n)$ and applying the definitions of conjunction and intersection,

$$\text{sol}(c_1 \land c_2 \land \cdots \land c_n)$$

$$= \{ x \mid (c_1 \land c_2 \land \cdots \land c_n)(x) = \top \}$$

$$= \{ x \mid c_1(x) = \top \land c_2(x) = \top \land \cdots \land c_n(x) = \top \}$$

$$= \{ x \mid x \in \text{sol}(c_1) \land x \in \text{sol}(c_2) \land \cdots \land x \in \text{sol}(c_n) \}$$

$$= \text{sol}(c_1) \cap \text{sol}(c_2) \cap \cdots \cap \text{sol}(c_n). \quad (5.2)$$

Recall that the pre-image is the set of solutions to a boolean satisfiability problem expressed in CNF. Let $m$ be the number of clauses in a SAT instance from the circuit and the current state. The pre-image with no signature is

$$P_{\text{all}} = \text{sol}(c_1 \land c_2 \land c_3 \land \cdots \land c_m)$$

$$= \text{sol}(c_1) \cap \text{sol}(c_2) \cap \text{sol}(c_3) \cap \cdots \cap \text{sol}(c_m). \quad (5.3)$$

The signature records $\text{sig}$ bits of the previous state. This adds $\text{sig}$ clauses to the SAT instance. The pre-image with a signature is

$$P_{\text{sig}} = \text{sol}(c_1 \land c_2 \land c_3 \land \cdots \land c_m \land b_1 \land b_2 \land \cdots \land b_{\text{sig}}) \quad (5.4a)$$

$$= \text{sol}(c_1) \cap \text{sol}(c_2) \cap \text{sol}(c_3) \cap \cdots \cap \text{sol}(c_m) \cap \text{sol}(b_1) \cap \text{sol}(b_2) \cap \cdots \cap \text{sol}(b_{\text{sig}}). \quad (5.4b)$$

Since each signature clause $b_i$ is of the form “bit $i = 1$” or “bit $i = 0$”, the intersection of their solutions is the set $S_{\text{sig}}$ which contains all the states that match the signature. That is,

$$S_{\text{sig}} = \text{sol}(b_1) \cap \text{sol}(b_2) \cap \cdots \cap \text{sol}(b_{\text{sig}}). \quad (5.4c)$$

Substituting (5.3) and (5.4c) into (5.4b),

$$P_{\text{sig}} = P_{\text{all}} \cap S_{\text{sig}} \quad (5.4d)$$
5.2.3 Fixed Partial Breakpoint

To reduce hardware overhead Gort [17] proposed using a small fixed set of state bits for the partial breakpoint instead of all the state bits. The optimizations presented above can be modified to use the intersection of the set of bits common to all states in the pre-image and the set of state bits available in the breakpoint hardware, BP, as the partial breakpoint. The bits loaded are $BP \cap \text{known}(P_{\text{sig}})$ and the other bits are set to “don’t care”.

5.2.4 Optimized Signature

We would also like to eliminate the need to compute the pre-image before determining what bits to load into the fixed partial breakpoint circuit. This can be accomplished by including all the bits in the partial breakpoint in the signature so that

$$BP \subseteq \text{SIG}, \quad (5.5a)$$

where SIG is the set of bits in the signature. Note that the bits in the signature will be the same in all the states in $P_{\text{sig}}$, so

$$\text{SIG} \subseteq \text{known}(P_{\text{sig}}). \quad (5.5b)$$

From (5.5a) and (5.5b),

$$BP \subseteq \text{known}(P_{\text{sig}}) \quad (5.5c)$$

then

$$BP \cap \text{known}(P_{\text{sig}}) = BP \quad (5.5d)$$

so we do not need to compute the pre-image $P_{\text{sig}}$ before loading the breakpoint circuit.

Since some signature bits are not used in the breakpoint circuit we set $\text{SIG} = BP$ and minimize the hardware overhead of the signature.
5.2.5 BackSpace-2bp

The result of applying the optimizations discussed earlier in this section to the baseline BackSpace algorithm is an algorithm we will refer to as BackSpace-2bp. The idea behind this algorithm is to record a small signature of the previous state, and rerun the chip to get a count of how many times this signature is seen. Using the signature and count, the chip can then be stopped once cycle earlier in its execution. The high level overview of what information gets collected during one iteration is shown in Figure 5.3. We start with the signature $\text{Sig}(s_{i+1})$ and count $n(s_{i+1})$ for a state $s_{i+1}$ that occurs immediately before the earliest state in the trace as shown in (a). Note that at this stage, the state $s_{i+1}$ is not yet known, but the states $s_i, s_{i+1}, \ldots, s_0$ are known. Next, we will obtain the signature $\text{Sig}(s_{i+2})$ for the state immediately before $s_{i+1}$ as shown in (b). Finally as shown in (c), the number of times $\text{Sig}(s_{i+2})$ is matched on a trace leading to $s_{i+1}$ is obtained. We then verify that a transition exists between $s_{i+1}$ and $s_i$, and add $s_{i+1}$ to the trace. As shown in (d), we now have the necessary information to start a new iteration.

A detailed flowchart showing the algorithm for BackSpace-2bp is shown in Figure 5.4. After the chip under debug crashes (1 in Figure 5.4) the signature of the state before the crash ($\text{Sig}(s_1)$) is loaded into the partial breakpoint counter circuit (2). The chip is rerun (3) until it crashes in order to obtain a count of the number of times the recorded signature is seen before the crash. If the signature is not matched in the previous cycle (4) then the chip must be rerun again since the signature in the previous cycle is different than what was recorded previously. This occurs because of non-determinism. Otherwise, we can now scan out the crash state $s_0$ and the number of times we have seen the signature of $s_1$, $n(s_1)$, (5).

Since we now have the signature and count for that signature for a predecessor to the earliest state in the trace ($s_i$) we can now enter the main loop shown in the right block of Figure 5.4. The breakpoint circuit is loaded with the signature of state $s_{i+1}$ and the count for that signature (6) and the chip is rerun (7). If the chip was stopped for some reason other than by the breakpoint circuit, then the chip is rerun (8). Other reasons why the chip could stop include exceeding a timeout threshold, reaching the end of the program, or reaching a crash state. If the breakpoint circuit stops the chip (9), then the state ($s'_{i+1}$) is scanned out and a transition check between $s'_{i+1}$.
Figure 5.3: The trace is built up by iteratively collecting information about earlier states using BackSpace-2bp. The information shown below the states is known. Step (6) from Figure 5.4 corresponds to (a), (10) corresponds to (b), (13) corresponds to (c), and (15) corresponds to (d).
Figure 5.4: Flowchart showing the BackSpace-2bp algorithm.
to \(s_i\) is performed (9) as described in Section 5.2.2. If no transition exists, the chip is rerun. If a transition exists then the signature of the state before the chip stopped (\(\text{Sig}(s_{i+2})\)) is loaded into the partial breakpoint counting circuit (10). The chip is rerun (11) until this signature is seen in the cycle before the breakpoint is triggered (12). If this signature is not seen in the previous cycle, then the count obtained would not be for this signature. When the chip stops at \(\text{Sig}(s_{i+1}), n(s_{i+1})\) while counting occurrences of \(\text{Sig}(s_{i+2})\) we scan out the count \(n(s_{i+2})\) and the state \(s'_{i+1}\) the chip stopped at (13). The state \(s'_{i+1}\) is a candidate to be added to the trace. We then determine whether there is a transition from \(s'_{i+1}\) to the earliest state in the trace \(s_i\) (14). If there is no transition, the chip is rerun again (11). If a transition does exist, \(s'_{i+1}\) is added to the trace and \(i\) is incremented. If the trace is long enough (16) then the algorithm is complete (17). If the trace is not long enough another iteration of the loop is started at (6). In our evaluations the desired length of the trace was known before BackSpace-2bp was run. In practice, this algorithm may be run until the trace is long enough to see the initial occurrence of the problem being debugged.

The first transition check (9) may seem unnecessary since the state that matches \(\text{Sig}(s_{i+1}), n(s_{i+1})\) times, is not added to the trace until after the second transition check (14). However, if we have a situation like that shown in Figure 5.5, keeping the signature obtained in the first run that stops at the breakpoint (8) could impede forward progress. In this example we assume that we have already obtained the signature and count for state \(s_{i+1}\) which is the unique predecessor of \(s_i\).
However, there is another state $s_{i+1}^*$ that shares the same signature and count as $s_{i+1}$ but does not lead to the earliest state in the trace $s_i$. Note that $s_{i+1}$ is reachable due to non-determinism. Now, state $s_{i+1}^*$ has a predecessor with a different signature than $s_{i+1}$’s predecessor. Let us assume that in the first two runs, the partial breakpoint circuit stops the chip at $s_{i+1}^*$ and that in the first run (7 in Figure 5.4) the signature of the predecessor $s_x$, $\text{Sig}(s_x)$, is obtained. Assume we do not perform the transition check at (9). After the second run (12), the transition check (14) between $s_{i+1}^*$ and $s_i$ will fail. If we keep the signature $\text{Sig}(s_x)$ we will not be able to add any more states to the trace. If the chip stops at $s_{i+1}^*$ the transition check (14) will fail. If the chip stops at $s_{i+1}$ the counting circuit will not match in the previous cycle since $\text{Sig}(s_x) \neq \text{Sig}(s_{i+2})$ and the chip must be run again (12).

The problem described above occurs because a signature of a state was retained before knowing that that state is the predecessor of a state in the trace. We ensure that this does not happen by checking for a transition first. Using the same example shown in Figure 5.5, after the first run, where the chip stops at $s_{i+1}^*$, we check for a transition between $s_{i+1}^*$ and $s_i$. No transition will be found so the signature $\text{Sig}(s_x)$ will not be retained. When the chip stops at $s_{i+1}$ a transition will be found so $\text{Sig}(s_{i+2})$ will be kept as a signature.

### 5.3 Reducing Processor Runs

The algorithm presented in Section 5.2 may require twice the number of processor runs that the baseline BackSpace algorithm requires. This section explains why this is the case, and presents an optimization that reduces the required number of processor runs to that of an ideal version of BackSpace.

#### 5.3.1 Pipelining

Adding one state to the trace using BackSpace requires only one processor run if the system was completely deterministic and the signature is large enough that there is only one state in the pre-image. For BackSpace-2bp, the best case only requires that execution be deterministic. In that case, adding a state to the trace would require two processor runs. The first run is to obtain the signature for the state two cycles before the earliest state in the trace. The second run is to obtain
the number of times the signature (partial breakpoint) is matched. A timeline for these events is shown in Figure 5.6. State $s_i$ is the earliest state in the trace, and the partial breakpoint and count for a state $s_{i+1}$ is known. The verification that the states $s_{i+1}$, $s_{i+2}$, and $s_{i+3}$ are predecessors is not shown here and is assumed to always succeed.

The extra number of runs needed in the best case could be reduced by recording a signature and counting the number of matches of a signature in the same run. It is not possible to record and count the number of matches for the same signature in the same run. However, we can record the signature for the state $s_{i+3}$ while we record the number of matches for its successor state $s_{i+2}$, where $s_i$ is the earliest state in the trace. A timeline is shown in Figure 5.7.
Figure 5.7: This figure shows the timeline for the steps to add four states to the trace using BackSpace-3bp. Initially, state $s_i$ is the earliest state in the trace.
5.3.2 BackSpace-3bp

The algorithm described in Section 5.3.1 pipelines the collection of a signature and the counting of the number of times a signature matches. We will refer to this algorithm as BackSpace-3bp. A high level overview of the information collected during one iteration of BackSpace-3bp is shown in Figure 5.8. Before the iteration begins (a), the signatures of two states, $\text{Sig}(s_{i+2})$ and $\text{Sig}(s_{i+1})$, preceding the earliest state in the trace ($s_i$) are known, as well as a count $n(s_{i+1})$ of the number of times the signature of the state immediately preceding $s_i$ was seen. At stage (b) the state $s_{i+1}$, a count of the times the signature of $s_{i+2}$ was seen, and the signature for the state that occurred two cycles before $s_{i+1}$ are also known. Next, we verify that a transition exists between $s_{i+1}$ and $s_i$. If it does, we add $s_{i+1}$ to the trace. We now arrive at (c) where we have the necessary information to
start another iteration.

A detailed flowchart describing the operation of BackSpace-3bp is shown in Figure 5.9. The shaded block on the left shows the steps to obtain the crash state $s_0$, the signature for the state before it $\text{Sig}(s_1)$, the number of times $\text{Sig}(s_1)$ is matched $n(s_1)$, and the signature $\text{Sig}(s_{i+2})$ for a state two cycles before $s_0$. The algorithm begins when the chip crashes (1). The signature of the state before the crash is read (2) and loaded into the partial breakpoint counter circuit (3). The chip is then rerun (4) until $\text{Sig}(s_1)$ is matched in the last cycle (5). We then scan out the crash state $s_0$ (6) which is the trace collected so far. The count for the signature of $s_1$, $n(s_1)$, and $\text{Sig}(s_2)$ are also read. With this information the algorithm can enter its main loop, which is shown in the shaded block on the right of Figure 5.9.

As shown in part (a) of Figure 5.8 we start each iteration of the main loop with the signature for a state $\text{Sig}(s_{i+1})$, its match count $n(s_{i+1})$, and the signature for the previous state $\text{Sig}(s_{i+2})$. We begin by loading the breakpoint circuit (7) with the signature $\text{Sig}(s_{i+1})$ and its count $n(s_{i+1})$. The partial breakpoint counting circuit is loaded with the signature of the state that occurred two cycles before $s_i$ (8). The chip is then rerun (9) until it is stopped by the breakpoint circuit (10). If the chip is stopped for another reason, for example a timeout or a crash, the chip is rerun (9) again. Also, if $\text{Sig}(s_{i+2})$ was not matched in the previous cycle (11) then the count $n$ obtained is not for the signature of the state before the one where the chip is stopped, so the chip is also rerun (9) again.

If the chip was stopped by the breakpoint circuit and $\text{Sig}(s_{i+2})$ matched in the previous cycle, then the state of the chip is scanned out (12). This state is a candidate to be $s_{i+1}$ and is referred to as $s'_{i+1}$. The count for the signature of $s_{i+2}$, $n(s_{i+2})$, and $\text{Sig}(s_{i+3})$ are also read. This corresponds to part (b) of Figure 5.8. The existence of a transition from $s'_{i+1}$ to $s_i$ is determined (13) as described in Section 5.2.2. If a transition does not exist, the chip is rerun (9). Otherwise, if a transition does exist, the state $s'_{i+1}$ is added to the trace (14) as $s_{i+1}$. Also, $i$ is incremented. If the trace is long enough (15) the algorithm is complete (16). Otherwise, if the trace is not long enough, we have the information needed to start another iteration as shown in part (c) of Figure 5.8.
Figure 5.9: Flowchart describing the operation of BackSpace-3bp. State $s_i$ is the earliest state in the trace.
5.4 Hardware

In this section, the hardware needed to implement the BackSpace-2bp and BackSpace-3bp algorithms will be described and the storage overhead required will be discussed. We will start by describing the partial breakpoint circuit and the counting circuit which are common to BackSpace-2bp, BackSpace-3bp, and BackSpace with partial breakpoints. Next, we will show how to use these components to implement BackSpace-2bp and BackSpace-3bp. Finally, the storage overhead required by these implementations will be discussed.

These hardware components were not synthesized. They were implemented in behavioral verilog for our evaluations.

5.4.1 Partial Breakpoint Circuit

The partial breakpoint circuit shown in Figure 5.10 is similar to the one described in previous work [17]. When the load target signal is asserted, a target partial breakpoint and target counter value are loaded into two registers. As the chip is running the value of the partial breakpoint register is compared with the corresponding bits of the state. If they match, the counter is incremented. The
value of the counter is also compared to the value of the target counter register every cycle, and when they match the breakpoint signal is asserted. This breakpoint signal is used to stop the chip so that the current state of the chip can be scanned out.

5.4.2 Counting Circuit

The purpose of the counting circuit is to obtain the target counter value used by the partial breakpoint circuit. The counting circuit shown in Figure 5.11 is similar to one proposed by Kuan [28]. The implementation of partial breakpoints described by Gort [17] did not have a dedicated counting circuit, but instead had two circuits that either performed the function of the partial breakpoint circuit or the counting circuit depending on what mode they were in.

When the load signal is asserted in Figure 5.11 the partial breakpoint (i.e., the signature for BackSpace-2bp and BackSpace-3bp) that we wish to obtain the count for is loaded into the register. As the chip runs, the bits of the state that correspond to the partial breakpoint (signature) are compared with the value stored in the register, and if they match the counter is incremented in the next cycle. The reason why the counter is not updated immediately is that we do not wish to count a match that occurs in the cycle when the chip is stopped. The algorithms described in Sections 5.2.5
and §5.3.2 require knowing if the counter was updated in the previous cycle, so there is an output that indicated whether the previous cycle matched the register. After the count is obtained, the partial breakpoint circuit will need both the count and the corresponding partial breakpoint, so the counting circuit also outputs the value of the partial breakpoint register.

### 5.4.3 BackSpace-2bp Hardware

The hardware needed to implement the algorithm shown in Figure 5.4 is shown in Figure 5.12. The signature, created by the signature creation circuit described in Section 1.2.3, is connected to the inputs for the partial breakpoint circuit, counting circuit, and the signature collection circuit. The input for the signature creation circuit itself is the state of the device under debug. The signature collection circuit in Figure 5.12 stores the signature for the previous state. The Load A signal
in Figure 5.12 is asserted in step (1) of Figure 5.4 and causes the counting circuit to load the signature of the previous state into its register. When the Load B signal is asserted in step (6) it causes the partial breakpoint to load the new target signature and count from the counting circuit into its registers. The Match Last Cycle, Load A, and Load B signals are connected to the off-chip BackSpace Framework Figure 4.1. The state bits come from the circuit under debug.

### 5.4.4 BackSpace-3bp Hardware

The hardware needed to implement the algorithm shown in Figure 5.9 is shown in Figure 5.13. The signature, created by the signature creation circuit, is connected to the inputs for the partial breakpoint circuit, counting circuit, and the signature collection circuit. The signature collection circuit in Figure 5.13 stores the signature for the two previous states. In Figure 5.9 (7) the Load
signal is asserted which causes the partial breakpoint circuit to load in the new target signature $\text{Sig}(s_{i+1})$ and count $n(s_{i+1})$. In Figure 5.9(a) the same Load signal is asserted which causes the counting circuit to load in the new signature $\text{Sig}(s_{i+2})$ to count.

5.4.5 Hardware Overhead

In this section, we will consider the hardware overhead needed to implement the various post-silicon trace generation techniques discussed so far. The hardware described in Section 5.4.3 and Section 5.4.4 was not synthesized so the area overhead will not be discussed. We will instead focus on the amount of information that must be stored in hardware structures.

All the post-silicon trace generation techniques discussed so far require storing additional information in hardware. Part of the storage is needed for the breakpoint circuit. A full state (or a part of it) must be stored in a register in order to compare with the current state. This overhead can be reduced using the partial breakpoint technique discussed in Section 1.2.4, but then requires the use of a counting circuit. In this evaluation the breakpoint circuit has one partial breakpoint circuit and a dedicated counting circuit. A signature of the previous state must be stored for each algorithm. BackSpace-3bp also requires storing the signature of the state two cycles before the current one. For BackSpace, either with full breakpoints or partial breakpoints, the signature is chosen to consist of the registers that can be overwritten as discussed in Section 5.1.1. This signature is used since it is a lower bound on what a practical signature would contain.

Although 32 bit counters may be sufficient to count all the matches that occur during debug, 64 bit counters are used to be conservative. With a 10 GHz chip, if every state matched the target partial breakpoint, a 64 bit counter could count the matches for over 50 years.

The storage required for the hardware components of each of the post-silicon trace generation algorithms is shown in Table 5.2. BackSpace-3bp requires 99.5% fewer bits of storage than the original BackSpace algorithm and 94.4% fewer bits than BackSpace with partial breakpoints.
Table 5.2: The storage overhead required for different post-silicon trace generation algorithms. BS is the baseline BackSpace algorithm with a signature containing only registers that can be overwritten as described in Section 5.1.1. BS-partial is BackSpace using the partial breakpoints described in Section 5.2.3. BS-2bp is the BackSpace-2bp algorithm described in Section 5.2.5. BS-3bp is the BackSpace-3bp algorithm described in Section 5.3.2.

<table>
<thead>
<tr>
<th>Component</th>
<th>BS</th>
<th>BS-partial</th>
<th>BS-2bp</th>
<th>BS-3bp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoint Circuit</td>
<td>Target State</td>
<td>88796</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>Counter</td>
<td>n/a</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>Target Count</td>
<td>n/a</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Counting Circuit</td>
<td>Target State</td>
<td>n/a</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>Counter</td>
<td>n/a</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Signatures</td>
<td></td>
<td>7744</td>
<td>7744</td>
<td>64</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>96540</td>
<td>8064</td>
<td>384</td>
</tr>
</tbody>
</table>

Table 5.3: Average number of runs needed to add one state to the trace.

<table>
<thead>
<tr>
<th>Program</th>
<th>BS-ideal</th>
<th>BS-2bp</th>
<th>BS-3bp</th>
</tr>
</thead>
<tbody>
<tr>
<td>arith-add</td>
<td>10.64167</td>
<td>21.56443</td>
<td>10.68333</td>
</tr>
<tr>
<td>fib-20</td>
<td>72.44083</td>
<td>57.70610</td>
<td>55.92547</td>
</tr>
</tbody>
</table>

5.5 Results

The performance of the post-silicon trace generation algorithms discussed in this thesis are studied in this section. The average (mean) number of runs to add one state to the trace is the metric used to measure performance. To evaluate the performance of the different algorithms, traces from various programs were collected. These traces all end (i.e., has as initial crash state) at the end of program execution. Trace collection ends when a trace of 25 states has been collected, or the algorithm times out after 48 hours. For each algorithm, 100 traces with different random seeds were collected for each program. The average number of processor runs needed to add one state to the trace is shown in Table 5.3. The implementation of BackSpace shown here, BS-ideal, uses all the bits of the previous state as the signature. This means that it will always have a single state in the pre-image.

The average number of runs per state added is useful, however, the distribution of runs per state added is also interesting. Histograms showing the distribution of average runs needed to add a state to a trace of the arith-add program are shown in Figures 5.14, 5.15, and 5.16. There are 100
Figure 5.14: Histogram of the number of runs to add one state to the trace of \texttt{arith-add} using BS-ideal.

data points shown in each histogram.

Figure 5.14 and Figure 5.16 show the average run distributions using the BS-ideal and BackSpace-3bp algorithms respectively. As expected, these distributions are very similar. Note that these distributions are only similar because this implementation of BackSpace uses the entire previous state as its signature. This means that there will only be one state in the pre-image, and multiple runs of the processor are required only to handle non-determinism. Figure 5.15 shows the distribution for BackSpace-2bp. Compared to the distributions for BS-ideal and BackSpace-3bp the peak is at about twice the number of average runs as expected.

The distributions for the average number of runs to add a state to a trace of the \texttt{fib-20} program are much wider. For example, Figure 5.17 shows the distribution for BS-ideal. It does not show very much detail except for the one outlier. To show more detail, a logarithmic scale is used for the average number of runs in Figures 5.18, 5.19 and 5.20. These graphs show the average run distributions for the \texttt{fib-20} program using the BackSpace, BackSpace-2bp and BackSpace-3bp algorithms respectively. We again see that the peak for BackSpace-2bp (Figure 5.19) is at about
Figure 5.15: Histogram of the number of runs to add one state to the trace of \texttt{arith-add} using BackSpace-2bp.

Figure 5.16: Histogram of the number of runs to add one state to the trace of \texttt{arith-add} using BackSpace-3bp.
**Figure 5.17:** Histogram of the number of runs to add one state to the trace of \texttt{fib-20} using BS-ideal.

twice the average runs for BS-ideal and BackSpace-3bp (Figure 5.18 and Figure 5.20) as expected.

An interesting feature of all the histograms shown are the long tails. Some of the traces take orders of magnitude more runs to collect than others. One reason why this might happen is that some of the traces may start (end with a crash state) that does not occur very frequently. In the next chapter we will discuss a potential solution to that problem.
Figure 5.18: Histogram with a logarithmic scale of the number of runs to add one state to the trace of fib–20 using BS-ideal.

Figure 5.19: Histogram with a logarithmic scale of the number of runs to add one state to the trace of fib–20 using BackSpace-2bp.
Figure 5.20: Histogram with a logarithmic scale of the number of runs to add one state to the trace of fib-20 using BackSpace-3bp.
Chapter 6

Crash State Selection

In this chapter we will explore the effect of non-determinism on the theoretical performance of the various trace generation techniques discussed so far and then propose a method to reduce the number of runs needed to add a state to a trace by selecting “good” crash states.

6.1 Number of Runs per State

The metric we will be using to evaluate the effects of non-determinism is the expected number of runs to add one state to the trace. Figure 6.1 shows the tree of possible execution traces that can occur with two random events. For the BS-ideal algorithm each run of the chip allows one more state to be added to the trace as long as the same series of events occurs as the last run that successfully added a state to the trace.

Let us compute the expected number of runs needed to add the first (non-crash) state to a trace leading up to one of the crash states in the example shown in Figure 6.1:

\[
E(\text{runs}) = E(\text{runs}|A) \cdot P(A) + E(\text{runs}|B) \cdot P(B) + E(\text{runs}|C) \cdot P(C) + E(\text{runs}|D) \cdot P(D)
\]

\[
= \frac{1}{P(A)} \cdot P(A) + \frac{1}{P(B)} \cdot P(B) + \frac{1}{P(C)} \cdot P(C) + \frac{1}{P(D)} \cdot P(D)
\]

\[
= 4
\]

(6.1)

where \( E(\text{runs}|X) \) is the expected number of runs to reach state \( X_{n-1} \). We assume that \( 0 < p_1 < 1 \)
Figure 6.1: This graph shows the possible execution paths a circuit may take to reach one of the four crash states $A_n$, $B_n$, $C_n$, and $D_n$. The circuit is non-deterministic i.e., at certain states the next state is randomly chosen from two or more states.

and $0 < p_2 < 1$.

A surprising result from Equation 6.1 is that the expected number of runs to add one state to the trace is independent of $p_1$ and $p_2$. It is equal to the number of states immediately preceding a crash. The intuition for this result is that although some crash states may not occur very frequently, when they do occur, they require many runs of the chip to reach again precisely because they do not occur very frequently. We could quantify how “random” an event is by using the Shannon entropy [44] of a set of probabilities

$$H(p_1, p_2, \ldots, p_n) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

(6.2)

This means that non-determinism events that are not very “random” (i.e., having low Shannon entropy) have the same effect on the number of runs needed as non-determinism events that are very “random” as long as the number of possible outcomes is the same.

This means that events which happen infrequently ($p$ close to 0) have the same impact on the expected number of runs to add a state to the trace as non-deterministic events that are more likely to occur. One example of such an infrequent event is a soft error corrected by Error Correcting Code (ECC) memory. Because errors can occur and be corrected by ECC memory in every cycle the number of possible execution paths can grow very quickly. However, most of these execution
paths are very unlikely to occur. In the next section we will explore a method that exploits the fact that some crash states occur more frequently to reduce the expected number of runs needed to add one state to the trace.

In general, we are interested in the expected number of times the circuit must be run to stop the circuit at one state in a particular set of states (e.g., the set of states that occur immediately before the crash state). For the following derivation we will assume that the baseline BackSpace algorithm described in Section 1.2 with a signature that includes all the bits of the previous state, is used.

Let us consider what happens when state $B_{n-1}$ in Figure 6.2 is reached as a breakpoint. It has two possible predecessor states $S_3$ and $S_4$. However, when the circuit stops at $B_{n-1}$ it is known which of the two possible predecessor states actually occurred during that run. We will refer to these two possibilities as $S_3/B_{n-1}$ and $S_4/B_{n-1}$. De Paula et al. [14] refer to these as states in the augmented state machine.

So for a set of states $S = \{s_1, s_2 \ldots s_{n-1}, s_n\}$ the expected number of runs to add the next state to the trace given that a state in $S$ is reached is

$$E(\text{runs}|S) = \sum_{i=1}^{n} \left( \frac{1}{P(s_i|S)} \cdot \sum_j P(s_i/t_{ij}|S) \right)$$

$$= \sum_{i=1}^{n} \left( \frac{1}{P(s_i|S)} \cdot P(s_i|S) \right)$$

$$= \sum_{i=1}^{n} 1$$

$$= n \quad (6.3)$$

where $P(s_i|S)$ is the probability of reaching state $s_i$ given that a state in $S$ is reached, $t_{ij}$ represents a successor state of $s_i$, and $P(s_i/t_{ij}|S)$ is the probability of reaching the state $t_{ij}$ through state $s_i$ given that a state in $S$ is reached. So $E(\text{runs}|S) = n$ which is the number of states in $S$. Note that $P(s_i/t_{ij}|S) = P(s_i|S)$ means that the probability of reaching a state $s_i$ is equal to the sum of the probabilities of reaching each of its successor states through $s_i$ given that a state in $S$ is reached.
6.2 Good Crash States

The expected number of runs to add the next state to the trace would be minimized if we could pick a crash state $X$ with the highest probability of occurring $P(X)$. The simplest method would be to run the chip until it reaches a crash state a fixed number of times and then pick the crash state that occurred most frequently. However, it is not known a priori how many possible crash states there are or what the probability distribution of crash states is so it difficult to determine a reasonable number of times to run the chip to find a good crash state $X$.

Another method is to run the chip until the same crash state is seen $m$ times. In Section 6.1 $m = 1$ was used. It only makes sense to pick likely crash states if some crash states are more likely than others. This occurs when different state transitions from the same state occur with different probabilities or when different execution paths converge as shown in Figure 6.2.

In Figure 6.2 the different crash states $A, B, C, D$ occur with probability $\frac{1}{8}, \frac{3}{8}, \frac{3}{8}, \frac{1}{8}$ respectively. Equation 6.1 also applies to this case, and the expected number of runs to add the next state to the trace before the crash is 4. The probability of reaching state $A_n$ twice before reaching another crash state twice is 0.07153 which is equal to the probability of reaching state $D_n$ twice. The probability of reaching state $B_n$ or $C_n$ twice is 0.42847. So the expected number of runs to add the next state to
the trace if we select the first crash state reached twice is

\[ E(\text{runs, } m = 2) = 2 \cdot \frac{1}{P(A)} \cdot 0.07153 + 2 \cdot \frac{1}{P(B)} \cdot 0.42847 = 3.4297 \quad (6.4) \]

which is a decrease over the case where the first crash state reached is used. Selecting the crash state in this manner to build the trace from requires running the chip multiple times before constructing the trace. However, the number of runs required is bounded by \( n + 1 \) where \( n \) is the number of possible crash states.

### 6.3 Results

To evaluate the crash state selection algorithm described above, 500 traces of 5 states were collected at the end of the `arith-add` program for varying values of \( m \), where \( m \) is the number of times a crash state must be seen before trace collection starts. The case \( m = 0 \) corresponds to the baseline where no crash state selection is performed and the first crash state seen is used. The data is shown in Figure 6.3. Although the long tails shown in Figures 5.14, 5.15, and 5.16 cause the 95% confidence interval for the average number of runs to be quite large, it can be seen that it is not likely that the crash selection algorithm reduces the number of runs needed to add a state to the trace.

One possibility is that, although a likely crash state is selected, all the other states in the trace are not selected for in any way and the algorithm is trying to reach a state that occurs very rarely. Another possibility is that the crash states are equally likely to occur. Further analysis on the data discussed in Section 4.3.2 reveals that out of 2000 processor runs, no crash state occurred more than twice and 1978 crash states only occurred once. This analysis suggests that the probability of reaching different crash states is very similar. Part of the reason this is the case is that the probability of the two random events (stall or no stall) are equal as discussed in Section 4.2.1. If a form of non-determinism is introduced that has random events with different probabilities, some crash states may be more likely to occur than others. The techniques discussed in this chapter may become useful if some crash states occur more frequently than others.
Figure 6.3: Average number of runs needed versus the number of times the crash state is seen before trace collection is started. The error bars show the 95% confidence interval.
Chapter 7

Conclusions

In this chapter, a summary of the this thesis is presented and then some possibilities for future work are discussed.

7.1 Summary

In this thesis, the number of processor runs and hardware overhead needed to generate a post-silicon trace are reduced compared to the original BackSpace algorithm [14]. First, an improvement to the number of runs BackSpace needs is achieved by sorting the pre-image. Then, new post-silicon trace generation algorithms that produce traces with the same properties as those generated by BackSpace, but requiring less hardware overhead are proposed.

BackSpace uses formal analysis to generate a set of possible previous states that are evaluated one at a time to find the actual prior state. This set of states is referred to as the pre-image. In Chapter 3, an algorithm for intelligently sorting the states in the pre-image was presented. This algorithm keeps track of the correlation between all pairs of states bits. Using this correlation information and the bits of the previous state that are known (common among all the states in the pre-image) a prediction of the previous state is made. The states in the pre-image are then sorted by their Hamming distance from this prediction.

In Chapter 4, a superscalar out-of-order processor using the Alpha ISA is introduced. BackSpace was implemented on this complex processor to understand its limitations. Non-deterministic mem-
ory access timing was implemented to study the impact of non-determinism on BackSpace and the trace generation algorithms proposed in this thesis. The impact of the non-determinism introduced on the processor itself was also studied. Three metrics were used to evaluate the non-determinism: number of unique states, number of bits that are not constant among all runs, and the average Hamming distance between states. It was found that the amount of non-determinism grows as the processor is run.

In Chapter 5 new post-silicon trace generation algorithms are proposed that require substantially less storage overhead compared to BackSpace. We began by looking at the performance limitations of BackSpace and found that overwriting some values in registers leads to prohibitively large pre-images ($\sim 2^{64}$ states). A series of optimizations are then made to the BackSpace algorithm that culminate in removing the need to compute pre-images entirely. We refer to this algorithm as BackSpace-2bp. One downside to the BackSpace-2bp algorithm is that it requires two runs of the processor to add one state to the trace in the best case (deterministic execution) compared to one run for BS-ideal (the original BackSpace algorithm with deterministic execution and one state in the pre-image). One run is needed for each of two steps: obtain partial breakpoint and count partial breakpoint matches. The BackSpace-3bp algorithm is then introduced that pipelines the two steps. The hardware overhead and performance of these algorithms are then compared.

In Chapter 6 a technique for selecting a “good” crash state to generate the trace to is discussed.

### 7.2 Contributions

A summary of the contributions of this thesis are listed below.

- A method of sorting the states in the pre-image, using the correlation between state bits, is proposed. This reduces the number of times the chip must be run by up to 51% [29].

- The BackSpace algorithm was implemented for a superscalar out-of-order processor that uses the Alpha instruction set. Non-determinism was introduced in the form of variable memory latency and the effect of this non-determinism evaluated.

- An algorithm that requires storing 99.5% fewer bits than the baseline BackSpace algorithm is
proposed. Relative to the state of the art BackSpace algorithm, that uses partial breakpoints, the new algorithm stores 94.4% fewer bits. This algorithm produces a trace that has the same properties as a trace produced by the baseline BackSpace. The new algorithm also requires fewer runs of the chip being debugged relative to a practical implementation of BackSpace.

7.3 Future Work

Some possible directions for future work include improving the tolerance to non-determinism of post-silicon trace generations, reducing the time it takes to simulate the processor, faster SAT solving, and modelling delayed breakpoints.

7.3.1 Tolerating Non-determinism

In Section 4.3 the effects of non-determinism on the IVM processor was studied. By the end of a simple program almost all 2000 runs of the processor resulted in a unique state. The post-silicon trace algorithms discussed in this thesis attempt to generate a trace of what actually occurs on-chip by repeatedly running the processor. If the same state is rarely reached again these algorithms will require many runs of the processor to add a single state to the trace. However, we also saw that a substantial number of bits remain the same across all runs at a particular point in the program. Also, the average Hamming distance between states at the same point in the program is low. These two results suggest that although the same state is rarely seen again, states that correspond to certain points in the execution may be very similar.

If we could identify which part of a state is incorrect, we could allow all previous states that would lead to the same error to be added to the trace rather than just the particular state that occurred one cycle earlier. Another possibility is that instead of generating a trace of complete states, only those bits that differ between the actual processor and an instruction level simulator are tracked.

7.3.2 Reducing Simulation Time

Although the post-silicon trace generation algorithms discussed in this thesis are designed to be run on real chips, they require additional hardware to be added in order to be used. For the evaluations presented in this thesis, a simulation of a chip was used instead. In order to collect significant
amounts of data on the performance of the algorithms presented in this thesis, the simulated processor must be run many times. A run of the simulated IVM processor can take over one minute. As we saw in Section 3.3 thousands of runs may be required to add a single state to the trace.

One way to reduce the time it takes to simulate the IVM processor might be to use behavioral (non-synthesizable) memory blocks instead of flip-flops for the register files. In Section 5.1.1 we saw that a substantial portion of the state bits in the IVM processor are contained in register files. Simulation of these registers might be faster if they and the control logic needed to read and write into them did not need to be synthesized into flip-flops and basic gates. Another way to speed up the simulation would be to add commonly used logic blocks to the technology library. For example, the IVM processor contains many multiplexors and they all had to be synthesized into basic gates because the current BackSpace library only contains basic blocks and flip-flops.

### 7.3.3 Faster SAT Solving

Solving each SAT instance only requires a few seconds for each transition being tested. However, if a real chip was used instead of a simulator, then solving the SAT instances more quickly may be desired. It may be possible to take advantage of the fact that all the SAT instances are very similar. 97.6% of the clauses for each SAT instance remain constant for the IVM processor. Currently, these clauses are read from disk for every SAT instance (transition check or pre-image computation). Instead, these clauses could be read and pre-processed once for every design.

### 7.3.4 Delayed Breakpoints

In a large circuit it may not be possible to stop the circuit within a single clock cycle as we have assumed. Since the goal is to run the circuit at full speed, the solution considered by Gort [17] is to pipeline the state comparison and/or the breakpoint signal propagation. Gort devised a trace generation algorithm that uses a delayed breakpoint circuit with a BackSpace algorithm that uses the partial breakpoints described in Section 1.2.4. It may be necessary to modify the algorithms presented in this thesis to work with delayed breakpoints in order to use them with real processors.
Bibliography


[2] Revision guide for AMD family 10h processors, June 2010. → pages 1


[10] Janice Castro, David S. Jackson, and Jane Van Tassel. When the chips are down. TIME, 1994. → pages 1


