An Effective Verification Solution for Modern Microprocessors

by

Ilya Wagner

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2008

To my Mom and Dad

ACKNOWLEDGEMENTS

I would like to thank my advisor Professor Valeria Bertacco, who guided me on the path towards my degree. For every project, paper and presentation that we worked on together she provided priceless advice, ensuring that the work was performed on time and with the best possible quality. I am also infinitely grateful to her for all the patience and tenacity with which she helped me improve my writing and presentation skills. I feel very fortunate to have her as my academic advisor, teacher and mentor.

I would also like to thank my committee members: Professor Todd Austin, Professor Scott Mahlke and Professor Nilton Renno. Throughout the years of graduate school I've enjoyed working with Professor Austin personally on several projects and always appreciated his brilliant ideas and advise. Likewise, I'd like to thank Professor Nilton Renno who supervised the Mars Rover Design Team - an extracurricular project that have been a major part of my life for four years. Although I have not worked personally with Professor Mahlke, I have always respected his opinion and advise, and would like to thank him for providing valuable comments that made the completion of this dissertation possible. Additionally, I thank Mark Brehob, one of the best teachers and lecturers I met in my life; it was his classes that sparked my interest to microprocessor design in the last year of undergraduate school.

I am deeply grateful to the all the people who collaborated with me on several of the projects: Andrew DeOrio, Kai-hui Chang, Joseph Greathouse and David Ramos. I also would like to thank all of the fellow Computer Science and Engineering students, who did not formally work with me, yet have become my good friends: Steven Plaza, Smitha Shyam, Kypros Constantinides, Andreas Moustakas, Andrea Pellegrini, Jin Hu, Jeff Hao, David Papa, Jarrod Roy, David Roberts, Adam Bauserman, Debapriya Chatterjee, David Meisner, Shobana Sudhakar, Bashar Al-Rawi, James Anderson, Hector Garsia and Geoff Blake. To those of you who have graduated already and those who remain in the lab – I bid you farewell and the best of luck. Likewise, I thank my friends in Russia, who, despite of being far away, still remain in my heart.

Finally, I'd like to thank my entire family, especially my brother Dmitriy and my sisters Vera and Anna, who always stood by me. I'd like to specially thank my sister-in-law Vita, whose help, hospitality and culinary skills I enjoyed on so many occasions, while living in Ann Arbor. I wish her the best of luck with her Ph.D. research and dissertation. Last, but not least, completion of this work would not be possible without the support of my mother and father who always encouraged me to follow the path of learning and discovery, supported me in the times of failure and celebrated my successes.

PREFACE

Over the past four decades microprocessors have come to be a vital and inseparable part of the modern world, becoming the digital brain of numerous electronic devices and gadgets that make today's lifestyle possible. Processors are capable of performing computation at astonishingly high speeds and are extremely integrated, occupying only a few square centimeters of silicon die. However, this computational power comes at a price: the task of verifying a modern microprocessor and guaranteeing correctness of its operation is increasingly challenging, even for most established processor vendors. Always attempting to deliver higher performance to end-users, processor manufacturers are forced to design progressively more complex circuits and employ immense verification teams to eliminate critical design bugs in a timely manner. Unfortunately, too often size doesn't seem to matter in verification, as schedules continue to slip and microprocessors find their way to the marketplace with design errors.

This work describes a novel verification framework targeting specifically today's complex microprocessors. The scope of the work spans many levels of verification and different phases of the processor life-cycle, from validation of individual sub-modules to complete multi-core system, and from pre-silicon design verification to in-the-field hardware patching. In particular, our StressTest and MCjammer approaches enable efficient generation of high-quality tests at the pre-silicon level for individual cores and multi-core systems, respectively, using machine learning techniques and making the process as automatic as possible. On the other hand, Reversi and Dacota enable low cost validation in post-silicon, while delivering even higher coverage than pre-silicon techniques. Finally, the Field-repairable control logic (FRCL) and Caspar techniques allow designers to patch different classes of escaped errors in processors that are deployed in the field.

The integrated set of solutions that we introduce with this thesis empowers processor vendors to drastically shorten their development timeline and, at the same time, to deliver more reliable and correct systems to their customers at a lower cost. Altogether, this work has the potential to solve the long-standing challenge of guaranteeing the complete functional correctness of modern microprocessors.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

Introduction

Over the past four decades microprocessors have permeated our world, ushering in the digital age and enabling numerous technologies, without which today's life style would be all but impossible. Processors are microscopic circuits printed onto silicon dies and consisting of hundreds of millions of transistors and wires interconnecting them. What distinguishes microprocessors from other silicon circuits is their ability to execute arbitrary programs written in software. In other words, processors make digital devices programmable and flexible, so one device can efficiently perform various operations, depending on the program. In our every day activities we encounter and use these tiny devices hundreds of times, often without even realizing it. Processors allowed us to untether our phones from the wired network and enabled mobile communications, while their counterparts deployed by the phone companies made communications richer and much more reliable. Processors monitor the health of hospital patients, control airplanes, tally election votes and predict weather. And, of course, they power millions of personal computers of all shapes and sizes, as well as the infrastructure of the Internet, a vital and inseparable part of the modern life. The computational power of these devices grows every year at an astonishing pace: not long ago processors were only capable of executing just a few thousands operations per second, while today they can perform billions of complex computations per second. Moreover, in the last two years many chip companies have introduced *multi-core* processors, which can execute several programs concurrently, thereby multiplying the overall performance of computer-based systems.

However, to be so powerful, processors have become extremely complex systems, making the design and manufacturing of these devices a major challenge for the semiconductor industry. Companies such as Intel, IBM and AMD are forced to dedicate hundreds of engineers for years at a time to continue to advance microprocessor technology and deliver better performance to end-users. Moreover, as these designs grow in complexity, it becomes increasingly harder to *verify* them and ensure that they operate properly. Design houses report that today verification efforts significantly overweigh design activities, and that they often staff their teams with two verification engineers for each design engineer. We can only expect the situation to exacerbate with future performance demands, to the point that high-quality verification of processors will no longer be possible with traditional means. Unless the demands of verification of the modern processor are answered, chips released to the public and used all around us will become more and more unreliable.

The impact of bugs in production microprocessors can range widely from innocuous to devastating, for several reasons. For instance, it is possible for a computer system to become compromised, in terms of safety and security, because of a hardware bug. As a result, a system with a buggy processor becomes vulnerable to security attacks. Attacks of this type could be perpetrated even on systems running completely correct software, since they rely exclusively on an underlying hardware flaw. Moreover, bugs can have a disastrous financial impact on the company that designed the processor and may trigger a costly recall of faulty hardware, as was the case in a past Intel processor, or a significant delay in product release, similar to what happened with AMD's latest chip. The impact in both cases is estimated in billions of dollars, due to the large volume of defective components that a functional bug always entail. To prevent devastating errors from seeping into the released chips, a variety of techniques have been devised to detect them during system's design and manufacturing. Conceptually, these verification approaches can be divided into three families, based on where they intervene in a processor life-cycle: *pre-silicon, post-silicon* and *runtime verification solutions*.

Pre-silicon techniques are heavily deployed in the early stages of a processor's design, before any silicon prototype of the device is available, and can be classified as simulation-based or formal solutions. Simulation-based methods are the most common approaches to locate design errors in microprocessors. Random instruction sequences are generated and fed into a detailed software model of a chip, results are computed by simulating this model and then checked for correctness. This approach is used extensively in the industry, yet it suffers from a number of drawbacks. First, the simulation speed of the detailed software model is orders of magnitude slower than the actual processor's performance. Therefore, only relatively short test sequences can be checked in this phase; for instance, it is almost impossible to simulate an operating system boot sequence or the complete execution of an application. More importantly, simulation-based verification is a non-exhaustive process: the number of configurations and possible behaviors of a modern microprocessor is too large to allow for the entire system to be fully exercised in a reasonable time.

The other family of pre-silicon solutions, formal verification techniques, solves the non-exhaustive nature of simulation using sophisticated mathematical derivations to reason about the design. If all possible behaviors of the processor could be described with mathematical formulas, then it would be possible to prove the correctness of the device's operation as a theorem. In the best scenarios, it is possible to guarantee that a design will not exhibit a certain erroneous behavior, or that it will never produce a result that differs from a known-correct reference model. The primary drawback of formal techniques, however, is that they are far from capable of dealing with the complexity of modern designs, and thus their use is limited to only a few, small components within the processor.

After a microprocessor is sufficiently verified at the pre-silicon stage, a silicon prototype is manufactured and subjected to *post-silicon validation*, where tests can, for the first time, be executed on the actual hardware. The key advantage of post-silicon validation is that its raw performance enables significantly faster verification than pre-silicon software-based simulation, thus it could deliver much better correctness guarantees. Unfortunately, post-silicon validation is plagued by the problem of limited observability, because at this stage it is impossible to monitor the internal components of the hardware prototype. Therefore, errors cannot be detected until they generate an invalid result, or cause the system to hang. The limited observability leads to extremely involved debugging procedures, with the result that today post-silicon validation and debugging has become the single largest cost factor for processor companies such as Intel.

Due to the limitations of pre- and post-silicon verification, and shrinking timelines for product delivery, processor manufacturers have started to accept the fact that bugs do slip into production hardware and thus they are beginning to explore *runtime verification solutions* that can repair a device directly at the customer's site. "Patching" microprocessor bugs, however, is a non-trivial task, since the functionality of the chip is already embodied in microscopic transistors on the silicon die, and thus it cannot be easily modified at this point. To enable in-the-field patching, designers create special processor components dedicated to detecting erroneous behaviors and recovering from them. Runtime verification is currently in its early stage: a few techniques have been recently proposed by academic research, while problem-specific solutions are starting to appear in commercial products.

Contributions of this thesis. This work presents a family of novel verification techniques targeting specifically modern complex microprocessors. They can address all phases of processor verification, pre- and post-silicon, and can handle a wide variety of hierarchical levels, from individual sub-modules to large multi-core chips. They also share several common traits, including scalability to high complexity designs, and the usage of self-tuning techniques, which lead to minimize the engineering effort required for verification. Moreover, this work dramatically improves the performance of post-silicon verification solutions, unlocking the full performance potential of this approach. Our post-silicon techniques are also tailored to provide very high verification quality, while minimally affecting the processor's characteristics of performance and area. Finally, this work presents a methodology that allows to patch escaped bugs in manufactured components that are already deployed in the field at an extremely low cost, and can guarantee the correctness of execution even on buggy designs. Therefore, the framework introduced by this thesis has the potential to solve the daunting challenge of verifying a modern processor and to enable vendors to shorten their design timelines, while delivering more reliable and correct designs to the customers.

1.1 Microprocessor design and verification: a short tutorial

A traditional microprocessor's design and manufacturing flow (Figure 1.1) consists of a series of steps that considers a high-level description of processor operation (*specification*), refines and transforms it, and, finally, implements the specified functionalities on a piece of silicon die. After each step, the design is progressively verified, to ensure that after all transformation and concretization steps the behavior of the device still adheres to the original specification. The process starts with a high-level specification of the microprocessor's required characteristics and functionalities, often described in natural language, and/or diagrams describing its basic structure and how the device should interface to other digital systems. This specification is then converted to an *architectural model* of the device, often written in a high-level programming language (such as C). This model represents the first formalized reference of the final system's behavior. Implementation in a hardware description language (HDL) can then start. The HDL description of the design describes the operation of individual sub-modules of the processor, as well as their interactions, and is also known as the *register-transfer level* (RTL) model of the chip. This RTL model is then verified to establish its equivalence to the architectural model through simulation-based and formal techniques. The outcome of simulation-based tests is compared to those of the known-correct (or "golden") architectural model and discrepancies, indicators of errors, are identified and fixed. In addition to simulation-based techniques, in *pre-silicon phase* of verification, engineers often employ formal methods, which can check correctness of a design using mathematical proofs and can thus guarantee the absence of certain types of errors. Unfortunately, formal methods cannot handle complex RTL models due to their limited scalability, thus their usage in pre-silicon verification is limited to a few, small critical blocks.

Once the RTL model is sufficiently validated, designers use synthesis software that maps HDL into individual logic gates, registers and wires, generating a *netlist* of the cir-

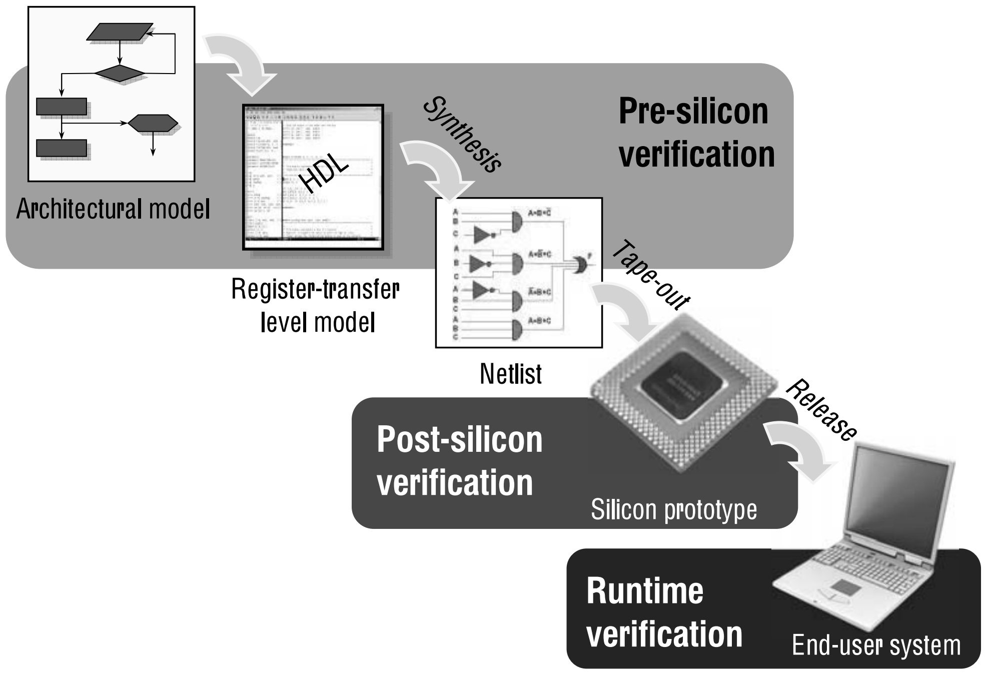


Figure 1.1: Modern microprocessor design and verification flow. In the pre-silicon phase an architectural, written based on the original design specification, is converted into an RTL implementation in a hardware description language (HDL). The RTL model then can be synthesized, producing design netlist. Placing and routing software calculates how individual gates and connections of the netlist could be placed on a silicon die and from this description a prototype of the processor is manufactured. Once a hardware prototype becomes available it is subjected to post-silicon testing. When hardware becomes sufficiently stable, the chip is released to the market.

cuit. Since conversion from an RTL model to a netlist may incur errors, specialized verification solutions intervene again to check that this new transformation is still equivalent to the previous model. Routing and placing software applications then calculate how individual logic elements in the netlist can be placed on the silicon die to produce a design that fulfills the required characteristics of power, are, delay, *etc.* After placement, the final description of the design is *taped-out*, *i.e.,* sent to a fabrication facility to be manufactured. When the first hardware prototypes become available they can be inserted into a computer system for *post-silicon* validation (as opposed to the pre-silicon verification that occurs before a prototype is available). One of the distinguishing features of post-silicon verification is its high speed; it is the same speed as the final product, which is orders of magnitude higher than simulation speeds in the pre-silicon domain. Typically, at this stage engineers try to use the hardware in real life-like settings, such as booting operating systems, executing legacy programs, *etc.* The prototype is also subjected to additional random tests, in an attempt to create a diverse pool of scenarios where differences between the hardware and the architectural model can be identified to flag any remaining errors. When a bug is found at this stage, the RTL model is modified to correct the issue and the chip often must be manufactured again (this process is called *re-spin*).

A processor design usually goes through re-spin several times, as bugs are progressively found in manufactured prototypes. Ultimately, the design is stabilized and it can go into production. Unfortunately, due to the complexity of any modern processor, it is impossible to exhaustively verify its behavior neither at the pre-silicon nor at the post-silicon level, thus subtle bugs often slip through all validation stages. Until recently, if bugs were found in the end-user's hardware, manufacturers had no other choice but to recall the device. Today vendors are starting to develop measures to avoid such costly recalls and allow chips to be patched in the field. Researchers in academia have also proposed solutions to ensure correctness of processor operation with special on-die checkers. Patching- and checker-based techniques are cumulatively classified as *runtime verification* approaches.

1.2 Overview of this Thesis

The goal of this thesis is to provide a scalable solution to the challenges of modern processor verification. We address the verification problem comprehensively throughout chip's life and, specifically, we leverage division the various phases of a processor's life-cycle (Figure 1.2) to attack bugs at all levels: *i*) in pre-silicon, where designs exist as hardware descriptions and are verified primarily with simulation-based techniques, *ii*) in post-silicon, where hardware prototypes are available for testing; and *iii*) at runtime in components deployed in customer's systems.

In our pre-silicon solutions, we use the architectural description of the design as the golden model to verify the implementation of individual blocks, ranging in size from small components (*e.g.,* ALU, register file, a single pipeline stage) to entire processor cores. *StressTest* [76, 77, 79] is the best suited technique for this task, for it delivers high-coverage verification with minimal human involvement. This solution can be applied to various processor modules, including blocks with multiple parallel interfaces (*e.g.,* arbiters, network switches), which are becoming an integral part of modern multi-core chips. When individual modules are integrated into an RTL description of the full multi-core system, our *MCjammer* [73] solution can be invoked to validate communication between the cores and on-chip interconnect. Unfortunately, due to the large number of design behaviors, simulation-based verification cannot provide complete coverage, therefore, in spite of the best efforts of pre-silicon tools, some design behaviors would be left unchecked. Moreover, in real-life industrial settings, the verification timeline is often shrunk due to

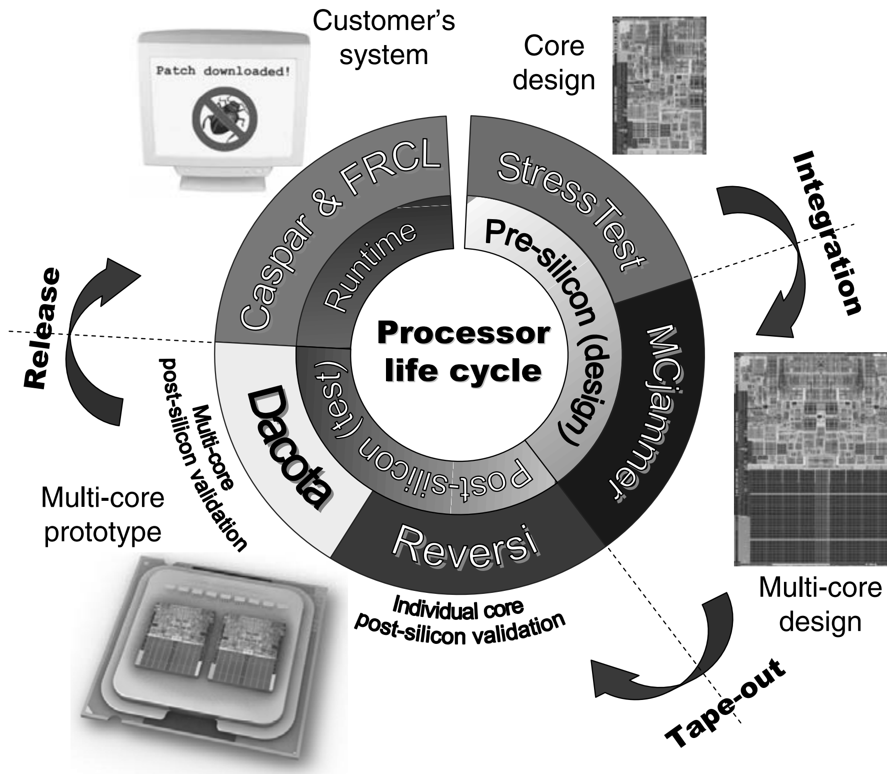


Figure 1.2: Verification throughout a processor's life-cycle. A processor's life-cycle can be divided into three phases: *pre-silicon verification*, *post-silicon validation* and *run-time verification* (hardware patching). The picture shows the phases of a processor's life-cycle in the inner circle, and our proposed verification solutions corresponding to each phase in the outer circle. Solutions presented in this thesis, enable efficient pre-silicon verification at different levels of the design's hierarchy: from individual sub-blocks and processor cores (using *StressTest*) to a full multi-core system (*MCjammer*). After the design is manufactured, cores are thoroughly validated with *Reversi* and, once the basic level of functionality of the cores is achieved, their communication is checked by *Dacota*. When sufficient level of coverage is reached the device can enter the production stage. If errors have escaped into production silicon, patching techniques (*FRCL, Semantic Guardians and Caspar*) allow for them to be corrected in the end user's system.

high time-to-market pressure. In these situations, processor vendors employing our approach may choose to augment the chip with additional small hardware components aimed for high-speed post-silicon validation, as well as hardware for patching escaped errors in manufactured devices.

After a design is sufficiently verified during the pre-silicon phase, a processor prototype is manufactured and subjected to thorough post-silicon validation. Recall that post-silicon validation can achieve orders of magnitude higher throughput and, therefore, can reach much higher coverage than simulation-based pre-silicon approaches. On the other hand, the drawback of post-silicon validation lies in its inability to monitor individual signals buried deep inside of the chip. The *Reversi* [74] and *Dacota* [26] solutions that we have developed to this end require access only to visible architectural state to detect and diagnose problems in individual cores and in the memory system of multi-core processors, respectively. More importantly, these solutions carry over the verification leveraging only minimal user guidance and exploit the inherent speed of the hardware prototype. Once an acceptable level of system coverage is reached, the device moves to production, where post-silicon validation features are disabled, while patching hardware is turned on. Patching, or runtime verification, allows vendors to remotely change the functionality of the processor in the third phase of its life-cycle, *i.e.,* when devices are deployed in end-customer's system. To achieve this, our techniques (FRCL [78, 80], Semantic Guardians [72] and Caspar [75]) augment the design with small bug detection components. When deployed in the field, this programmable detectors are capable of recognizing functional errors in processor operation and reconfiguring the chip to avoid buggy states, guaranteeing correctness of execution. Altogether, the integrated system of solutions described in this work creates a novel verification framework, which comprehensively addresses the challenge of validating a modern multi-core microprocessor.

1.3 Guiding Principles in Developing the Verification Solutions

This section outlines the common guiding principles that we adopted throughout our research work in investigating novel solutions.

1.3.1 Adherence to Traditional Flow

All techniques in this thesis were specifically developed to fit well in a traditional industrial verification flow. In particular, as mentioned above, formal approaches are not used in industry is very limited due to their limited scalability and requirement of a formal design specification. Therefore, the pre-silicon tools described here rely in their core on simulation, which can easily scale to handle the design sizes of modern processors and is well understood by processor manufacturers. Moreover, all of our solutions have modular structure and can be easily integrated into the simulation environment of developer's choice. Likewise, post-silicon solutions presented in this thesis fit well into traditional post-tape-out validation, and improve it by unlocking the full potential of hardware execution's performance. Finally, our runtime techniques rely on novel methods for hardware patching and can effectively utilize traditional patch distribution schemes, such as BIOS updates.

1.3.2 Minimization of Human Involvement

Another common aspect of all of these solutions, is that they try to minimize the effort demanded from verification engineers, a major bottleneck of traditional verification flows. Obtaining guidance from a human is very time consuming, and often automatic reasoning and learning techniques can achieve comparable results much faster and at a fraction of the cost.

Our framework uses several solutions that stem from machine learning: In StressTest we employ a technique, based on Markov models, that is able to evaluate the quality of test stimuli based on the stress observed in key design points and dynamically generate higher-quality sequences with reinforcement learning. Our approach to verification of multi-core processors (MCjammer) uses distributed intelligent agents that cooperate with each other trying to achieve their own verification goals and help other agents do the same. In MCjammer we also use closed feedback loops at fine granularity to direct tests towards interesting corner cases of processor behavior while trying to expose bugs.

Similarly to pre-silicon verification solutions, our approaches for post-silicon validation try to minimize the time a designer spends creating high-quality test stimuli and acquiring design's state to check for correctness. For example, Reversi, a post-silicon validation solution for individual processor cores, automatically produces test sequences with known correct final state. As a result, the validation process is dramatically simplified and streamlined, becoming virtually a push-button operation. Likewise, Dacota, a solution for post-silicon validation of cache-coherence and memory consistency uses inherent resources of the design under test to log and analyze memory accesses. Only when an error is detected, the engineer is alerted and must become engaged in the debugging activity.

1.3.3 Minimization of Area and Performance Overhead

In the post- and runtime verification approaches presented in this thesis we strived to minimize the hardware overhead to make these solution practical and attractive for processor designer companies. Since verification features do not add to the device's performance, manufacturers are less likely to deploy them in production hardware, preferring features increasing the throughput of the chip. Thus, to be adopted, the impact of validation and patching features on the processor area and performance must be minimal. To this end, the post-silicon techniques described in this work step aside from traditional state acquisition techniques, and attempt to either discover errors through an architectural state mismatch (as it is the case for Reversi) or use portions of on-chip caches to record information for error detection and analysis (as was done with Dacota). Likewise, runtime approaches

(Field-Repairable Control Logic, Semantic Guardians, and Caspar) incur minimal area overhead, trading off for it either error coverage or device's performance. The first one of these approaches, Field-Repairable Control Logic (FRCL), augments the device with a small programmable matching hardware that is pre-loaded at start-up with descriptions of the known bugs, making it possible to detect and bypass errors in the control logic of processor pipelines. An extension of this work introduces semantic guardians, combinational matcher circuits, synthesized automatically based on the results of design verification. In this case, the matching hardware encodes all of the verified configurations of the design and switches the processor to a safe mode when an un-verified, and therefore potentially buggy, configuration is encountered at runtime.

In the next six chapters of the thesis we uncover the proposed approaches following the flow of Figure 1.2. We start with verification of individual processor cores at pre-silicon level with StressTest in Chapter II, followed by post-silicon validation of such designs with Reversi. Chapter IV presents the field-repairable control logic solution for patching processors, as well as the extension of this solution to semantic guardians. We then review in detail the multi-core processor verification flow with MCjammer at pre-silicon and with Dacota at post-silicon level in Chapters V and VI respectively. An approach to memory subsystem patching in multi-core processors, called Caspar, is then detailed in Chapter VII. Finally, in Chapter VIII we briefly summarize the contributions of this thesis, draw conclusions, and outline future research directions.

CHAPTER II

Pre-silicon Verification of a Single Core

In this chapter we begin our journey through the pre-silicon verification solutions that are used in the early stages of the processor life-cycle to verify the adherence of the design model to the specification. In particular we will focus on the validation of single core processors, a type of design that has been developed for many decades, yet is still extremely difficult to verify due to the complexity of the computational core. To tackle the challenge of pre-silicon verification of single core microprocessors we introduce a novel test generator tool called StressTest. This solution uses a form of machine learning technique to dynamically obtain feedback from the design under test and progressively create better test sequences, which stress various design behaviors and expose complex bugs within a processor core. StressTest requires minimum interaction and control from the user, and it is easily fine-tuned and highly portable, since it considers the design under test at a very high abstraction level. Our approach only requires the engineering team to provide a simple *template* describing the interface protocol of the design. To assist the engineer in describing concise and meaningful programs, our template language includes a number of helpful features, including parameterized dependency variables. Using this template, StressTest generates a very broad spectrum of testing programs to verify the design. The underlying generation engine of StressTest uses a dynamically-adjusted *Markov model* representing the set of valid inputs for the design. This engine implicitly limits the generator to only produce valid test sequences, excluding the risk of false-negatives. Additionally, this approach combines advantages of both probabilistic and self-guiding stimulus generation techniques, which allows us to improve design coverage while lowering the overall verification effort. Finally, the template-based approach allows for a very compact representation, even for designs with complex input constraints, and increases the portability and flexibility of StressTest.

For guidance in the test generation, StressTest uses a novel technique based on *activity monitors*, which are simulation monitors that probe the internal state of the design at specific points. A first selection of probing nodes is made by the user, based on the key aspects or components which he wants to test. These nodes typically include key internal signals that reflect major changes in the design's state or outputs (for instance, a write-enable signal to the register file). StressTest complements this user selection with additional probing nodes highly related to the first ones so to achieve better performance and coverage. During simulation, StressTest observes the activity of all the probing nodes. Closed-loop feedback techniques are used to direct the test generator engine towards stimuli generating higher switching activity at the probing points. Unlike previous approaches, which usually adjust the stimuli generator based on results of a full, completed test run, StressTest is capable of adjusting the Markov model dynamically during the test. The use of activity monitors enables StressTest to gradually increase the stress on the automatically selected probing nodes and, subsequently, on the user-specified nodes themselves. We find that our approach achieves better coverage for complex bugs in fewer cycles than constrained open-loop random generation techniques. Moreover, the addition of automatically selected probing points is key in boosting StressTest's performance and in generating effective tests in an assertion-based verification methodology.

2.1 StressTest Structure

StressTest provides a convenient platform for specifying the set of valid inputs for a design under test by mean of templates. A number of activity monitors observe a small set of relevant circuit's internal signals and drive the generator toward scenarios that excite those signals. StressTest's self-guiding generation engine consists of two major components: a Markov model and a set of activity monitors (see Figure 2.1). The Markov model encapsulates the set of legal inputs of the design as well as probabilities of generating different sequences of inputs. The activity monitors bind to several key nodes, or signals, of the design and evaluate the "stress" on the design based on the switching activity of these signals. The information collected by the activity monitors is used as feedback to the Markov model, which adjusts the probability to generate stimuli which maximize the stress on these nodes. The signals selected for the activity monitors drive the focus of the test generation and, consequently, which components of the design will be most thoroughly tested. Therefore, in order to locate bugs within a critical component, activity monitors should be selected from key signals which activate the component, and within the component itself.

As with other RTGs, the stimuli generated by StressTest are supplied to both the design under test (DUT) and a golden model, which is a functional description of the design usually provided in a high-level language. The output and the architectural state of the two

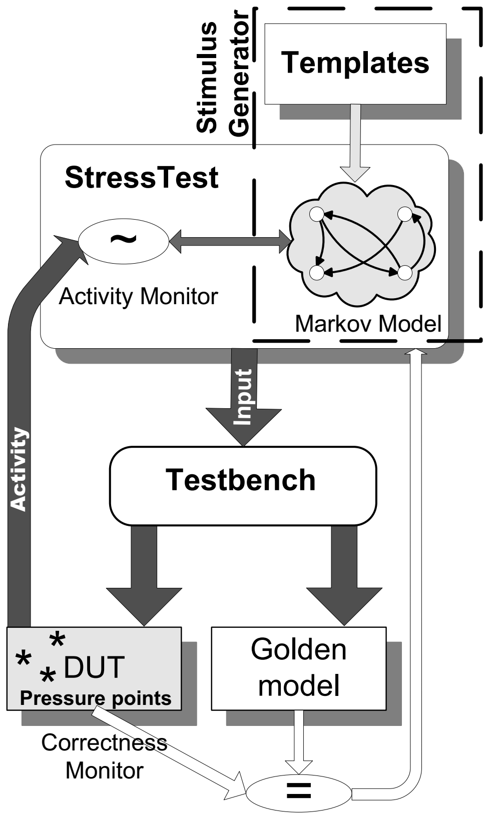


Figure 2.1: StressTest structure. Templates are used to create a Markov model which generates valid stimuli: templates' transactions are mapped to the vertices of the model. Stimuli are fed in parallel to the design under test (DUT) and the golden model. Activity monitors observe the behavior of the DUT during the simulation and adjust the Markov model improving the quality of the stimuli. The outputs of DUT and golden model are compared to expose bugs.

descriptions are monitored and discrepancies are flagged as potential design errors. For instance, in our tests on microprocessor pipelines, the golden model was a single-cycle functional model, and StressTest was connected to the external instruction bus interface of DUT and golden model. Hence, neither model could detect that the instructions were provided by a test generator instead of main memory. This greatly simplifies the test setup, allowing StressTest to reuse the framework of directed tests. When the correctness monitor detects a mismatch in the architectural states of the two descriptions, it halts simulation and outputs a trace leading to the problem.

2.2 Stimulus Generator

The stimulus generator engine of StressTest comprises an adaptive Markov model whose vertices describe monolithic blocks of stimuli, or transactions. The transactions corresponding to each vertex are generated based on the template files provided by the user. Template files are loaded by StressTest at the beginning of each simulation run and are used to generate the initial Markov Model. The probabilities associated with the model's edges are then adjusted over time based on the direction from the activity monitors. Several reasons led us to choose Markov models as the underlying engine for stimulus generation. First of all, Markov models are simple to implement, and straightforward to train; second, they provide a convenient abstraction for modeling instruction generation: Instructions (or classes of instructions) constitute the vertices of the Markov model, while programs correspond to paths in the model's graph. In contrast, structures such as Bayesian networks must be "trained" on an actual circuit to determine the cause/effect relation between inputs and observed outcomes; therefore, unlike Markov models, they depend on the circuit's implementation. On the other hand, a noticeable drawback of Markov models is their lack of "memory". Each generated stimulus is independent from past stimuli and relates only to the last vertex visited in the model. Therefore, we introduced the concept of dependency variables to provide deterministic dependencies between past stimuli and present input, as discussed later in this section.

2.2.1 Markov Model

StressTest uses a Markov Model as the main engine for generation of stimuli to the design. In general, a Markov model is a directed graph, where each edge has an associated probability of transitioning from the its source vertex to its sink vertex. In StressTest, the nodes of the Markov model corresponds to valid stimuli or groups of stimuli for the design. During the simulation, a stimulus associated with the current node is generated and supplied to both DUT and golden model. Within the framework of microprocessor verification, the stimuli corresponding to a single vertex can be individual instruction, or groups of instructions forming a program fragment. A single vertex can represent a range of similar individual instructions by parameterizing the various instruction fields. An example of a simple Markov model for an abstract instruction set architecture (ISA) is shown in Figure 2.2. The vertices in this case represent different types of assembly instructions, such as branches, arithmetic instructions, and so on. Edges are labeled with the probability of being traversed. For instance, the probability of generating a branch instruction after an arithmetic instruction is p_1 which is equal to 0.6. Note that the sum of the probabilities associated with the outgoing edges from each vertex must be equal to 1.

This Markov model-based approach is not limited to microprocessors only, and can be used to represent transactions through an interface of any digital circuit. In the experiments we used mutually disjoint and cumulatively exhaustive ISA partitioning, so that we could

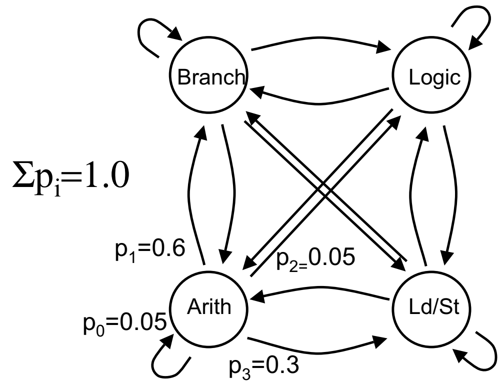


Figure 2.2: Example of Markov model. The model has 4 vertices, each capable of generating a particular instruction type. The sum of probabilities to transition from a single vertex must be 1.0. In the example, once an arithmetic instruction is generated, a branch has probability of 0.6 to occur, a load/store has probability 0.3, while other instructions are less probable.

generate every instruction type, increasing the possibility of catching a variety of design errors. In simulations aimed at exposing a specific kind of bug, the Markov model can be structured to contain only a small subset of inputs, representing distinct transactions. At the beginning of a simulation the model is a clique, with each transition being equally probable. A starting vertex is selected at random, the system produces the corresponding inputs and randomly selects a transition to the next vertex. During the simulation probabilities associated with the edges are adjusted based on the feedback from the activity monitors. StressTest allows for a sequence of instructions to be associated with a single vertex. When such a vertex is reached, StressTest generates instructions until it reaches the end of the sequence, and only then it transitions forward. This allows for generation of deterministic instruction sequences. For instance, a vertex could map to a sequence of a branch instruction followed by a *noop* to generate legal assembly code for a microarchitecture relying on the compiler to resolve control hazards. Notice that, since the entire sequence is clustered into a single vertex, all the activity observed when executing the sequence will affect the probability of reaching that sequence again.

The probability adjustment algorithm considers a weighted sum of the switching activity reported by the monitors and converts it to a value which we refer to as "score". The score is used to adjust the probability associated with the edge E traversed during the last transition. The update is computed as follows: the score is first scaled as a fraction of the maximum score that all edges could achieve; then the probability of the relevant edge P_E is adjusted; finally, the probability values on other edges starting from this vertex are adjusted, so that their sum is normalized to 1. The probability increment is then equal to:

$$P_{inc} = \frac{score_{monitor}}{score_{max} * N_{edges}} \tag{2.1}$$

where N_{edges} is the number of outgoing edges from the vertex under consideration, and $score_{max}$ is the maximum score achievable. The adjusted probability is the saturated sum:

$$P_{E\_new} = max\{T_{sat}, P_E + P_{inc}\} \tag{2.2}$$

where T_{sat} is the saturation threshold unique for the system. We set the saturation threshold T_{sat} to a value slightly less than 1, so that we can always attribute at least a small probability to any edge. Doing so guarantees that we never eliminate potentially useful transitions from the system. In most of our experiments we set $T_{sat} = 0.95$, allowing other edges to have probability in the range $[0.05/(N_{edges}-1), 0.95]$. If a vertex has an outgoing edge with probability T_{sat}, then all other edges are set at $P_{min} = (1 - T_{sat})/(N_{edges} - 1)$.

The last phase of the probability adjustment requires that the sum of the outgoing edges' probabilities is re-normalized. We do so by decrementing all eligible edges (the ones with $P_{edge}>P_{min}$) by an amount proportional to P_{inc}. Specifically, we first compute the "slack" available as

$$slack = \Sigma_{i \neq E}(P_i - P_{min}) \tag{2.3}$$

where the contributing edges include all but the one we just incremented (note that edges for which $P_i = P_{min}$ contribute 0). The adjustment is then performed by decrementing each P_i proportionally to their contribution to the "slack":

$$P_{i\_new} = P_i - \left(\frac{P_i - P_{min}}{slack}\right) * P_{inc} \qquad \forall i \neq E \tag{2.4}$$

This computation normalizes the probability while guaranteeing that each edge has probability $P_i \geq P_{min}$ associated with it.

2.2.2 Template Files

We use a special template language to describe the stimuli to generate at each vertex of the Markov model. StressTest's templates describe a short sequence of stimuli for each vertex, as shown in the example of Figure 2.3. Stimuli definitions are in binary format, and specify the values of each bit of the input signals. Values can be specified as 0, 1 or through a parametric field. Each bit field is identified by a single alphabetic character repeated for the whole width of the field. Note that the stimulus generator only describes legal inputs, requiring no information about the DUT structure, making template files highly portable among multiple designs. The user must only specify the desired partitioning of the input set and no explicit sequencing control is required. In addition, StressTest templates do not specify the biasing of input stimuli, since this is derived during the simulation.

This is different from industry tools, such as Genesys-Pro [3], where sequencing control and biasing information must be provided in the template to achieve high quality test sequences. The structure of StressTest templates is particularly suitable for describing instruction set architectures (ISA) and structured interface protocols.

```
immVal   (cacheSize=5,probCache=0.9,lambda=2,
          minVal=-8192,maxVal=8191);

destIndex(cacheSize=30,probCache=0.8,lambda=4,
          minVal=0,maxVal=31);

randIndex(probCache=0,lambda=0,minval=0,maxVal=31);
r-funcs(probCache=0,lambda=0,minVal=0,maxVal=63);

i-funcs  (probCache=1,lambda=0) =
{  'b001000 /ADDI/,  'b001001 /ADDIU/,
   'b001010 /SLTI/,  'b001011 /SLTIU/,
   'b001100 /ANDI/, 'b001101 /ORI/,
   'b001110 /XORI/ };

vertex(r-type-inst)
{  input = 'b000000ssssstttttddddd00000ffffff;
   field(s) = $destIndex.read();
   field(t) = $randIndex.read();
   field(d) = $randIndex.read();
   $destIndex.write(field(d));
   field(f) = $r-funcs.read(); }

vertex(i-type-inst)
{  input = 'bffffffssssstttttiiiiiiiiiiiiiiii;
   field(f) = $i-funcs.read();
   field(s) = $destIndex.read();
   field(t) = $randIndex.read();
   $destIndex.write(field(t));
   field(i) = $immVal.read(); }
```

Figure 2.3: Example of a StressTest template file. This file defines a Markov model with two vertices: *r-type-inst* (register type instructions) and *i-type-inst* (immediate type instructions). The vertices use five dependency variables declared at the top. In particular, variable *$dest-Index* creates a potential dependency between consecutive stimuli.

Template files can also force interactions between vertices, allowing StressTest to generate stimuli sequences with complex interdependencies. To describe these interactions, we have created a structure very flexible and easy to use, called dependency variables. Dependency variables, declared at the beginning of a template, generate values for the parametric fields, and include special support for specifying locality and dependency characteristics. These variables and their operation are detailed in Section 2.2.3. Note that edges do not appear anywhere in template files, since our Markov models always starts as a single clique. In designing the template language we strove to keep a simple and intuitive structure, since this is the only part of StressTest which requires user input. In spite of its simplistic structure, the template language retains the ability of describing a very broad range of stimuli, and different interaction constraints.

The template language allows to represent groups of stimuli through multiple vertices, each using different parameters. This enables StressTest to associate distinct probabilities to stimuli with different characteristics. For instance, arithmetic instructions could be highly dependent on previous instructions in one vertex, while nearly always independent in another, allowing the activity monitors to selectively adjust transitions to and from vertices with specific individual properties. An example of a template file is given in Figure 2.3. It defines five dependency variables, each with different sets of parameters. After the declarations, the template file contains the vertices' specifications, which correspond to nodes in the Markov model. In the example we define two vertices: *r-type-inst* and *i-type-inst*, each generating a 32-bit input stimulus. Below the bit structure definition, each of the fields is assigned a value from dependency variables. For instance, vertex *r-type-inst* binds field *f* to dependency variable *r-funcs*, while field *f* of the *i-type-inst* vertex is bound to variable *i-funcs*.

2.2.3 Dependency Variables

Dependency variables, such as those declared at the top of the template file in Figure 2.3, provide a concise mechanism to specify the generation of values from i) a list of constants, ii) a uniform random distribution, iii) a randomly generated locality set, or iv) some combination of the previous three constructs. The variables are used to pass information between the template's vertices and fields and create complex interactions, such as locality and dependency between generated inputs. All of the variables have global scope and can be accessed from anywhere in a template file. The functional block used to generate a value for a dependency variable is illustrated in Figure 2.4. The underlined labels in the figure represent the five declaration parameters of a variable:
probCache is the probability that a *read()* to a dependency variable will retrieve the value from the locality cache.
cacheSize is the size of the locality cache, which contains the most recently generated values that can be reused to simulate locality and dependency.
lambda represents the length of the locality window, and it corresponds to the rate parameter to an exponential distribution which selects the cache indexes: The larger the value of *lambda*, the greater the probability that the selection is skewed towards recent element inserted, the smaller this value, the more uniform the distribution of selection among all of the cached elements.

minVal, maxVal are the bounds over which the uniform random samples are produced when the cache is not used.

Some of the parameters can be omitted when declaring the variables, in which case a default value is used. For example, *minVal* is omitted when declaring variable *i-funcs* in Figure 2.3, hence the value of that parameter defaults to 0. When the variable is *read()*, the returned value is either taken from the sample cache with probability *probCache*, or produced by the random value generator with probability *1-probCache*. Thus, by setting the parameter *probCache* to 0 we can create a perfect random generator, as it is the case for variable *randIndex* in Figure 2.3. On the other hand, we can completely disable the random generator and select values exclusively from a pre-defined list, similar to case of variable *i-funcs*.

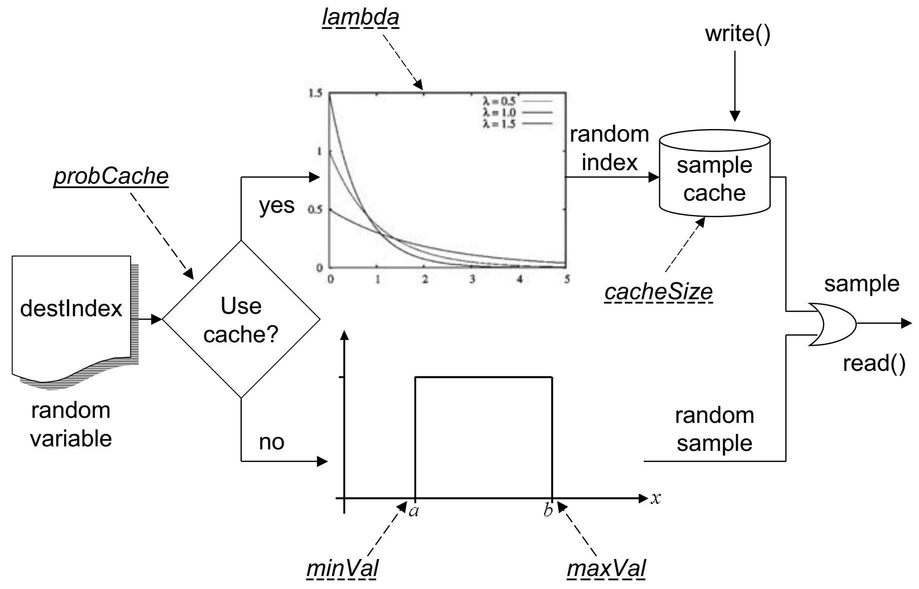


Figure 2.4: Dependency variable functional block. A *write()* access to the variable inserts a value in the cache. On a *read()*, the parameter *probCache* is used to select whether the value should be generated randomly or retrieved from the cache. In the latter case, the *lambda* parameter affects the distribution of the selection from the cache. In the former, the distribution is uniform between *minVal* and *maxVal*

When a value is taken from the sample cache, *lambda* is used as the parameter of an exponential distribution function to generate the index of the cache entry storing the value to return. Note that high *lambda* values correspond to a high probability of generating low indexes, and thus retrieving more recent data. When the value of *lambda* is small, cache reads are almost uniformly distributed among all entries. Thus, it is possible to control the level of locality in the sample cache by varying *lambda*. In the context os dependency variables, this mechanism can be used to control the degree of register dependence between generated instructions. For instance, in the example in Figure 2.3 there is a dependency between field *d* of *r-type-inst* and field *s* of *i-type-inst* through *destIndex*. Moreover, the probability distribution of the returned values allows to efficiently control the strength of the dependency between instructions which operate on the same register and, therefore, the "stress" on the control and forwarding logic of such architectures. The constructs of sample cache and parametric fields proved to be sufficient for representing any instruction in the ISAs that we evaluated, due to the natural breakdown of the instruction formats into fields which have a fixed number of legal values and fields which can assume any value within a range. This is especially true of any RISC ISAs, because of their structured approach to instruction formats. For these ISAs, a large number of instructions can be compactly represented with a simple template file, using just a positional field notation, hence minimizing setup time and verification effort.

2.2.4 Hierarchical Markov Models

In the basic StressTest structure described above, random input generation is based on a Markov model, which partitions the set of inputs of a microprocessor core. However, it is often the case that a design has multiple parallel input interfaces, unlike pipelines, which can be viewed as having only one stimuli entry point. A good example of such a design is a network switch or a crossbar that has multiple input ports. For this design it is critical to be able to generate stimuli at each individual port that are time- and data-independent. Therefore, it is often more desirable to generate multiple input streams in parallel and observe the interactions between them. Since often the hardest bugs in such designs are found when multiple input requests are competing for the same hardware resources, we deemed crucial to allow the designer to create sequences of input stimuli where the traffic at each interface is independent, so that it becomes easier to produce relevant tests.

To cope with this problem we extended the StressTest framework to employ a multi-level Markov model. Each of the parallel inputs of the circuit is assigned to an individual Markov model for generating valid input stimuli according to the interface specifications. However, some information between these models can be shared to increase the competition for the resources and intensify the pressure on the design. To allow this information passing between the models, we use a global variable space that is accessible from any model. Note that the Markov models can still generate valid sequences independently of their peers through local variables.

In addition to the individual Markov models assigned to input ports of the design, a global model is used to encode possible scenarios of simultaneous stimuli generation. For example, the model can determine the number of ports of the design activated simultaneously and values of the controlling inputs to the design. The objective of the global

Markov model is to coordinate local models assigned to individual design ports. Note that individual models supply sequences of stimuli with dependencies between them, while the global model orchestrates them to exert simultaneous pressure on the design. The activity feedback from individual inputs of the design in this framework is used to reinforce transitions in local models, while the combined activity from different points in the circuit is used for adjusting edges in the global Markov model.

/ Global variable space /
```
global {
  dest(probCache=0.7,cacheSize=8,lambda=.5,
                                  minVal=1,maxVal=15);
  srcW(probCache=1,cacheSize=20,lambda=3)={'b0000, 'b1111};
}
```

/ Global Markov model/
```
none : TopModel(global) {
  rand-send-one(probCache=1)={'b1000,'b0100,'b0010,'b0001};
  command[3:0] : { portD, portC, portB, portA };
  vertex(send_one_pkt) {  command = 'bCCCC;
                          field(C)=$rand-send-one.read(); }
  vertex(send_all) { command = 'b1111; } };
```

/ Local Markov models descriptions/
```
switchPort {
  using global::dest;
  using global::srcW;
  vertex (message) {  input='bDDDDSSSS;
                      field(D)=$dest.read();
                      field(S)=$srcW.read();
                      $dest.write(field(D)); } };
burstingPort {
  Bsrc  (probCache=0,minVal=0,maxVal=15);
  vertex (message){  input='b0100SSSS;
                     field(S)=$Bsrc.read();
                     $Bsrc.write(field(S)); } };
```

/ Local Markov models binding /
```
dut.port_a : switchPort(portA) ;
dut.port_b : burstingPort(portB) ;
dut.port_c : switchPort(portC) ;
dut.port_d : switchPort(portD) ;
```

Figure 2.5: Example of an extended StressTest template file. Shown are a global Markov model along with two local models bound to each port of a network switch. Note the differences with template shown in Figure 2.3: Global variable space is used to pass information between low-level models, top-level Markov model encodes execution scenarios (*send_one_pkt* and *send_all*), while low-level models encode individual messages.

An example of a hierarchical template file shown in Figure 2.5 indicates how hierarchical Markov models can be used to model different interfaces of a DUT. In the case shown in the example, the DUT is a network switch with four simultaneous interfaces *port_a* to *port_d*. The two input-generating models *switchPort* and *burstingPort* produce different kinds of data packets, while the global top level Markov model creates different scenarios and orchestrates the work of the packet generators. The messages described here consist of an eight-bit packet.

The top portion of the template contains the global variable space with two random variables, *dest* and *srcW*. They are visible to any of the local models that declare using them via a *using* clause, as shown in model *switchPort*. The global Markov model either runs a scenario where input is sent to only one port or where it is sent to all ports simultaneously, by signaling to the low level models with command bits. Model *switchPort* produces packets with source and destination fields generated by accessing the global variables, while model *burstingPort* produces all messages to destination 0100.

2.3 Activity Monitors and Feedback

This section presents the concept of activity monitors and describe how they are used to close the feedback loop. We also show how switching activity observed in the DUT can be related to transitions in the Markov model, and how multiple verification goals can be pursued simultaneously using this concept. Finally, we present an approach for the automatic extraction of additional relevant activity signals which allow to increase the quality of the tests generated.

2.3.1 Activity Monitors

In designing StressTest, we made an assumption that many bugs arise from complex interactions between instructions, which create high activity at the interface between design units and control blocks. Thus, we are more likely to find bugs by generating patterns that cause a high level of activity in those signals, rather than by simulating random input sequences. The activity monitors are responsible for identifying interesting input sequences which occurred by chance, and for reinforcing the appropriate transition edges, so that those sequences may occur more often. Figure 2.6.a illustrates the approach on one of the microprocessor cores that we targeted in our experiments. The Markov model sends stimuli, in the form of instructions, to the DUT. At each cycle, the activity monitors assess the control signals in the DUT for specific activities of interest. In particular, they monitor a set of "pressure points", that is, user-specified, relevant nodes within the design. A monitor for a single bit pressure point simply tracks its switching activity by counting the number of transitions occurred at the node due to the past stimulus. A monitor for a vector signal computes the sum of the single bit transitions and scales it by the vector's width. We found useful to have the possibility to control a monitor through an enable signal. For instance, in our experiments, the activity monitor for the memory interface would only be enabled if a memory transaction was executed during a cycle. The activity monitors let us accurately re-create a range of real world "stresses", such as physical register file pressure, recurring dependent instructions in the register renamer, or high cache miss rates.

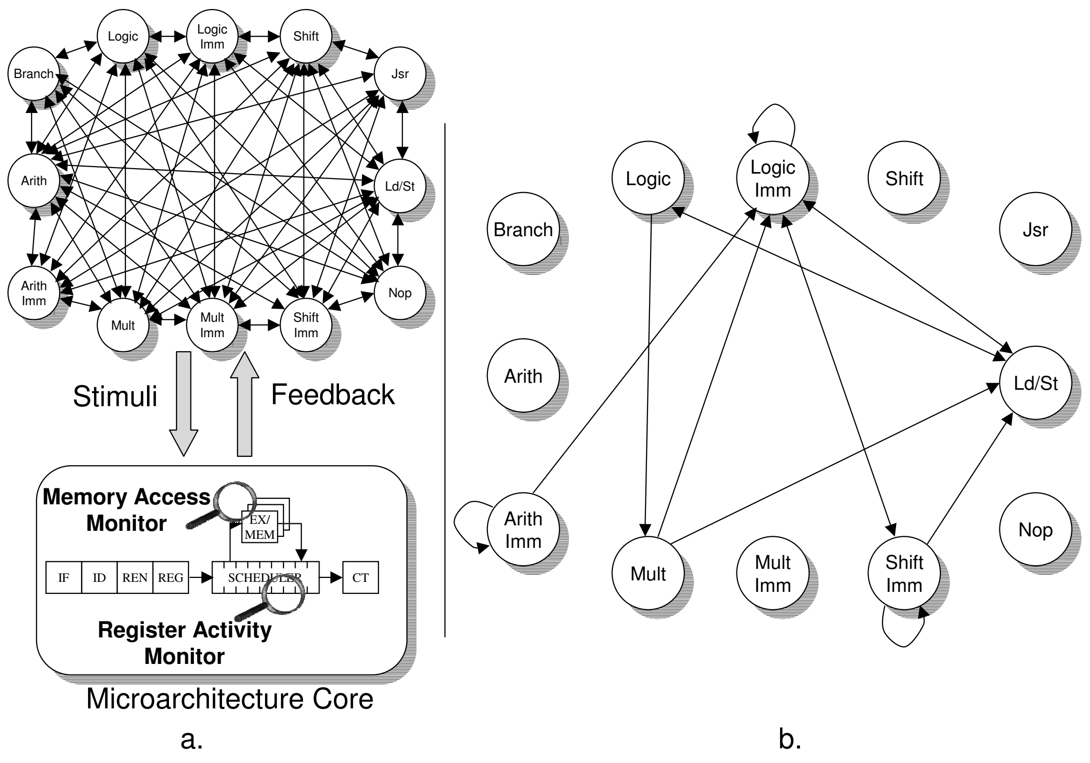


Figure 2.6: Impact of activity monitors. A simulation using two activity monitors observing register file and memory interface transforms a Markov model started as a clique in **a** to the one generating dense register and memory writes in **b**. The diagram in **b** has extremely low probability associated with all edges but the ones indicated.

One challenging aspect of implementing the activity monitors was in maintaining the association between a particular Markov model transition and the resulting activity event. The challenge exists because input stimuli are generated many clock cycles before the corresponding activity can be observed. To maintain this binding we use a small cache where we store a map of recently generated instructions and transitions leading to them. When sampling an observed activity at a pressure point, we also track the opcode of the instruction in the corresponding pipeline stage. The opcode is used to match an <*instruction*, *transition*> pair from the small cache. Once a transition is retrieved, the proper probability adjustments are applied. If more than one match is found, all the matches are adjusted. We found that multiple matches occur with less than 3% frequency, making the inaccuracy in the activity to stimulus mapping negligible.

Figure 2.6 shows an example with two activity monitors and their impact on a Markov model. In the experiment, a Markov model for an Alpha ISA is used to generate stimuli for a testbed microarchitecture. Two activity monitors are engaged: a memory access monitor, which encourages frequent memory accesses, and a register file monitor, to push for the generation of tests with many accesses to the register file using diverse data values. In the initial Markov model (Figure 2.6.a) each transition is equally probable. However, due to the adjustment dictated by the activity monitors, after 8000 cycles of operation a few transitions have much higher probability compared to the others. Figure 2.6.b shows the few high probability transitions that are left. As shown, the Markov model quickly morphs into a graph which generates many memory accesses, due to the edges pointing toward the load/store generation node. Moreover, the use of instructions with immediate operands increases the range of values written to registers (immediate fields have random values), thereby reinforcing the monitor on the register file. Shift and multiply operations contribute to this monitor, too, by generating high variations in the computed output values. Finally, branches and jumps are less frequent since they cause little excitement in the register file.

The flexible construct of activity monitors enables StressTest to aim for multiple verification goals simultaneously. For instance, the example above targets both registers and memory activity. When targeting multiple goals as in the example, StressTest computes a "score" (see Section 2.2.1) which is a weighted sum of the activity reported by the individual monitors. While this approach is often fruitful in discovering complex bugs due to unforseen components interactions (as we experienced in our experiments), it could theoretically lead to inadequate exercising of individual coverage targets. If this latter scenario occurred, StressTest could be easily adopted to focus on one verification goal at a time by using dynamic weights in computing the activity monitors' score. This flexibility enables the application of StressTest in a wide range of contexts. For instance, if a cycle-accurate description of the DUT is available, activity monitors can be used to target performance bugs. In this case a bug can be found by observing the difference in performance of the golden model and DUT. Other examples include generating frequent collisions among packets routing through a network switch and checking correctness of the switch operation at high utilization points, or "stressing" a pipeline recovery mechanism with frequent mispredicted branches. In general, the "pressure points" to monitor during simulation should be selected based on the role they play in the operation and state of the DUT. For example, in our experimental evaluation, we selected three pressure points located at the register file interface, memory access control logic, and program counter control logic. The three corresponding activity monitors would tune the system to generate a wide variety of tests.

2.3.2 Depth-Driven Activity Monitors

We found experimentally that a small number of key activity points is sometime insufficient in directing the verification process towards interesting scenarios. The reason lies in the coarse “scores” collected when the system is run with few monitors showing bipolar behavior (very low or very high activity). The result is that often the model would “saturate” and produce very similar stimuli corresponding to the highest scores possible, without exploring surrounding areas of relatively high activity. To correct these situations, we expand the pool of user-selected pressure-points with additional circuit nodes selected automatically by StressTest. The additional pressure points are nodes that directly influence the activity of the user-selected pressure-points. For instance, if a pressure-point is produced in the circuits’s netlist as the logic AND of two other signals, then StressTest would add activity monitors for the two signals as well. It is obvious that generating high activity for the two additional signals would enforce StressTest’s ability to stress the user-selected pressure-point.

a.

```
wire mem_wr;
wire store;
wire stall;
wire store_byte;
wire store_word;
assign store = store_byte | store_word;
assign mem_wr = store & (~stall);
...
```

b.

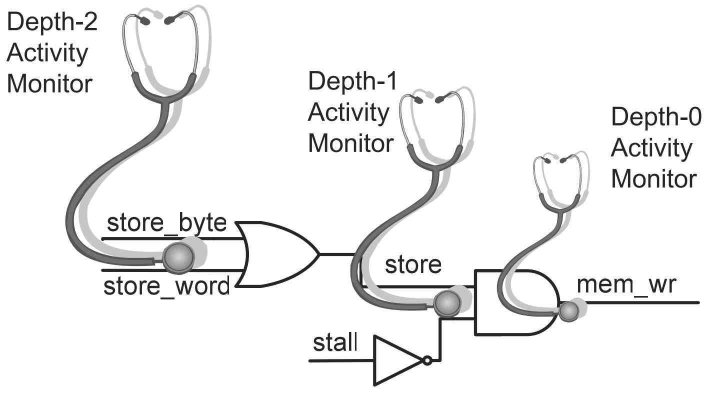


Figure 2.7: Depth-driven activity monitors. StressTest complements the set of user-selected pressure-points with additional nodes to refine the granularity of the measured activity scores. In the example, signal *mem_wr* is selected by the user; signal *store* and *stall* are identified as a depth-1 signals, based on the assign statement in the RTL description. Similarly, *store_byte* and *store_word* are depth-2 signals.

Another context where this technique is useful is an assertion-based verification methodology. Selecting the output of an assertion, or a checker, as a pressure-point would not provide any relevant feedback to StressTest's stimulus generator. In fact, the output of the checker would only transition once, at the end of the simulation when a design error is uncovered. In this scenario, the use of additional pressure points in the logic cone of influence of the assertion is critical for StressTest to be effective. In general, increasing the number of sampling points produces activity scores with fine-grain resolution which reach smoothly the goal of interest. The additional pressure points are identified by StressTest by analyzing the either the circuit structure or the behavioral RTL description of the design under test, whichever is available. We call *depth-1 signals* those signals which are direct inputs to the block whose output correspond to a user-selected pressure point. Here a block could be a simple logic gate, a logic/arithmetic expression, or a control statement, depending on which DUT description is available. By expanding this construction recursively, *depth-2 signals* are signals in the cone of influence of depth-1 signals, etc. Activity monitors connected to these StressTest-selected pressure points have a lower impact in the activity score (see Section 2.2.1). More precisely, their weight is inversely proportional to their depth. Figure 2.7 shows an example of how these additional pressure points are identified and connected to activity monitors. In the example, signal *mem_wr* is a user-selected pressure point, two additional monitors are created by analyzing the RTL description of the design under test.

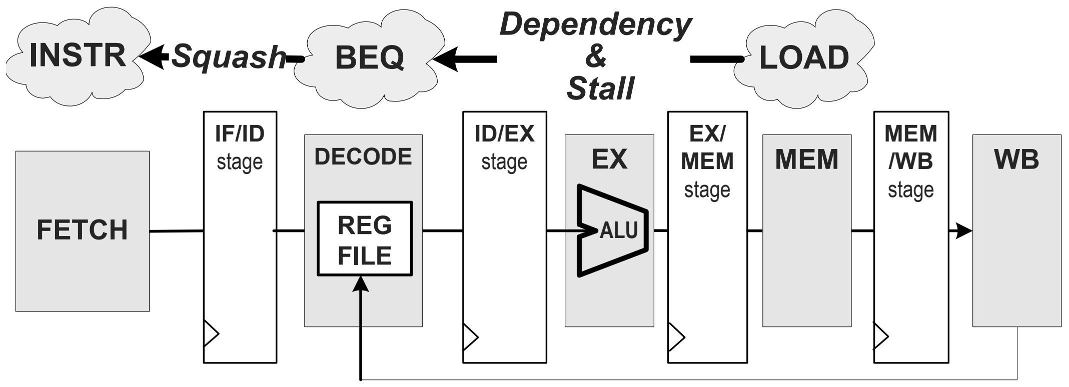


Figure 2.8: Pipeline snapshot for case study. Schematic of the 5-stage pipeline used for the case study. The bubbles show the instructions exposing the design error: a branch instruction dependent on a load is stalled in Decode stage, and when the forwarded value shows that the branch is taken the instruction in Fetch should be squashed.

2.4 Case Study

In this section we present a simple case study of how StressTest is used to detect an error related to a specific instruction dependency. The design under consideration is a five-stage MIPS pipeline similar to the one described in [59], whose simplified structure is reported in Figure 2.8. For this design, branches are resolved in the decode stage and always predicted as not-taken. When a branch instruction follows a load, it is possible to have a race condition if the load's destination matches the branch's condition register. In this case the correct execution of both instructions requires to stall the pipeline for one cycle after load so that the proper value can be read from memory before it is tested for the branch. When the condition is finally evaluated the branch is in the decode stage and the instruction at the fetch stage might need to be squashed if the branch was mispredicted (*i.e.*, it is taken). A schematic of the specific situation is presented in Figure 2.8. Note that both squashing and stalling should occur in the same clock cycle. However, due to an implementation error, the priority of the control signals (stall and squash) was incorrectly encoded and a stall would prevent a simultaneous squash. This encoding error allows the instruction following the branch to be executed causing an error. Given the very specific setup required to expose the problem, it would be unlikely to generate a test for it in a purely random verification environment. In fact, we would need to 1) generate the proper sequence of instructions, 2) have a dependency between a load and a branch, and 3) have the branch be taken. The probability of all these conditions is computed below.

1. If we were to group instructions in 4 classes – loads, branches, noops, and all others – then the probability of generating the required sequence is:

   $$p_{seq} = 1/4 * 1/4 * 3/4 \tag{2.5}$$

   (note that the last instruction can be anything but a *noop*). Any other grouping, would lead to an even lower probability for the desired sequence.
2. The probability to generate a dependency between the load and the branch is $\frac{1}{N_{regs}}$. The architecture in [59] has 32 registers, hence p_{depend} = 1/32.
3. The probability of the branch being taken requires the two branch operands to have the same value. If the comparison is between two distinct registers, then this probability is $\frac{1}{2^{32}}$, which is negligible (each register stores 32 bit values). However, if the register operands of the instruction are the same, then the branch is always taken. The probability of generating such a branch is p_{taken} = 1/32

By putting all together, the probability of generating the desired snippet of assembly program is

$$P = p_{seq} * p_{depend} * p_{taken} = 4.57763672 * 10^{-5} \tag{2.6}$$

Hence, we can calculate the average number of cycles of random simulation before the sequence is generated as expected value of a geometric distribution:

$$Cycles_{avg} = \frac{1-P}{P} = 21844. \tag{2.7}$$

On the other hand, in StressTest information passing between load and branch instructions can be setup by proper dependency variables. For example, if *lambda*=2, and *cacheSize*=5, the average distance between dependent instructions is 1.2, which means that virtually every pair of instructions depend on each other, exactly what we need in this case. Even if we offset that by setting *probCache*=0.5, hence there is 50% probability of not exploiting locality, the probability of having a dependency between a load and a branch is still significantly higher than 1/32. Similarly, if the same dependency variable is used for both operands of the branch, the probability of them being equal is much higher than 1/32. Finally, since the Markov model stores paths leading to favorable activities, by using a memory interface activity monitor and a branch logic one, the probability of having a load/branch/!noop sequence is much higher than that of a random approach. We found experimentally that, with the setup described, StressTest was capable to expose the design error after only 126 simulation cycles using a range of initial condition seeds. In addition, note that using this setup StressTest can generate longer dependency intervals, and thus discover further potential bugs in the control logic. The setup presented here was simplified for presentation purposes; a full design scenario would penalize random simulator even further, while StressTest could still use all of its tuning features to narrow down on critical execution scenarios.

2.5 Experimental Results

In this section, we first introduce our experimental evaluation framework, and the test designs we used for the analysis. Then we provide insights on a range of aspects in StressTest: the impact of the various parameters involved in dependency variables and a study of the convergence of the initial stimulus generator to a stable Markov model. Section 2.5.4 evaluates the performance of our proposed technique against an open-loop random instruction generator, comparing both coverage of bugs and number of simulated instructions required to expose those bugs. The next part of this Section reports on the impact of depth-driven activity monitors on the speed of convergence of the system on a range of design errors. Finally, the performance of StressTest extended with hierarchical Markov models is compared to that of an open-loop generator using a design with multiple parallel interfaces.

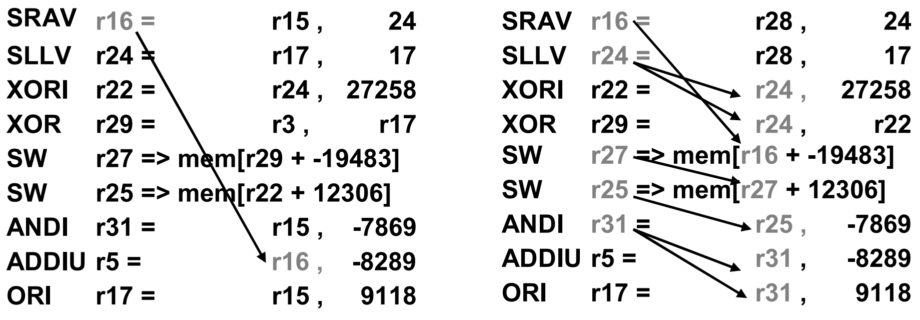
2.5.1 Experimental Testbeds

To evaluate the performance of StressTest we conducted a series of simulations on two processor core designs described in Verilog RTL. The two systems have both 5 pipeline stages; the first is based on the MIPS-Lite ISA and branches are resolved in the ID stage, The second design runs an Alpha ISA with branches resolved in the EX stage and has two-cycle store instructions. We also built single-cycle golden models (that is, functional descriptions) for both systems, to evaluate the correctness of the designs. The design under test (DUT) and the golden model were connected to independent data memories, and interfaced to StressTest through the instruction bus. We connected the two implementations, DUT and golden model, through a small Verilog testbench interface. We also implemented the stimulus generator (template file parser and Markov model data structure) and the activity monitors (analyzing the activity on the pressure points) in a C++ program. Sampling of switching activity at the pressure points was accomplished through Vera's binding constructs [31], which work as a glue between the Verilog description and StressTest's C implementation. We selected a range of pressure points to connect to the activity monitors, including key register file interface signals, memory system interface, branch/jump resolution logic, and pipeline stalling and flushing logic. The selection was driven by the wide variety of potential design errors that we were hoping to uncover in the testbed designs. To target more specific bugs, users would choose activity monitors at very similar points. The activity feedback from the DUT was observed using Vera and then passed to C functions for adjusting the edges of the Markov model.

For the final set of experiments in this section we compared performance of StressTest extended with hierarchical Markov models against an open-loop random stimulus generator. Both generators were used to supply stimuli to an on-chip network switch design, which featured simultaneous parallel interfaces. The switch was initially designed for testing performance and traffic patterns of different routing algorithms. In all experiments, we used a version of the switch utilizing an adaptive cut-through minimal-path routing algorithm for two-dimensional mesh networks. The design consists of five input ports with three virtual channels each, five output ports, and crossbar logic. The packets have fixed length of 64 bits, and, the header fields can be modified by the switch to allow for adaptive routing. The virtual channels in input ports of the switch can contain up to seven packets, each consisting of header flit and several payload flits. The crossbar logic considers the address of a packet stated in the header by accessing the static routing table and then selecting the destination from two possibilities. The switch might be prevented from transmitting by asserting back pressure signals of the output ports. Whenever the switch is unable to latch a flit due to buffer overfill it will assert outgoing back pressure signals of input ports. The switch consists of about 2000 lines of Verilog in total. The golden model for the switch is an equivalent design written in C.

2.5.2 Dependency Variables: Parametric Evaluation

The first set of experiments evaluates the impact of the parameters involved in a dependency variable. To this end, we set up an experiment on the Alpha pipeline testbench: The template file includes only one dependency variable which is used to create a dependency between source and destination operands in sequences of instructions. We fixed *probCache=1*, and used a range of cache sizes and *lambda* values for the dependency variable. For each simulation produced with this setup, we recorded the average distance between dependent instructions, that is the average interval between the insertion of a value into the locality cache and the retrieval of that value. In addition, we disabled activity monitors for this experiment, so that the sequence of instruction generated would be unchanged across different runs, as long as we started with the same random seed. Portions of sequences generated in two of such runs are shown in Figure 2.9. It can be noted that the run shown in part b) has many more dependent instructions very closely spaced. This is due to the smaller value of *cacheSize* and larger value of *lambda*. The average dependency distance is plotted as a function of *lambda* and *cacheSize* in Figure 2.10. The results of this analysis allowed us to better select the configuration of dependency variables for the experiments presented in the following sections. Specifically, knowing that our testbed designs are 5-stage pipelines with 32 registers, we were only interested in generating dependencies between pairs that were 4 instruction apart or less. A *lambda* value of 2 provides a majority of dependency intervals within the range of interest. Also we set the locality cache size to 20, so that we could have a broad set of values available. Larger values of *lambda* are more suitable for longer pipelines.



```
SRAV  r16 =               r15 ,      24
SLLV  r24 =               r17 ,      17
XORI  r22 =               r24 ,   27258
XOR   r29 =               r3 ,      r17
SW    r27 => mem[r29 + -19483]
SW    r25 => mem[r22 + 12306]
ANDI  r31 =               r15 ,   -7869
ADDIU r5 =                r16 ,   -8289
ORI   r17 =               r15 ,    9118
```

```
SRAV  r16 =               r28 ,      24
SLLV  r24 =               r28 ,      17
XORI  r22 =               r24 ,   27258
XOR   r29 =               r24 ,     r22
SW    r27 => mem[r16 + -19483]
SW    r25 => mem[r27 + 12306]
ANDI  r31 =               r25 ,   -7869
ADDIU r5 =                r31 ,   -8289
ORI   r17 =               r31 ,    9118
```

a. Cache size = 20, lambda =0.1   b. Cache size = 10, lambda = 5.0

Figure 2.9: Results of two sample runs with different values of cacheSize and lambda. The arrows represent dependencies among instructions in a sequence. Note how larger values of *lambda* generate shorter dependency intervals. A cache of larger size allows for more complex dependency patterns.

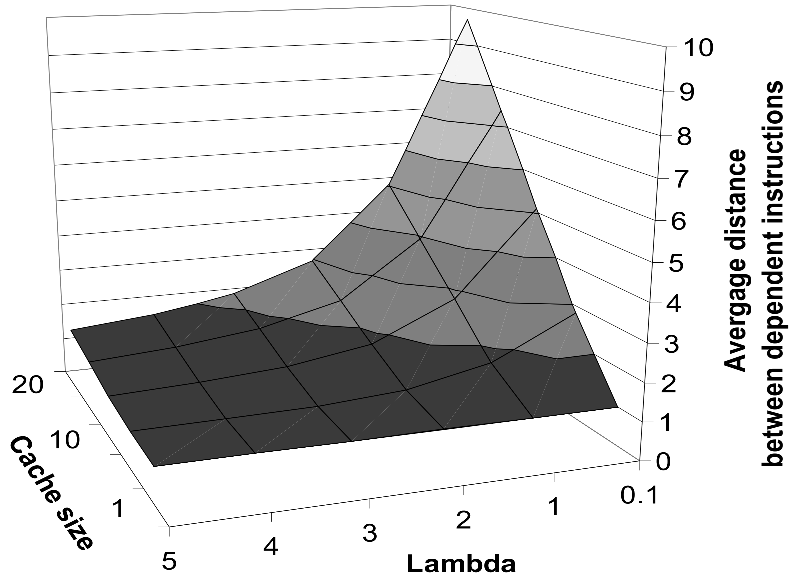


Figure 2.10: Average distance between dependent instructions. Higher values of *cacheSize* and lower values of *lambda* result in larger intervals between dependencies.

2.5.3 Stability of the Activity Monitors

The objective of our second set of experiments is to evaluate the change of the Markov model over time during simulation. More specifically, we study if the model converges to a stable shape (that is, if the probabilities associated with the transitions stabilize) in the long run, or if it keeps oscillating. In addition, we evaluate the impact of simultaneous multiple activity monitors on the stability of the model. We generate stimuli for a RISC pipeline for 3000 cycles and observe the probabilities marking the edges of the Markov model every 200 cycles. After the simulation, we calculate how much, on average, the values associated with the transition edges changed during each 200-cycle interval. The Markov model used for our experiments includes only 3 vertices, each representing a different class of instructions, namely, memory operations (loads and stores), register-to-register instructions and branches. We also used three distinct activity monitors, practically corresponding to each of these instruction types: a memory accesses monitor, a register file one, and a program counter control monitor. We ran several simulations varying the weights assigned to each monitor, effectively changing the impact of the activity observed. The results of this study are shown in Figures 2.11 and 2.12.

Figure 2.11 shows that the model stabilizes for all the scenarios (a range of different weights associated with the activity monitors) that we explored within 2000 cycles (probability variation is below 0.02 on average). This consistent stabilization of the Markov model has both advantages and drawbacks. The advantage is that a particular "path" is set up within 2000 simulation cycles and from then on sequences of instructions with similar

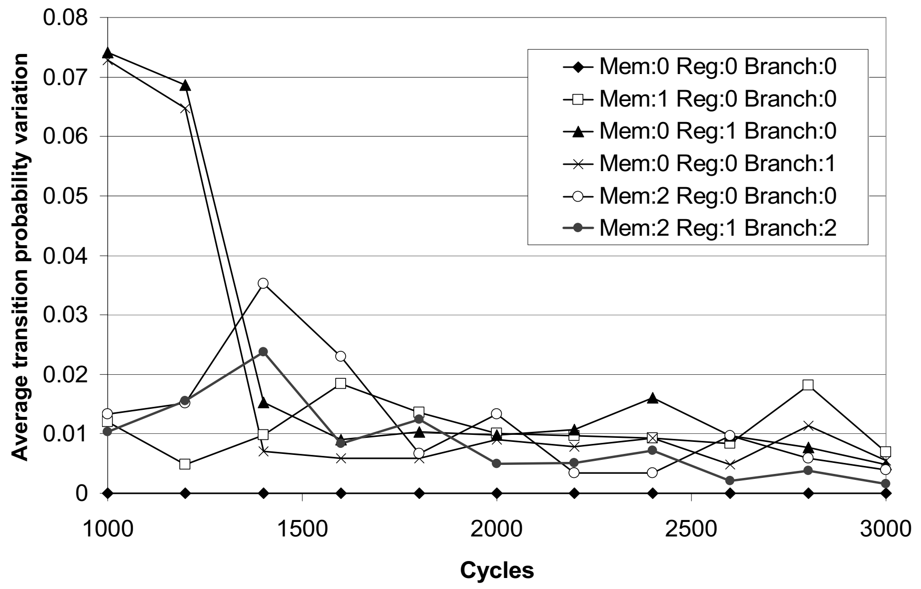


Figure 2.11: Impact of activity monitor's weights on the Markov model stability. The diagram shows that the Markov model consistently converges to a stable form over time (simulation cycles). Each trend line has been generated using different weights for three distinct monitors: memory accesses (Mem), register file (Reg), and branching (Branch). A value zero in the legend means the corresponding monitor is suppressed. Note that the Markov model stabilizes in all cases and that higher weights result in faster stabilization.

structure are generated repeatedly. However, since the various operands of the instructions are generated via dependency variables, different dependency combinations are still explored, which is crucial for thorough testing. On the other hand, the Markov model would not transition to another stable point any longer, because the activity feedback forces it to remain stable. Therefore, after a while the simulation starts loosing diversity. As expected, higher weights cause the Markov model to converge faster, since activity scores are higher.

During this set of experiments, we also monitored the number of each type of instruction generated during a simulation run. Ideally a high weight on a memory access monitor should lead to generating more load/store instructions. A high weight register file monitor should provoke many register-type instruction, and the PC control logic monitor should generate many branch instructions. Figure 2.12 shows the relative distribution of each type of instruction generated during simulation. When the weights are all 0 (as in the leftmost bar), the model does not receive any feedback and instructions are roughly 1/3 of each type. It is also evident from Figure 2.12 that the relative frequency of occurrence of each instruction is related to the weight associated with a particular activity monitor reinforcing the corresponding type of instruction. Note also, that when the weight of the memory interface activity monitor is increased from 1 to 2, the number of loads and stores increases, because of the stronger reinforcement connected to the corresponding Markov model edges.

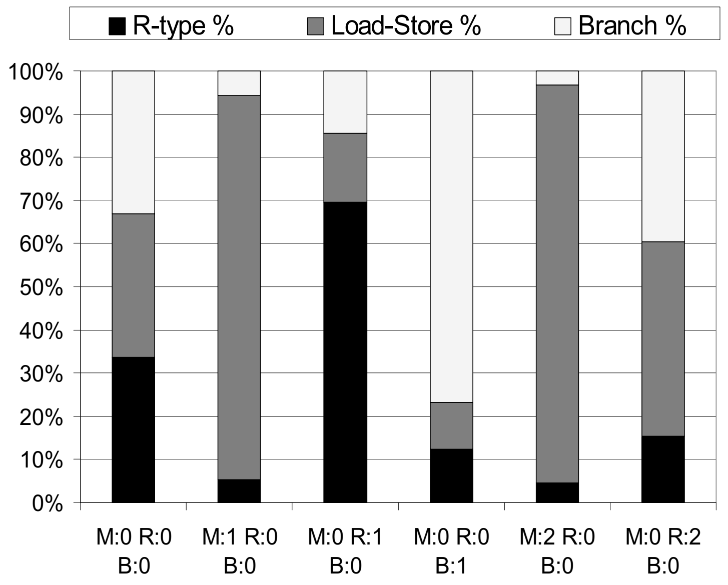


Figure 2.12: Impact of the monitor's weight on stimuli generated. The graph reports the distribution of the stimuli among three types of instructions: register, memory and branch. The experiment is set up with three activity monitors, a memory (M), register access (R) and branching activity (B). Each bar consider a distinct setup with different weights associated with the monitors. The analysis shows that a higher weight result in a larger fraction of the stimuli generating activity for it.

2.5.4 StressTest Coverage Density and Performance

We now examine the coverage density and performance of StressTest compared to other approaches based on random simulation. The techniques we compare are: plain random simulation, simple constrained generation, open-loop and StressTest. For each approach evaluated, we quantify the effort (in simulation cycles) required to expose bugs, and analyze how many bugs the technique is capable to expose. The testbenches used for these experiments are the DLX and Alpha processors described in Section 2.5.1. The set of tests for the DLX design consist of 30 distinct versions of the DLX core, each containing a different bug. Bugs vary from simple (such as incorrect operation for a given arithmetic opcode) to complex ones, involving forwarding logic and interactions through memory. Ten of the simplest bugs for DLX were taken from a testsuite that is part of an advanced hardware verification course at the University of Michigan, while the others were handcrafted for the experiment. For each of the buggy variants and each of the considered techniques we performed 25 runs, using distinct random seeds, and evaluated the average effort and standard deviation. In each run we simulated for a maximum of 75000 cycles, or less if the bug was exposed sooner. We selected the number of distinct seeds and the length of the simulation runs so that to ensure stabilization of the Markov model and to permit each experiment to complete in less than 3 hours. The set of tests for the Alpha design used ten buggy variants including moderate and very complex bugs, mostly very specific corner cases in the pipeline's forwarding logic. The four techniques compared in the study are:

- **Random** utilizes only a static evenly-distributed Markov model of the ISA for the instruction generation and does not collect feedback from the DUT. It also uses several dependency variables, but without caching. Random represents a capable open-loop testing solution.
- **Simple** relies on a feedback-adjusted Markov model, based on activity feedback from the DUT. However, it still does not use caching for the dependency variables.
- **Open-loop** is an open-loop setup which uses dependency variables with caching, but does not have activity feedback, so the Markov model is has evenly-distributed edges across the entire simulation.
- **StressTest** is the full-fledged implementation using both a feedback-adjusted Markov model and dependency variables with caching. Variables are used to transfer destination register indices to source register fields and to share arithmetic immediate values and memory and branch offsets among instructions in the same test.

The tests setup use an initial Markov model with 7 and 11 vertices for DLX and Alpha, respectively. Each vertex represents a particular class of instructions. No single vertex included multiple instructions, since the ISAs did not impose any sequencing constraints on the input, *i.e.*, branches did not require a *noop* instruction to follow. The dependency variables were set with *cacheSize*=20 and *lambda*=2.0, values which we derived from the analysis described in in Section 2.5.2. We also set *probCache*=0.66, allowing random values to be freshly generated in 30% of the uses of dependency variables, to introduce variation in the generated test programs. Variables that store static information, such as opcodes, had *probCache*=1.0 and *lambda*=0.1, so to obtain uniform random distribution among the values in the cache (which was initialized with all the legal opcodes in the ISA). We used three activity monitors located at the memory control logic, the register-file interface, and the PC control logic. We used an analysis similar to that of Section 2.5.3 to select a set of weights for the monitors, and finalized them to 2, 1 and 2, respectively. In setting up the experiments for each of the four random testing techniques, we exposed bugs by by running in lockstep with a golden model and flagging any discrepancy in the committed results by comparing register file, program counter and memory writes.

Figures 2.13 and 2.14 show the results of the four random test generation approaches, applied to the DLX and Alpha processor pipelines. For each of them, the graph illustrates the cumulative effort (in total run-time) versus the total number of bugs detected. To distinguish between easy-to-find bugs from harder ones, we have sorted them in ascending order of total number of instructions required to locate a bug. As a result, the bugs on the left part of the graph were easier to locate than the bugs on the right. When a technique was incapable of finding some of the bugs (for instance, Random), the curve stops short. In addition, we are only showing the 15 hardest bugs in Figure 2.13, since the performances were virtually indistinguishable for simpler bugs. As shown in Figure 2.14, StressTest and Open-Loop achieve better coverage than Random for the DLX processor, detecting five extra bugs. StressTest is also far more efficient than Open-Loop at detecting all the bugs, requiring approximately half the time. Interestingly, Simple appears to be the worst approach, despite of its use of activity feedback. We believe that this is due to the inability of generating interesting correlations between instructions due to the lack of caching in dependency variables.

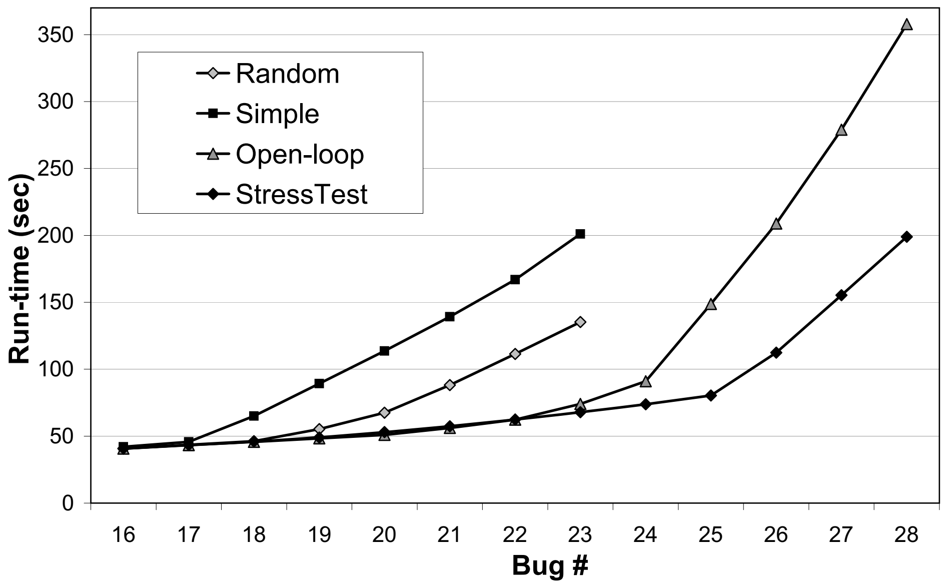


Figure 2.13: Effort versus bugs covered for the DLX processor. The diagram compares four random testing-based techniques in their effectiveness at uncovering design errors. StressTest and Open-loop can expose more bugs than Simple and Random, and StressTest can do so faster than all other techniques.

The experiment led to exposing three hidden bugs in the forwarding logic of the DLX pipeline, which initially was assumed to be correct. Although this design was a subject of verification projects for several years, these bugs were unknown, until they were exposed with StressTest tool:

1. *Forwarding through Reg.0.* Register 0 in a DLX architecture should always retain the value 0. When forwarding through this register, the second instruction should always receive a 0 value, no matter what the first instruction computed. However,

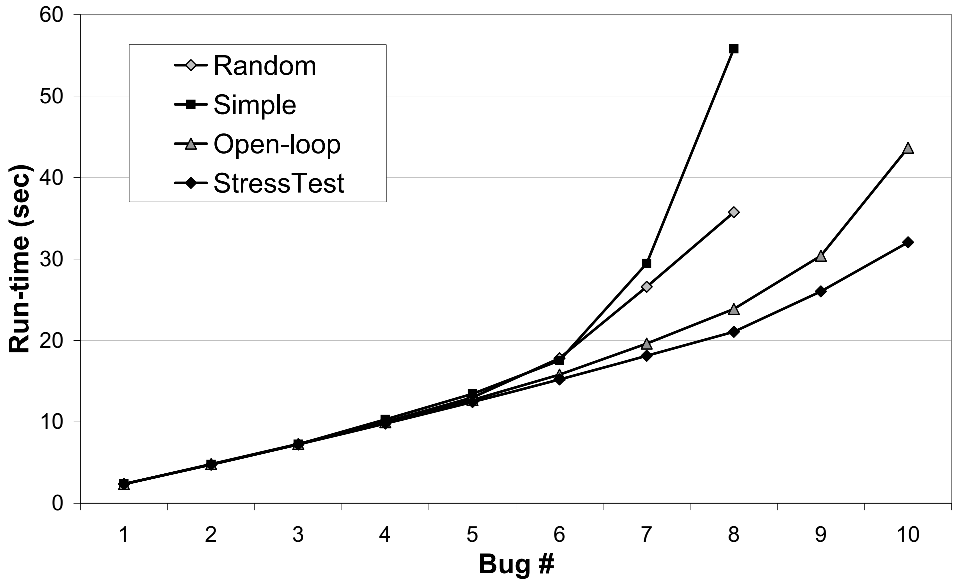


Figure 2.14: Effort versus bugs covered for the Alpha processor. The diagram compares four random testing-based techniques in their effectiveness at uncovering design errors. StressTest can expose more bugs in less time than all other solutions.

the original design allowed bypassing and the value forwarded to the second instruction would be the result of the first instruction's operation. Most experiments with Random and Simple could not locate this bug because the generated operand fields had a very small chance of creating the dependencies through architectural register 0 necessary to expose the issue.

2. *Stalling logic*. Because of the timing between memory accesses and branch resolutions, it is possible in DLX to create a scenario where the pipeline is stalled due to memory contention while its front-end is being flushed due to a mispredicted branch. We found that our initial pipeline design ignored pipeline flushes in this context, allowing the instruction following the mispredicted branch to proceed to execution. An analysis of this situation is discussed in the case study of Section 2.4.
3. *Forwarding through unused register*. Some MIPS instructions use only one source register, which is ignored in the execution and memory stages. However, when a dependency through this register existed, the forwarding logic would still trigger and forward the bogus value to a following instruction. Dependency variables played a key role in discovering this bug.

The features which led to find these bugs were: 1) simulating the DUT in lockstep with the golden model to check correctness, 2) generating instructions using templates, and 3) passing information between instructions using dependency variables. Without these techniques the bugs would have been extremely hard to find and would have required significant user effort in directing the test towards them. Figure 2.14 shows the results of the same experiment on 10 variants of the Alpha pipeline, each including a different bug. Again, we computed average cumulative effort and sorted the bugs from easiest to hardest. The analysis confirms once again, that the advanced features of StressTest are critical in exposing the more complex bugs.

2.5.5 Depth-driven Activity Monitors

We also investigated the impact of the depth-driven activity monitors on the performance of StressTest. We used the same 30 variants of DLX cores and 10 of Alpha pipelines and compared the full fledged StressTest used for the previous section against StressTest with depth-driven activity monitors of depths 1, 2 and 3. The experiment setup was the same as the one in Section 2.5.4. Figures 2.15 and 2.16 show the results for DLX and Alpha pipelines, respectively. Note that in both cases deeper monitors perform better for harder bugs, but perform slightly worse for medium-difficulty bugs. For example, in Figure 2.15, the performance of StressTest up to bug 25 is better than other approaches, but it is worse for all other harder bugs (26 and up). The reason lies in the amount of additional information which is accrued by the depth-driven monitors: Easier bugs are usually exposed by simpler scenarios which can be reached by watching only a handful of critical signals, and additional information is just "distracting" the system. However, in harder bug scenarios, any available additional information is helpful in narrowing the bug scenario. Finally, the experiments also showed that StressTest finds bugs more consistently than Random and other techniques, which rely mostly on chance.

2.5.6 Evaluation of Hierarchical Markov Models

To verify the on-chip switch design, we used a hierarchical Markov model with the top level model specifying the number of packets to send simultaneously. The top model also specified the state of the back pressure signals in the network surrounding the switch. The local Markov models were used to generate valid packets that were likely to have similar destinations, thereby exerting high pressure on the switch. The amount of effort to write templates for both global and local Markov models for the switch was around 16 man-hours. To guide the test generation during switch verification, we utilized checkers written in Verilog. We derived ten distinct combinational checkers from the high level description of the switch routing algorithm and buffer functionality. Each property module had a single-bit output, which depended only on particular signals in the design. This study also used activity monitors of different depth, with *Depth-0* experiment only monitored the

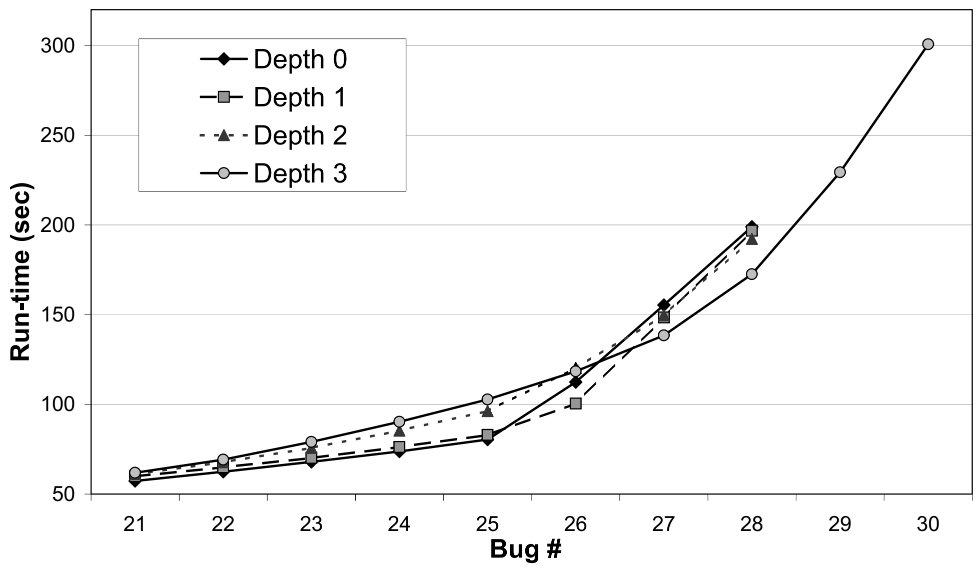


Figure 2.15: Effort versus bugs covered for StressTest with depth-driven activity monitors (DLX processor). The diagram compares a range of depths for the activity monitors and show that depth-driven activity monitors allow StressTest to find complex bugs faster.

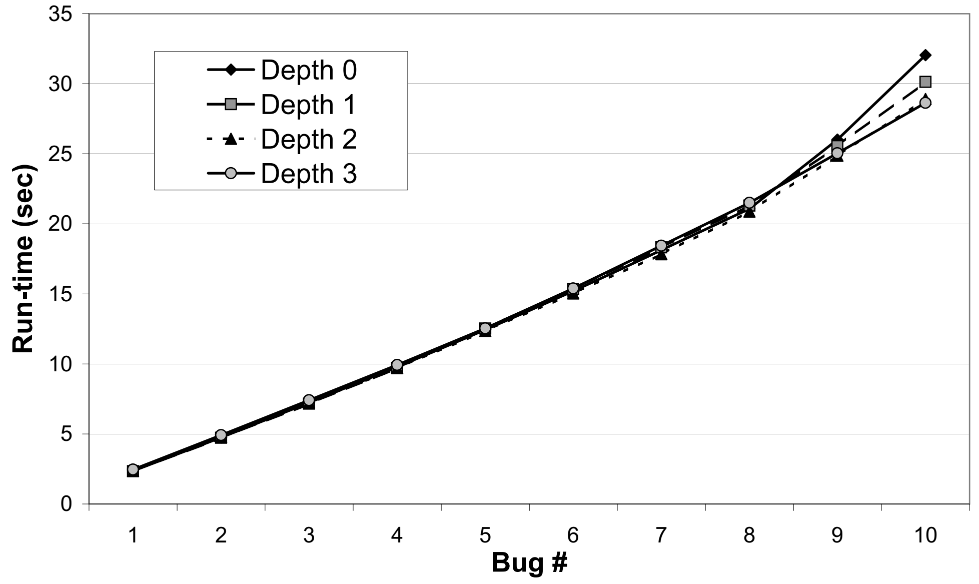


Figure 2.16: Effort versus bugs covered for StressTest with depth-driven activity monitors (Alpha processor). The diagram compares a range of depths for the activity monitors and show that depth-driven activity monitors allow StressTest to find complex bugs faster.

output bits of the properties, while *Depth-1* design monitored the inputs to the properties as well.

Incidentally, in this study we were able to find three actual design bugs during verification of the switch: one in the buffer control logic and two in the routing logic of the crossbar. All of these bugs were hard corner cases of the switch's behavior, for example, in one bug several internal counters were incorrectly handled during a buffer-overflow situation, however from the error could be seen several tens of cycles later.

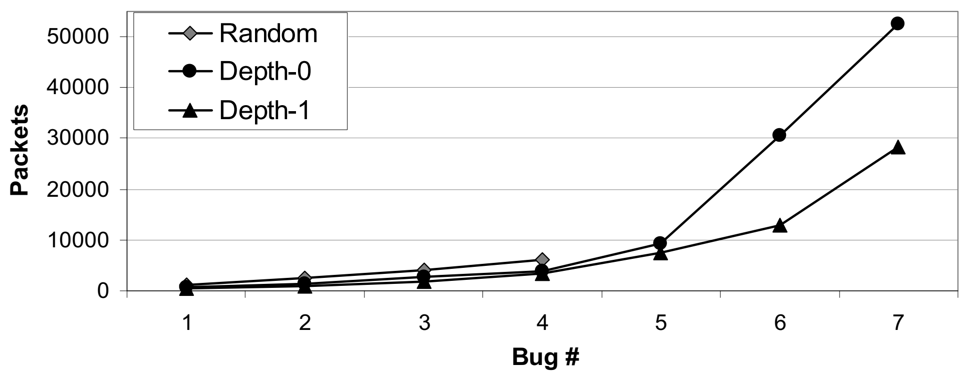


Figure 2.17: Effort vs. Bug coverage for the switch design. StressTest with hierarchical Markov models is able to find more bugs faster than Random and performs better with deeper activity monitors.

The results of the switch verification experiment are presented in Figure 2.17. The effort in this case is measured in number of packet transactions required to find a bug. Again only the hard to find bugs are reported, since all three systems performed equally well in discovering easy bugs. The three bugs on the far right of the graph are the three original bugs found during the verification of the switch. Once again, the *Random* configuration was unable to discover the last three bugs and requires significant effort for covering easier bugs. That explains why the bugs were not discovered during the design of the switch where only random testing was used. As can be seen in the graph, the approach of monitoring just the output bits of the property modules performs relatively well for intermediate bugs. We believe that is partially because our approach enabled information sharing between local Markov models. Moreover, this system explores more distinct input sequences than *Random*, due to reinforcement learning that dynamically changes the Markov models, thereby continuously steering the generation to explore new stimuli sequences.

2.6 Related Work

Traditional verification of hardware designs relies mostly on simulation and constrained-random testbenches, which exercise a design with streams of stimuli resembling real-life applications. Recent advances in scalable constrained-random generation have focused on generating feedback on previous tests to influence the generation of future tests. Tools such as Specman Elite [34] and Vera [31] provide on-the-fly data assertion and checking and methods for validation of generated tests; however, they still require a significant amount of user guidance at run-time. Other techniques employ coverage-directed test generation processes, and use sophisticated methods to relate input generation to coverage, such as Bayesian networks and computer learning approaches [28] and Markov-models [70]. In industry there are several examples of such tools being used in verification of single-core processors [11, 12, 49, 47, 71] and multi-core [1, 63, 52, 4, 3, 82] chips. Unfortunately, even the best simulation-based tools cannot create exhaustive tests due to the immense complexity of today's hardware.

Alternatively, formal methods, such as symbolic simulators, model checkers and theorem provers, attempt to prove mathematically that a device behaves correctly under any operating condition. Unfortunately, so far these methods had only partial success in microprocessor verification, because they suffer from the "state explosion" problem and cannot handle such complex designs. For example, even a simple safety critical processor such as the VIPER microprocessor [17] had to be validated with "intelligent exhaustive simulation", since it was too complex for formal tools. In 1995, Ho, *et al.* [33] attempted to verify a processor by generating all possible control logic transactions. Although this proved effective for a simple RISC processor, this approach doesn't seem to scale to handle today's commercial designs, which include complex features such as wide issue, out-of-order execution, simultaneous multi-threading, and complex memory hierarchies. In addition to scalability problems, one of the major drawbacks of formal tools is that they usually require the design to be formally specified using a dedicated language, for example, Sugar [22], PSL [2], or ForSpec [7]. This, however, means that such techniques cannot be easily integrated into today's industrial design flow, because they would require a major engineering re-training, and, even then, the writing of a correct specification in a formal language would be extremely time-consuming. Moreover, in many cases, formal approaches still require human guidance to successfully proove design invariants, as reported in [21] and [29], where Murφ system was used to verify cache coherence protocols of industrial designs. Similar experiences with formal methods, based on the TLA+ language and model checking approaches, were reported in [42] and in [60, 27], respectively. Thus, due to the limited scalability of formal techniques, non-exhaustive simulation-based approaches still remain the workhorse of the verication effort in industry. However, the need for solutions that can address both the high-coverage and design size requirements has been recently voiced by Intel, which in recent years started to deploy designs that exceed 100M transistors in size and require 2000 person-years of effort for verification [68].

2.7 Summary

This chapter presented a novel, scalable approach to pre-silicon constraint random validation of individual processor cores and sub-modules. The approach, implemented in a tool called StressTest, is a closed-loop technique based on a Markov model, which generates instruction sequences based on templates. These templates are designed by verification engineers to resemble legal directed tests. The template language of StressTest is particularly expressive, in that it supports the generation of a wide range of input types with varied dependency and locality characteristics and can be used in the verification of processor cores or other digital circuits. Moreover, the language can be used to describe hierarchical Markov models, which can be used for effective verification of designs with multiple simultaneous interfaces, such as on-chip routers/crossbars. In addition to stimulus template, the verification engineer needs to identify key activity signals in the design, *i.e.,* signals that are indicators of "stressful" activity, or are suspected to be indicators of performance or design bugs. A closed-loop feedback engine adjusts the Markov model continuously during simulation based on the activity observed at the activity points, to produce effective and efficient tests. For more effective feedback, StressTest automatically extracts the cone of logic influencing the user-selected activity points and monitors them as well, smoothly guiding the simulation towards interesting scenarios. Experimental evaluation shows that StressTest is capable of finding more bugs in fewer simulation cycles than open-loop random simulation or less sophisticated closed-loop test generation techniques. All together, various features of StressTest, *e.g.,* Markov-model based reinforcement, flexible dependency specification, depth-driven activity monitors, *etc.*, make this approach a viable and scalable solution for pre-silicon verification of individual cores and modules of modern multi-core microprocessors.

CHAPTER III

Post-silicon Verification of a Single Core

Once design components are verified satisfactorily in pre-silicon, the first handful hardware prototypes are manufactured, so that additional verification and electrical testing can be carried out on the prototypes themselves to determine if the system is fully functional and ready to go into full production, or if additional fixes are necessary. In the latter situation, the design is corrected and then new prototypes are produced (this process is commonly called a "re-spin"). The verification effort that is applied in determining the correctness of these hardware prototypes is called post-silicon validation. Traditionally, post-silicon validation approaches are used to identify electrical faults and manufacturing defects, however, in recent years processor manufacturers began to increasingly deploy these methods to detect also complex functional bugs. Post-silicon validation relies on a concept similar to pre-silicon simulation: the hardware prototype executes as many input vectors as possible and its responses are validated to determine if the system's behavior is correct. However, there are a few key differences between this approach and pre-silicon verification. First, the execution on a hardware prototype is several orders of magnitude faster than any functional simulator, therefore, significantly more test vectors can be checked. On the other hand, this high speed comes at the price of limited observability: the internal state of the prototype cannot be easily nor fully observed, forcing the engineers to employ silicon state acquisition solutions such as scan-chains [20, 10], JTAG [58], cycle breakpoint [13] or on-chip-logic analyzers [48] at the cost of die area. To minimize the area impact of post-silicon verification features, designers commonly try to diagnose errors from the architectural state of the system obtained after test program execution. Such tests usually consists of directed tests checking specific features of the processor, compatibility checks, such as operating system boot-up and tests with legacy software[66], as well as automatically generated random tests [11, 62]. Due to the unpredictable outcome of these random programs, engineers must also simulate them on a known-correct model of the design to obtain the correct final state of the hardware prototype to identify discrepancies, potentially revealing a bug. While tests can be run at-speed on the hardware, test generation and simulation constitute the bottleneck in this process, limiting it to the performance level of pre-silicon simulation. Consequently, design houses are forced to spend enormous computational resources on test generation and simulation servers [62] to ensure an adequate level of post-silicon validation.

In this chapter we present a solution that takes a first step towards a novel high-throughput post-silicon validation methodology, called Reversi, which allows for test generation to match the performance of execution of the silicon prototype and eliminates the costly simulation step required to obtain a known-correct final state. To this end, tests are generated by Reversi in such a way that at the end of the execution, the initial state of the machine is restored. Therefore, the final state of such a *reversible program* is known *a priori* and, the simulation phase of the validation process is bypassed. Since our program generation algorithm is agnostic to any particular instruction set, it can be easily ported between processors with different instruction and feature sets. Moreover, the absence of the simulation step in our framework allows for tests to be generated directly by hardware residing on the same system board as the prototype, eliminating the need for costly test generation servers. Consequently, the validation speed becomes only limited by the speed of communication between the prototype and the testing board. Moreover, once the system under test is sufficiently validated, the test generator can run directly on it. In this latter case, tests can be produced in one portion of the chip's cores and transferred to other cores for execution. If the generator cores were flawed, they would not produce proper reversible programs, exposing the issue. In this chapter we present our techniques to develop reversible programs and we compare a post-silicon verification methodology against a traditional post-silicon flow, noting that Reversi can lead to a performance improvement of 20 times.

3.1 Reversi Test Generation System

Typically post-silicon functional validation in industry has been conducted with two types of tests: parameterized directed tests and constrained-random (or pseudo-random) tests. Although the former ones can provide high coverage, they require significant human effort to be developed. The pseudo-random tests, constrained to produce only valid instruction sequences, can be generated automatically, however often suffer from lower coverage. More importantly, the final state of the processor after executing a random test sequence is unknown. Therefore, engineers must resort to simulating the design's golden model to compute the final processor state and check it against state dumps of the actual hardware prototype (as shown in Figure 3.1.a).

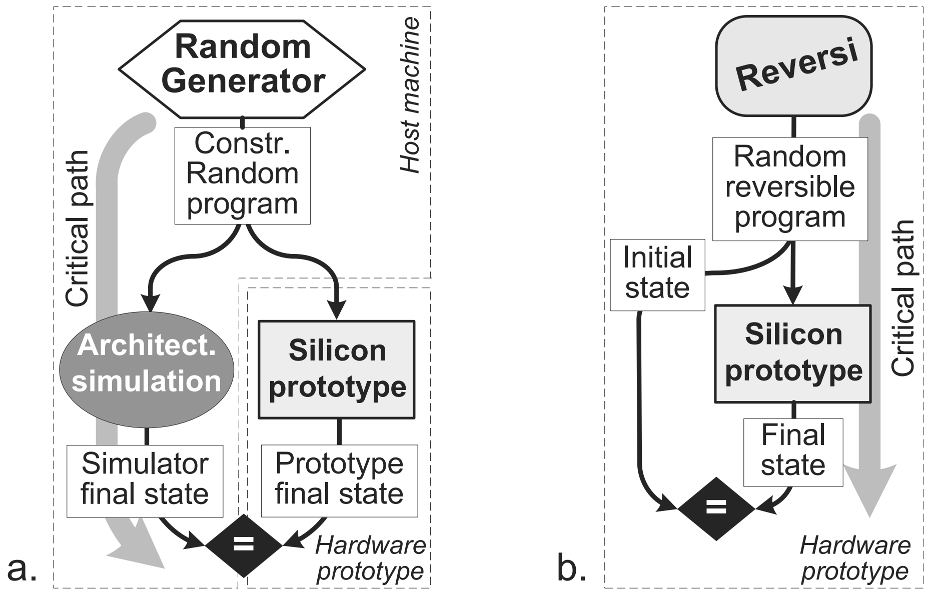


Figure 3.1: A typical post-silicon validation flow vs. a Reversi-based flow. a. In a typical post-silicon methodology, random tests are produced by a test generator and fed to both a golden model simulator and a silicon prototype. Bugs are flagged by differences between the prototype's and simulator's final states. Both test generation and simulation are done on a host machine at relatively slow speed. **b.** A Reversi-based flow does not require a simulator: random reversible programs can be generated on a tester board or on the hardware prototype itself. Bugs are flagged by differences between final and initial states of the prototype.

Unfortunately, as was mentioned above, the simulation of the golden model is several orders of magnitude slower than the hardware execution, therefore, the computation of the final state becomes a bottleneck for the entire effort. We address this issue in our methodology by developing a post-silicon solution which fully exploits the performance of the hardware under test. We designed a test generator, called *Reversi*, that produces tests whose outcome is known by construction. This allows us to bypass the simulation step and speed up the overall validation flow (Figure 3.1.b).

The main observation that we made in developing Reversi is that many instructions in a processor's ISA have counterparts, *i.e.,* operations whose functionality is the inverse of the former, such as restoring a value in a particular register, clearing a set of flags, *etc*. Moreover, if no single instruction exists to reverse the action of another, one can devise a small program sequence to be used to the same effect. This was the case, for example, for the integer multiply instruction in one of the ISAs that we used in our experimental evaluation. No instruction for integer division was implemented, but we could resort to software emulation of division to revert the effect of multiplication. Note that, if the emulation routine exposed any error in the hardware prototype, the result of the multiplication would not be reversed correctly. The presence of inverse functions enables us to design programs that include every instruction in an ISA, and for which the final register values match exactly the initial ones. In other terms, if $\overline{x}$ is a vector representing the processor state, and each F_i / F_i^{-1} pair represents a distinct function (either an ISA instruction or an instruction block) and the corresponding inverse, then a program generated by Reversi applies the following sequence of functions to the state $\overline{x}$:

$$\overline{x} = F_1^{-1}(F_2^{-1}(...(F_n^{-1}(F_n(...(F_2(F_1(\overline{x})..) \tag{3.1}$$

3.1.1 Reversible and Non-reversible Instructions

In order to create reversible programs, we first analyze each ISA and identify inverse instructions (or instruction sequences) for each of the operations. By applying these operations in the manner discussed above we can modify the state of the processor and then properly restore it (in the absence of bugs). This allows us to create a *block database* containing pairs of *functional blocks*: for each operation block, there is a corresponding inverse block. Each block contains either a single instruction or a small program sequence. An operation block modifies the value of a register, called the *focus register*, while its inverse restores its initial value. The ID of the focus register for each block is a parameter set by Reversi dynamically during test generation. Therefore, the same block may appear in the test program multiple times, each time modifying a different register, which allows a varied set of programs to be created. Note that blocks operate only on a single focus register at a time to maintain the reversibility of our program and track the correctness of its execution. Thus, for instructions with multiple operands, only one of the registers is the focus register, while other operands are randomly generated by Reversi according to the instruction format. The flexible and robust structure of the block database allows the Reversi algorithm to be agnostic to the functionality of individual blocks and the underlying ISA, making our framework readily adaptable to different processor architectures. Moreover, since blocks in Reversi may contain multiple instructions, we can populate the database with complex functions, including loops, procedure calls, *etc.*, and create elaborate tests representative of real software. In the remainder of the section we discuss individual classes of instructions and implementation details of operation and inverse block verifying them.

Arithmetic and logic instructions. The design of blocks containing arithmetic and logic instructions is summarized in Table 3.1.1 and is fairly straightforward, since the majority of these operations have a simple inverse directly in the ISA. For example, *add* can be reversed by *sub*, *inc* by *dec*, *ror* (rotate right) by *rol* (rotate left) and so on. If an instruction

Table 3.1: Reversi blocks for arithmetic and logic instructions.

Instruction	Operation Block	Inverse Block
add	*add*	*sub*
sub	*sub*	*add*
inc	*inc*	*dec*
dec	*dec*	*inc*
xor	*xor*	*and/or emulated xor*
not	*not*	*nand emulated not*
neg	*neg*	*-1 mult emulated neg*
and	*and/or emulated xor*	*xor*
or	*and/or emulated xor*	*xor*
mult	*mult*	*emulated division*
rol	*rol*	*ror*
ror	*ror*	*rol*
sll	*store lost bits, sll*	*srl, restore lost bits*
srl	*store lost bits, srl*	*sll, restore lost bits*
sra	*1.store lost bits,*	*1.rol*
	2.create mask	*2.apply mask*
	3.sra	*3.restore lost bits*

does not have a counterpart in the ISA, a small routine can be used to emulate its inverse. Some Boolean logic instructions, such as *and* and *or*, do not have direct inverses, however, these operations can be used to construct an *xor* logic function, which can then be reversed by an *xor* instruction. Such structure is also beneficial for verification of the *xor* instruction itself, since operation and inverse block in this case exercise different hardware modules. Situations where the same processor modules are used in the function and its inverse should be avoided to prevent bugs being masked by faulty hardware.

Some ISA operations, for example *sll* and *srl* cause some of the data bits to be lost. In order to be able to restore fully the initial value of the focus register, we must mask out these bits and store them in the scratchpad memory before applying the operation. When the program reaches the inverse block, it first applies the reverse operation (*i.e.,* shift in the opposite direction in this case) and then loads and restores the bits from memory. Finally, the outcome of an instruction may depend on the sign or value of the focus register, which is not known at generation time. For example, shift-arithmetic right (*sra*) will preserve the sign of the value by replicating its most significant bit. Blocks verifying such value-dependent operations can be built to execute differently based on the operand's value, saving and restoring all the bits required to deterministically retrieve the initial data.

Load/store instructions. In Reversi the correctness of load and store instructions is checked by copying a data structure: a region of memory is initialized with random values and load/store pairs are used to copy it to a new location. We do not require that the copy preserves the order of the bytes, rather, we treat the data structure as a pool of values, which can appear out of order at destination (see Figure 3.2). This allows programs generated by Reversi to closely resemble real software applications where loads bring data from memory to the processor, and stores copy results of the computation back. Moreover, because of their random nature, Reversi programs contain a variety of cache and memory access patterns, that can expose corner-case bugs in the memory subsystem. To check the correctness of the final state of the memory, we simply compare an *xor-hash* of the memory values before and after test execution. This approach allows Reversi to expose load/store related issues such as illegal memory accesses and/or data corruption.

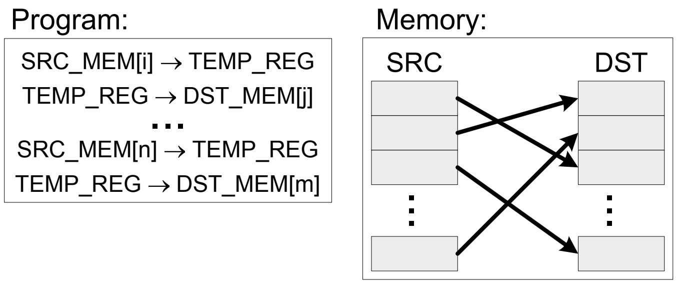


Figure 3.2: Blocks for load and store instructions. Load/store pairs are used to copy bytes from source to destination data structures. Bytes may be reshuffled, but their *xor-hashes* must match.

Branch instructions. The block database of Reversi also contains templates that test branches with different properties: forward/backward, taken/nottaken *etc.* For example, an operation block for a forward taken branch (Figure 3.3.a) contains a store operation that saves the value of the focus register to scratchpad memory, followed by a load that overwrites the focus register with a predetermined constant and then by the branch itself. Although the constant is generated randomly, its value is dependent on the type of the tested branch. For example, a template for a *beq* (branch if equal) instruction, overwrites the focus register with a random constant and then loads a temporary register with the same value to test the branch. With reference to Figure 3.3.a, the destination of the branch is located in the inverse block, which also contains a load operation restoring the focus register and a return jump. Therefore, if all control flow instructions are executed correctly, the value of the focus register after execution of the block is preserved. If, however, the branch is not taken by a faulty hardware, the focus register would not be restored. Note that the inverse block is only accessible via the proper branch and is skipped otherwise. If, however, the unconditional branch in Figure 3.3.a is not taken due to a bug, the *halt* instruction is executed and the test stops without fully reverting processor's state. The structure of the blocks for forward not-taken branch (Figure 3.3.b) is similar to the one described above differing in the position of the restoring load. We use a similar technique to detect other faulty control flow operations, initializing all unused locations in the program to *halt* instructions.

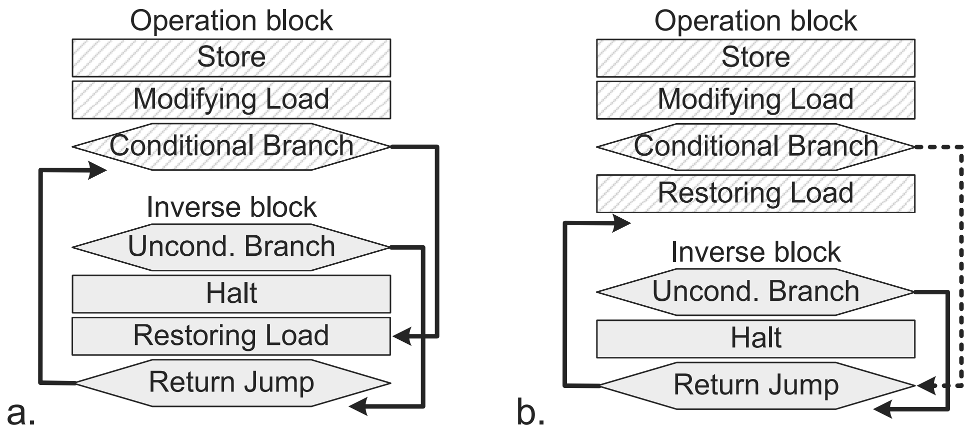


Figure 3.3: Branch operations. a. Block pair for forward taken branch. The operation block includes a modifying load, a branch and the return label, while the inverse block contains a restoring load and a return jump. **b.** Structure of a forward not-taken branch. The dashed line indicates the program flow for a case when the branch is taken by the faulty hardware.

Control register manipulation. In many modern processors there exist several control registers. In general terms we can classify them into two groups: *mode control* registers, that can only be accessed by special instructions and specify the machine's mode of operation; and *Execution flag* registers, that cannot be changed by the user but are affected indirectly by executed instructions. For instance, a register enabling/disabling the first level cache is a mode control register, while a register storing the ALU overflow bit or outputs of the comparator (*equal, greater than, etc.*) is an execution flag register.

Reversi exploits the fact that the value of the mode control registers can be modified only through specific instructions, and must remain unchanged throughout other parts of program execution. To test the proper operation of a control register, the following sequence of steps is taken (Figure 3.4.a). In the operation block the old value of the control register is first stored to memory, then the new bit-mask is loaded to the control register and also *xor*ed with the focus register. In the inverse block, Reversi first accesses the value of the control register and *xor*s it with the focus register, restoring the previous mode of operation from memory afterwards. Thus, if the control register was erroneously modified between the execution of the operation and the inverse block, the focus register's value would reflect this error.

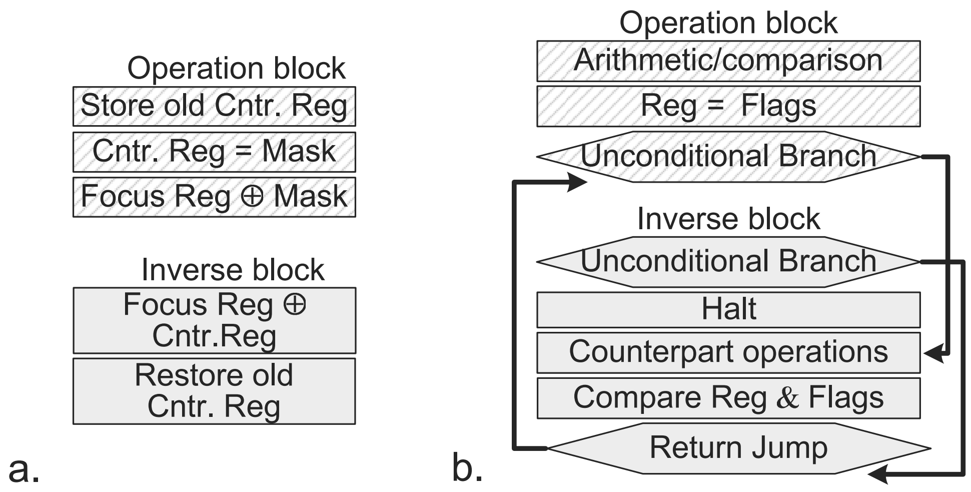


Figure 3.4: Handling of instructions affecting control flags. a. Block pair for testing mode control registers. An erroneous register operation is reflected in the focus register's value. **b.** Block pair for instructions affecting execution flag registers. The operation block includes an arithmetic/comparison instruction setting the flag bits and copying the resulting flag vector. The inverse block performs the counterpart action and compares resulting flags with the vector from the operation block.

Reversi can also check the correctness of execution flag registers, because instructions affecting them (arithmetic, logic and comparison) have counterparts in terms of which flags they set. For instance, if *comp $r1, $r2* sets the *greater-than* bit, then *comp $r2, $r1* must set the *less-than* bit. Similarly, knowing that $a + b > c \equiv b > c - a$, we can check that an *add* sets the overflow bit in the execution flag register correctly. In this case

add $r1, $r2, $r3 # overflow *i.e. $r3 > MAX*

can be checked through subtraction and comparison:

sub MAX, $r1, $r3
comp $r2, $r3 # must set greater-than bit

where *MAX* is the largest number that can be stored in the register. Thus, individual bits of the flag register can be used to check other flag bits. The Reversi block structure for execution flag register validation is presented in Figure 3.4.b. The operation block executes a comparison or an arithmetic operation that affects the flags, stores the flags values in a register and jumps to the inverse block. The inverse block then executes the counterpart operations, obtains the resulting flags and checks if they correspond to previously computed ones. Recalling the example above, first the *add* executes in the operation block and then, in the inverse block, we check that if an overflow bit was set by the *add*, then a greater-than bit is set by the *comp* instruction.

Floating point instructions. Floating point operations present a unique challenge to Reversi due to the their inherent imprecision: In the majority of cases the result of the computation is rounded, making it impossible to restore the operands exactly. To address this issue we must recognize this intrinsic approximation and take into account the relative error that is introduced by these operations. We do so by constructing a table indexed by the exponents of the operands and checking the relative error after every operation against the expected boundaries. Although this solution will not lead to a strictly reversible program, the approach is still viable for floating point error detection.

3.1.2 Limitations

Although the Reversi framework allows to create high-coverage tests with a verifiable final state, the proposed approach has a few limitations. The most important one stems from the fact that Reversi relies on the existence of inverse functions that can fully and precisely restore the internal state. For example, if the integer division operation was implemented in such a way that the remainder is lost, the value of the dividend could not be restored precisely. Similarly, floating point instructions do not exhibit such precision, however, they can still be partially verified by the approach described above. Unfortunately, input and output operations are inherently irreversible and cannot be easily covered by Reversi. For example, external interrupts cannot be expected to arrive at a certain time and cannot be "undone" by the core. In addition, Reversi may fail in targeting special execution cases for instructions whose output depends on the operands' values. For instance, a divide-by-0 operation may trigger an exception or set an error bit. Due to the randomness of operand values generated by Reversi, it is unlikely that a zero divisor occurs. To address this, the Reversi database can be augmented with specialized blocks to exercise corner case situations deterministically.

3.1.3 Reversi Generator

As described above, reversible programs generated by Reversi consist of sequences of operation and inverse blocks instantiated from the block database. However, a single sequence of blocks only alters a single focus register, therefore, to create complex programs, Reversi generates multiple block sequences (called *stacks*), each altering a different focus register. The stacks are then interleaved into a complex reversible test program, as Figure 3.5 illustrates.

Stack generation. During the test generation, Reversi randomly selects functional blocks from the database and creates a user-specified number of *stacks*, each consisting of several

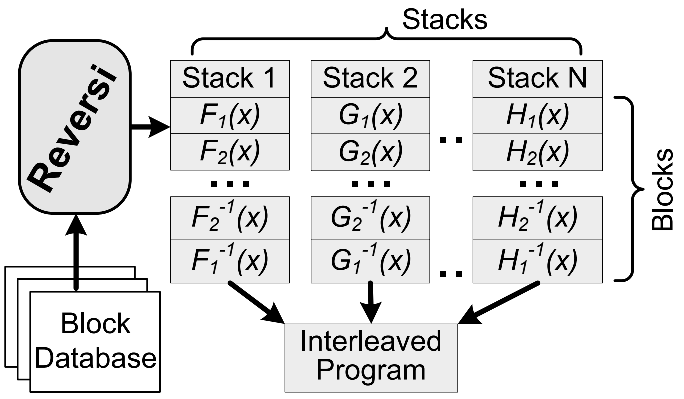


Figure 3.5: Reversi operation. Given a database of functional blocks, Reversi produces a set of *stacks*, consisting of blocks and inverse operations assembled in reverse order. Each stack operates on a single *focus register*, modifying it in such a way that its final value matches the initial one. The stacks are then interleaved into a reversible program.

blocks and their inverses. Each stack has only one focus register selected at random. The blocks are then arranged so that inverse blocks follow operation blocks in inverse order (see Eq. 3.1). On a properly working processor the focus register should be restored to its original value once a stack execution completes. Reversi also allocates a set of temporary registers to each of the stacks, based on the requirements of its blocks. We chose to allocate completely disjoint sets of registers to each stack to simplify the interleaving.

Stack interleaving. After the required number of stacks is generated, Reversi interleaves them by selecting instructions from all stacks and chaining them together to form a single test program. Note that some instructions may be grouped together into "atomic operations", meaning that the interleaving phase cannot insert instructions between them. The atomicity indicator is provided in the block definition in the database. To balance the selection algorithm, we attribute different probabilities of selection to each stack, based on its length, so to avoid a long tail from a single stack at the end of the program. The probability of selecting the next instruction from a given stack j is:

$$P_j = \frac{|stack_j|}{\sum_i |stack_i|} \tag{3.2}$$

where $|stack_j|$ is the number of atomic operations in $stack_j$, and all P_j's are adjusted after each removal of an atomic operation. Note that the requirements of using disjoint sets of registers in each stack limits the total number of stacks that we can have in Reversi. We chose to forego more complex dynamic register set partitioning (as in some compiler techniques) in favor of faster test generation.

The test program includes one last routine that calculates the final *xor-hash* of the destination memory data structure. When the program terminates the final values of the focus registers and the hash of the destination memory are compared to the initial state computed by Reversi during the generation to determine if the test executed successfully. It is also important to note that Reversi programs can provide more aid in debugging than traditional randomly generated programs. If the test results indicate that there is a bug in the processor, a validation engineer can quickly check if the exposing instruction sequence is located in an individual stack, by re-running the program without interleaving. Insights into the nature of the bug can also be found by "peeling" operation and inverse blocks from the program. Therefore, a reversible program exposing a bug can be dramatically shortened to alleviate debugging. In contrast, in a traditional flow a costly re-simulation is required to obtain the new golden state after each change of the test program.

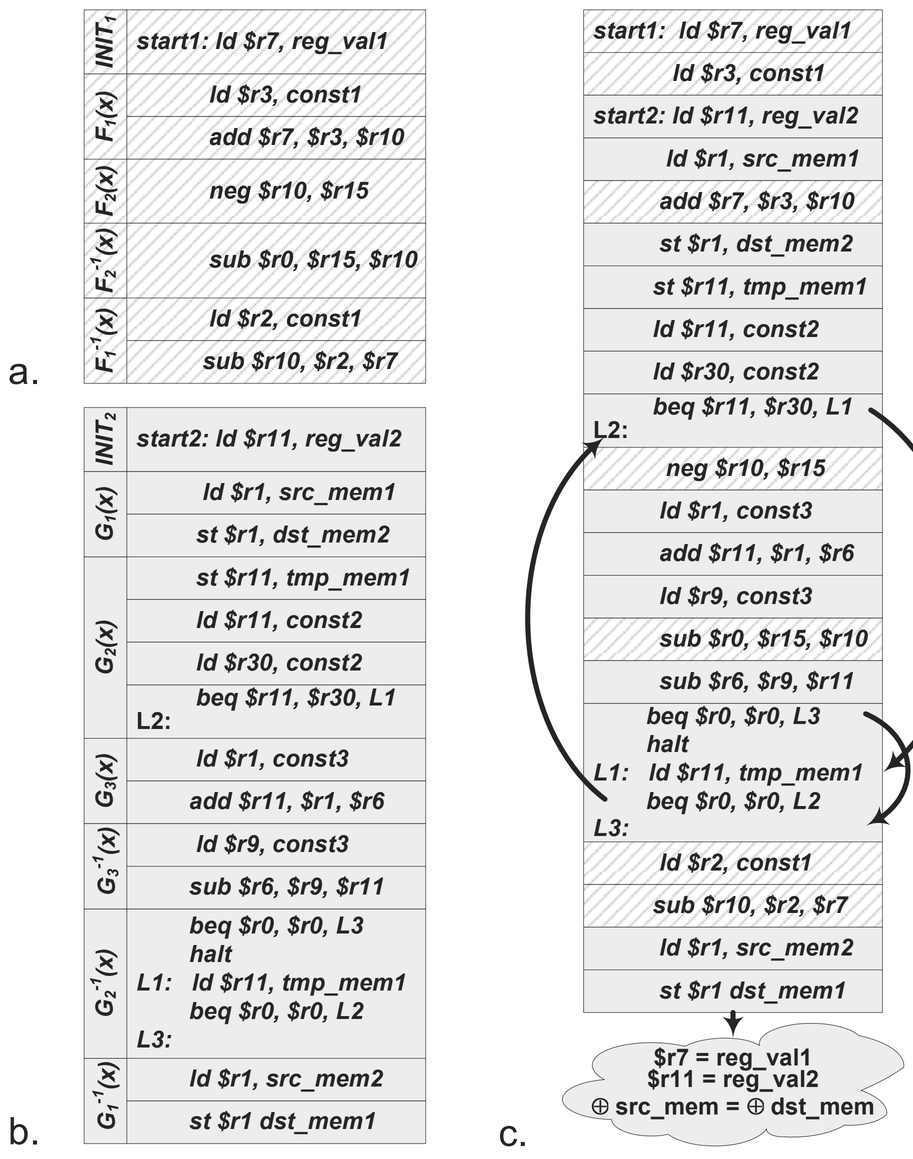
3.2 Example

This section presents an example of a program generated by Reversi for a simple instruction set presented in Table 3.2. Two stacks for this ISA using focus registers $r7 and $r11 are shown in Figure 3.6.a and 3.6.b. For both stacks the function blocks are indicated in the left column and boxes mark atomic actions. The stack in Figure 3.6.a contains simple arithmetic/logic operations, while the stack in 3.6.b includes logic instructions, load/store pairs and forward taken conditional branches. Sets of register IDs for both stacks are allocated dynamically by Reversi and are disjoint. Initial focus register values (*reg_val1* and *reg_val2*), constants (*const1*-*const3*) and location accessed by the loads and stores in the program are also selected at random.

Table 3.2: Example instruction-set architecture.

Instruction	**Semantics**
halt	Stop the execution
add $r1, $r2, $r3	$r3=$r1+$r2
sub $r1, $r2, $r3	$r3=$r1-$r2
neg $r1, $r2	$r2= -$r1
ld $r1, var	$r1=MEM[var]
st $r1, var	MEM[var]=$r1
beq $r1, $r2, label	PC=($r1==$r2) label : PC+1
Register $r0 is hardwired to the value 0	

An interleaving of the stacks into a program is shown in Figure 3.6.c. Conditions that must hold after this program executes are: $r7=*reg_val1*, $r11 = *reg_val2* and $\bigoplus$*src_mem* =$\bigoplus$*dst_mem*. So, by using the resulting values of the focus registers $r7 and $r11 and



```
start1: ld $r7, reg_val1
        ld $r3, const1
        add $r7, $r3, $r10
        neg $r10, $r15
        sub $r0, $r15, $r10
        ld $r2, const1
        sub $r10, $r2, $r7
```

```
start2: ld $r11, reg_val2
        ld $r1, src_mem1
        st $r1, dst_mem2
        st $r11, tmp_mem1
        ld $r11, const2
        ld $r30, const2
        beq $r11, $r30, L1
L2:
        ld $r1, const3
        add $r11, $r1, $r6
        ld $r9, const3
        sub $r6, $r9, $r11
        beq $r0, $r0, L3
        halt
L1:     ld $r11, tmp_mem1
        beq $r0, $r0, L2
L3:
        ld $r1, src_mem2
        st $r1 dst_mem1
```

```
start1: ld $r7, reg_val1
        ld $r3, const1
start2: ld $r11, reg_val2
        ld $r1, src_mem1
        add $r7, $r3, $r10
        st $r1, dst_mem2
        st $r11, tmp_mem1
        ld $r11, const2
        ld $r30, const2
        beq $r11, $r30, L1
L2:
        neg $r10, $r15
        ld $r1, const3
        add $r11, $r1, $r6
        ld $r9, const3
        sub $r0, $r15, $r10
        sub $r6, $r9, $r11
        beq $r0, $r0, L3
        halt
L1:     ld $r11, tmp_mem1
        beq $r0, $r0, L2
L3:
        ld $r2, const1
        sub $r10, $r2, $r7
        ld $r1, src_mem2
        st $r1 dst_mem1
```

Figure 3.6: Test program for the example ISA. a. Stack with arithmetic/logic operations. **b.** Stack with arithmetic operations, load/store pairs and forward taken branches. **c.** Interleaving of atomic operations in stacks a. and b. and exit condition of the test.

the xor-hash of the *dst_mem* data structure, we can quickly determine if the program has exposed any functional bugs. Note also that the branch in block G_2 was generated by Reversi to be taken. Thus, during correct operation, the execution should modify the value of \$r11 and jump to the label $L1$. Then the processor restores the value of the focus register and takes the unconditional branch returning to $L2$. When operating properly, the processor should not visit line $L1$ again and skip directly to $L3$. Moreover, if the branch in G_2 is not taken, then the exit condition described above does not hold, exposing a bug.

3.2.1 Experimental Framework

To evaluate the performance of our Reversi approach, we created two reversible instruction block databases: one implementing a subset of the Alpha instruction set and another implementing a subset of the x86 ISA. The database for the Alpha instruction set contained 17 distinct blocks for arithmetic and logic functions testing a range of instruction formats (reg/reg and reg/imm) and 5 blocks for each type of compare instructions. In addition to that, the database included 3 blocks for load and store instructions, an unconditional jump block and 16 branch blocks containing 4 distinct branching instructions, each in four possible modes (fw/bw and taken/nottaken). Similarly, the x86 block database contained 32 logic-arithmetic blocks testing multiple instruction formats (reg/reg, reg/imm, reg/ mem, mem/reg), 3 load-store blocks, 1 compare block and 40 branch blocks. Reversi itself is implemented as an optimized program in C that created and interleaved a specified number of stacks and contained routines to set a random initial state and perform the final check. The blocks are partially pre-assembled in binary, and Reversi is responsible for setting the appropriate bit-fields with register IDs, randomly generated constants, *etc.*

3.3 Experimental Evaluation

In this section, we first present our experimental evaluation platform and two of our Reversi setups. Then, we evaluate the performance of these setups against a traditional solution based on a constrained-random instruction sequence generator. Finally, we investigate bug-finding capabilities of Reversi in our last experiment.

To compare Reversi with a traditional post-silicon validation flow (Figure 3.1.a), we created an assembly-level constrained-random test generator. In addition, for the architectural simulation phase of the traditional post-silicon flow we used M5 2.0b3 [50] and Bochs-2.3.5 [14] for Alpha and x86 systems, respectively. Test generation and simulation for both Reversi and the traditional post-silicon flow was performed on a 3.2GHz Pentium 4 machine with 2GB of memory.

3.3.1 Performance Evaluation

In our first experiment we compared the validation performance of the traditional post-silicon flow with the Reversi flow. In this case the total time for the traditional flow consisted of i) the time to create a program on the constrained-random test generation, ii) the time for the instruction set simulator (either M5 or Bochs) to obtain the golden state and iii) the execution time on the silicon prototype. For the Reversi flow we need to include i) the Reversi generation time and ii) the time to execute on silicon. Performance for the Alpha

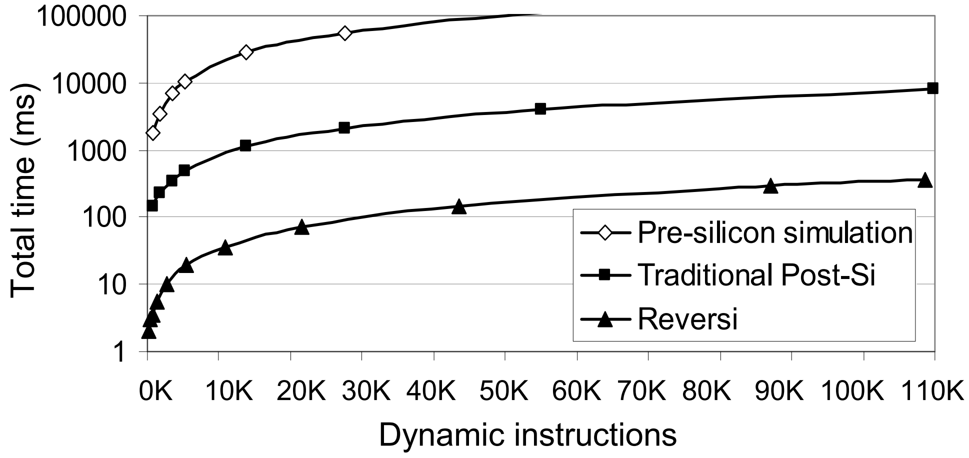


Figure 3.7: Total testing time for a traditional post-si flow and Reversi: Alpha instruction set. The total time for the traditional post-silicon flow includes the test program generation, simulation and execution time. The time for Reversi includes test generation and execution. For comparison we plot the speed of a pre-silicon validation technique based on RTL simulation.

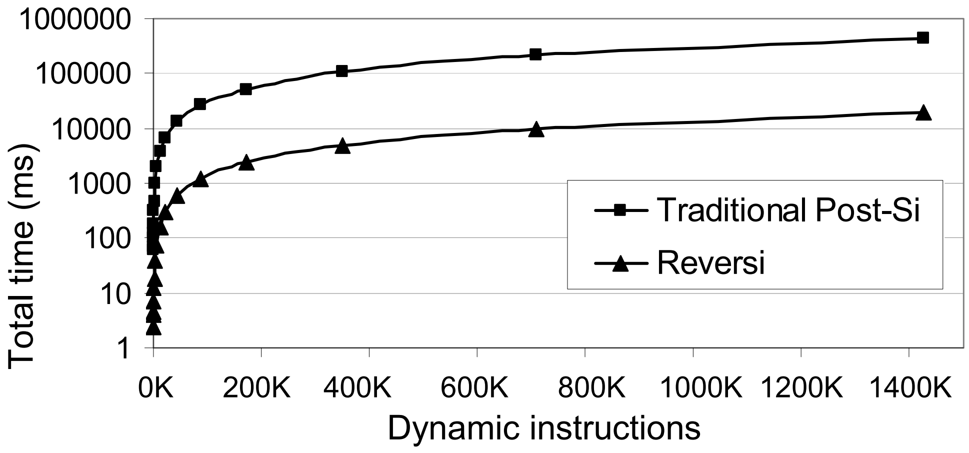


Figure 3.8: Total testing time for a traditional post-si flow and Reversi: x86 instruction set. The total time for the traditional post-silicon flow includes the test program generation, simulation and execution time. The time for Reversi includes test generation and execution.

design was measured over shorter program sequences, while x86 used longer testing programs. The results of the experiments are presented in Figures 3.7 and 3.8. In Figure 3.7 we also plot the performance of a typical pre-silicon simulator (using a behavioral Verilog model of the Alpha design) for comparison. As these figures demonstrate, the Reversi-based approach flow provides a 19.5x and 21.5x performance improvement for Alpha and x86 designs, respectively. It should be noted that in addition to eliminating the simulation step from the flow, Reversi is more efficient because it operates on pre-assembled blocks. In a traditional approach, on the other hand, the generator must frequently solve fairly

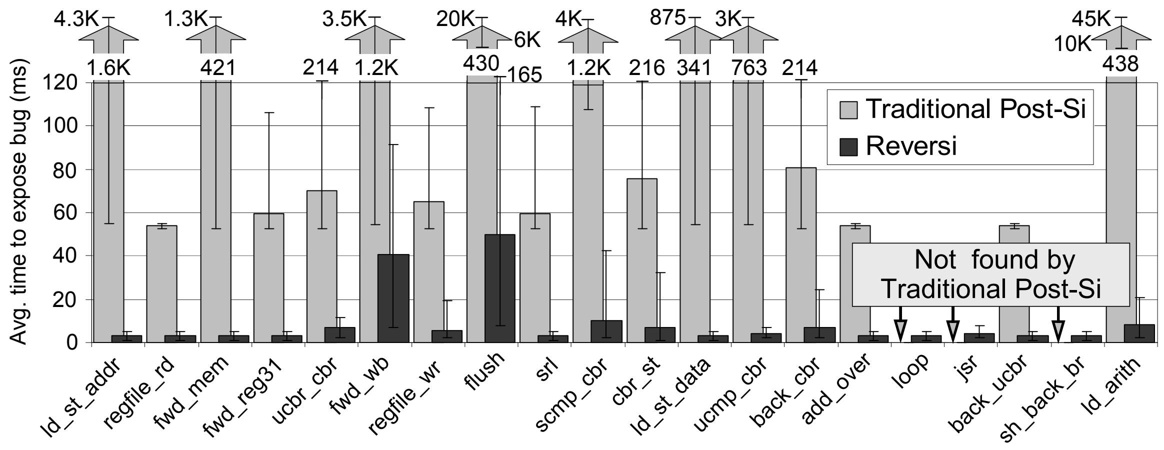


Figure 3.9: Average time to discover bugs in the traditional post-silicon flow and Reversi. The experiments were run 10 times with different random seeds and the minimum, average and maximum times to expose each bug are plotted. Note that bugs *loop, jsr* and *sh_back_br* were not exposed by a traditional post-silicon flow based on a constrained-random test generator.

complex constraints to produce valid and meaningful tests. Moreover, due to the presence of branching instructions and PC-relative branches, the program generator must produce tests in assembly language and then call an assembler to convert it to machine codes. Reversi, however, does not need an external assembler, since it implements internally all functions required to generate the binary code.

3.3.2 Design Error Coverage

In the second experiment, we use an RTL implementation of a 5-stage pipeline running Alpha ISA to create 20 designs, each containing a single bug from Table 3.3.2. During the test, both the traditional flow and Reversi generated code of increasing length until the bug was exposed. Note that, in order to identify an error with a randomly generated program, we first need to compute the correct final state by running it on a known-correct model. We run the experiment 10 times with different random seeds and calculate the minimum, average and maximum time required for the traditional flow and Reversi to expose the fault (Figure 3.9).

As the results in Figure 3.9 demonstrate, Reversi can find all errors faster than the traditional post-si flow. Furthermore, some of the bugs, such as *loop, jsr* and *sh_back_br*, were not exposed by the post-si flow in any of the runs. We believe that this was due to the unique nature of the programs generated by Reversi - they are designed so that only correctly operating hardware produces an easily verifiable result. Thus, incorrect operations can be detected immediately at execution completion. Moreover, Reversi creates complex programs with multiple interleaved execution flows that exercise all instructions in the ISA, exposing these corner-case bugs.

Table 3.3: Bugs introduced in Alpha design.

Bug	Description
ld_st_addr	load to store address forwarding fault
regfile_rd	faulty internal forwarding in register file read port A
fwd_mem	error in forwarding dependency resolution
fwd_reg31	forwarding through register 31 (const 0)
ucbr_cbr	unconditional branch after conditional branch fails
fwd_wb	unnecessary forwarding from wb stage
regfile_wr	invalid write access to register file
flush	pipeline flush on specific register file access
srl	invalid execution of logical right shift
scmp_cbr	invalid forwarding from signed compare to a branch
cbr_st	backward conditional branch after a store is not taken
ld_st_data	load to store data forwarding fault
ucmp_cbr	invalid forwarding from unsigned compare to a branch
back_cbr	specific backward conditional branch is never taken
add_over	incorrect handling of overflow on add
loop	incorrect execution of looping sequence
jsr	incorrect handling of jsr with invalid address
back_ucbr	fault in backward unconditional branch
sh_back_br	fault in branch resolution for short backward branch
ld_arith	invalid execution of a load followed by arithmetic

It's worth observing that, in several experiments with the traditional flow (such as *fw_wb*), a shorter random program exposed a bug, while a longer sequence of instructions did not. This is possible due to the random nature of the test: later instructions may overwrite registers/memory locations that contain incorrect values, thus eliminating the evidence of the bug. Therefore, a longer random program does not necessarily find more bugs than a shorter one. Reversi programs, on the other hand, are designed so that any behavior corrupting the processor state is propagated to the exit point and exposed.

3.4 Summary

This chapter presented a novel post-silicon validation methodology that exploits the performance potential of hardware prototypes and bypasses the design simulation step required by traditional flows. Test programs that our Reversi framework generates explore complex execution scenarios and, most importantly, have identical initial and final architectural states, eliminating the need for a simulator to check the correctness of the test. The programs are built from sequences of functional blocks, which modify the state of the machine, combined with inverse blocks to undo earlier operations and restore the original machine state. Individual blocks are parameterized and may consist of one or several instructions, selected randomly from a block database during test generation. Reversi handles all types of instructions: arithmetic (integer and floating point), logic, memory accesses, control flow and control register operations. As our results demonstrate, Reversi creates programs capable of finding more bugs faster than traditional constrained-random test generation techniques. Moreover, due to the omission of the architectural simulation step, Reversi can generate and run tests 20x faster than tools based on a traditional post-silicon flow. Moreover, with Reversi we take the first step towards a novel high-throughput post-silicon validation methodology, which allows for test generation to match the performance of execution of the silicon prototype.

CHAPTER IV

Hardware Patching of a Single Core

Once a system is verified to satisfaction during its development, a design house will ramp up its production and start distributing it to the customers. Unfortunately, due to shortening development timelines and rapidly increasing complexity of modern processors, released designs are never fully and exhaustively verified and often contain subtle errors. These *escaped bugs* are eventually identified when the system is already deployed in the field and cannot be easily corrected by the manufacturer, since they require modification of the actual silicon chip. Recognizing the inevitability of such errors and the need to efficiently fix them without a costly product recall, researchers in recent years started to develop hardware patching solutions. Our analysis in Section 4.1 demonstrates, that the majority of such escaped errors occur in the control logic portion of the chip, which traditionally has been the hardest portion of the design to verify. To enable efficient in-the-field patching of processor control logic, in this chapter we introduce a reliable, low-cost and extremely expressive mechanism called Field-Repairable Control Logic (FRCL). In our framework, when an escaped bug is found in the field, the support team investigates it and generates a pattern describing the control state of the processor which causes the bug to manifest itself. The pattern is then sent to the end customers as a patch and is loaded into the on-die *state matcher* at startup. The matcher constantly monitors the state of the processor and compares it to the stored patterns to identify when the pipeline has entered a state associated with a bug. Once the matcher has determined that the processor is in a flawed control state, the processor's pipeline is flushed and forced into a *degraded mode* of operation for the execution of the next instruction.

In the degraded mode, the processor starts execution from the first un-committed instruction and allows only one operation to traverse the pipeline at a time. Therefore, much of the control logic that handles interactions between operations can be turned off, which enables a complete formal verification of the degraded mode at design time. In other words, we can guarantee that instructions running in this mode complete properly, and thus can ensure forward progress, even in the presence of design errors by simply forcing the pipeline to run in degraded mode. After the error is bypassed in degraded mode, the processor returns to *high-performance* mode until the matcher finds another flawed control state. In designing the state matcher, we have put special care into creating a system that can detect multiple design errors with minimal false positive triggering. In addition, for cases when the number of patterns of design errors exceeds the capacity of a given matcher, we developed a novel compression algorithm which compacts the erroneous state patterns, while minimizing the number of false-positives introduced by this process.

In this chapter, we extend the ideas of the field-repairable control logic technology to a solution that provides protection even from unknown escaped bugs in systems deployed at the end-customers site. Instead of requiring a company to identify an escaped bug before we can repair it, we simply assume that *any configuration that was not verified at design time is potentially a buggy configuration*. Thus, in our semantic guardian solution, we protect a system against all those configurations that were not verified at design time. We will show in Section 4.5.6 that we can indeed deliver such protection at minimal performance costs since, even if the unverified portion of a design is extensive, unverified configurations tend to occur rarely at runtime. Moreover, we discuss how we can match a large set of unverified configuration with little area overhead.

4.1 Background

In this section we first analyze escaped design errors reported in errata of several commercial processors, thereby identifying which modules of modern microprocessors are most prone to error. We then overview approaches to runtime verification being researched in academia, as well as two patching solutions used by processor vendors as a stop-gap measure against escaped errors in their products deployed in the field.

4.1.1 Analysis of Escaped Errors in Commercial Processors

Despite the impressive efforts of microprocessor manufacturers to build correct designs, bugs do escape the verification process. This section examines the reported escaped errors of a number of ARM, x86, and PowerPC processors. When these bugs are classified we can observe that a fairly large fraction of them is related to the control portion of the design. The results of the study are listed in Figure 4.1. This chart summarizes the bugs reported in x86 [24, 44, 6, 39, 38, 40, 37], StrongARM-SA1100 [36], and PowerPC 750GX [41] processors. Errors are classified into one of the following categories:
Processor's control logic: These bugs are the result of incorrect decisions made at the occurrence of important execution events, and of bad interactions between simultaneous

events. An example of this type of escape could be found in the the Opteron processor, where a reverse REP MOVS instruction in certain cases causes the following instruction to be skipped [6]. The solution proposed in this work addresses precisely this type of bugs.
Functional units: These are design errors in functional units which can cause the production of an incorrect result. This category includes bugs in microarchitectural components, such as branch predictors and TLBs. An (infamous) example of this type of bug is the Pentium processor FDIV bug, where a lookup table used to implement the division algorithm contained incorrect entries [24].
Memory system control: These are bugs in the implementation of the on-chip memory system, including caches, memory interface, and instruction pre-fetcher. An example of this type of bug could be found in the Pentium III processor, where certain interactions of Instruction Fetch Unit and Data Cache Unit on the main bus could cause the entire system to hang [40].
Microcode: These are (software) bugs in the implementation of the microcode for a particular instruction. An example of can be found in the 386 processor, where microcode incorrectly checked the minimum size of the TSS segment, which must be 103 bytes, but, due to a flaw, segments of 101 and 102 bytes were also incorrectly allowed [44].

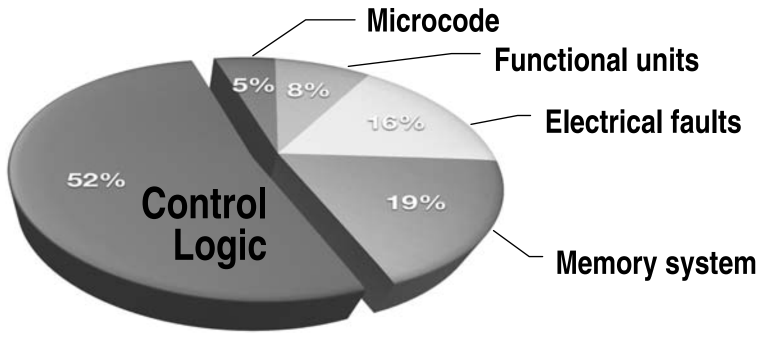


Figure 4.1: Classification of escaped bugs found in commercial processors. The chart shows the number of occurrences and percentage of each particular type of bugs. Analysis from x86 [24, 44, 6, 39, 38, 40, 37], StrongARM-SA1100 [36], and PowerPC 750GX [41] processors.

Electrical faults: These are design errors occurring when certain logic paths do not meet timing under exceptional conditions. Consequently, if a processor runs well below its specified maximum frequency, these faults will often not occur. An example is the Load Register Signed Byte (LDRSB) instruction of the StrongARM SA-1100 which does not meet the required timing constraint when reading from the pre-fetch buffer [36].

As the analysis above demonstrates, control logic escapes dominate the errata reports for these processors. The high frequency of such escapes can be explained by the complexity of the control logic blocks that handle interactions between multiple instructions and the inability of formal techniques to handle complex interactions between multiple logic blocks in a design. Related studies on sources of design errors corroborate these findings, an example being the work by Van Campenhout [19], reporting that many design flaws are the result of incorrectly implemented interactions between major components or an unforseen combination of rare events. Furthermore, analysis of the multi-core processor errata reveals that 10% of the bugs in these designs occur in the memory subsystem [25]. All of these studies clearly indicate the inability of pre-release verification to produce a completely bug-free design, before a component is shipped to the end user. Thus, there exist a need for effective solutions to augment processors with hardware checkers that can detect, diagnose and correct errors at runtime.

4.1.2 Hardware patching approaches

Runtime verification used in industry and proposed by researchers can be divided into two groups: checker-based [8, 56, 57] and patching [64, 30, 55] solutions. In the former, the detection is done by a special checker block, which dynamically analyzes the execution trace, detects anomalies or violations of invariants and triggers recovery. DIVA [8] is one of the first checker-based solutions for ensuring correctness of execution in a single core processor. This framework augments a complex processor pipeline with a simplified second core, which validates the results of execution and recovers from detected errors. A recent work by Meixner, *et al.* [56] describes another checker-based runtime verification solution for simple processor cores. It leverages compile time information about program execution flow that is then compared against signatures generated dynamically by checkers distributed throughout the core. When an error is detected through signature mismatch, the system invokes a backward error-recovery scheme known as SafetyNet [67]. SafetyNet can also be used as the underlying recovery scheme in checker-based solutions for runtime verification of multi-core chips, such as the one in [57]. Patching-based approaches, on the other hand, do not rely on checking of system-wide invariants, but try to change the software or the semantics of its execution to prevent the processor from executing sequences of instructions that may expose the issue To date, there are two techniques in this domain that have been deployed commercially:

Instruction Patching: Software patching can sometimes correct the execution of an instruction which has an erroneous implementation [69]. In this approach, the program code is inspected, and if a broken instruction is encountered, it is replaced with an alternative implementation, typically through a function call to a correct emulation of the instruction.

This technique was used as the initial work-around for the Pentium FDIV bug using software recompilation. Linux- and Windows-based compilers were updated to generate code which would run a preliminary test, to determine if the underlying processor suffered from the FDIV bug. If the test indicated so, a divide emulation routine would be called to avoid using the hardware divider [69]. A similar technique was used to port Windows NT to Alpha processors [16]: A bug in the underflow exception mechanism forced Alpha software developers to make the operating system step in and handle the offending instructions in software. A specific advantage of this approach was that it could operate in a completely transparent fashion to the user (beside the requirement of installing an operating system patch). Performance-wise, however, this approach does not seem very promising. For example, the FDIV fix [61] in the Microsoft Visual C++ compiler incurs 100% worst case performance overhead on a flawed processor. Moreover, on a correctly working chip it still causes up to 10% overhead.

Microcode Patching: Intel and AMD processors reportedly have the ability to update their microcode after deployment in the field [30, 55, 20]. During system startup, microcode patches are loaded into a small on-chip buffer, which overrides existing microcode in on-chip ROMs. A microcode patch can change the semantics of any instruction, similar to instruction patching. An added advantage of microcode patching is that no changes are necessary to existing software, since the patching occurs during the instruction's decode stage. The concept of patchable microcode is not new, as many early computers such as the Xerox Alto and DEC LSI-11 supported writable micro-stores, thus allowing engineers to update the implementation of individual instructions at any time [9].

While these techniques have proven their positive impact in commercial solutions, they have limited value because of their high performance impact, and due to their inability to cope with complex control bugs. For example, in the case of the Pentium FDIV bug, all divide instructions had to be tested for susceptibility to the bug and replaced with an emulated routine if needed, which resulted in significant slowdowns. Additionally, many control logic bugs cannot be easily associated with a particular instruction, and thus they could not be fixed with any of these techniques. For example, on the 486 processor, if a non-maskable interrupt (NMI) occurred in the same cycle as a global segment violation, the violation would not be detected [24]. Short of emulating every instruction, this bug could not be fixed with instruction patches. To address these shortcomings solutions patching control logic of the chip directly (such [64]) have recently started to appear in academic research community.

4.2 Field-repairable Control Logic (FRCL) flow of Operation

This section presents usage flow and the process to correct escaped bugs for a design incorporating field-repairable control logic (FRCL) technology. We also show the structure of the state matcher circuit and present a pattern compression algorithm for cases when the number of patterns exceeds the size of the matcher. Finally, we analyze an example of an actual bug that is repaired using our approach.

Field-repairable control logic is designed to handle flaws in processor control circuitry for components already deployed in the field. The flow of operation that we envision for this approach is summarized in Figure 4.2. When an escaped error is detected by the end customer, a report containing the error description, such as the sequence of executed operations and the values in the status registers, is sent to the design house. Engineers on the product support team investigate the issue, identify the root cause of the error and which products are affected by it, and decide on a mechanism to correct the bug. As was mentioned above, instruction or microcode patching are valid approaches, however, they can have a very high performance overhead or be too costly. We propose that the engineers instead use our solution, field-repairable control logic. By knowing the cause of the bug and which signals are monitored by the matcher in the defective processors, the engineers can create patterns that describe the flawed control state configuration. The patterns then can be compressed by the algorithm presented in Section 4.2.3, and sent to the customers as a patch. The patches in the end system are loaded into the *state matcher* at startup. Every time the patched error is encountered at runtime, a recovery via degraded mode, detailed in Section 4.2.4, is initiated, effectively fixing the bug.

4.2.1 Pattern Generation

The pattern to address a design error can be created from the state transition graph (STG) of a device. The correct STG consists of all the legal states of operation, where each state is a specific configuration of internal signals, that are crucial to the proper operation of the device. In addition, these states are connected by all the legal transitions between them. Within this framework an error may occur because of an additional erroneous transition from a legal state to an illegal state, that should not be part of the STG, or when an invalid transition connects two legal states, or by the lack of a transition that should exist between states (Figure 4.3). In our solution we add hardware support which uses *patterns* to detect both the illegal states and the legal states which are sources of illegal transitions. A pattern is a bit-vector representing the configuration of the internal signals that is associated with erroneous behavior of the processor. Note that in this framework

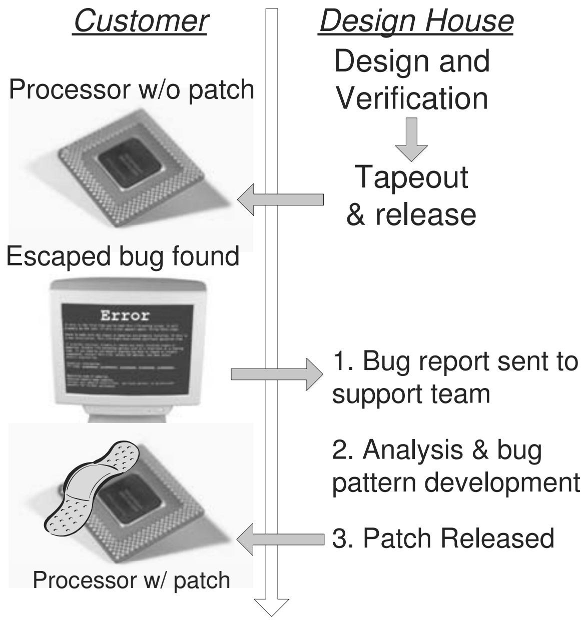


Figure 4.2: Field-repairable control logic usage flow. After a component is shipped to the end customer and a new bug is found, a report detailing the bug is sent to the support team. The error is analyzed and patterns representing the control states associated with the bug are issued as a patch. On every startup, the processor loads the patterns into the state matcher and, if a bug is encountered, it is bypassed through the reliable degraded mode.

a single bug can be mapped to multiple patterns, if it is caused, for example, by multiple illegal states. To cope with this problem we incorporated a range of features into our technology, including a novel pattern compression algorithm presented in Section 4.2.3. In a real-world scenario, after receiving a bug report, a product support team would analyze the issue, try to reproduce the error and understand what caused it. Tools such as trace minimizers can be very helpful for this analysis, since they can significantly shorten a trace that leads to a bug, which helps immensely in the debugging process. Moreover, some of these tools, for example Butramin [35], investigate alternative simulation scenarios that reach the same bug. This allows the support team to pinpoint multiple processor control states associated with the bug and identify how these states map to the critical signals observed by the matcher in the design. Afterwards, the configurations of the critical control signals are compactly encoded and issued as a patch to the end customer. The process is repeated when new bugs or new scenarios exposing known bugs are discovered.

4.2.2 Matching Flawed Configurations

As was mentioned above, design errors and patterns describing them in our framework are defined through configurations of control signals of the processor and transitions between these configurations. At run-time these signals are continuously observed by a *state*

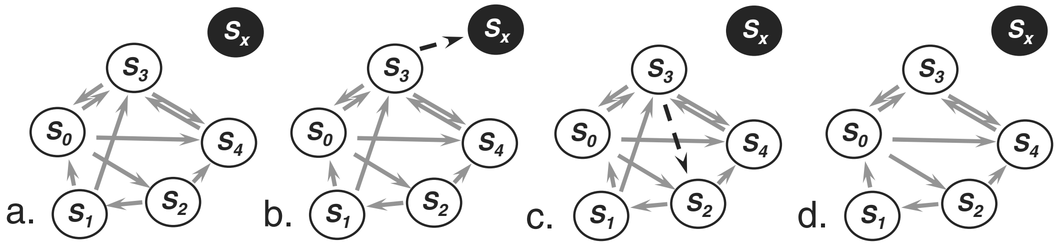


Figure 4.3: Error representation in the state transition graph (STG) framework. a. Correct STG of the device. (S_x is an unreachable illegal state). **b.** Erroneous STG due to a transition to an illegal state S_x **c.** Erroneous STG due to an illegal transition between legal states S_3 and S_2 **d.** Erroneous STG due to the absence of a legal transition $S_1 \rightarrow S_3$.

matcher and compared to pre-loaded patterns describing bugs. Therefore, only the bugs that manifest themselves on these critical signals can be detected by the matcher. Ideally, all of the design's control signals could be used for this purpose, however, complexity and stringent timing constraints of modern chips prevent such extensive monitoring, allowing only for a small portion of the actual control state to be routed to the matcher. In Section 4.3.3 we present techniques to intelligently select these critical state bits among the prohibitively large control state of a processor.

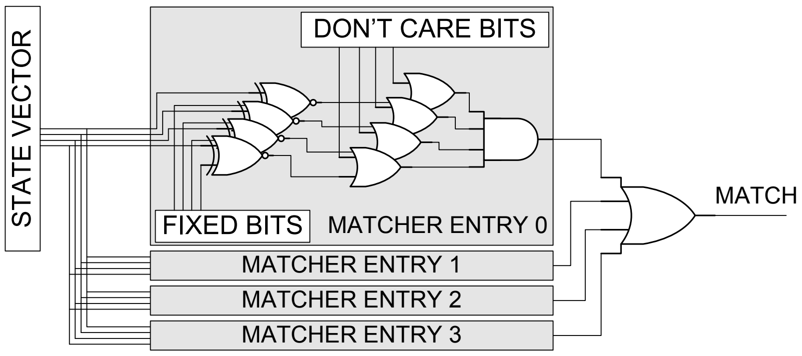


Figure 4.4: State matcher structure. The critical control state vector is first compared against the fixed bits in a bug pattern. Then the don't care bits in the pattern are overlayed, and the result is reduced to a single match bit. The matcher contains multiple independent entries to allow for multiple simultaneous comparisons.

The state matcher can be thought of as a fully-associative cache with the width of the tag being equal to the width of the critical control state vector, which in our experiments was just several tens of bits long. The tag in this case is the pattern describing an erroneous configuration, thus if such a tag exists in the cache, then a hit occurs and a potential bug is recognized. In order to improve the performance of the matcher we structured it to allow the use of don't care bits in the patterns to be matched. The don't care bits help to make a compact representation of multiple individual configurations of the critical control state that differ in just a few bits. Using our state matcher, designers issuing a patch can specify a bug pattern through a vector of 0's, 1's and don't care bits (x): 0's and 1's represent the fixed value bits, while x's can match any value in the corresponding control signal. Note, however, that the control state observed by the matcher at run-time contains only fixed bit values 0 and 1. Figure 4.8 shows several examples of bug patterns loaded into a four-entry state matcher.

We also anticipate that a single patch may consist of multiple bug patterns since a single bug may be associated with several patterns, as was mentioned above, or the design may contain multiple unrelated bugs. To handle this situation we developed a matcher with multiple independent entries, as shown in Figure 4.4. On startup, each of the matcher's entries is loaded with an individual pattern containing fixed bits and don't cares. At run-time the matcher simultaneously compares the actual critical control bit values to all of the valid entries and asserts a signal if at least one entry matches the control state. The number of entries in the matcher is set at design time and is one of the engineering trade-offs. A larger matcher can be loaded with more patterns, however, it occupies larger area on the die and has longer propagation delay. A smaller matcher, on the other hand, might not be able to load all of the patterns and compression would be needed.

4.2.3 Pattern Compression Algorithm

The pattern compression algorithm that we developed was inspired by the two-level logic minimization techniques described in [54]. Our algorithm compresses a number k of patterns into a state matcher with r-entries, where $k > r$. This process, however, often over-approximates the bug pattern and introduces *false positives*, *i.e.* error-free configurations that will be misclassified as buggy, and incur some performance impact. Nevertheless, this compression is necessary to fit the patching patterns into an available state matcher of smaller size.

To map k patterns into an r-entry matcher, the algorithm first builds a *proximity graph*. The graph is a clique with k vertices, once for each of the k patterns, and weighted edges connecting the vertices. The weights on the edges are assigned using a variant of the Hamming distance metric. Specifically, we use an additive metric whereby corresponding bits are compared one to one, and each $0-1$ pair contributes 1 to the weight, while each $1-x$ or $0-x$ pair contributes 0.5 to the weight. Matching pairs ($0-0$, $1-1$, and $x-x$) do not contribute to the weight. As an example, consider the two patterns $101xx1$ and $1001x1$ shown in Figure 4.5. The two leftmost and two rightmost bits of the patterns are identical, thus they contribute 0 to the weight. Bits 3 of the patterns, on the other hand, form a $0-1$ pair, contributing 1 to the weight, while bits 4 form a $x-1$ pair, making the total weight on the edge between these patterns 1.5. The reasoning behind this weighing structure is fairly straightforward: if we were to compact the two patterns connected by an edge, we would have to replace every discording pair ($0-1$, $x-0$, and $x-1$), with an x, basically creating the minimum common pattern that contains both of the initial ones. Matching pairs, however, would retain the values they had in the original patterns. For example, for the two patterns $101xx1$ and $1001x1$ mentioned above, the common pattern is $10xxx1$, since we have two discording pairs in the third and fourth bit positions. With this algorithm, each $0-1$ pair contributes the same degree of approximation in the resulting entry generated. However, pairs such as $1-x$ or $0-x$, will only have an approximating impact on one of the patterns (the one with the 0 or 1), leaving the other unaffected, hence the corresponding weight is halved.

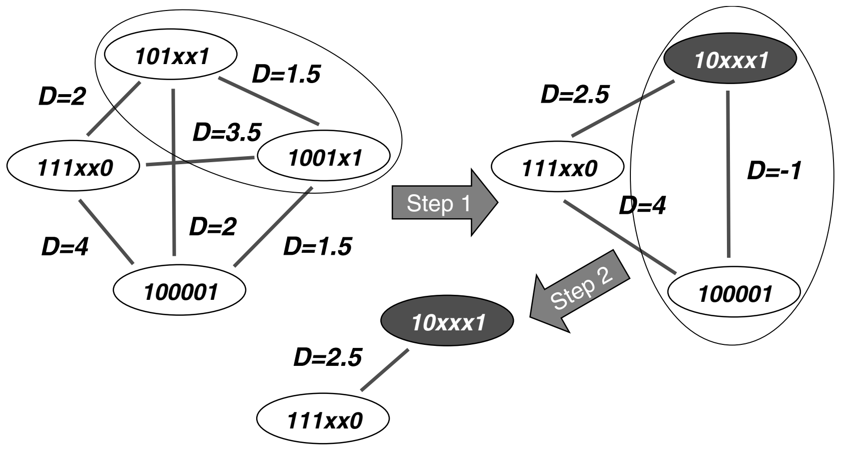


Figure 4.5: Pattern compression example. Four bug patterns are compressed to fit into a two-entry matcher. A complete graph of the initial four patterns is computed and is labeled with a variant of distance. The first compression step combines the two closest (in terms of distance) patterns *101xx1* and *1001x1*. The resultant patterns *10xxx1* has fixed bits in every position where original patterns were identical and don't care bits (x) in all other positions. In the second step, pattern *100001* is eliminated, since it is a subset of the pattern *10xxx1*, as the -1 label indicates.

An exception to the above metric is a case when one pattern is a subset of another pattern. This is possible, because we allow patterns to have don't care bits that essentially represent both 0 and 1 values. In our framework we set the distance between such proximity graph vertices to -1, guaranteeing that these vertices will be chosen for compression, and the more specific pattern be eliminated from the graph.

Once the proximity graph is built, the two patterns connected by the minimum-weight edge are merged together. If $r \leq k$, the compression is completed, otherwise the graph is updated using the compressed pattern just generated, instead of the two original ones, and the process is repeated until we are left with a number of patterns that fits in the matcher.

An example of a compression is given in Figure 4.5. Here, for simplicity, we assume that the matcher can only contain two entries and initially there are four bug patterns. After the proximity graph is initially built and edges are labeled, the algorithm selects the edge with the smallest distance ($D = 1.5$) and merges patterns $101xx1$ and $1001x1$ connected by it. As was shown above, the resulting pattern is $10xxx1$. When the graph is updated after the first step, it has three vertices and is still too large for the matcher. Note, however, that pattern that was added ($10xxx1$), completely overlaps pattern 100001, thus the edge between them is labeled with distance -1. When the algorithm searches for the edge with the smallest weight for the second step, this edge is selected and vertex 100001 is eliminated. Compression then terminates, since the resulting set of patterns can fit into the two-entry matcher.

Figure 4.2.3 shows a pseudo-code for the pattern compression algorithm. Lines 2 to 7 generate the initial proximity graph by computing the weights of all the edges either by detecting that vertex i contains vertex j (*contains* function) or computing the distance using the algorithm described above (*compute_distance* function). Lines 9 to 11 select the pair to merge, remove one pattern from the set and update the graph. The procedure is repeated until we reach the desired number of patterns. Function *merge* in line 10, generates a pattern that is the minimum over-approximation of the two input patterns. The function first must check for containment, in which case it returns the former one. If there is no containment between the two patterns, their approximation is computed by generating an x bit for each non-matching bit pair. It is worth noting that the performance of the algorithm described could be optimized in several ways, for instance by eliminating all edges with $D = -1$ in the graph at once.

As was mentioned before the compression algorithm generates a set of patterns that *over-approximates* the number of erroneous configurations. The resulting pattern will still be capable of flagging all the erroneous configurations, however, it will also flag additional correct configurations that have been included by the merging function (*false positives*). The impact on the overall system will not be one of correctness, but one of performance, particularly if the occurrence of the additional critical control configurations is frequent during a typical execution. We measure the amount of approximation in the matcher's detection ability as its *specificity*. The specificity is the probability that a state matcher will not flag a correct control state configuration as erroneous. Specificity can also be thought of as $1 - false\_positive\_rate$. Hence when there is no approximation, the matcher has an ideal specificity of 1; increasing over-approximation produces decreasing specificity values. It is important to note that by virtue of our design and the pattern compression algorithm, our system never produces a false negative, that is, it never fails to identify any of the bug states observable through the selected critical control signals.

```
 1  PatternCompress(){
 2  for each (pattern i)
 3    for each (pattern j != i) {
 4      if (contains(i, j))
 5        weight(i, j) = -1
 6      else
 7        weight(i, j) =
              compute_distance (i, j) }

 8  while (num_patterns > matcher_lines) {
 9    (i, j) = edge_with_minimum_weight
10    pattern i = merge(pattern i, pattern j)
11    delete pattern j
12    update graph
13    num_patterns -- }}
```

Figure 4.6: Pattern compression algorithm. A proximity graph is initially generated and labeled in lines 2-7. The two closest patterns are merged and the graph is updated in lines 9-12. The cycle is repeated until the patterns can fit into the fixed size matcher.

4.2.4 Processor Recovery

At this point, the set of patterns generated and compressed is issued to the end customers as a patch. We envision this step being similar to current microcode patching flow, where a patch for the processor is included into BIOS (Basic Input-Output System) updates. Updates are distributed by operating system or hardware vendors and are saved in non-volatile memory on the motherboard. At startup, when BIOS firmware executes, the patches are loaded into the processor by a special loader. FRCL can use an almost identical mechanism, and we expect FRCL patches to be approximately of the same size of a microcode update (~2KB or less). After the patch is loaded at startup into the matcher and the processor starts running. While none of the configurations recorded in the matcher is detected, activity proceeds normally (we call this mode of operation *high-performance*). However, when a buggy state is detected, the pipeline is flushed and the processor is switched to a reliable degraded mode of execution. Figure 4.7 shows an example of the execution flow when a bug pattern is matched in a FRCL-equipped processor. In the example we consider a simple in-order single-issue pipeline, and we further assume that the interaction between a particular pair of instructions *INST2* and *INST3* triggers a control bug which has been detected and encoded in a pattern already uploaded in the matcher. The Figure shows that, when the pattern is detected by the matcher (Figure 4.7.a), the pipeline is flushed (Figure 4.7.b), and the processor is switched to the degraded mode. This mode is formally verified at design time; hence, we can rely on it to correctly

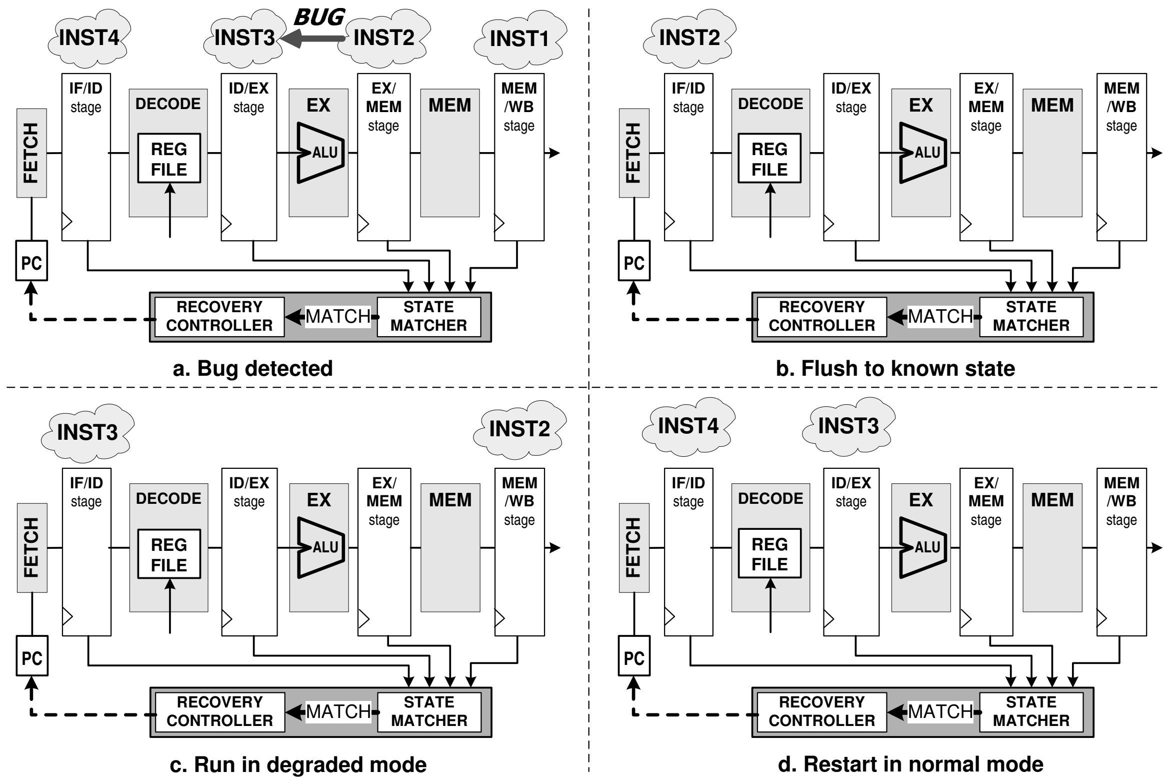


Figure 4.7: Field-repairable control logic (FRCL) in operation. a. Matcher detects a state associated with a bug described in a pre-loaded pattern. **b.** Pipeline is flushed to a known state. **c.** Processor runs in degraded mode allowing only one instruction in the pipeline at a time. Degraded mode is formally verified, guaranteeing forward progress and correctness. **d.** After offending instructions are bypassed normal operation is resumed.

complete the next instruction (Figure 4.7.c). Finally, the high-performance mode of operation is restored (Figure 4.7.d). Without FRCL technology, a problem such as the one just described would probably have required re-writing the compiler software or the microcode related to the instructions, to circumvent the bug configuration. Note that it is sufficient to complete only one instruction before re-engaging normal operation since, in the event that the pipeline steps again into an error state, it will once again enter the degraded mode to complete the following instruction. On the other hand, a designer may choose to run in degraded mode for several instructions to guarantee bypassing the bug entirely in a single recovery.

It also should be noted that, unlike some implementations of the microcode update mechanism which allow for buggy patches to be loaded [5], our technique cannot introduce new flaws into the processor, since our patches only specify when a processor switches to degraded mode. In the worst case, the processor runs in degraded mode all the time, with notable performance impacts, but providing correct functionality.

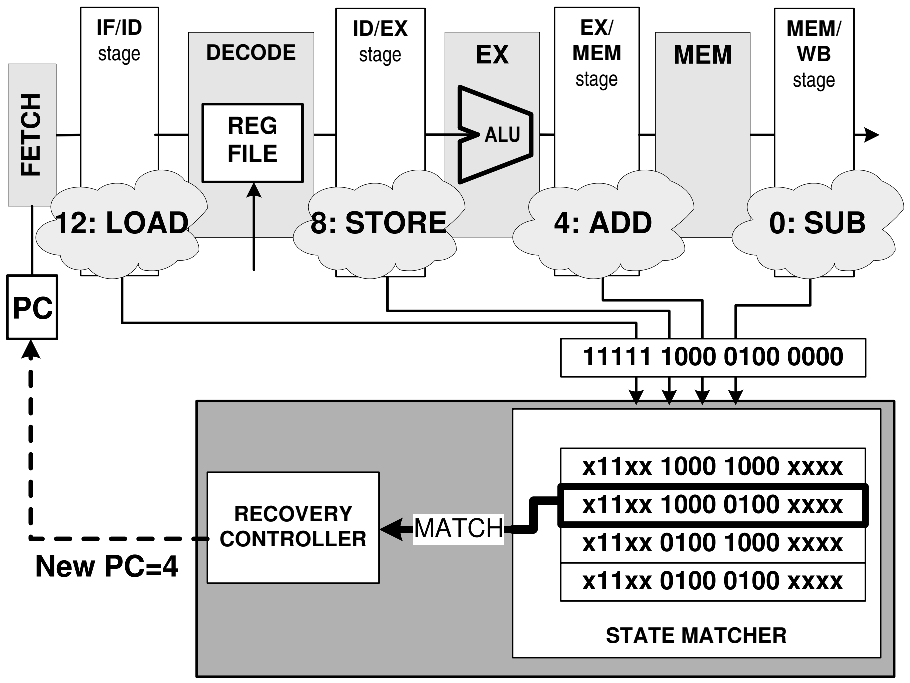


Figure 4.8: FRCL for a memory access bug. Without FRCL two consecutive memory accesses (8:STORE and 12:LOAD) would be erroneously allowed to proceed back-to-back in the pipeline. When the bug is recognized by the state matcher, the pipeline is flushed and execution restarts at the first uncommitted instruction (4:ADD). In degraded mode instructions do not go through the pipeline back-to-back, avoiding the bug.

4.2.5 Example

We now show the use of field-repairable control logic through an example similar to the Intel Celeron bug listed in [24], which we adapt, for simplicity reasons, to a five stage pipeline. In this example, the processor has a flow that does not always enforce a necessary stall between two successive memory accesses. A stall is required, since all memory operations are performed in two cycles: during the first one the address is placed on the bus and the data from or to memory follows during the second cycle. If a memory operation is followed by a non-memory instruction, they are allowed to proceed back to back, since the second operation does not require memory access while advancing through the MEM stage of the pipeline.

In the example, the program that is being run contains a store and a load back to back, which triggers the bug described. The matching logic in this case contains four entries that describe all possible combinations of having two memory instructions in the ID and EX stages of the pipeline. For instance, the first entry matches valid instructions in the ID and EX stages of the pipeline which are both memory reads. The second entry matches a store in EX followed by a load in ID, which is triggered during the program execution (Figure 4.8). The pipeline is flushed, then the recovery controller restarts execution at the instruction preceding the store, that is, the first uncommitted instruction. Note that in this case the bug is fully and precisely described by the four patterns loaded in the matcher, thus no false-positive matches are produced.

4.3 Design Flow

In this section we describe a design and verification flow that incorporates field-repairable control logic technology. First, we show how the traditional design process needs to be changed to incorporate field-repairable control logic and then investigate formal verification of the degraded mode of operation. Then we move on to overview control state selection techniques, including our novel automatic selection algorithm. Finally, we present some insights on incorporating performance critical execution into an FRCL-protected hardware design.

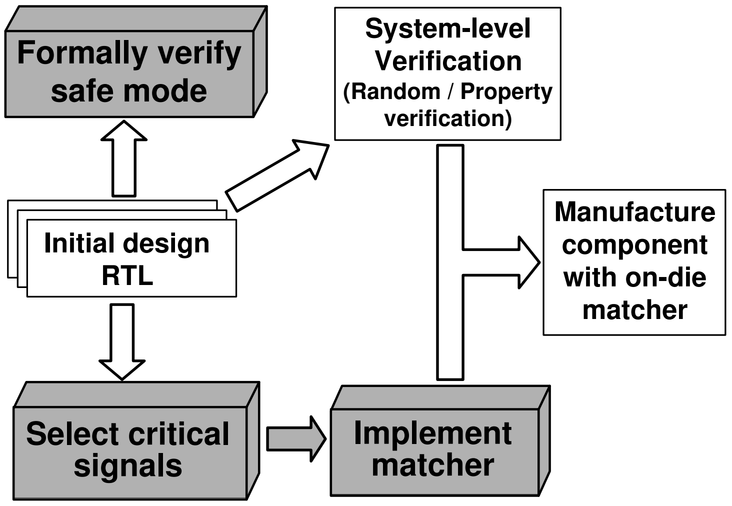

Formally verify safe mode
System-level Verification (Random / Property verification)
Manufacture component with on-die matcher
Initial design RTL
Select critical signals
Implement matcher

Figure 4.9: Field-repairable control logic design flow. Using the initial RTL designers formally verify the degraded mode and select control signals to be monitored by the state matcher. A matcher is incorporated into the final design shipped to the end customer.

4.3.1 Overview of the Design Framework

The overall design flow of a component augmented with field-repairable control logic is shown in Figure 4.9. As was mentioned above, verification of complex hardware components such as microprocessors relies today on a variety of formal and simulation-based methods. The deployment of FRCL technology in a processor design requires the addition of two steps to the mainstream design flow. The first step requires to formally verify the processor when operating in degraded mode, needed by FRCL to recover from patched design errors. Note that we setup the degraded mode so that instructions are never interacting, hence verification is greatly simplified. For the most part this verification effort is reduced to the verification of individual functional blocks, which are, already today, heavily addressed by formal verification techniques. The system-level verification of the entire processor is still performed using the same mainstream methodology that was used before the deployment of FRCL, typically a mix of random simulation and formal property verification.

The second additional task during the system design is the selection of the signals that should become part of the "critical control state". These signals are then routed to a state matcher, which was sketched in Figure 4.4. The number of entries in the matcher is subject to a tradeoff between total design area and overall performance of the deployed component, since a smaller matcher might require compression and reduce the processor's performance because of increased false positives.

4.3.2 Verification Methodology

In addressing the formal verification of the degrade mode of operation, we exploited a series of optimizations made available by its specific setup. Most of the complex functionality of the processor is disabled in this mode and only one instruction is allowed in the pipeline at any time, greatly reducing the fraction of the design involved in each individual property proof. To this end, it is important to note that it is not necessary to create a new, simplified version of the design. Instead, all of the simplifications are achieved either as a direct consequence of the nature of the input stream – only one instruction is in flight at any one time – or by simply disabling the advanced features through a few configuration bits. For example, modules such as branch predictors and speculative execution units can be turned off with a variant of the "chicken bits", which are control bits used in many design developments to enable and disable features. On the other hand, control logic responsible for data forwarding, squashing and out-of-order execution would be abstracted away by the formal tools, due to the fact that only one instruction appears in the pipeline at a time and these blocks are irrelevant. These two major simplifications make the degraded mode operation simple enough for traditional formal verification tools to handle.

In our experiments we used Magellan from Synopsys to verify both our testbed processor designs. Magellan is a hybrid verification tool that employs several techniques, including formal and directed-random simulation first presented in [32]. Since instructions are executed independently, we use Magellan to verify the functionality of each instruction in the ISA, one at a time.For each instruction, we wrote assertions in the Verilog hardware design language to specify the expected result. Constraint blocks fixed the instruction's opcode and function field, while immediate fields and register identifiers were symbolically generated by Magellan to allow for verification of all possible combinations of these values. The first module, *add_valid* guarantees that only valid instructions, ADDitions in this case, are in execution. The second checker, *add_forward* enforces forward progress by forcing the instruction to complete in a set number of clock cycles. Finally, *add_sem* enforces the correct semantic for additions by checking that the correct result is written to the register file during the writeback stage. For more complex instructions, such as loads and branches, additional checkers are needed to prove that the execution of the operation on the degraded pipelined machine matches exactly the ISA specification. While we could completely verify the degraded mode for both our testbeds, it should be pointed out that neither could be verified in high-performance mode, because of the much greater complexity involved.

4.3.3 Control Signal Selection

A critical aspect of deploying a field-repairable control logic is determining which control state signals are to be monitored by the matcher. On one hand, it would be ideal to monitor all the sequential elements of a design; however, given the amount of control state in complex designs, such approach would be either infeasible or extremely costly. For FRCL to be practical, the set of critical control signals should be just a handful, selected among any internal net of the design; although this limitation could potentially be the source of false positives at runtime. An example of the impact of a poor signal selection is discussed in Section 4.5, where we describe a bug, *r31-forward*, used in our experimental evaluation, which describes an incorrect implementation of data forwarding through register 31. In the Alpha ISA register 31 has a fixed value 0, and hence cannot be a reference register for data forwarding. If the critical signal set does not include the register fields of the various instructions in execution, it is impossible to repair this bug without triggering all those configurations which require any type of forwarding, causing an extremely high rate of false positives.

We envision two possible solutions to address this problem. The first and simplest is to monitor the destination register indices of the instructions at the EX/MEM and MEM/WB stage boundaries by including them in the critical signal set. The downside of this solution is that the critical signal pool would grow and possibly impact the processor's performance, for our in-order experimental testbed this would be a 30% increase in the signals monitored. The alternative solution entails including a comparator asserting when a forwarding on register 31 is detected and one additional single bit – the output of the comparator – to the critical set. The additional overhead in this case would be less than

```
//a.  RTL checker for ADD validity
module add_valid ( INST, valid, fail );
  assign fail = valid & (INST[‘OPCODE] != ‘ADD);
endmodule

//b.  RTL checker for ADD forward progress
module add_forward (clock, reset, IR, valid, ...);
  reg [3:0] count_committed_adds; //saturating
  reg [3:0] clk_cnt; //saturating
  assign fail = (clk_cnt == 4’d5) &
             (count_committed_adds == 4’d0);
endmodule

//c.  RTL checker for ADD semantics
module add_sem (clock, reset, add_in_id,
          add_in_wb, write_dest, write_data, ...);

  reg [63:0] read_a_, read_b_; //operands from RF
  //result register ID read in decode stage
  reg [4:0] dest_id;

  //shows that destination is chosen correctly
  assign fail1 = !((!add_in_wb) || ((add_in_wb)
               & (write_dest == dest_id)));
  // shows that addition is performed properly
  assign fail2 = !((!add_in_wb) || ((add_in_wb)
            & (write_data == read_a_ + read_b_)));
endmodule
```

Figure 4.10: RTL checkers to verify the correctness of the ADD instruction with Synopsys's Magellan. Checkers verify a. the presence of only a valid ADD instruction in flight, b. forward progress, and c. correctness of execution.

the previous alternative. Both approaches would eliminate the false positives for the r31-forward bug and hence improve the processor's performance. Therefore, a designer using the FRCL approach should keep in mind possible corner cases such as this and design and select his critical control pool for a broad range of bugs. A possible approach for this task would be analyzing the previous designs to gain a sense of where bugs have been exposed.

4.3.4 Automatic Signal Selection

Since the critical signal selection is of key importance for FRCL, we have developed a software tool to support a designer in this task. The tool considers the register-transfer level (RTL) description of the design and it narrows the candidate pool for the critical control set. It does so by first automatically excluding poor candidates such as wide buses, and then ranking the remaining candidates in decreasing relevance. The rank is computed based on the width of the cone of logic that a signal drives and the number of sub-modules that they feed into. For example, for the RTL block shown in Figure 4.11, the critical state selection tool marks signal A as data, and signals B and C as control. However, B will have a higher control signal ranking, since it drives more signals than C – B drives C plus all the nodes that C drives, indicating that it is probably a more important control signal.

When comparing our manually selected critical signal set with the output of the automatic signal selector tool, we noted an 80% overlap. It should be noted that the manual selection was performed by a designer who had full knowledge of the micro-architecture, while the automatic selection tool was only analyzing the RTL design.

```
module example (A, B, C)
    input [64:0] A;
    input B;
    output C;
        assign C = !B & (A == 64'h0);
endmodule
```

A	data
B	Control rank 1
C	Control rank 2

Figure 4.11: Example of automatic control selection for a simple module. Signal A is labeled as data because of its width, and signal B is a higher ranked control signal than C, since it drives C.

In Section 4.5 we present an experiment comparing the performance, in terms of *specificity* (precision of the bug detection mechanism), of a range of variants of manual and automatic selection. In particular, we looked at the average *specificity per signal*, or the measure of how much each signal is contributing to the precision of the matcher. Solutions with higher average *specificity per signal* provide higher specificity, which translates into higher performance, and require less area, for fewer signals need to be routed.

4.3.5 Performance-critical Execution

In some systems the speed of execution may be more critical than its correctness. For example, in real-time systems, it is important to guarantee task completion at a predictable time in order to meet scheduling deadlines. In streaming video applications, the correctness of the color of a particular pixel may also be less crucial than the jitter of the stream. In these situations, the field-repairable control logic approach that trades off performance for correctness may be undesirable. For these scenarios we propose having an extra bit to enable/disable the matcher (Figure 4.12). The matcher-enable bit, however, should only be modifiable in the highest privileged mode of the processor operation, to ensure that user code cannot exploit design errors for malicious reasons.

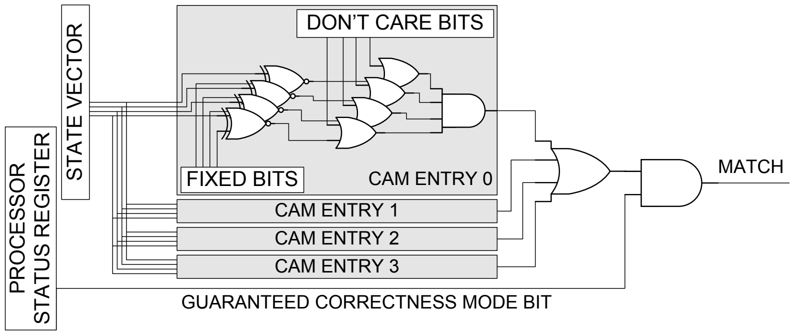


Figure 4.12: A state matcher for performance-critical execution. Matcher functionality is controlled by asserting or de-asserting a special bit in the processor status register.

4.4 Trusted Hardware Design with Semantic Guardians

Although the Field-Repairable Control Logic framework enables effective patching of a variety of complex processor bugs in the field, it can only do so with patterns generated off-line by the design support team. The most critical consequence of this gap between error discovery and availability of the remedy is a possibility that a malicious user can exploiting the bug to bypass security or otherwise attack the system before it is patched. To prevent this and enable design of truly trusted hardware, we extend the FRCL solution with *semantic guardians*. The combinational circuits of the guardians are generated automatically at design time based on the validation effort and encode all insufficiently validated configurations of the processor control state. At runtime, the guardians, similarly to FRCL matchers, monitor the critical control signals, forcing the chip to switch to the degraded mode each time an un-verified configuration is encountered. Thus, the machine always remains in a state either validated in high-performance mode, or a state validated formally in degraded mode, eliminating the chance of incorrect processor operation and security risks associated with it.

The design flow we envision for our solution is shown in Figure 4.13 and is similar to traditional design and verification flows. We require that the design team formally verifies the units that provide the key functionality of the device, *i.e.* the *degraded mode* operation. For example, core units required for a processor's operation include the datapath blocks, while forwarding logic, pipelining, branch prediction, *etc.* are performance enhancement units needed only in *high-performance mode*. As it is the case for FRCL, in degraded mode the device can perform all its necessary functions, but only at a baseline performance. Moreover, since the degraded mode provides only barebone functionalities, it is simple

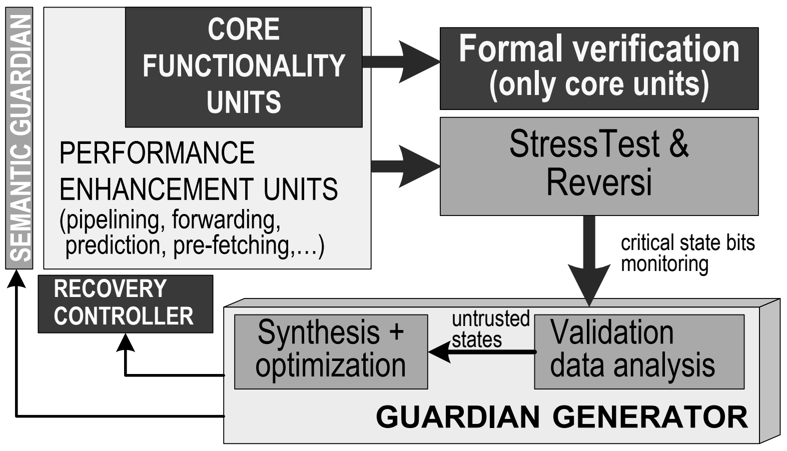


Figure 4.13: Trusted hardware design flow. The degraded mode is verified thoroughly with formal tools, while the high-performance mode is validated with focus on the most common functionality by StressTest and Reversi solutions. A semantic guardian is then automatically generated and manufactured with the design. The guardian, together with a recovery controller switches the design into the degraded mode when any non-validated scenario is observed at runtime.

enough to be tackled by modern formal verification tools. The semantic guardian solution, on the other hand, differs from FRCL framework by requiring that the high-performance mode of operation, including all the performance enhancements, is validated extensively (as it is common practice today) through a mix of semi-formal and simulation tools. The main purpose of this effort is to verify that the most typical, and frequently occurring, operation scenarios are designed correctly.

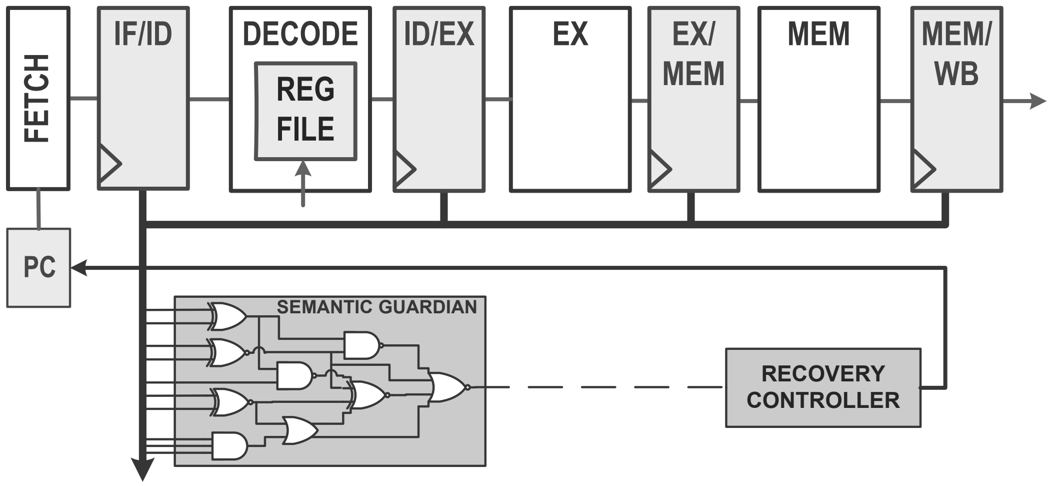


Figure 4.14: Trusted execution model with a semantic guardian. When an un-trusted state in the pipeline's operation is observed by the guardian, the recovery controller switches the processor into a safe single-cycle, single-instruction degraded mode). Once the un-validated configuration is bypassed, the device transitions back to normal execution in high-performance mode.

An important step in this process is the identification of the signals which represent the critical internal state of the design, which can be handled similarly to that of FRCL (see Section 4.3.3). The values observed on these signals during validation are monitored by the guardian generator. The guardian generator tracks which configurations of the design have been explored during the validation, and considers those to be *trusted* states, since it assumes that the verification team has validated the behavior of the device for those states. All the configurations that were not validated are considered *un-trusted*. Afterwards, the guardian generator creates a guardian, as a combinational logic block, which flags all the un-trusted states. The generator also creates a recovery controller, which is connected to the output of the semantic guardian. The controller has the responsibility of switching the design into the degraded mode whenever the guardian flags an un-trusted state. It does so by disabling all design's blocks except for the core functionality units. For example, for the pipelined processor shown in Figure 4.14, the guardian monitors the critical state set, and, when an un-trusted state is encountered, it signals the recovery controller. The controller squashes all unfinished instructions and restarts execution from the first uncommitted operation, forcing the degraded mode. When the un-trusted state is bypassed, the recovery controller restores normal mode operation.

To optimize area and propagation delay of the semantic guardian block, we introduced a range of heuristic optimizations. First, we synthesize both our un-trusted set and its complement, and then we select the smaller circuit. Then, if any of the un-trusted configurations has been proven unreachable at design time, we use it as don't care in optimizing the guardian design. Additionally, designers might choose to trade off the specificity, or accuracy, of the semantic guardian for better physical parameters of the matching circuit. In this scenario, some of the trusted states might be *re-labeled* as un-trusted and included in the set, which the guardian will flag. This operation has the potential to increase the effectiveness of the optimization algorithm, and therefore generate a smaller and faster semantic guardian. On the other hand may cause false positives in the guardian detection mechanism, hence, it is important to take the frequency of occurrence of a state into consideration when performing this type of optimization.

4.4.1 Combining Semantic Guardians and Hardware Patching

In spite of several benefits, Semantic Guardian hardware described above has one noticeable drawback - its rigidity. Once the combinational circuit of the guardian is synthesized and manufactured, it cannot be altered, restricting performance of the processor. To cope with this challenge we suggest augmenting the guardians deployed in the field with FRCL-style programmable matchers. Since semantic guardians ensure that the system is always operating in a verified state, the patching mechanism in this case is used to overrule the guardian’s decision for selected configurations. The configurations of choice would be those that have been proven correct after the design’s release, resulting in an overall performance boost for the processor.

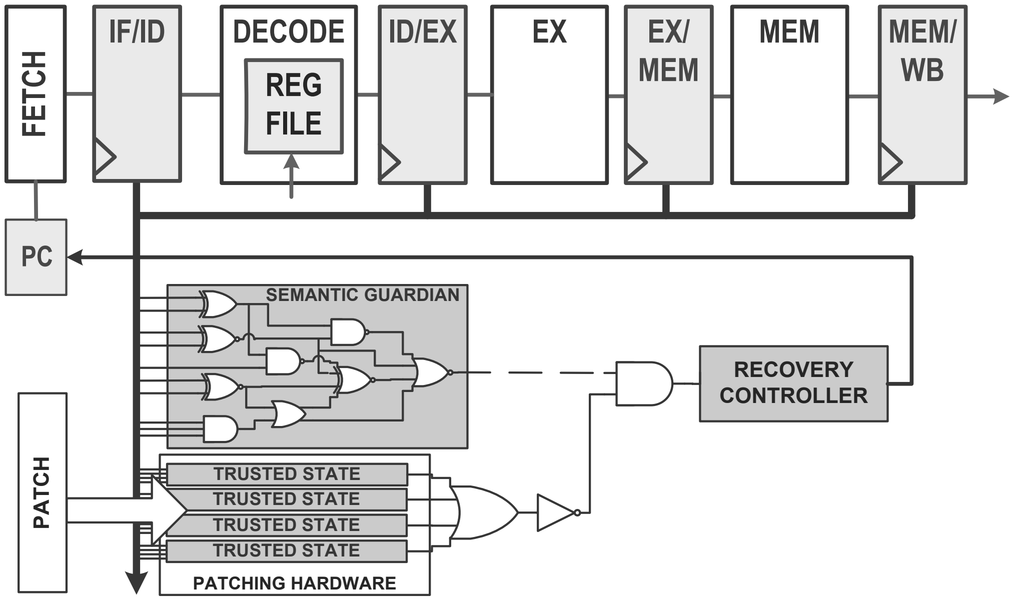


Figure 4.15: Semantic guardian and patching hardware working in synergy. A patch extending the set of trusted configurations is uploaded to the system deployed in the field. A match in the guardian can be overwritten by the FRCL matcher as a trusted state, avoiding the transition to degraded mode.

This scenario is illustrated in Figure 4.15: the hardware designer continues the device validation after the release. If and when an un-trusted configuration becomes sufficiently validated, it is encoded and uploaded onto the specialized processor memory at runtime. At this point, if the configuration is ever flagged by the guardian, the patching mechanism overwrites the decision, preventing the transition to degraded mode. Therefore, this approach allows the design team to expand the set of trusted configurations even after the device has been manufactured and shipped to the end customer.

4.5 Experimental Evaluation

In this section we detail two prototype systems with field-repairable control logic support. Using simulation-based analysis, we examine the error detection accuracy of FRCL for a number of design error scenarios and varied state matcher storage sizes. We also examine different criteria for selecting the control state, including an automatic selection heuristic outlined in Section 4.3.4. In addition, we examine the area costs of adding this support to simple microprocessors. We examine the performance impact of degraded mode execution, to see the extent of error recovery that can be tolerated before overall program performance is impacted. Finally, we investigate the semantic guardian methodology outlined in Section 4.4, evaluating several guardian area/delay optimization approaches.

4.5.1 Experimental Framework

To gauge the benefits and costs of the field-repairable control logic, we added this support to two prototype processors. Although experimental in nature, these processors have been already deployed and verified in several research projects. While these prototype processors do not have the complexity of a commercial offering, they are non-trivial robust designs that can provide a realistic basis to evaluate the field-repairable control logic solution. For our experiments we implemented two variants of the state matcher, with four and eight entries, and integrated it into the two baseline processor designs.

The first design is a 5-stage in-order pipeline implementing a subset of Alpha ISA with 64-bit address/data word and 32-bit instructions. The pipeline had forwarding from MEM and WB stages to ALU and resolves branches in EX-stage. The pipeline utilizes a simple global branch predictor and 256-byte direct mapped instruction and data caches. For this design, we handpicked 26 control bits, which govern operation of different logic blocks of the pipeline (datapath, forwarding, stalling, *etc.*), to be monitored by the matcher. These signals were selected through a two-step process: we first analyzed a variety of escaped bugs reported in commercial microprocessor errata, and then selected those control signals that would have been good indicators of those bugs. This analysis relies on the assumption that future escaped bugs are correlated to past escapes. In addition, in making our selection, we were careful to choose signals which encoded critical control situations in compact ways: for instance we chose not to monitor the indices of source and destination registers of each instruction (which require several bits each), but instead we decided to track the occurrence of each data forwarding (only a handful of bits). To limit the monitoring overhead, we also chose not to observe any of the instruction opcode bits that are marched down each pipeline stage. As detailed in Table 4.5.1, the majority of the critical control signals were drawn from the ID and EX stages of the pipeline, where the bulk of computation occurs. For example, in ID stage we selected some of the output bits of the decoder, which represent in compact form what type of operation must be executed, and in EX stage we selected the ALU control signals. Although this potentially limited our capability to recognize a buggy state before the instruction is decoded in ID stage, it allowed us to reduce the number of bits monitored. Note also that, while we did not to modify the original design in any way, it is possible to enhance the precision of the error detection with minimal additional logic. Examples are the solution to the *r31-forward* bug described

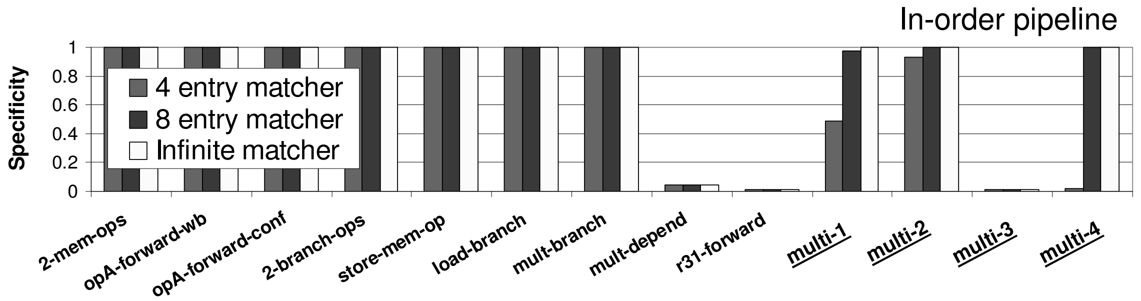


Figure 4.16: Specificity of detection for a range of bugs in the in-order pipeline. Low specificity can be due to insufficient critical control monitored through the matcher (for instance mult-depend and r31-forward) or to insufficient size of the matcher (for instance the 4-entry matcher in bugs multi-1, multi-2, multi-4).

Table 4.1: Control state bits monitored in the in-order processor pipeline.

Name	Number of bits	Pipeline stage
IF_valid	1	Fetch
ID_valid	1	Decode
EX_valid	1	Execute
MEM_valid	1	Memory
WB_valid	1	Write-back
ID_rd_mem	1	Decode
ID_wr_mem	1	Decode
ID_cond_br	1	Decode
ID_uncond_br	1	Decode
EX_rd_mem	1	Execute
EX_wr_mem	1	Execute
EX_cond_br	1	Execute
EX_uncond_br	1	Execute
EX_ALU_function	5	Execute
EX_br_taken	1	Execute
EX_hazard_source_a	2	Execute
EX_hazard_source_b	2	Execute
MEM_br_taken	1	Memory
MEM_bus_command	2	Memory
Total	**26**	

in Section 4.3.3, and also the addition of pipeline latches to propagate more complete information on the instruction being executed through the pipeline, with the result that it is possible to capture more precisely the specifics of an instruction leading to a bug.

The second processor is a much larger out-of-order 2-way super-scalar pipeline, implementing the same ISA. The core uses Tomasulo's algorithm with register renaming to re-order instruction execution. The design has four reservation stations for each of the functional units and a 32-entry re-order buffer (ROB) to hold speculative results. The flushing of the core on a branch mispredict is performed when the branch reaches the

Table 4.2: Control state bits monitored in the two-way super-scalar out-of-order processor pipeline.

Name	**Number of bits**	**Pipeline module**
ROB_head_commit	2	Re-order buffer
ROB_head_store_address	1	Re-order buffer
ROB_head_store_data	1	Re-order buffer
ROB_head_load	1	Re-order buffer
ROB_full	1	Re-order buffer
RS_full	1	Reservation Stations
RS_complete	2	Reservation Stations
RS_br_miss	2	Reservation Stations
Issue_branch	2	Rename
Issue	2	Rename
Total	**16**	

head of the ROB. The memory operations are also performed when a memory instruction reaches the head of the ROB, with a store operation requiring two cycles. The re-order buffer can retire two instructions at a time, unless one is a memory operation or a mis-predicted branch. The design also includes 256-byte direct mapped instruction and data caches and a global branch predictor. The signals hand selected for the critical control pool include signals from the retirement logic in the ROB as well as control signals from the reservation-stations and the renaming logic as reported in Table 4.5.1.

Similarly to the in-order design, no opcodes and instruction addresses were monitored, to minimize the number of observed signals. The matcher developed for this design was capable of correctly matching scenarios involving branch misprediction, memory operations, as well as corner cases of operation of the ROB and reservation stations, for example, when they were full and the front-end needed to be stalled. Again, a larger set of signals could be used to gather more detailed information about the state of the machine, however, for this design, the benefit would consist of a shorter recovery time by recognizing problems earlier on. On the other hand, the ability to identify erroneous configurations precisely would not be improved significantly, since errors can still be detected when instructions reach the head of the ROB.

The processor prototypes were specified in synthesizable Verilog, and then synthesized for minimum delay using Synopsys Design Compiler. This produces a structural Verilog specification of the processor implemented with Artisan standard logic cells in a TSMC 0.18um fabrication technology.

For performance analysis, we ran a set of 28 microbenchmark programs, designed to fully exercise the processor while providing small code footprints. These programs included branching logic and memory interface tests, recursive computation, sorting, and

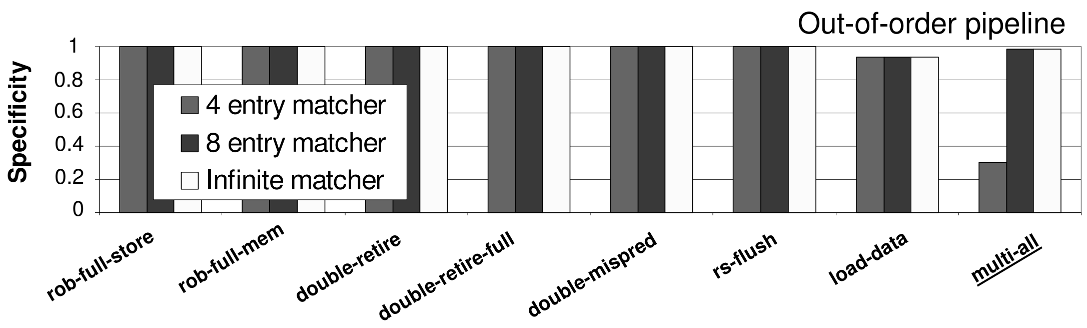


Figure 4.17: Specificity of detection for a range of bugs in the out-of-order pipeline. Low specificity can be caused by insufficient number of critical control state bits monitored through the matcher (for instance *load-data*) or by insufficient size of the matcher (for instance the 4-entry matcher with *multi-1* bug).

mathematical programs, including integer matrix multiplication and emulation of the floating point computation. In addition, we ran both of the designs for 100,000 cycles with the StressTest stimulus to verify correctness of operation as well as provide a more diverse stream of instruction combinations.

4.5.2 Design Defects

To evaluate the performance of the field-repairable control logic solution, we equipped the designs with a matcher block, manually inserted a variety of bugs into our designs, downloaded the appropriate patch to the matcher, and then examined their overall performance. For each bug or set of bugs, we created a variant of the design which included them. In crafting the bugs, we emulated the bugs reported in commercial microprocessor errata and strived to target all levels of the design hierarchy. Usually, high-level bugs were the result of bad interactions between instructions in flight. For example. *opA-forward-wb* breaks forwarding from WB stage on one operand, and *2-branch-ops* prevents two consecutive branching operations from being processed properly under rare circumstances. Medium-level bugs introduced incorrect handling of instruction computations, such as *store-mem-op*, which causes store operations to fail. Low-level bugs were highly-specific scenarios in which an instruction would fail. For example, *r31-forward* is a bug causing forwarding on register 31 to be performed incorrectly. Finally, the multi-bugs are combined bugs, where the state matcher is required to recognize larger collections of bug configurations. For instance, multi-all is a design variant including all bugs that we introduced. A summary of the bugs introduced in both of the designs is given in the Table 4.5.2. It can be noted that even for these simple designs, some of the bugs require a very unique combination of events to occur in order to become visible.

Table 4.3: Bugs introduced in in-order and out-of-order pipelines.

Bug	Description
In-order pipeline	
2-mem-ops	Two consecutive memory operations fail
opA-forward-wb	Incorrect forwarding from WB stage on operand A
opA-forward-conf	Incorrect hazard resolution on operand A
2-branch-ops	Two consecutive taken branches fail
store-mem-op	Store followed by another memory operation fails
load-branch	A conditional branch depending on a preceding load fails
mult-branch	A branch following a multiply instruction fails
mult-depend	Multiply followed by a dependent instruction fails
r31-forward	Forwarding on register 31 is done incorrectly
multi-1	*2-mem-ops* + *opA-forward-wb* + *opA-forward-conf* + *2-branch-ops*
multi-2	*store-mem-op* + *load-branch* + *mult-branch*
multi-3	*mult-depend* + *r31-forward*
multi-4	*2-branch-ops* + *mult-branch* + *load-branch*
Out-of-order pipeline	
rob-full-store	Store operation fails when ROB is full
rob-full-mem	Any memory operation fails when ROB is full
double-retire	Double-issue and double-retirement in the same cycle fails
double-retire-full	Retirement of two instruction fails if two non-branch instructions are added to full ROB at the same time
double-mispred	ROB incorrectly flushes the pipeline if two branches are mispredicted at the same time
rs-flush	Reservation stations do not get flushed on a branch mispredict if rs_full signal is asserted
load-data	Loaded data is not forwarded to dependent instructions in the reservation stations
multi-all	All out-of-order bugs combined

4.5.3 Specificity of the Matcher

The matcher has the task of identifying when the processor has entered a buggy control state, at which point the processor is switched into a degraded mode that offers reliable execution. In this section we study the specificity of the state matcher, that is, its accuracy in entering the degraded mode only when an erroneous configuration is observed.

Figures 4.16 and 4.17 graph the specificity of the state matcher for bugs in the in-order and out-of-order processor designs. Recall that the specificity of a bug is the fraction of recoveries that are due to an actual bug. Thus, if the specificity is 1 the state matcher only recovers the machine when the bug is encountered. On the other hand, a matcher with low specificity would overshoot in its analysis and enter the degraded mode more often than necessary. For instance, a specificity of 0.40 indicates that an actual bug was corrected only during 40% of the transitions to degraded mode, while the other 60% were unnecessary. In order to gather a sense of the correlation between specificity and matcher size, we plot our results considering a 4-entry, 8-entry and a infinite-entry matcher.

It can be noted that for both processors, many of the bugs can be detected and recovered with a specificity of 1.0, even when using the smallest matcher, thus no spurious recoveries were initiated. Some combinations of multiple bugs (*e.g.*, *multi-1* and *multi-2*) had low specificities, but when the matcher size was increased, the specificity reached again 1.0. For these combinations of bugs, a four entry matcher was too small to accurately describe the state space associated with the bugs, but the larger matcher overcame this problem.

Finally, for a few of the bugs, *e.g.*, *mult-depend* in Figure 4.16 and *load-data* in Figure 4.17, even an infinite-size state matcher could not reach the perfect specificity. For these particular bugs, the lack of specificity was not the result of pressure on the matcher, but rather insufficient access to critical control information, as was described in Section 4.3.3. Thus, these experiments had to initiate recovery whenever there was a potential error, leading to the lower specificities.

To evaluate the impact of various critical control signal selection policies and compare them to the automatic approach described in 4.3.4, we developed a range of FRCL implementations over the in-order pipeline using different set of critical signals. The results of this analysis are shown in Figure 4.18.

In the first configuration developed, *single-instr*, the critical control consists exclusively of the 32-bit instruction being fetched. The second solution, called *double-instr* monitors the instructions in the Fetch and Decode stages (64 instruction bits and 2 valid bits). The third configuration (*auto-select*) includes all of the signals selected automatically by our heuristic algorithm from Section 4.3.4 for a total of 52 bits. For this set up the automatic selection algorithm was configured to return all RTL signals with non-zero control rank and width less than 16 bits. The *manual-select* implementation corresponds exactly to the one from the experiment in 4.5.2, including all the signals of Table 4.5.1, thus its matcher performance is the same as in the experiments above. The final configuration, *manual-select w/ID* is as *manual-select*, but it includes 10 extra signals to monitor the destination registers in stages MEM and WB.

Matcher sizes for all of the variants contained enough entries to accommodate even the largest patches, so compression was never required. For each design variant, we developed individual patches for the first 9 bugs listed in Table 4.5.2 (all but the *multi-bugs*). For each bug and each design variant, we measured the *average specificity per signal*, that is specificity divided the number of signals in the critical control pool. This measure gives us an intuition on how to select the approach with the best performance/area tradeoff.

As shown in Figure 4.18, the *manual-select* variant produces the best results for most

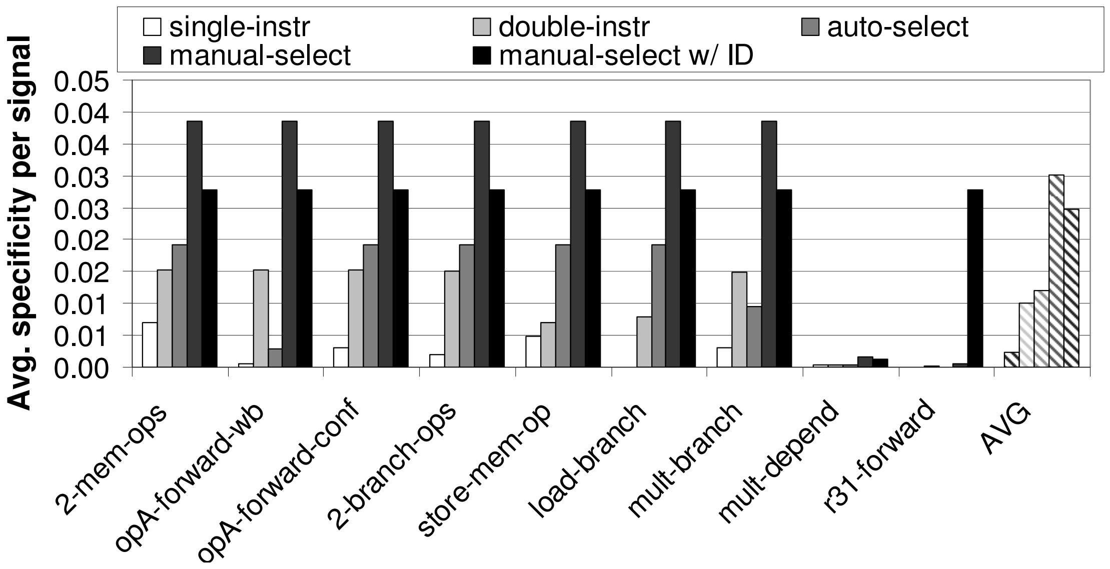


Figure 4.18: Average specificity per signal for a range of critical signal sets in the FRCL implementation of the in-order pipeline. In most cases the *manual-select* solution achieves best specificity at lower cost. However, *auto-select*, based on the automatic selection algorithm of Section 4.3.3 achieves good results with no effort from the designer.

bugs. The *manual-select /w ID* solution has better specificity than *manual-select*, but at a higher price. Its main advantage is the good result over *r31-forward*, which is made possible by its tracking destination register indices. Note also that the automatic selection algorithm performs quite well, especially taking into account that this approach does not require any engineering effort.

4.5.4 State Matcher Area and Timing Overheads

Implementing a field-repairable control logic solution requires the addition of the critical control matcher logic, that is, the matcher itself and the recovery controller, which cause an area overhead for the final design. Table 4.5.4 tabulates the area overheads of a range of FRCL implementations, including matcher size of 4 and 8 entries built over both the in-order and out-of-order designs and considering 256B and 64kB instruction and data caches. As shown in the table, the overhead of FRCL is uniformly low. Even the larger state matcher with small pipelines and caches (in-order-256B) results in an overhead of only about 2%. Designs with larger caches and more complex pipelines have even lower overhead. Given the simplicity of our baseline designs, we would expect the overhead for commercial-grade designs to be even lower. Table 4.5.4 also presents the propagation delays through the matcher block. Note that all solutions have propagation delays that are well below the clock speed, hence they do not affect the overall system's performance. Note that the matcher for the out-of-order processor performs faster because

Table 4.4: Area overheads and propagation delays for a range of FRCL implementations on the in-order and out-of-order pipeline when synthesized on 180nm technology.

	Critical control state matcher area (% design area)			
	In-order		**Out-of-order**	
	256B	64kB	256B	64kB
4 entry matcher	1.10%	0.01%	0.34%	0.01%
8 entry matcher	2.20%	0.02%	0.68%	0.02%

	Propagation delay of the matcher (ns)	
	In-order (clk=11.5ns)	**Out-of-order** (clk=6.5ns)
4 entry matcher	1.18ns	1.17ns
8 entry matcher	1.43ns	1.21ns

it monitors fewer control signals. It should be also pointed out that FRCL matching is performed in parallel with normal pipeline operation, and given the observed propagation delays through the matcher, they do not affect the overall design frequency.

4.5.5 Performance Impact of Degraded Mode

During recover the processor is switched into degraded mode to execute the next instruction, and then returned to normal operation. During recovery, only one instruction is permitted to enter the pipeline, thus instruction-level parallelism is lost and program performance will suffer accordingly. Figure 4.19 graphs the performance of the in-order and out-of-order processors as a function of increasing recovery frequency. As shown in the graph, for performance impact to be contained under 5%, the rate of recovery could not exceed 6 per 1000 cycles for the in-order pipeline and 1 per 1000 cycles for the out-of-order pipeline. For a more stringent margin of 2% impact, recovery rates should not exceed 2/1000 and 4/1000 for the in-order and the out-of-order processors, respectively. In addition, the in-order pipeline is more quickly affected by recovery than the out-of-order pipeline, due to the fact that the out-of-order pipeline is able to better tolerate the loss of parallelism due to its more capable instruction scheduler.

Finally, Figures 4.20 and 4.21 show the CPI (clock cycles per instruction) of the FRCL-equipped in-order pipeline. The CPI has been normalized to the average CPI achieved when no patch was uploaded on the matcher (hence degraded mode was never triggered). By comparison with Figure 4.16, it can be noted that low specificity often results in increased CPI. However, the worst case scenario (4 entry matcher and *multi-1* bug) occurs

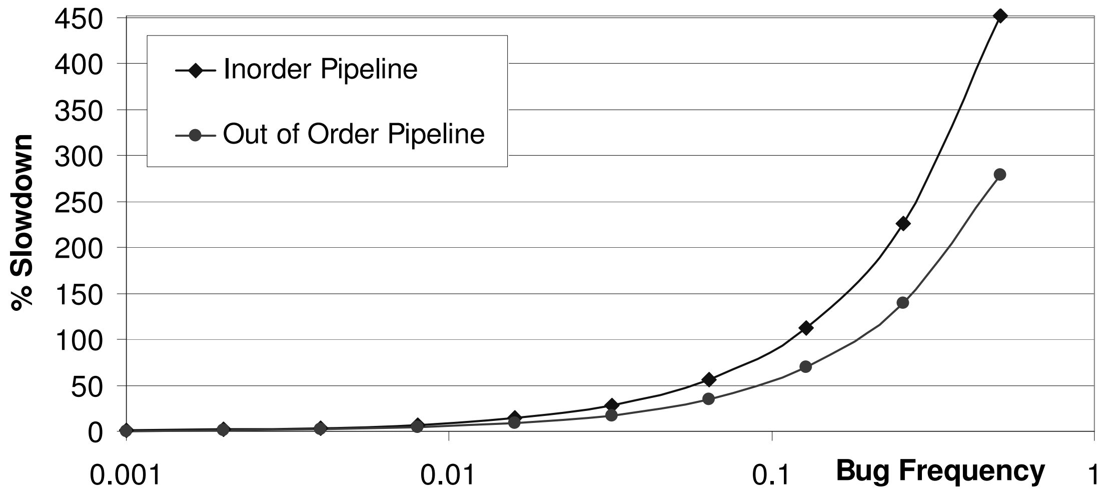


Figure 4.19: Impact of recovery on processor performance. Field-repairable control logic technology incurs less than 5% performance impact as long as the frequency of the bug does not exceed 6 per 1000 cycles in the in-order pipeline and 10 per 1000 cycles in the out-of-order pipeline.

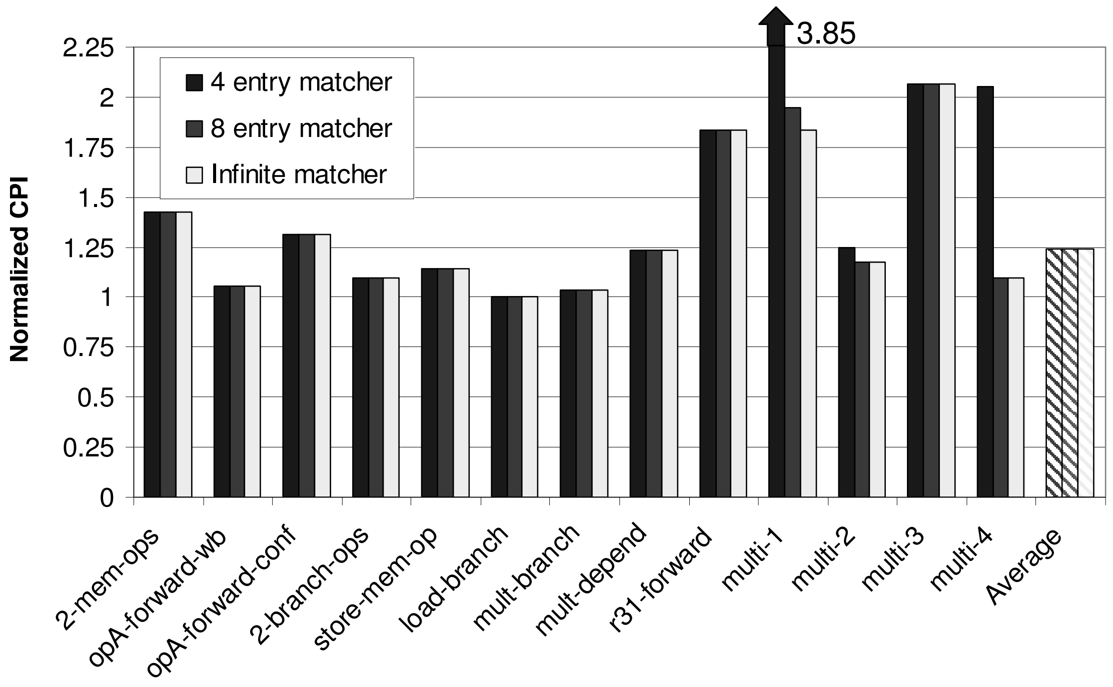


Figure 4.20: Normalized CPI for the in-order pipeline. Average CPI increase is computed only over design variants with a single bug.

because of an insufficiently sized matcher and not because of the critical control selection.

4.5.6 Semantic Guardian Framework Analysis

For our experiments with semantic guardian technology we used the same setup as in the previous FRCL study, including the the two processor cores, critical signal sets for each of the pipelines and degraded mode verification with Synopsys Magellan for all instruction

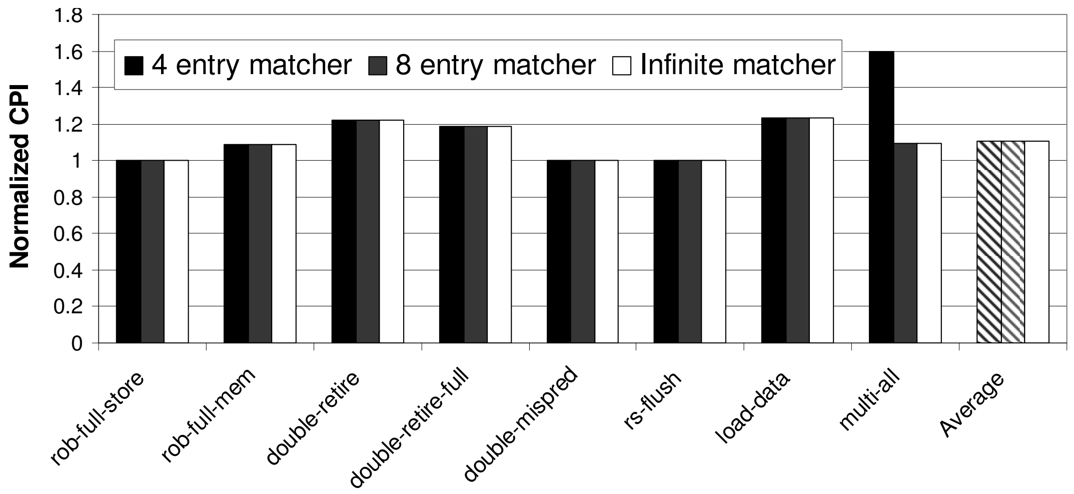


Figure 4.21: Normalized CPI for the out-of-order pipeline. Average CPI increase is computed only over design variants with a single bug.

types. For semantic guardian's synthesis and optimization we used a combinations of several techniques, including different configuration of Espresso [15] computing the ON- or OFF-set of the combinational function. We also developed a proprietary heuristic which progressively collapses pairs of states with Hamming distance of 1. Also, if the un-trusted set encompassed more than 50% of the total state space, the trusted set was used instead to generate the guardian. Finally, we developed a script which considers the output of Espresso or of our synthesis heuristic and produces an register-transfer level description of the guardian circuit, which is then synthesized with Synopsys' Design Compiler and mapped to TSMC 0.18 and 0.13 μm libraries.

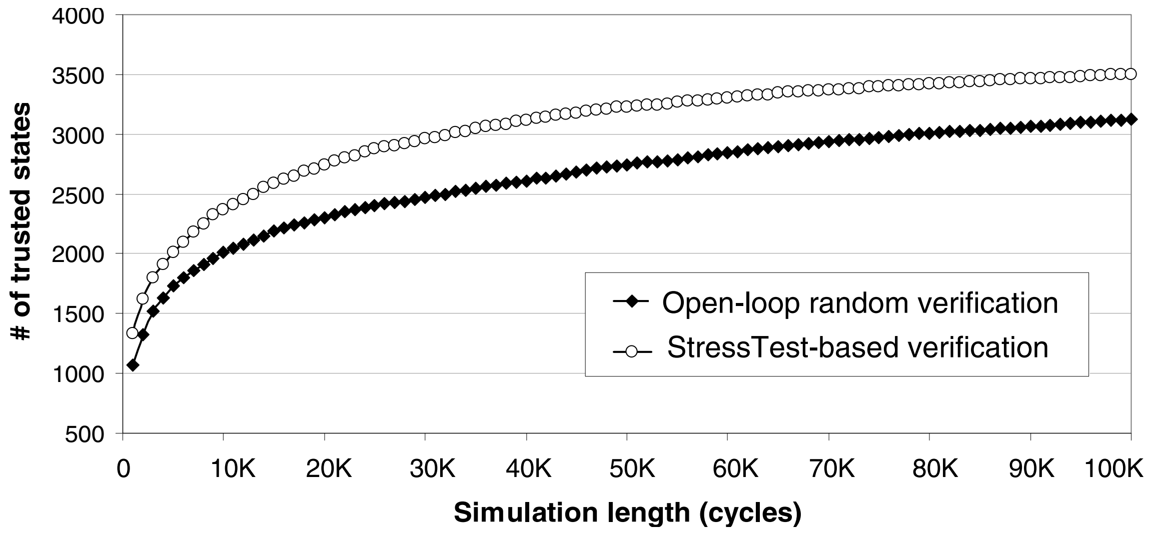


Figure 4.22: Trusted states vs. simulation effort. With increasing number of simulation cycles more states can be checked and labeled as trusted. Moreover, a more complex closed-loop verification technique performs better then simple constrained random approach.

In our first experiment we calculated the number of distinct critical signal values combinations (trusted states) observed during the validation of the high-performance mod. To produce stimuli we used both an open-loop constrained random generator and StressTest closed-loop generator. In our framework, a state was considered trusted if it is observed at least once during the simulation, since the signals we selected thoroughly describe the behavior of the processor control FSM. The results of this experiment are shown in Figure 4.22. It can be observed that, with increasing simulation length, *i.e.* increasing validation effort, the number of states labeled as trusted increases, leveling off at the far right of the graph. Note that we could not perform a formal verification of this design and, therefore, cannot estimate the fraction of the overall reachable state-space covered during simulation.

In the second experiment we analyzed various compression techniques for best area-delay parameters of the semantic guardian matching logic. The results of this study are presented in Table 4.5. The columns list the compression technique, number of set bits and don't care bits in the in the Boolean function for the guardian and area in mm^2 and propagation delay through the semantic guardian in ns. Espresso+ and Espresso- indicate the circuit was obtained optimizing the ON-set or the OFF-set with Espresso. Compaction is our heuristic optimization technique described above, and the last are combination solutions. The top half of the table shows results with no additional optimizations by Design Compiler, while the bottom half shows the best circuit that we were able to synthesize with the most narrow timing and area constraints.

In general, we found that with the more stringent area and timing constraints, Design Compiler gives higher priority to delay optimization and produces a significantly faster guardian. We also noted that Espresso generating the ON set (Espresso+) or Espresso in combination with our compaction approach (Comp & Espr+) generated the circuits with the smallest delay, at a tolerable area penalty. For comparison, the area of the in-order core design, excluding caches, in 0.18 μm technology was 0.5 mm^2. Thus, with Design Compiler optimization the best guardian can incur as little as 3.5% area overhead in the processor design.

In addition, we investigated the possibility of generating a smaller and faster circuit by re-labeling some of the trusted states as un-trusted. Note that in this experiment, the semantic guardian was generated from the trusted set, therefore, when trusted, but rare, states are removed from the set, the guardian becomes smaller. This effect is amplified due to better compressibility of the set with rare states removed. In other words, Espresso and our compaction heuristic were capable to achieve a more effective simplification without these states. In addition, it is possible that when these states were removed, more DC combinations became available, further simplifying the guardian design.

Table 4.5: Area and delay of the semantic guardian generated with different design optimizations.

Optimization method	#ON-set	#DC-set	TSMC $0.18\mu m$ area mm^2	TSMC $0.18\mu m$ delay ns	TSMC $0.13\mu m$ area mm^2	TSMC $0.13\mu m$ delay ns
Without Design Compiler optimization						
No optimization	52104	0	0.0186	4.07	0.0099	3.78
Espresso+	2455	11091	0.0194	2.79	0.0101	**2.42**
Espresso-	10586	1556	0.026	4.13	0.0144	3.24
Compaction	18137	843	**0.0121**	**2.76**	**0.0062**	2.47
Comp & Espr+	2449	11097	0.0196	2.78	0.0101	2.46
Comp & Espr-	10586	1556	0.0252	4.44	0.0143	3.23
With Design Compiler optimization						
No optimization	52104	0	0.0268	1.55	0.0168	1.28
Espresso+	2455	11091	0.0212	1.14	0.0171	**0.93**
Espresso-	10586	1556	0.0324	1.59	0.0264	1.27
Compaction	18137	843	**0.0173**	1.19	**0.0143**	1.01
Comp & Espr+	2449	11097	0.0246	**1.08**	0.0160	0.96
Comp & Espr-	10586	1556	0.0316	1.56	0.0269	1.3

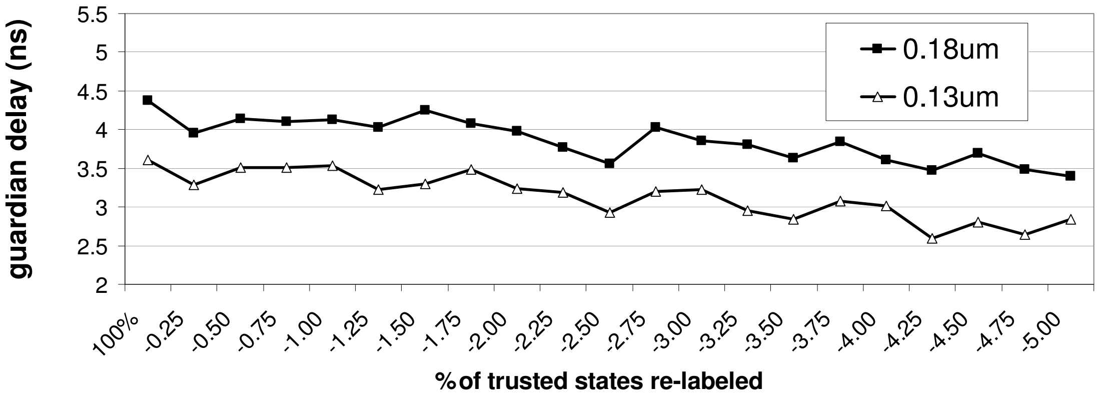


Figure 4.23: Impact of trusted state re-labeling on the guardian circuit delay.

For this experiment, each state observed in validation is labeled with its observation frequency, that is, the number of times the state has been seen, divided by the total number of simulation cycles. The baseline matcher included all trusted states and hence its cumulative observation frequency was equal to 100%. Then we grouped the states in clusters whose cumulative observation frequency was 0.25%, starting from the lowest frequency states. Each of these clusters was then progressively removed, one at a time, from the pool used to generate the semantic guardian, and we created a new guardian each time without timing or area constraints. The results of this experiment are presented in Figures 4.23 and 4.24, where the X-axis shows the trusted-set size reduction in percent of observation frequency. It can be observed from the diagrams that there is a definite reduction

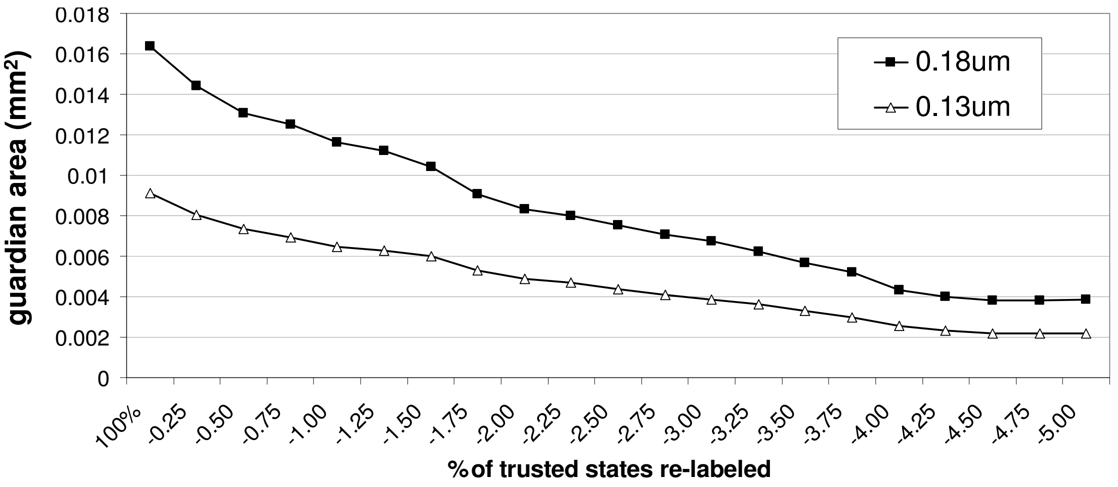


Figure 4.24: Impact of trusted state re-labeling on the guardian circuit delay.

in area for smaller trusted sets, enabling up to a 4x area reduction for just a 5% penalty in observation frequency. This trend can also be observed in the matching delay. The trust re-labeling algorithm allows for the designer to trade off the precision of the matcher to achieve improved area and delay for the guardian circuit.

4.6 Summary

This chapter of the thesis outlined a technology called field-repairable control logic (FRCL), which is a novel microprocessor design solution to detect erroneous control configurations and recover correct execution through a low-complexity, reliable *degraded-mode*. We described a low-cost state matching mechanism that can detect when to bypass bugs. The technique consistently has an area cost of less than 2%. Moreover, with moderately sized matchers, we can ensure highly accurate detection of bug states, with nearly all of our experiments. We also showed how the framework can be extended to include semantic guardians - combinational circuits forcing the processor into degraded mode of operation whenever a control state un-verified at design time is encountered. The guardians can be deployed separately or they can augment the field-repairable control logic to prevent security breaches in systems with known, but currently un-patched bugs or even protect processors from unknown escapes. Finally, we examined the performance impacts of running programs in degraded mode, and we found that, if recovery frequency is less than ten per 1000 instructions in the out-of-order design, and less than 6 recoveries per 1000 instructions in the in-order design, the performance impact is below 5%. Additionally, the analysis of the semantic guardian demonstrates that these circuits can be subjected to a variety of area/delay optimizations that improve their performance and minimize the silicon footprint. In particular, our experimental evaluation showed that guardians can be optimized to occupy as little as 3.5% of the die and can be further optimized using the re-labeling heuristic that trades off guardian performance for silicon area. We also showed that the guardian circuit does not have any impact on the performance of the processor, while it is running in a verified state and, as with FRCL, the impact of invoking the degraded mode is proportional to the frequency of occurrence of untrusted states. We feel that this work makes a strong case for field-repairable control logic and semantic guardians and shows that the approach holds a great promise to insure chip designers and manufacturers against the potential disasters of releasing buggy silicon.

CHAPTER V

Pre-silicon Verification of Multi-Cores

For three decades the challenge of pre-silicon verification of processors has grown exponentially with each new generation of systems. In recent years, however, this daunting task jumped to a new plane of complexity due to the introduction of multi-core chips. Validation of memory coherence of such systems, which include multiple levels of cache and complex protocols, constitutes a major fraction of the verification process. Unfortunately, current tools are incapable of addressing these challenges, allowing bugs, which cause unpredictable software behavior and wrong computation results, to slip into hardware. We feel that a thesis in processor verification is not complete unless it addresses the new challenges that multi-core designs have raised. For this reason, in this and the next two chapters, we present solutions in pre-silicon, post-silicon and runtime that focus on the complex memory sub-systems of multi-core processors. By deploying the solutions discussed in the previous chapter on the individual processor cores, and then applying our techniques for memory sub-system verification to verify the memory protocols, a design house can achieve better coverage across the entire system under development and hopefully deliver better quality products at lower cost.

In this first chapter focusing on the challenges of multi-core processor validation, we present MCjammer - an adaptive verification tool for Multi-Core designs that uses closed-loop feedback to dynamically adjust simulation to effectively test corner cases of design behavior. MCjammer is a novel scalable approach to stimulus generation and coverage analysis, which is specifically designed for the verification of the shared memory subsystem, namely cache controllers, memory controllers and interconnect. MCjammer relies upon multiple cooperating agents, each of them containing a simplified model of the system that is linear in the number of processors/cores in it. Agents create and correlate with each other sequences of memory accesses, attempting to maximize coverage of transitions in their respective simplified system models. The use of a distributed simplified model allows MCjammer to be more scalable than techniques that are based on the full description of the system coherency protocol.

In addition, coverage and frequency of conflicting memory requests are analyzed by the agents, so that they can track progress on their goals, produce test sequences with large amount of "stress" on the system, and try to expose errors. Finally, the data that agents supply to the design under test is uniquely tagged and can be used to detect a variety of bugs, including violations of memory coherence, or even faults in the interconnect. Both simplicity of the system and data tagging enables MCjammer to easily scale and be adapted to large multi-processor designs. In addition, these features make our solution applicable to a variety of coherence protocols and different design representations, including RTL or high-level C models.

5.1 Background

In a shared-memory multi-core or multi-processor system several processors communicate via an interconnect structure (bus, network, *etc.*) to the main memory or with each other, as shown in Figure 5.1. Unfortunately, the latency of a memory access in such a system can be significantly higher than in a single-processor machine, since memory is physically located much further away. A processor's request often must go through a network interface and make multiple hops to reach the memory controller and then return back with data. Therefore, caches, which reside within each core/processor and amortize the access time, become vital for performance. This also complicates the interaction between processors since some of them might have more recent data in their caches than what main memory has.

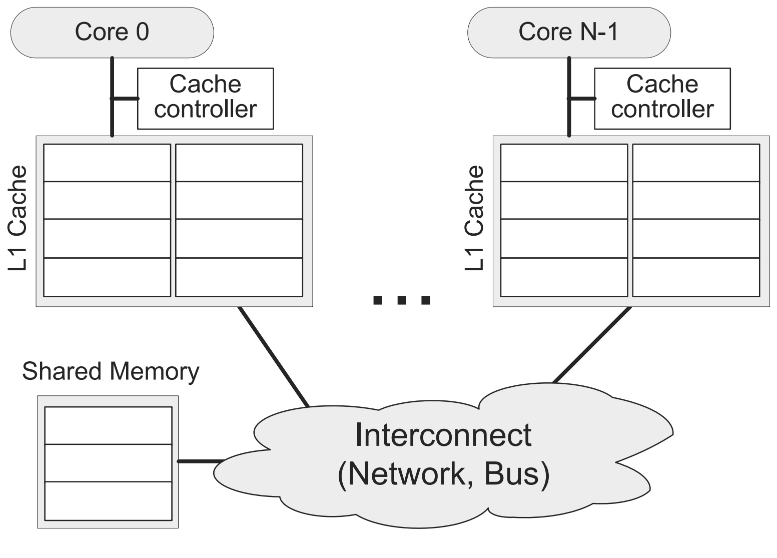


Figure 5.1: Structure of a multi-core/multi-processor system. Multiple cores/processors (core 0 through core N-1) have separate caches, but communicate with each other and the shared main memory via interconnect.

To make sure that all processors have a coherent view of each memory location, and all data changes are propagated through the entire system with the best possible performance, a variety of *cache coherence* protocols have been proposed. Figure 5.2 presents a model of the *MESI* invalidation coherence protocol (described in [23]), where (from the point of view of each cache controller) a particular memory location can be in one of four states: 'Modified', 'Exclusive', 'Shared' or 'Invalid'. For example, if processor P2 has a memory location 0x1000 in its cache in state 'S', the value in P2's cache is the same as in the main memory and the same memory location is potentially in the shared state in other caches.

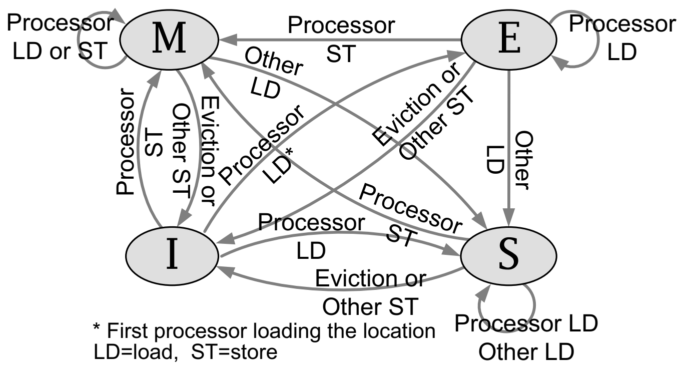


Figure 5.2: MESI cache coherence protocol. Each memory location can be in one of the following states in each cache controller: 'Modified', 'Exclusive', 'Shared' or 'Invalid'. 'I' when the location is not available in the cache, 'E' when only the corresponding processor can modify the data, and 'M' after the value has been updated in the local cache. The cache line is in state 'S' when the data is in the cache, but it may also be present in caches of other processors in the system.

Note that the finite state machine of the protocol shown in Figure 5.2 only reflects the view of a single processor on the state of the memory location. Therefore, the logic for the full system protocol for a single memory location is a *product* of n finite state machines (FSM), where n is the number of processor nodes in the system. A product FSM for a MESI-based system with three nodes is shown in Figure 5.3. For example, a scenario where processors P1 and P2 have a particular memory location in state 'S', while P3 does not have it in the cache ('Invalid' state), corresponds to state 'SSI'.

Verification of the memory coherence in a multi-core/multi-processor system includes verification of this type of system-level FSMs, which encode all possible interactions of the nodes with respect to one memory location. The main aspects to verify in this case are absence of invalid transitions and invalid states (for example, states where several processors have the same memory location marked 'Modified' simultaneously).

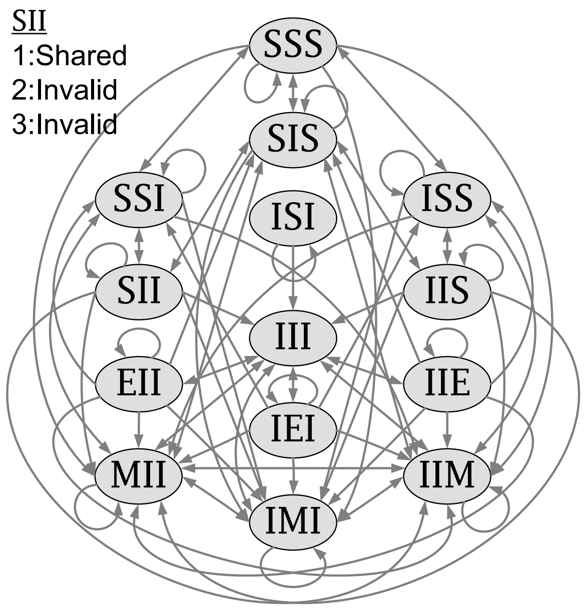


Figure 5.3: Finite state machine for the full system cache coherence protocol for a three processor MESI-based system. Each processor follows the MESI protocol shown in Figure 5.2.

5.2 MCjammer Tool

MCjammer is a novel simulation-based verification tool designed specifically to target multi-core and multi-processor systems. MCjammer requires only high-level knowledge of the coherence protocol, and it is easily portable to multiple representations of the same multi-core design: C model, protocol-based RTL or full RTL implementation. The tool employs multiple agents that generate concurrent and colliding memory access patterns, trying to expose incoherent cache states or errors in data or address manipulation. At the same time the agents support each other in achieving individual coverage goals, thus attempting to maximize coverage of the full protocol state machine. This section gives an overview of the tool.

5.2.1 Overall Structure

MCjammer instantiates a collection of cooperating adaptive agents, generating test sequences for each of the processors of the design (see Figure 5.4). To allow for scalability and efficiency of this technique, each agent is designed to have a novel simplified view of the system under verification, called *dichotomic finite-state machine (DFSM)*. DFSM allows an agent to distinguish only between its own actions and the actions of the “environment”, *i.e.,* all other agents. The DFSM in each agent is used for an internal representation of coverage. Agents keep track of DFSM transitions traversed in the past and use this information to direct testing towards unexplored scenarios of execution. At the beginning of each simulation run, each agent selects an insufficiently verified transaction in its DFSM, and generates load/store instructions to cover it. MCjammer also allows collaboration between agents, allowing them to generate stimuli in a coordinated fashion to quickly achieve common coverage goals. At the end of the run, the agent checks the coverage report of the simulation and adjusts its actions so that *i*) it increases the likelihood of observing the desired transition in its DFSM, and *ii*) it increases the pressure on the memory system to maximize the number of collisions and expose possible design errors.

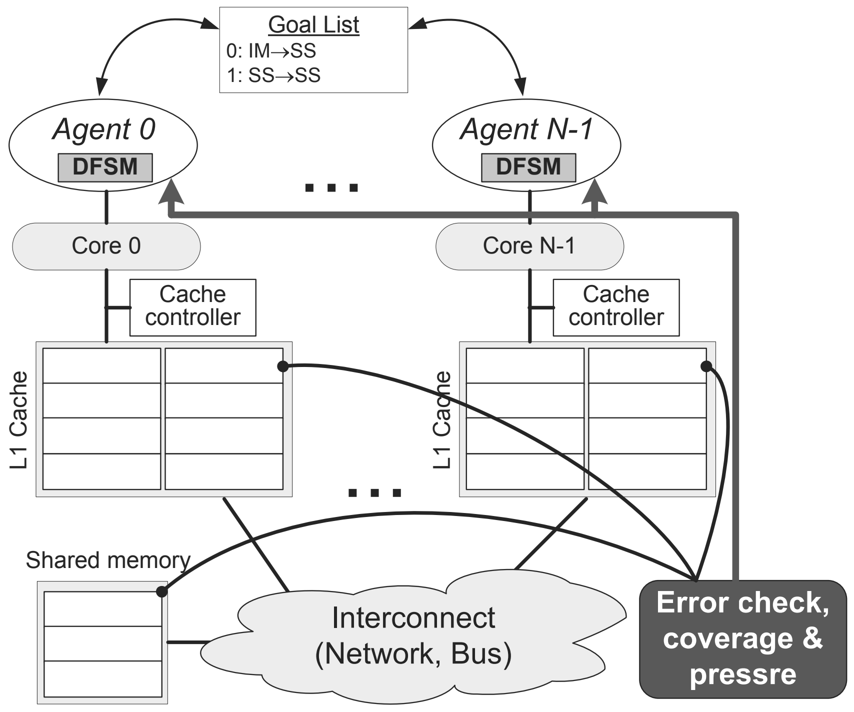


Figure 5.4: Structure of MCjammer. Each core in the system is assigned an agent. Agents formulate their goals in terms of transitions in the *dichotomic finite-state machine (DFSM)* of the memory coherence protocol specified by the user. During each run, agents choose if they want to attempt to achieve their own goals, provide support to another agent, or execute a random transition.

5.2.2 Dichotomic Finite-State Machine

As was pointed out in the introduction of this work, the number of processors and individual cores used in today's chips is increasing rapidly every new generation. Therefore, the number of possible states and state transitions in the global coherence protocol increases exponentially. As a result, tools that evaluate coverage on the global system level become unscalable. In designing MCjammer, we decided to divide the large problem of validating all possible transitions in the full system's coherence protocol (*full system FSM*) into a set of smaller problems, each with a simplified FSM. Instead of one agent formulating test sequences for the multi-processor system based on coverage or collision metrics, MCjammer consists of a set of simpler agents cooperating with each other. However, for simplicity reasons, agents don't have an understanding of the full system FSM and instead use a *dichotomic finite-state machine (DFSM)* to represent their perspective of the protocol and coverage. States in a dichotomic finite-state machine are only comprised of the states for the local node and the state of the "environment", *i.e.,* all the other nodes.

An example of a DFSM for MESI protocol is shown in Figure 5.5. A state in this figure represents the protocol for a single memory location at the agent's cache (first letter) and at some other agent's cache (second letter). For instance, in the state 'SI' the agent has the cache line marked as 'Shared', while some other agent has it as 'Invalid'. Transitions between states correspond to actions of the agent itself (subscript s - *self*) or other agents (subscript o - *other*) and include such actions as load (LD), store (ST) or cache eviction (E). A transition between states in DFSM represents a change in the state of the system, for example, transition from 'II' to 'IM' to 'SS' corresponds to a scenario where first some other node wrote to the location and then the agent itself loaded the location in the cache. Moreover, if in a four-node system a memory location transitioned from state 'SSSS' (all nodes have location as 'Shared') to 'SSSI', then in the first agent's DFSM both the edge from 'SS' to 'SI' and a loop from 'SS' to 'SS' are marked as visited. This is because in state 'SSSI' relative to the first agent there is another node that has the location as 'Shared' (either 2 or 3) and there is another node that has the location as 'Invalid' (node 4). Note that in a dual-processor/dual-core system, the DFSM corresponds exactly to the full system protocol, since there are only two nodes present. However, if the number of processors increases, the full system protocol FSM grows (recall Figure 5.3), while individual agents retain the same DFSM structure.

For more precise feedback and better cooperation between the agents, each DFSM is augmented with additional information: each edge in the Dichotomic Finite State Machine is associated with a *coverage vector* of n entries, where n is the number of cores in the system. An ith element of the vector is equal to the number of times a given edge was traversed by cooperation of the current agent and agent i, preserving pairwise edge coverage of the full system FSM. The coverage vector is used to bias the probability distribution in the agent's algorithm and make the pairwise coverage of the DFSMs more even. Notice that although this additional information makes the size of the DFSM linear, in terms of the number of nodes in the system, it provides much more precise coverage information, while retaining the simple goal and action formulation algorithm of the DFSM.

The division of the complex protocol state machine into simple DFSMs allows MCjammer to retain the simplicity of individual agents and their interactions regardless of the

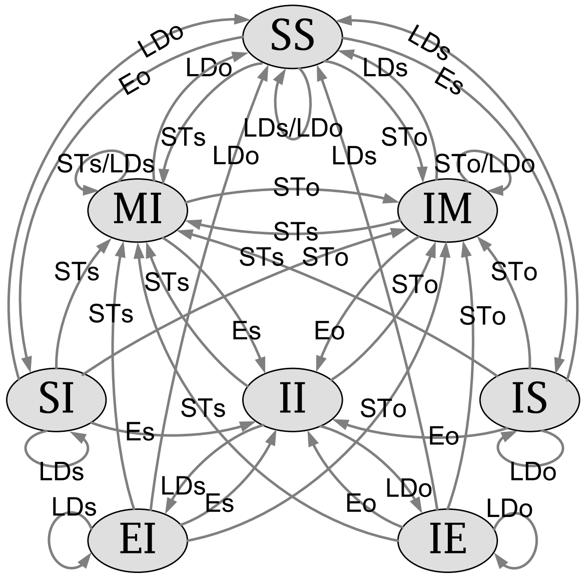


Figure 5.5: Dichotomic Finite State Machine for MESI protocol. DFSM represents a simplified view of the global states that each memory location in the system might have. An agent using this DFSM only distinguishes between actions of its own and actions of the "environment" or the rest of the agents. Transitions in the DFSM are labeled with the corresponding action that the agent (subscript s) or one of other agents (subscript o) must take. Actions include load (LD), store (ST), and eviction (E).

number of agents in the system. On the other hand, if a single agent had precise knowledge of the entire system, *i.e.,* the full system FSM, the complexity of its decision and communication processes would not be scalable beyond just a few cores. For example, a MESI system containing just four processors would have 211 edges. If each agent had a full view of the system, the collaboration between agents and the decision making algorithm would be extremely complicated. In the same case, MCjammer, on the other hand, has partial but overlapping DFSMs, each with only 37 edges, leading to a total of 148 edges. By dividing the problem into a set of smaller problems, MCjammer retains a manageable complexity, while, hopefully, maintaining high full system FSM coverage.

5.2.3 Agents' Goals

Each agent selects its verification goals by indicating which transition it would like to cover in the node's DFSM. For example, in Figure 5.4, agent 0 had chosen transition 'IM'→'SS' as its goal. Since transitions in the DFSM are labeled with actions that the agent and/or other agents need to perform for the transition to occur, generating actions to test a particular goal is straightforward. Note, however, that covering all the transitions of all DFSMs is not equivalent to covering all the transitions in the full system FSM. To enhance the coverage in the full FSM (which is impossible to represent with reasonable resources), MCjammer observes each transition within each DFSM several times, partially compensating the effect of DFSM's abstraction of the entire system.

Since each agent relies on a simplified view dictated by the DFSM, the algorithm governing its activity is also fairly straightforward. Prior to each simulation run, individual MCjammer agents formulate their goals by choosing insufficiently verified transitions in their DFSM's. Then all agents exchange their goals and make a probabilistic decision to either pursue their own goal, or generate a stream of instructions to allow another agent to achieve its goal, or execute a random stream of memory accesses. In the agent algorithm the probability of a random stream is constant, while probability of helping another agent is inversely proportional to the number of own goals that an agent had reached. In other words, agents that reach their own goals early are more likely to help other agents that have fewer goals covered. Actions of individual agents are then translated into streams of loads and stores timed in such a way that those of two or more collaborating agents have little overlap with other memory accesses. This is done by partitioning the set of agents into collaboration groups and allocating a time-frame for each group, so that interference between groups is reduced. After MCjammer sets up this list of timed loads and stores, the simulation starts with this distributed "program".

If, during a particular simulation run, an agent chooses to work on its own goal, there is no guarantee that another agent will help, or that a transition of interest will occur. However, if an agent does not observe the desired transition, it will attempt to change the timing between its load/store instruction and request any of the helping agents to do the same. The process of delay reduction is based on the pressure metric discussed below. If the coverage report indicates that a transition of interest occurred sufficiently often, the agent chooses another action based on the list of unreached goals and signals all helping agents that the goal has been accomplished. After a preset number of simulation runs, all agents are required to change their goals and repeat the procedure. In addition to coverage and pressure analysis, simulation results are analyzed for errors that may be detected with the data tagging technique discussed in Section 5.3.2.

In designing MCjammer, we decided to avoid strict partnership, where agents deterministically choose partners to test various transitions. This simplifies the complexity of the agent algorithm. and allows the possibility to generate unforseen interactions, which strengthen the verification of the coherence protocol. Moreover, the implementation of MCjammer is simulator-independent and can be easily ported between simulators of different level (architecture, RTL, and even gate-level). Moreover, by partitioning and abstracting the full protocol's FSM into a collection of DFSMs, the agent's algorithm is made independent from any specific protocol, which enables MCjammer to be easily portable to other cache coherence protocols. In fact, the portability to other verification environments is executed by simply modifying the DFSM description, with no other modification. To test this aspect in the experimental setup, MCjammer was seamlessly deployed in a range of systems based on both MESI and MOSI protocols.

5.3 Feedback and Correctness

This section introduces the coverage model and coverage-directed feedback in MCjammer. We also discuss pressure, which is a metric of "stressfulness" of a simulation run. Finally, the section demonstrates how MCjammer uses pressure as additional feedback parameter to increase the quality of generated tests.

5.3.1 Coverage and Pressure

Coverage is the measure of thoroughness of the verification process and full coverage is an assurance that all monitored design behaviors have been tested. Often, coverage is only reported by verification tools to the engineer, who then designs the next test based on unverified regions of operation of the design. Unfortunately, this human intervention becomes a very high-latency and high-cost part of the verification process. In MCjammer this process is automated to close the loop with coverage and pressure feedback.

A natural coverage metric for a cache coherence is the coverage of the states and transitions in the full system state machine. However, as was shown in Section 5.1, the number of states in this FSM grows exponentially when the number of nodes increases. Therefore, using protocol state machine coverage to automatically evaluate the test thoroughness is a prohibitively complex task. To overcome this issue in MCjammer, agents evaluate coverage on their individual DFSMs. After a run, each agent identifies which DFSM transitions were explored and records the information. As discussed above, a single traversal of an edge in the DFSM is generally insufficient to gain high coverage of all the corresponding full system. To boost the full system coverage, MCjammer agents are required to cover each DFSM transition several times before marking it as verified, so this transition cannot be chosen as the agent's goal again. The goals of the agents in MCjammer are adjusted dynamically with every coverage report obtained from each simulation run. This fine granularity of feedback allows MCjammer to direct tests towards insufficiently verified areas of design operation and requires no human effort or oversight.

In addition to coverage feedback, MCjammer uses a measure of pressure on the memory system to adjust the generated tests between consecutive simulation runs. In designing MCjammer we strove to create "stressful" activity on the system by having actions from different nodes interfere with each other. This allows MCjammer to exercise a rich set of unexpected interactions between the nodes for improved coverage. Moreover, empirical evidence suggests that complex bugs are often exposed by such situations. With reference to Figure 5.5, an example of such situation arises when one processor attempts to load a previously un-cached location from the memory and is in the process of going from 'Invalid' to 'Exclusive' state, meanwhile another processor tries to store to the same location at the same time. Pressure in MCjammer is computed as a mean time between colliding events at the caches and memory controller and it is used to maximize the "stress" on the system. If the pressure is low, the delay between agents' actions is reduced, hence increasing probability of collision in future simulation runs. As the experiments demonstrate, pressure and coverage feedback help MCjammer to quickly create high-quality tests and achieve thorough coverage of complex multi-processor protocols.

5.3.2 Error Check

To detect bugs in data or address manipulations and check for potential errors with memory consistency MCjammer employ a data tagging technique. The data for each store in the system contains the unique ID of the agent issuing the store, the unique ID of the store operation at that processor, and a subset of the address bits of the store. Given the result of a load the tool can quickly identify which agent issued the last store that wrote to this location. For example, if the first agent is issuing a store with unique ID=2 to address $0x00AB3456$, the data written to this location will be $0x01023456$. The most significant byte in this case carries the ID of the agent, second byte - store ID and the last two bytes carry lower bits of the address. Therefore, if this location is accessed later and the two lower bytes do not have value of 0x3456, it will indicate a problem with data transmission/manipulation. Unfortunately, agents do not have access to the current *state* of the memory location in the cache, since this information is not architectural and not available to the processor core. To check for errors in cache coherence protocols, MCjammer's coverage analysis system tracks the state of each accessed memory location and reports invalid states, for example, a state where one agent sees the memory location transition to 'Modified' state, while another still observes it in 'Shared' state.

Data tagging can be beneficial for diagnosing memory consistency problems. Memory consistency, which defines the order of memory accesses that are legal in a particular machine, is also a crucial aspect of multi-core computing. The issue of consistency arises from the fact that scalable interconnects in multi-processors may re-order request messages, thus different processors may see the global sequence of loads and stores in different orders. For example, in a system implementing Sequential Consistency [45], all processors must see all store operations in the same order. By checking the tags of the data loaded by each processor, MCjammer can quickly establish if this rule was violated. Therefore, the error checker does not need a fully-specified reference model to establish that a violation has occurred, and only the axioms of load/store ordering are required.

5.4 Experimental Results

To analyze the performance of MCjammer we conducted several experiments on two multi-processor protocols using the Multifacet GEMS architectural simulator [53]. In particular, the Ruby simulator was used to model the interconnect, caches and memory and coherence controllers. The Ruby tester program was modified to allow multiple nodes in the system to initiate overlapping memory operations. Two protocols that we used in these experiments were MOSI, (*MOSI_SMP_Bcast _1level* in the Ruby model), and MESI, (*MESI_SMP_LogTM _directory*). Both systems were configured to include only two banks of fully-associative L1 caches. Descriptions of the DFSMs for both MESI and MOSI designs were derived from the protocol FSM specifications. For performance comparison we created a constrained-random (*ConstRand*) generator that did not feature collaborating agents and feedback, but, for fairness, produced the same type of accesses to the same memory locations with timing comparable to MCjammer.

In the experiments, MCjammer is compared to the constrained-random stimulus generator in terms of transition and state coverage of the full protocol FSM for a single memory location of a 16-core system. The results of the experiments are shown in Figures 5.6.a and 5.6.b, for MESI and MOSI protocol, respectively. The x-axis of the graph shows the number of instructions executed by MCjammer or *ConstRand*. The y-axis (log scale) evaluates the number of covered states or transitions of the full system protocol state machine. Note that in both experiments MCjammer achieves significantly higher coverage with lower effort than the random generator.

In the final experiment, we inserted eight bugs of ranging complexity into a sixty-four-processor MOSI system and investigated how quickly MCjammer could find these bugs compared to *ConstRand*. The results of the study are presented in Table 5.1. The first six bugs were inserted into the logic of the directory controller (*dc*) of the system. The last two bugs, with prefix *cc*, were inserted into individual cache controllers. We ran both systems several times with a range of random seeds and measured the average number of instructions each system executed to expose the bugs. For this experiment the *ConstRand* generator was configured to label the data similarly to MCjammer and used the same correctness checker to detect the bugs. As Table 5.1 shows, *ConstRand* was not able to

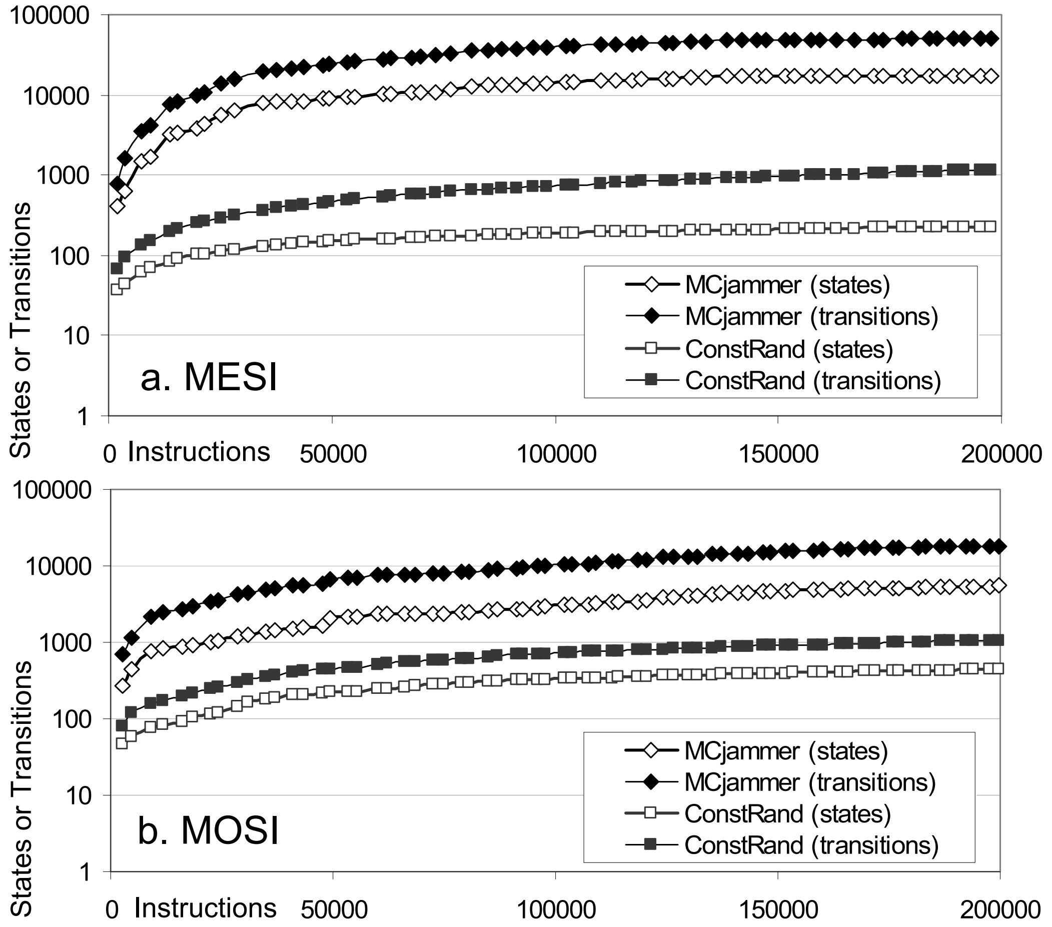


Figure 5.6: Comparison of state and transition coverage vs. effort for MCjammer and constrained-random simulation. MCjammer can achieve higher state and transition coverage on the 16-node full system state machine, with less effort.

Table 5.1: Bug coverage for MCjammer and ConstRand. MCjammer is capable of finding more bugs, and finding bugs in fewer instructions than a constrained-random simulator.

Bug name	MCjammer # instructions	*ConstRand* # instructions
dc_write	1248	6235
dc_two_writes_1	116	3272
dc_two_writes_2	116	240
dc_read_write	363	710
dc_write_read	3134	—
dc_two_reads	1215	10095
cc_data_write	116	180
cc_data_forward	4227	—

find two of the bugs and required significantly more instructions than MCjammer to find the others. This was due to *ConstRand*'s lack of coverage-based feedback that directs the test towards unexplored system behaviors, which is available to MCjammer.

5.5 Summary

This chapter shifted the verification focus from single core processors to multi-core ones. We started to address the challenges they pose at the pre-silicon level by presenting MCjammer, a novel scalable tool designed specifically for verification of memory coherence in multi-core/multi-processor systems. MCjammer uses multiple adaptive agents that are connected to individual processor nodes in the system and that work together to generate concurrent, and often conflicting, memory accesses. This coordination allows MCjammer to thoroughly cover the behavior of the design under test while also gradually increasing pressure on it to test "stressful" operations of the design. To set verification goals and to evaluate coverage, each agent owns a simplified view of the full system coherence protocol, whose complexity is linear in the number of processors/cores in the system. MCjammer also features unique data tagging that allows it to quickly detect errors in the design and verify consistency rules. The experiments on several MESI and MOSI systems of varying size demonstrate the coverage and scalability potential of MCjammer, showing that it achieves higher coverage with lower effort than a constrained-random generator.

CHAPTER VI

Post-Silicon Verification of Multi-Cores

The introduction of multi-core processors pushed computer systems up to a new level of performance, and brought the chip manufacturers face to face with the challenge of verification of these designs, featuring complex and sometimes non-deterministic memory subsystems. The deteriorating situation with an increasing number of subtle, yet devastating, bugs slipping into production silicon called for high-efficiency, high-coverage validation. To address the challenge, post-silicon functional validation, where validation experiments are run directly on newly manufactured prototype hardware, is becoming a vitally important aspect of today's design and manufacturing process. The key advantage of post-silicon validation is that its raw performance provides significantly faster testing and analysis than pre-silicon software-based simulation, thus promising to deliver much higher coverage before releasing the part to customers.

In this chapter we present the first post-silicon solution to validate the correctness of the shared memory subsystem in multi-core processors. Our solution, called Dacota (Data-coloring for COnsistency Testing and Analysis) has minimal performance impact, provides high coverage of consistency and coherence related bugs, and has extremely low silicon area overhead. Dacota incorporates a simple hardware activity logging mechanism that examines the system during program execution. In addition, Dacota periodically invokes a software-based analysis engine that detects violations in memory consistency and cache coherence of the system.

The activity logging mechanism stores a compact encoding of memory accesses at each L1 cache of the multi-core processor. First, we connect to each cache line an *access vector*, which contains a counter "color" value, incremented on each store to the line. The access vector is used to disambiguate accesses to the line throughout the execution. In addition, after each load and store, individual cores of the processor log the address and color values of accesses to an *activity log*, maintained in the local cache. The log contains a history of individual core memory accesses in program order.

To detect memory consistency and coherence bugs, Dacota periodically collects the cores' activity logs and runs a software-based memory checker, taking advantage of the available processing power of the processor. The procedure requires building a consistency model-specific graph in which vertices represent memory accesses and edges indicate the observed ordering between them. Finally, Dacota inspects the graph for cycles that occur because of memory consistency and coherence bugs.

Our overarching goal in designing Dacota was to provide an effective solution for post-silicon validation, where the raw performance of the hardware prototypes is not hampered by frequent, time-consuming scan chain and/or logic analyzer operations. To achieve this, we realized early in the project that the monitoring and diagnostic harness needed to be embedded in the hardware prototype itself. Moreover, our philosophy was to create a system where logging is seamlessly integrated into the caches of a multi-core such that it minimally perturbs the timing of program execution. This allows Dacota to conduct high-coverage validation of scenarios that closely resemble real-life execution flows. Note that, while performing validation through embedded hardware components brings a great performance advantage, it also constitutes a pure cost once the system goes into production. Hence, we strove to keep the hardware cost of Dacota extremely low, since it provides no direct benefit to the end-user in the field (except, possibly, for in-field diagnostics).

6.1 Dacota Overview

In developing Dacota we assumed the underlying processor architecture illustrated in Figure 6.1.a. Multiple simple processing elements (cores), each with a private L1 cache, are connected via an on-chip interconnect fabric to a shared L2 cache. When Dacota is enabled, the system is reconfigured as shown in Figure 6.1.b, so that a portion of the cache resources is reserved for exclusive use by Dacota to store information about the order of memory accesses. A possible partition of the memory is shown in the Figure, where half of the cache is dedicated to Dacota and the other half to mainstream data storage. Other solutions, where Dacota only uses a smaller fraction of the cache, are also possible.

When Dacota is active, each cache line is partitioned to include additional information side-by-side with the data block. This information is structured as an *access vector* that records the number and the order of store operations issued to the line. Each core modifying the data in the cache line, must also update the corresponding access vector. We use the access vector, which travels throughout the system along with its data block, to track the order of store operations to individual cache lines. In addition, we use the storage space allocated to Dacota in each local cache to log a snapshot of the access vector upon each load or store operation. This logs are later analyzed to validate memory consistency

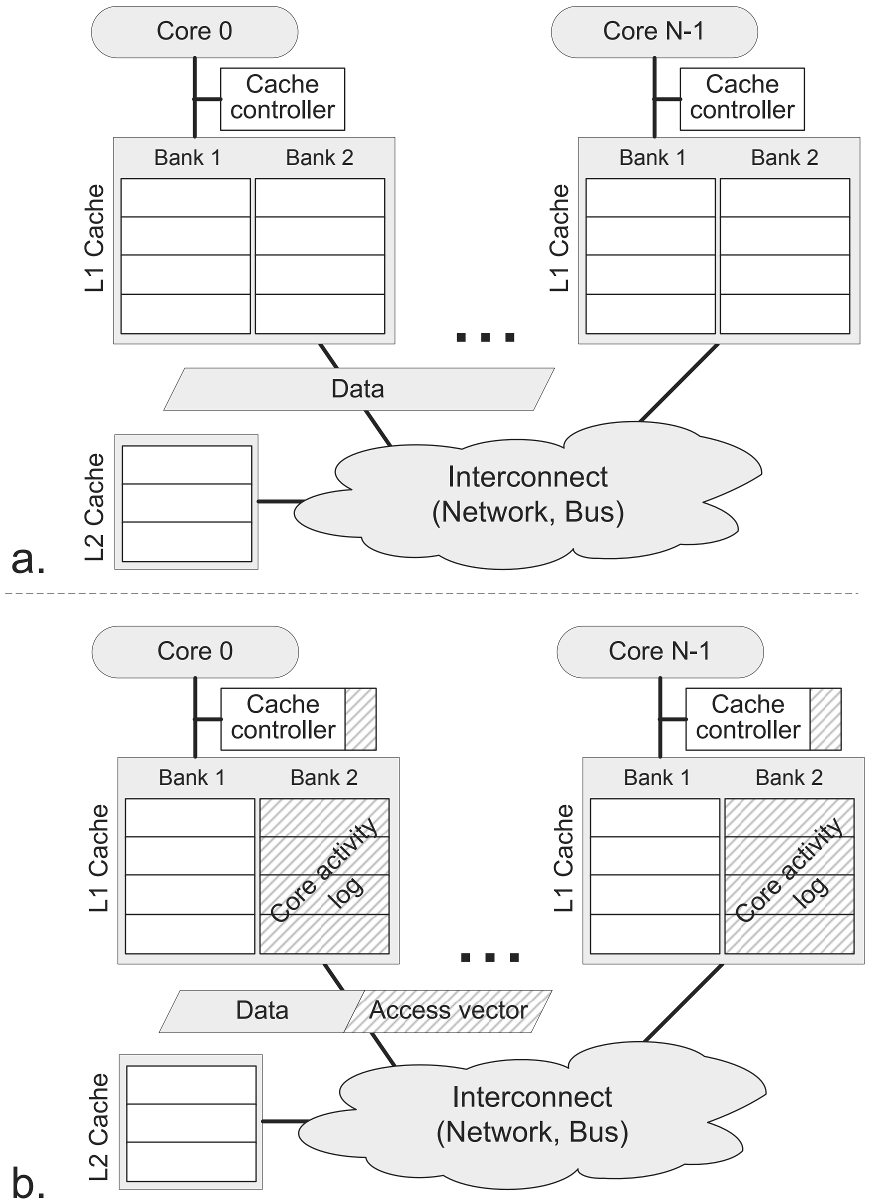


Figure 6.1: Typical multi-core processor architecture and system reconfiguration for Dacota post-silicon validation. a. Multiple processing elements (cores), each having a private L1 cache are connected with on-chip fabric to each other and an L2 cache. **b.** Each cache line is partitioned to include an *access vector* to track the access order to the line. A portion of the local caches is reclaimed and used as *activity log* storage for load/store operations. Finally, the cache controllers are augmented to include supporting hardware.

and coherence. We call this information the *core activity log*. The log records the address of each location to which an access has occurred, and the content of the corresponding access vector at the time of access. Additionally, when required to support specific consistency models (such as Weak Consistency), Dacota may also record special memory synchronization instructions.

The analysis algorithm in Dacota is applied at regular intervals and it relies on a *consistency graph* as its underlying data structure. Consistency graphs are built based on the aggregate information gathered from all the core activity logs over a period of time and on the consistency model in use. The vertices of the graph represent memory accesses, while the directed edges indicate relative operation sequencing as perceived by different cores. A subsequent graph analysis allows to determine if a memory consistency violation has occurred (manifesting as a loop in the graph). The analysis engine can also expose coherence violations if they are exposed through the consistency graph or by incompatible access vectors in the activity log.

Graph construction and analysis in Dacota are executed in software on the cores of the processor itself, requiring no additional hardware for this phase of the memory subsystem validation. In summary, the only hardware overhead incurred by Dacota consists of a small block in the local cache controllers to maintain access vectors and activity logs. All other resources in our solution are gathered by reconfiguring the system only for the duration of the post-silicon validation process.

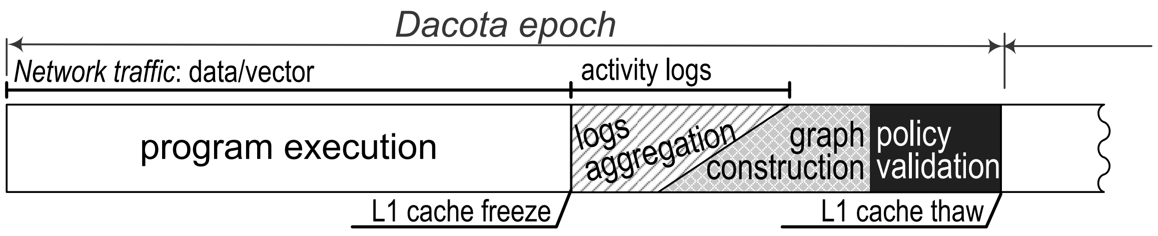


Figure 6.2: Dacota execution flow. During normal program execution data and access vectors are transferred together and activity is logged. When log resources become exhausted, program execution stops and data in local caches is frozen. The activity logs are then aggregated into un-cacheable memory and used to construct the consistency graphs, which are then analyzed to detect possible violations. If an error is found, the logged information can support system debugging, otherwise execution resumes.

While Dacota is active, processor activity is organized into epochs, as shown in the schematic of Figure 6.2. During the phase of normal program execution, Dacota monitors activity in the background, by updating and transmitting access vectors along with data and by logging individual cores' activity. When any of the log storages is exhausted, program execution stops, data in transit is allowed to reach its destination, and the data portion of all caches is frozen. All cores then drain their activity logs into a dedicated region of un-cacheable memory (log aggregation). Graph construction can start as soon as at least some of the logs have been aggregated. When the graph construction is completed, Dacota runs its policy validation algorithm to expose coherence and consistency errors. If the analysis exposes an error, information from the activity logs can be leveraged to support subsequent debugging. On the other hand, if no error is detected, all activity logs and access vectors in the system are cleared and normal execution may resume.

6.2 Activity Logging

The purpose of the activity logging system in Dacota is to record the order of shared memory accesses observed by different cores. To this end, we maintain both an access vector attached to each cache line as well as activity logs residing in the cores. The information in the access vectors and logs is updated concurrently with program execution and incur no performance overhead during the program execution phase. However, when the log storage resources are exhausted, normal execution is suspended and the activity logs are aggregated and analyzed by the policy validation engine, discussed in Section 6.3.

6.2.1 Access vector

To identify coherence violations, Dacota logs the order of accesses to a particular cache line using a scheme based on data coloring. Each cache line is partitioned into two parts: the program data occupies one portion of the line, while the other part is configured as a vector. This *access vector* propagates together with the data through caches and interconnect of the multi-core processor. The access vector comprises N+1 entries, where N is the number of the cores in the system. Each core has a corresponding entry in the vector, updated when that core performs a store access to the cache line. The last entry is reserved for a counter tracking the total number of stores to the line since the beginning of the epoch. Figure 6.3 shows several examples of access vectors, where the counter is the entry shown in gray color. The counter and all entries of the vector are initialized to zero in the beginning of the epoch and are updated as follows: on each store, Dacota automatically increments the counter and copies its value to the vector entry associated with the issuing core. Therefore, we use monotonically increasing counter values to establish the chronological order in which the cores modified the line during the epoch. During the policy validation phase any disagreement between cores on the order of stores exposes a coherence error, since the order of stores to a line in a coherent system must be unique. Note that, when the counter reaches the maximum value, we must stop the execution to preserve the uniqueness of the line's access order and invoke the analysis algorithm.

To illustrate how access vectors operate, we show several examples in Figure 6.3.a-d. Part 6.3.a shows the vector associated with an unmodified cache line: all of its N entries and the counter have a zero value, indicating that the line was not yet modified by any of the cores. Figure 6.3.b shows the vector of a cache line that experienced three stores (as indicated by the counter value 3) issued by cores 0, 2, and 1 in this order. Initially, the line starts with a vector as in 6.3.a, and after the Core0 store, the counter and the first entry are updated to 1. When Core2 issues a store, the counter is incremented to 2 and this value is copied into the third vector entry. Finally, the store by Core1 changes the access vector to its final state shown in Figure 6.3.b. Observe that in this example the vector uniquely identifies the order of all store operations to the address block. In contrast, Figure 6.3.c shows a situation where three store operations have occurred, yet we can only determine which core executed the last one. Fortunately, in this specific example, the fact that all other entries are at zero indicates that Core0 is indeed responsible for all the stores. In the last example in Figure 6.3.d, we cannot determine precisely the unique order of the accesses to the cache line, since we cannot determine which one among Core0 and Core1 issued the first store operation. Note, however, that if we had a snapshot of this access vector before the third store operation was issued, then we would be able to establish the sequence of operations precisely.

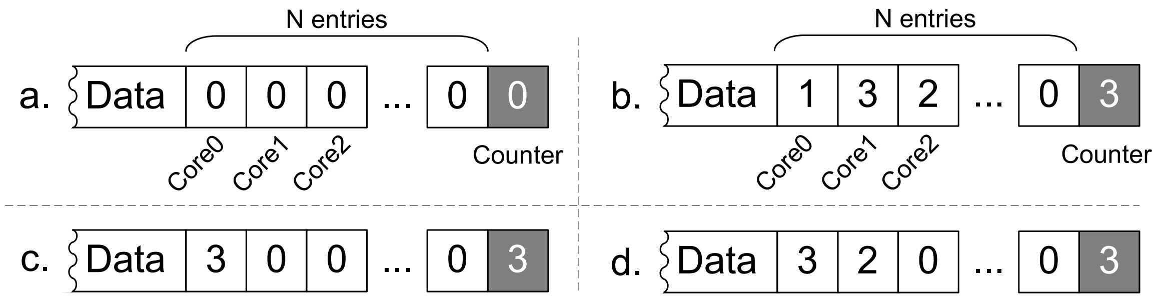


Figure 6.3: Dacota access vector. An access vector is associated with each cache line in the system and is used to track the order of store operations to a line. **a.** Access vector at the beginning of an epoch. **b.** Access vector indicating that the corresponding cache line has been modified by Core0, Core2 and Core1 in this order. **c.** and **d.** Examples of aliasing due to multiple stores by a same core. In c. the order can be inferred because of the 0s in the other entries; in d. the complete order cannot be recovered.

To manage access vector updates we require the addition of a small hardware block to the native local cache controller. On a store operation, this hardware component is responsible for incrementing the counter and updating the entry corresponding to the local core. From a structural standpoint, it can be implemented as an adder or an up-counter with saturation. This circuitry is sufficiently simple that it can easily be made to operate concurrently with the main cache accesses. Note that, at the cost of increased hardware complexity, it would be possible to eliminate the counter entry from the access vector and simply retrieve the counter value by extracting the highest value in the access vector entries. This alternative approach incurs higher area cost, but could be interesting for system where the resources for the access vector are extremely limited.

6.2.2 Core activity log

As presented above, Dacota access vectors associated with individual lines allow to monitor the order of accesses issued to those lines and detect coherence violations. Unfortunately, they provide no information about the interleaving of the accesses to different lines and are by themselves insufficient to validate memory consistency. To this end, we also record the sequence of operations issued by individual cores and the values of the access vectors associated with the accessed lines in an *activity log*. To incur minimal area cost we place this activity log in a portion of each core's L1 cache, that is made inaccessible to the executing program while Dacota is active. When a core issues a load or a store, Dacota allocates space in the log and copies there the updated access vector together with the type of access (load/store) and the cache line tag. The activity log is maintained as a queue and entries are allocated in program order. On the other hand, the access vectors of lines are added to the log in the order in which they complete, which may be different from the program sequence. The values of the vectors allow Dacota to observe data dependencies and identify the last store that modified the line. By leveraging program order and the entries content in the activity log we can reconstruct the order of memory operations perceived by a single core. When logs are aggregated at the end of the epoch, memory consistency invariants can be validated by the policy validation engine. When multiple cores access a same shared location, we can leverage the access vector snapshots stored in their activity log to detect a broad range of coherence violations.

To reconfigure Dacota's portion of the L1 cache as a queue, we augment it with a simple up-counter that cycles through the allocated ways and sets. The tag array stores a portion of the location's address, while the data block stores spill-over address bits and the instantaneous value of the access vector associated with the line. In addition, since the order of completion of memory operations may be different from program order, we add a small index table for conversion between outstanding memory accesses and entries in the cache. Because we only need to index the outstanding memory accesses, the table can be quite small.

An example operation of the activity log is shown in Figure 6.4.a-d. In the beginning, the core issues a store to location A, allocating an entry for it in the log and recording the conversion in the index table. Before the store completes, a load to B is issued and completed (Figure 6.4.b), logging the access vector of line B. When the store completes in Figure 6.4.c, the vector of line A is copied to its pre-allocated entry. When the log fills (Figure 6.4.d), a signal is asserted and the policy validation algorithm is invoked.

In developing Dacota, we observed that logging each memory access led to prohibitively

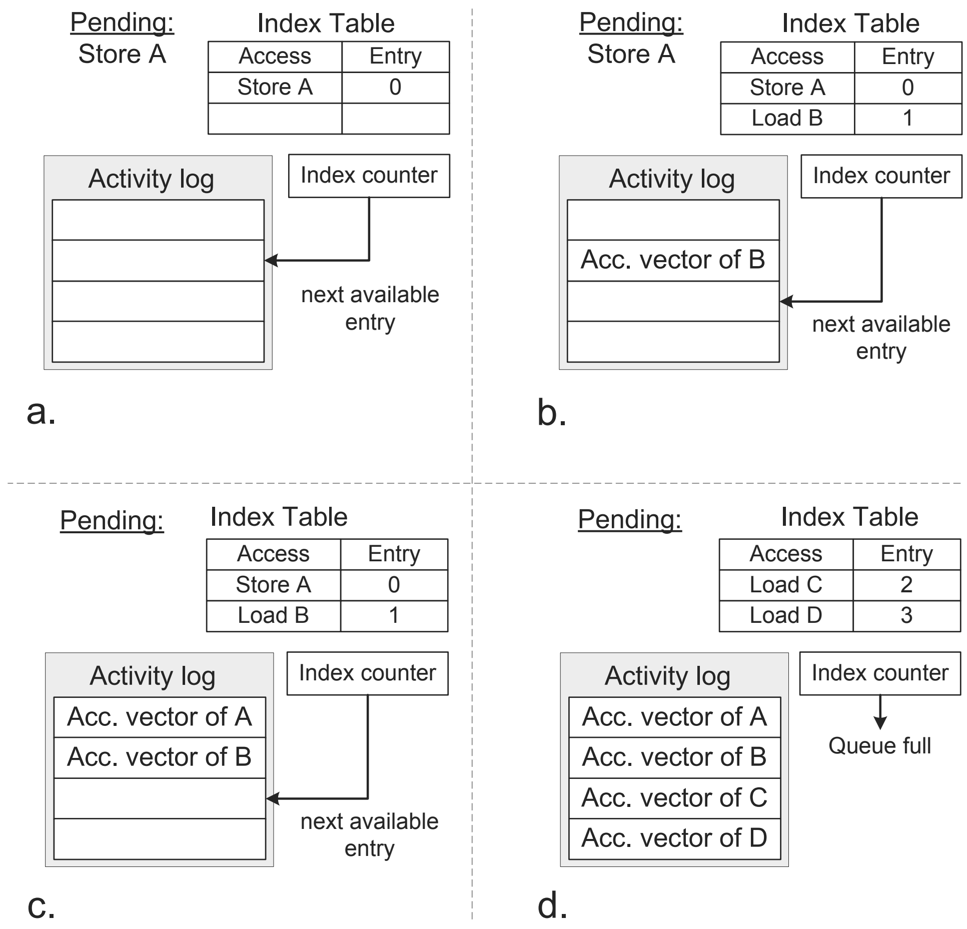


Figure 6.4: Dacota activity log operation. The activity log records addresses and access vectors associated with accessed lines. The log reflects program order, while data dependencies are exposed by the access vectors' content. The index table connects accesses that have not yet completed to their corresponding log entry. These constrains are later used by the policy validation algorithm to check system coherence and consistency. **a.** Store to line A is issued and the counter and the index table are updated correspondingly. **b.** Load to line B is issued and completes before Store A. **c.** Store to line A completes and the updated vector is logged. **d.** After loads to C and D the log fills and system initiates the policy validation algorithm.

large storage requirements. Thus, we optimized our design to log a load access only if it triggered a cache miss, either because the data block is not cached, or because the copy in the cache is obsolete due to a modification by another core. While this optimization may lead to hiding certain cache coherence faults, we hope that the input stimuli's redundancy will expose them in other scenarios.

6.2.3 Activity Logging Example

An example of Dacota's logging system in operation is presented in Figure 6.5. We assume a processor with two cores, MESI coherence protocol and a load/store buffer that enforces strict program order. For this and subsequent examples we adopt the following notation for the access vector format: $c0|c1\,\underline{cnt}$, where $c0$ and $c1$ are the entries for Core0 and Core1, and $\underline{cnt}$ is the counter. For example, access vectors in Figure 6.3.a and 6.3.d are recorded as $0|0\,\underline{0}$ and $3|2\,\underline{3}$, respectively. For simplicity we do not show the index table and the index counter of the log.

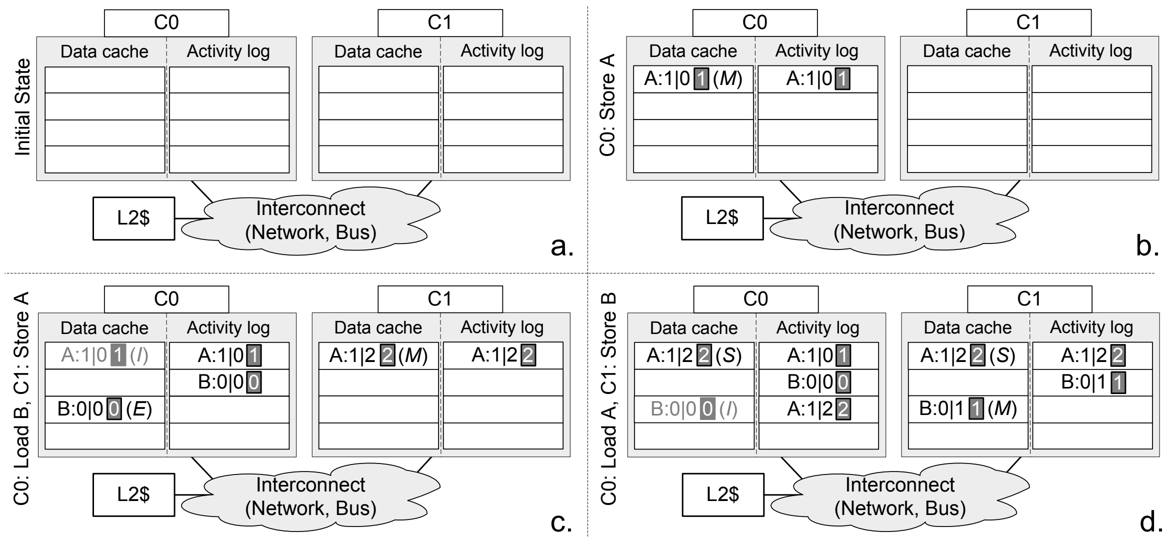


Figure 6.5: Example of Dacota activity logging system. The example is based on a system with 2 cores, MESI protocol and in-order execution of individual cores' memory operations. **a.** Initial state of the system: all caches and logs are empty. **b.** Core0 issues a store to address A, updating and logging the line's access vector. **c.** Core0 issues a load to cache line B and records its vector in the activity log. Core1 issues a store to cache line A, updates its vector and logs it. **d.** Core0 loads A with the updated access vector, while Core1 modifies line B, and inserts its updated vector in the log.

Initially, both L1 caches, as well as the logs, are empty (Figure 6.5.a). The first access (store to A) is issued by Core0 in Figure 6.5.b and brings line A's into the L1 cache in modified state. The access vector of the line is then updated: the counter is incremented and the new value (1) is copied into Core0's entry. The resulting value $1|0\,\underline{1}$ and the address of A is copied to the core's activity log. Subsequently, Core0 issues a load to B and Core1 issues a store to A (Figure 6.5.c). As the result, line B (with access vector $0|0\,\underline{0}$) is brought to the local cache and recorded in the activity log of Core0. Note that since a load operation is performed, the access vector is not updated. The consequence of Core1's store, on the other hand are as follows: line A is invalidated in Core0's cache and is moved Core1 in modified state. The vector is transferred together with the data and is updated to $1|2\,\underline{2}$. Note that the log of Core0 still preserves a log entry for a previous store to A, while the line itself is no longer in Core0's cache. Finally, in Figure 6.5.d Core0 issues a load to A and Core1 issues a store to B. The latter invalidates line B in Core0's cache, transfers it to Core1 and updates the vector to $0|1\,\underline{1}$. The former, puts line A in shared state in both caches and records its access vector in the activity log of Core0.

At the end of the execution we can use the information from the logs to establish how the order of store operations was perceived by individual cores. For example, we observe that Core0 has the following log entries $A : 1|0\,\underline{1}$, $B : 0|0\,\underline{0}$, $A : 1|2\,\underline{2}$. We can thus determine that when Core0 fetched line B, this was not yet modified by stores in Core1.

6.3 Policy Validation

The values of access vectors logged during Dacota operation represents the order of accesses perceived by the individual cores of the processor. When one of the access vectors saturates (*i.e.*, the counter reaches the maximum value) or one of the activity logs fills, the cores stop execution and aggregate the logs in an un-cacheable memory region. Log aggregation partially overlaps the graph construction process, which is followed by the policy validation through graph analysis. To minimize the area overhead of Dacota we implement the checking algorithm in software running on the processor itself.

6.3.1 Access log aggregation

When a core detects that its log is full or the counter of the accessed line's vector reached the maximum value, it broadcasts a message requesting validation of coherence and consistency. Upon receipt of the message all the cores are required to pause the execution and finish all pending memory operations. Afterwards, the data portions of the cores' caches are frozen, so that after the check program execution can restart with exactly the same state of the caches. The private activity logs are then transferred to a region of un-cacheable memory and accessed by all cores for graph construction and analysis. Note that only activity logs are drained, while the values of access vectors remaining in the frozen portion of the cache are cleared.

6.3.2 Graph construction

After the activity logs from all cores are drained to a shared memory region, Dacota engages a policy validation algorithm to determine if the execution in the preceding epoch was error-free. First, Dacota builds a directed graph, whose construction varies with the consistency model. The graph's vertices are unique memory accesses issued throughout the epoch, and edges are the ordering constraints on them. The constraints are derived from both the program order imposed by the activity logs and the data dependencies observed by loads and stores through access vector updates. As graph construction progresses, Dacota conducts a coherence invariant check using the access vectors of the individual lines.

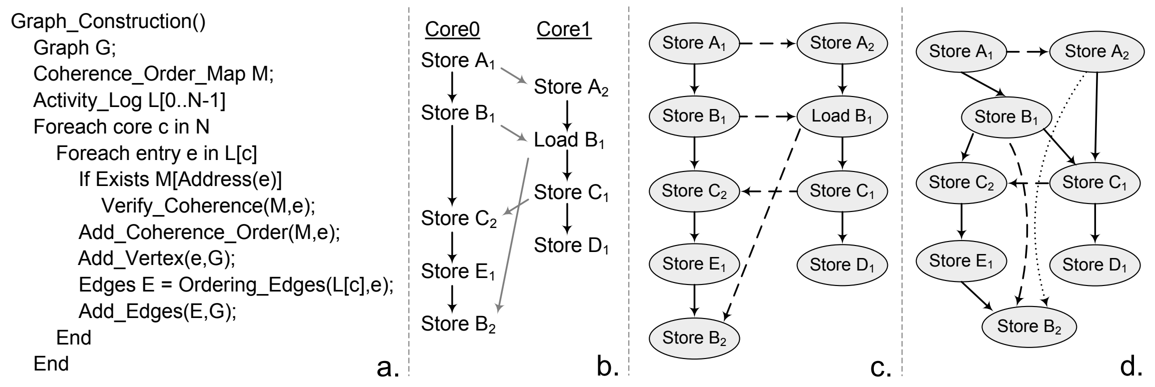
The pseudocode for the graph construction algorithm is given in Figure 6.6.a. For each core, the algorithm iterates through every entry of the core's log in the recorded (program) order. For all entries, a preliminary check is performed to verify the uniqueness of the order of stores to the individual cache lines. In other words, the algorithm checks that for a single line all cores agree on the order of the write operations. For this coherence check, we employ a data structure that maps the line address to a complete list of stores issued to this line. Each time the line's address is encountered in the logs, its access vector is compared to the list to see if there are any violations. If the entries of the vector describe an access that was not seen previously, the ID of the core that issued it is added to the list. After the preliminary check, we use the information in the log entry to augment the graph for the given system consistency model.

Sequential Consistency. Sequential consistency requires that the order of operations of all cores is perceived uniquely throughout the system. For systems employing this model we construct a graph from the aggregated logs as follows. Loads and stores become vertices of the graph and edges are of two types: program order edges and address reference edges. Program order edges are imposed by the order of entries in the activity logs, while the address reference edges are derived from load and store operations issued to the same location and represent data dependencies . Any loop in the consistency graph indicates a disagreement on the unique order of loads and stores by different cores, which violates the consistency policy.

Total Store Ordering (TSO). This consistency model allows reads to complete before previously issued writes. In the graph construction for TSO, we do not create vertices corresponding to loads, but rather use them to impose edges in the graph. Therefore, the vertices and program order edges of the consistency graph for Total Store Ordering are constructed only from the log entries describing write operations. Address reference edges are also constructed from store entries, connecting subsequent writes to the same cache block. Finally, the loads are used to construct chronological edges between the vertices. In the access vector of a load we identify the last operation to modify a location and infer that any later stores to it did not execute yet. Thus, we create edges from all completed write operations to the next store to this location. As with Sequential Consistency, any cycle in the graph indicates an error.

Processor Consistency. This model demands stores issued by a single core to be perceived in a unique order throughout the system. Therefore, for Processor Consistency we construct separate graphs for stores from each of the processor cores using information from the aggregated logs. The construction process is similar to that of TSO, however only stores of a single core are used as vertices in each of the graphs, while edges are imposed by accesses from all cores. Note that a loop in one of the graphs built for such a system indicates that stores of the corresponding core were not perceived uniquely, flagging the problem.

Other consistency models. Weak consistency models require that only special instructions (such as memory barriers) are perceived in a unique order throughout the system, while the observed interleaving of accesses between the synchronization operations may be different for different cores. For those systems, in addition to loads and stores, we log the synchronization instructions and use them as vertices in graph construction. The edges between the operations are derived from the log entries of ordinary loads and stores. A loop in a graph would indicate an error where an access incorrectly bypassed a memory barrier, violating the consistency. Note that coherence of a single line is still validated by our algorithm during the graph construction.



```
Graph_Construction()
    Graph G;
    Coherence_Order_Map M;
    Activity_Log L[0..N-1]
    Foreach core c in N
        Foreach entry e in L[c]
            If Exists M[Address(e)]
                Verify_Coherence(M,e);
            Add_Coherence_Order(M,e);
            Add_Vertex(e,G);
            Edges E = Ordering_Edges(L[c],e);
            Add_Edges(E,G);
        End
    End
```

Figure 6.6: Graph construction algorithm for systems implementing Sequentially Consistency and TSO. a. Pseudocode for graph construction algorithm. The algorithm iterates through all history logs and checks the coherence invariant for every entry. Ordering edges are then derived and added to consistency graph. **b.** Example of an interleaved sequence of loads and stores issued by two cores. Black lines represent program order constraints, while gray lines represent data dependencies. **c.** Consistency graph for a sequentially consistent system for execution in b. Solid lines represent program ordering edges, while dashed indicate single location constraints. **d.** Dacota consistency graph for a system implementing Total Store Ordering for execution in b. Dotted lines represent chronological edges.

An example of an interleaving of loads and stores is shown in Figure 6.6.b, where black lines indicate program order constraints and gray lines show data dependencies. In Figures 6.6.c and 6.6.d, we show the graphs constructed by Dacota for a sequentially consistent multi-core processor, as well as one implementing TSO. In the figures, the program ordering constraints are noted with solid edges, while dashed and dotted lines stand for single location and chronological edges, respectively. The chronological edge in Figure 6.6.d between A_2 and B_2 is derived as follows: when the load B_1 completes, it obtains the access vector last updated the store B_1, indicating that store B_2 was not yet executed. Thus, store A_2 is perceived by Core1 to occur chronologically before the write B_2, as shown by the dotted edge.

6.3.3 Graph analysis

Consistency graphs in Dacota are constructed to reflect the order in which accesses performed by individual cores are perceived in the system. The graphs are constructed in accordance with the consistency model and must be acyclic to indicate the absence of errors. Thus, to find bugs Dacota searches the constructed graphs for loops, employing a modified version of the Depth First Search (DFS) algorithm [43]. At each iteration of the algorithm we check whether the current vertex has already been seen on the search path. Since the existence of the vertex on the path is represented efficiently with a bit vector, the algorithm retains the complexity of the underlying DFS, equal to $O(E)$, where E is the number of edges in the graph. Therefore, the number of edges of the graph becomes the determining factor in the algorithm's speed. To ensure maximum performance, we aggressively apply transitive closure during the construction of the graph, thus reducing the number of edges. Note that graph construction can be started before the logs from all cores are aggregated, overlapping the computation and communication overheads of Dacota-based verification.

A crucial feature distinguishing Dacota as a post-silicon solution from runtime approaches based on comparable graph analysis techniques is the fact that we do not use any additional hardware for policy validation. While a software-only algorithm may not be as fast as the hardware checkers suggested by runtime approaches, it dramatically reduces the silicon area overhead of Dacota. Moreover, this allows us to parallelize the analysis for more common weaker consistency models where the perceived order of memory operations is not unique and multiple graphs need to be constructed. Even for systems with a single graph, such as the those enforcing Sequential Consistency, Dacota uses cores of the processor to run multiple parallel searches for loops starting from different vertices. Once one of the cores finds the loop or visits all paths in the consistency graph, the entire search terminates. Such parallelization does not increase the search time in the absence of bugs and can expose existing bugs faster. Finally, during the graph construction and analysis period, we allow Dacota to reconfigure the activity logs of individual cores to be used as regular cache space. Recall that when the program execution pauses, the data portions of the L1 caches are frozen and cannot be used until the end of the analysis. However, once the log is transmitted to a shared memory region, we no longer need to maintain it locally and can reclaim this space for improved performance of Dacota graph algorithms. When the check completes the cache is again reconfigured as an activity log, the data cache is thawed and the next Dacota epoch commences.

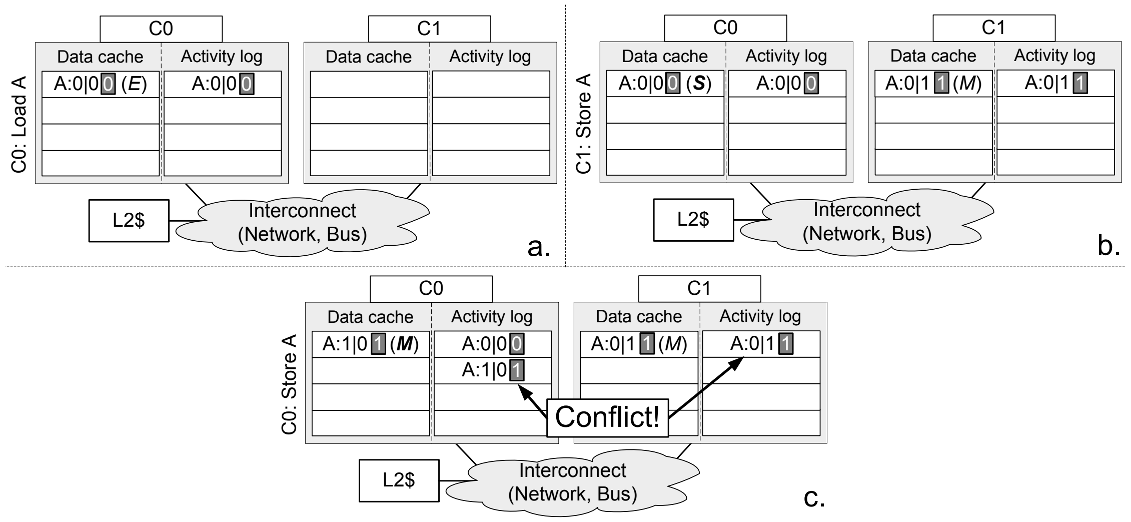


Figure 6.7: Example of coherence conflict detected by Dacota. a. Store to line A is executed by Core0. **b.** Core1 requests exclusive access to line A to perform a store, however, the cache of Core0 is updated erroneously and exclusivity is violated. **c.** Core0's store to line A updates the access vector, leading to a conflict detected by Dacota at graph construction time.

6.3.4 Examples of error detection

In this section we overview two examples of errors that may occur in a multi-core processor and show how Dacota is able to identify them through consistency graph construction and analysis. The example in Figure 6.7 shows how Dacota detects a coherence violation in a system. The figure demonstrates a case when the state of the cache line A is not updated properly when Core1 requests exclusive access to the line. Normally, in the situation shown in Figure 6.7.b, Core0 must invalidate the line in its cache, yet the line's state is changed to shared (here we assume MESI coherence protocol). This, in turn, leads to the situation in Figure 6.7.c, where two modified copies of the cache line exist in different caches at the same time. In other words, the cores disagree on the order of stores they issued to the line, which is detected by Dacota from the access vector of A recorded in the cores' activity logs. Note that if Core0 does not perform a store (Figure 6.7.c) and retains a stale copy of the data, Dacota will not be able to flag the bug through a coherence check. However, if other write operations are performed to different lines this coherence error will manifest itself as a loop in the consistency graph, since the cores will disagree on the order in which line A was modified with respect to the other stores.

A more complex fault, this time in memory consistency, is shown in Figure 6.8, where we assume the Sequential Consistency model, *i.e.*, all cores must agree on the order of accesses that occurred. When Core1 issues its loads, their execution order is erroneously reversed by the system because of a non-deterministic interconnect or incorrect behavior of the memory controller. Thus the load to A completes first, bringing an unmodified vector associated with the line into Core1's cache and activity log. Then, stores of Core0 complete, and, finally, the load to B returns to Core1 with the line's data and access vector. Thus, the reversal of the load execution sequence causes the order in which stores to A and B are perceived by the cores to be different, resulting in the loop in the consistency graph. Dacota's graph analysis algorithm then detects the cycle and flags the error.

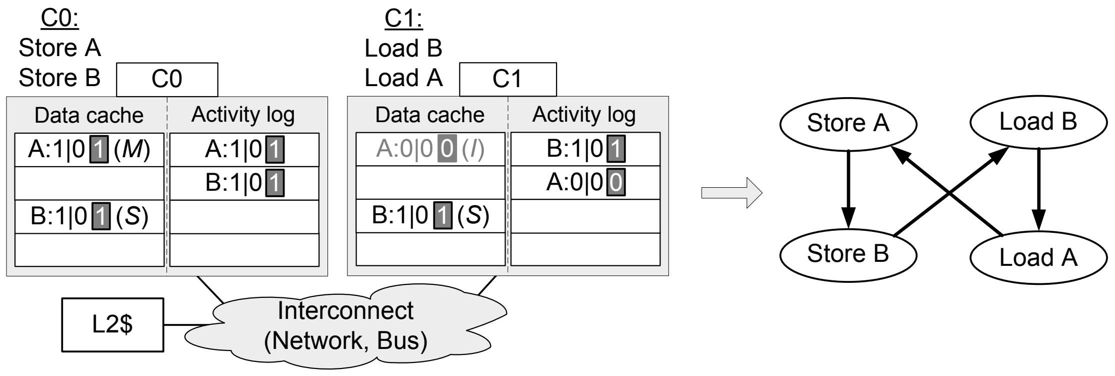


Figure 6.8: Example of consistency violation detected by Dacota. Assuming a system with sequential consistency model and MESI cache coherence protocol, a load to A is erroneously reordered to execute before load to B and completes first. Then, stores from Core0 execute (invalidating A in Core1's cache), and, finally, load to B completes and updated access vector of B is recorded by Core1. The order in which accesses to A in B are observed by Core0 and Core1 in this case is different, which leads to a loop in the consistency graph constructed by Dacota. The loop is then detected during the policy validation and the error is flagged.

6.3.5 Requirements for checking algorithm

To minimize the area overhead of Dacota, its checking algorithm is designed to run in software, rather than on dedicated hardware. Therefore, for our approach to be reliable we require computational correctness from the cores, *i.e.*, correct operation of arithmetic/logic units, pipeline, *etc.* The cores also must be able to access main memory in order to build and analyze the graph. Note that these accesses use the same subsystem that Dacota is verifying. However, bugs in the memory subsystem are unlikely to cause analysis errors due to the absence of memory race conditions during Dacota analysis. The cores drain the activity logs into disjoint memory regions and then use this information without overwriting it. Moreover, since the logs are aggregated in the uncacheable memory, they will not be transferred core caches during the analysis, eliminating the chance that coherence or consistency bugs corrupt the analysis process.

6.4 Strengths and Limitations

While Dacota is effective in catching a diverse range of functional errors in the memory subsystem of a multi-core processor architecture, there are several limitations to this approach. First, its bug catching ability is dependent on the quality of the stimulus, that is the program running on the system under investigation. More specifically, Dacota can only find bugs that are uncovered by the workload. For example, workloads without shared data will not uncover coherence or consistency related bugs, nor will short programs which do not stress the memory system. Bugs that result in deadlock or system hang are also not flagged by Dacota, as we assume forward progress is always maintained. Additionally, bugs that do not change execution semantics evade capture by our validation system. Consider one cache line shared by two cores, both in the shared state of the MESI protocol. If one processor silently (and incorrectly) transitions the line's state to exclusive the system becomes incoherent. Yet, if no store is performed to the line, the execution semantics of the program is not violated. A store, however, will result in a flagged error since it updates the core's activity and the access vector.

6.4.1 Debugging with Dacota

One of the major strengths of Dacota is its support of the debugging effort. In addition to the detection of complex coherence and consistency bugs, Dacota can also help find the root cause of an error once it has been discovered. When an epoch of execution completes, the core activity logs are aggregated for analysis, and additionally become an extremely useful resource for debugging. They provide detailed information about which cache lines were accessed, which cores accessed them, and when stores occurred. With this, an engineer can clearly see the activity of all the processor cores, as well as system level events leading up to the bug. Moreover, since the data portions of the cores' L1 caches are frozen during Dacota analysis, this information can be also be used to provide even more details about the violation. Finally, Dacota can be augmented with hardware to log the state of each core's internal registers at the end of each execution epoch, providing the engineer with even more details about the program execution. These features allow Dacota to be efficiently used for debugging of complex and non-deterministic multi-core systems.

6.4.2 Design Considerations

In order to make Dacota a viable post-silicon solution, its design was driven by two primary goals: high coverage and debuggability. Performance was an additional consideration, as high execution speed is critical to our primary goal of high coverage. The interplay between the logging scheme and the analysis algorithm required us to consider tradeoffs between them: a terse and efficient log leads to a lengthy and inefficient analysis algorithm. Early in the design phase, we considered attaching only a sequence counter to each cache line and storing these values in the activity log with each memory operation. This minimalist setup eliminates the need for the aforementioned access vector and potentially makes the activity log smaller, enabling longer periods of execution between checks. We quickly found that the analysis algorithm corresponding to this storage mechanism was grossly inefficient. Since the sequence counter did not record the order of accesses to the line, the order had to be inferred from the logs of every other core in the system. Thus, to determine the order of operations in the system, the algorithm was required to walk the entire set of aggregated access logs during the construction of every vertex in the graph. With the logs placed in un-cacheable memory, the overhead of the graph construction process outweighed the savings in log storage space. Furthermore, this scheme lost writer identification information, thus significantly hampering debuggability. In response to these performance and debuggability problems, we designed the access vector, which contains the number and order of stores issued to the line. With the addition of the access vector, graph construction became much faster, since the extensive memory search was not required. Moreover, the access vector provided additional information crucial for root-causing an error. Our design decisions were later validated by our experimental results, which showed that longer periods of execution between checks is not necessarily better for the performance of the graph analysis algorithm.

6.5 Experimental Evaluation

We evaluate the efficiency of Dacota in a simulated multi-core system, determining its error coverage, performance and area impact. Investigating how different configurations affect Dacota, we explore several activity log lengths and their affect on consistency graph size and policy validation algorithm runtime. We also measure how the amount of communication and computation overhead incurred by Dacota ranges across multiple benchmarks and log sizes.

6.5.1 Experimental Framework

Our simulation framework was based on a multi-core system modeled with the Wisconsin Multifacet GEMS architectural simulator [53]. The processor contained 16 cores, each with a 16 entry load/store buffer, 128KB L1 cache, a single 4MB L2 cache and a 4x4 on-chip mesh interconnect. The MOESI directory protocol was used as the coherence protocol and the Total Store Ordering consistency model was assumed. For several additional experiments, we evaluated systems with token coherence as well as crossbar and switch-based interconnects. Dacota itself was implemented as a plug-in for the simulator and included methods for access vector manipulation, core activity log management and the policy validation algorithm, implemented with the Boost Graph Library [65]. The runtime of the algorithms was calculated using the SimpleScalar architectural simulator [18].

The set of workloads used to test Dacota was a combination of real world programs and random stimulus. First, we used the ten SPLASH2 benchmarks [81], each 10,000,000 instructions long and generated by the Virtutech Simics simulator [51]. In addition, to induce more stress on the memory subsystem, we created eight tests of directed random stimulus with varying degrees of sharing, each containing 1,000,000 memory accesses. Three of these benchmarks had a fairly small address space that could fit into the cores' L1 caches without eviction, while the other five used significantly larger memory ranges and could not be fully contained in the L1 caches. Additionally, we used the GEMS built-in random test generator executing the "barrier" and "locks" patterns.

6.5.2 Design Error Coverage

In our first experiment, we introduced eight coherence and consistency related errors into our simulation model, inspired by known issues with industrial multi-cores. We then ran our SPLASH2 and random stimulus benchmarks with Dacota fully enabled and recorded the number of cycles required to discover the bug (Table 6.5.2). We found that the activity logging scheme of Dacota is capable of quickly finding complex coherence and consistency bugs.

6.5.3 Performance Evaluation

We also investigated the computation and communication overhead of Dacota relative to normal program execution time. Additionally, we measured the amount of extra traffic that our solution adds to the system under test. In this study, we assumed that one half of the L1 cache (64kB) was devoted to the data and associated access vectors and the activity log was limited to 256 entries. As you can see from Figure 6.9, the performance over-

Table 6.1: Design error coverage by Dacota.

Bug name	Description of the error	Avg. cycles to expose
shared_store	store to a shared line may not invalidate other caches	0.252M
invisible_store	store message may not reach all cores	1.32M
store_alloc_1	store allocation in any core may not occur properly	1.93M
store_alloc_2	store allocation in a single core may not occur properly	2.27M
reorder_1	invalid store reordering (all cores)	1.38M
reorder_2	invalid store reordering (by a single core)	2.82M
reorder_3	invalid store reordering (to a single address)	2.87M
reorder_4	invalid store reordering (to a single address by a single core)	5.61M

head was well within the acceptable range for post-silicon solutions: the total overhead (communication and computation combined) for SPLASH2 benchmarks is roughly 26%. The overhead for random benchmarks is somewhat higher due to the nature of these artificially created tests designed specifically to stress the system. Note that this is the worst case communication and computation overhead, when all graph construction is done after the logs are fully transferred to shared memory. Thus, a Dacota implementation where communication and computation overlap would have a smaller aggregate overhead. Since Dacota can be disabled upon shipment, these performance overheads are only realized during in-house analysis of a prototype.

In our next study, we varied the number of entries allocated for Dacota's activity logs, measuring the communication and computation overhead of our solution. The results of this analysis for the SPLASH2 benchmarks and random tests are presented in Figures 6.10.a-b and 6.11.a-b. We found that for all benchmarks, the communication overhead remains nearly constant despite different queue sizes. On the other hand, the computation overheads, *i.e.*, time to construct and analyze the consistency graphs, exhibit several interesting trends shown in Figure 6.10. For some benchmarks (*fft, lu, locks*), the ratio of analysis time to normal program execution time decreases as the activity log size grows. Interestingly, the computation overhead time for several other benchmarks, such as *cholesky, radix, etc.*, exhibits a local minimum at medium log sizes. This can be explained by growing complexity of the consistency graph, resulting in an increased time to build and analyze it. Our study demonstrates that the average aggregate overhead (computation and communication) is minimal when the activity log is 256 entries long.

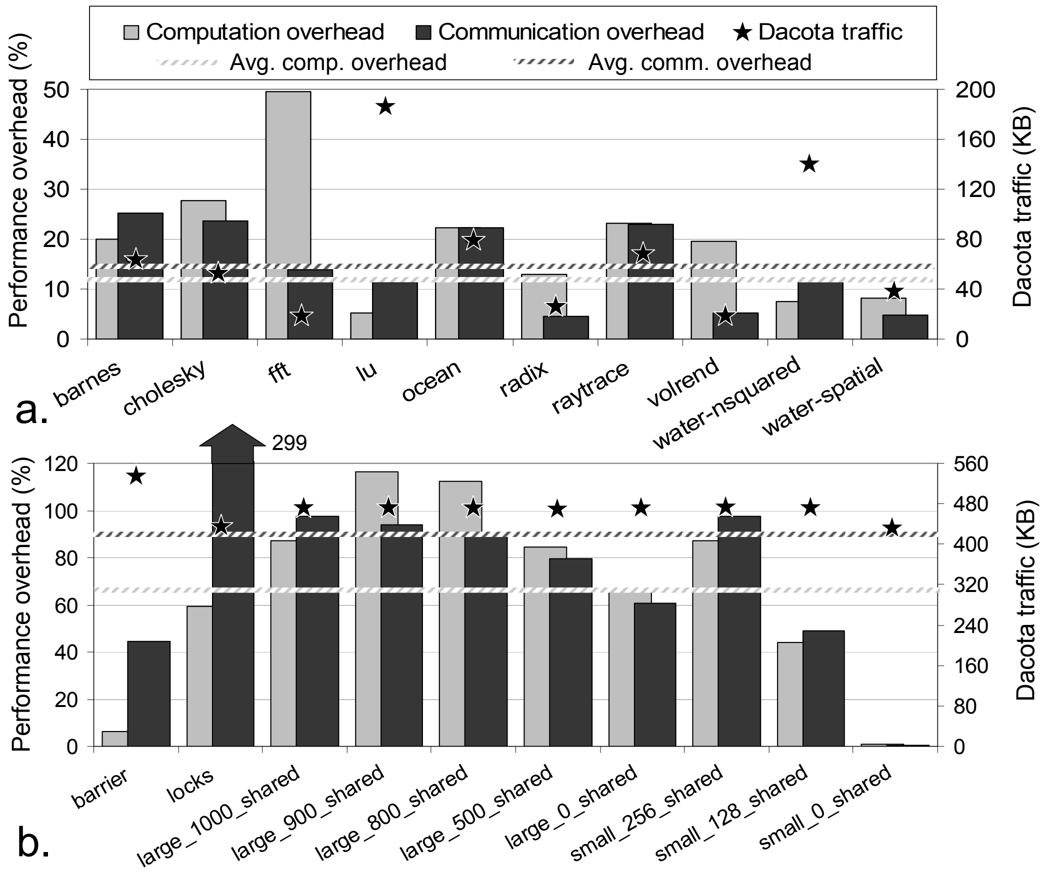


Figure 6.9: Dacota performance overhead and additional traffic for activity log size of 256 entries. a. Overhead for SPLASH2 benchmarks. **b.** Overhead for random stimulus.

In the next experiment, we ran the same benchmarks and activity log lengths, but varied the interconnect topology (crossbar or hierarchical switch) and used a different underlying coherence protocol (token coherence). Our results do not demonstrate any appreciable difference in Dacota's performance for different network topologies, however, changing the coherence protocol to token coherence resulted in an 8% reduction in communication overhead. We found that in this case, the ratio of Dacota traffic to normal system traffic is lower due to the larger number of control messages required to implement the token coherence scheme.

Finally, we measured the number of cycle between invocations of the policy validation algorithm as well as the average size of the resulting consistency graphs. The average time between analyses in simulation cycles is presented in Figure 6.12.a for SPLASH2 benchmarks and Figures 6.12.b and 6.12.c for the random tests. As you can see, with the growing size of the activity log, the time between checks also increases. An interesting observation presents itself in the *barrier* benchmark in Figure 6.12.b. When the log increases past 512 entries, the periodicity of Dacota checks remains the same. The plateau

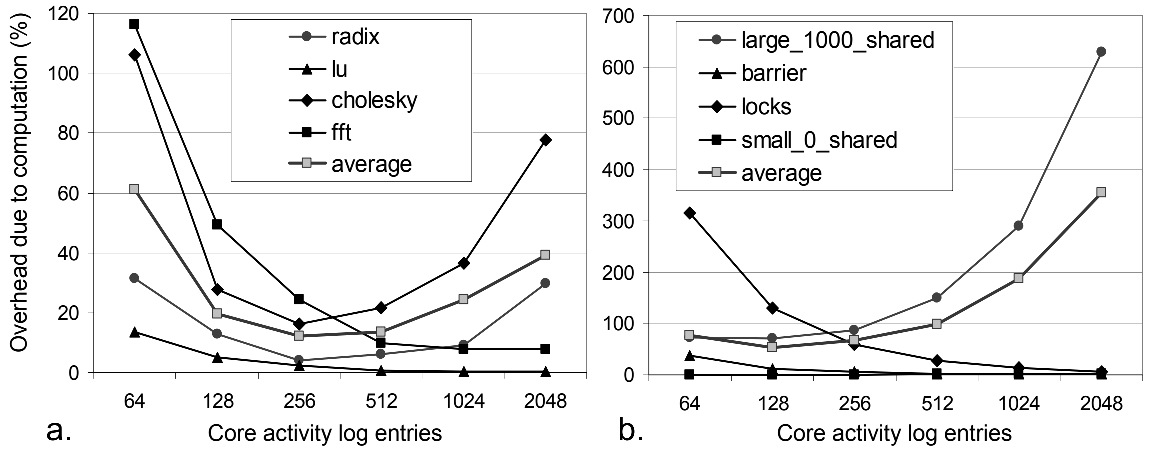


Figure 6.10: Dacota computation overhead. a. Overhead for SPLASH2 benchmarks (4 individual benchmarks and the average of all 10 benchmarks is shown) **b.** Overhead for random stress-tests (4 individual tests and the average of all 10 tests is shown).

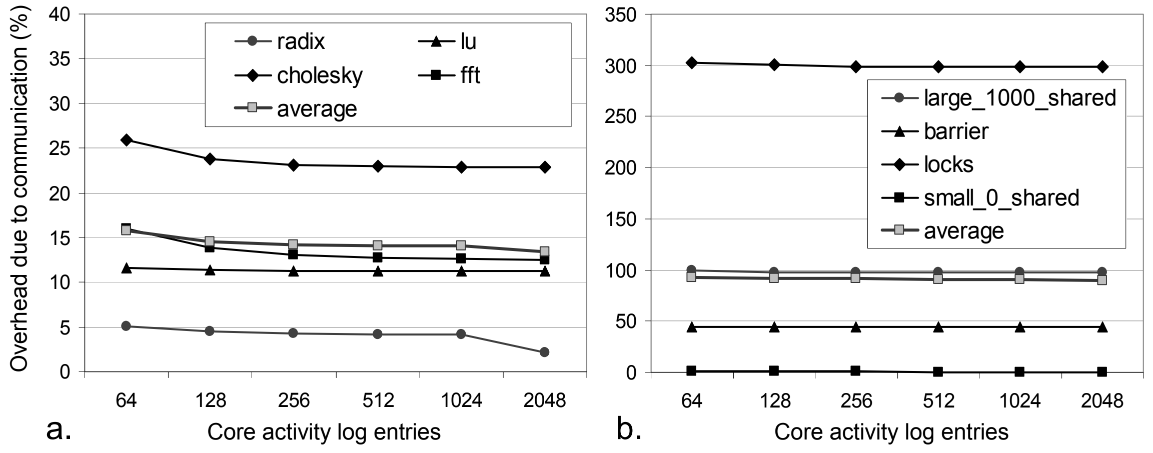


Figure 6.11: Dacota communication overhead. a. Overhead for SPLASH2 benchmarks (4 individual benchmarks and the average of all 10 benchmarks is shown) **b.** Overhead for random stress-tests. (4 individual tests and the average of all 10 tests is shown).

in this benchmark occurs because the check is triggered by access vector counter saturation, not due to the filling of the log. Another special case is illustrated in Figure 6.12.c by benchmark *small_0_shared*. Recall that small random benchmarks in our experiments have address spaces that fit completely in the L1 caches of individual cores. Since this particular benchmark has no shared data, lines cached by the cores are never invalidated or evicted. Thus, with optimizations discussed in Section 6.2.2, the core activity logs never fill up throughout the entire execution time of the benchmark and Dacota analysis is invoked only once, after the benchmark's completion.

Analogous to the previous study, we explored the number of memory accesses between checks (Figure 6.13), as well as the average size of the graphs (in terms of edge count) built

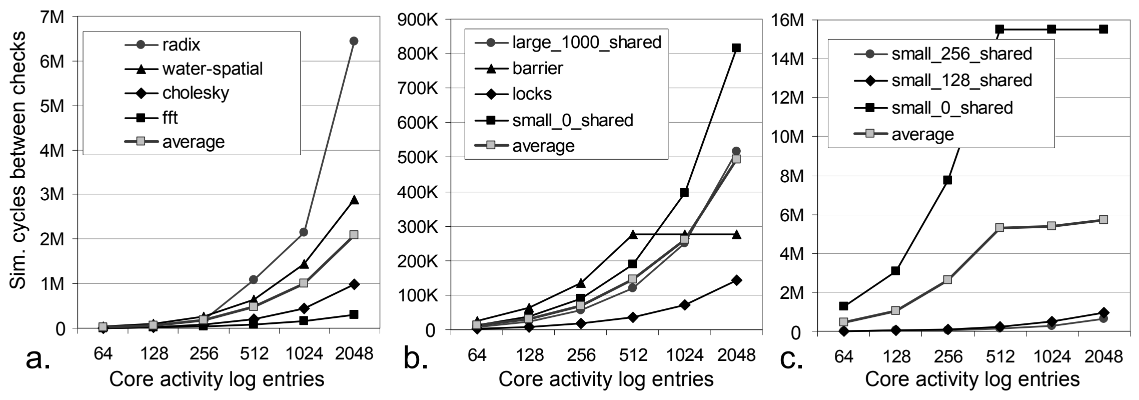


Figure 6.12: Number of simulation cycles between checks. Policy validations are invoked by Dacota when an activity logs fills or an access vector counter reaches the maximum value. **a.** Number of simulation cycles between Dacota checks for SPLASH2 benchmarks (4 benchmarks and the average of all 10 benchmarks is shown) **b.** Number of simulation cycles between Dacota checks for *barrier, locks* and larger random benchmarks with different degrees of sharing. **c.** Number of simulation cycles between Dacota checks for smaller random benchmarks with different degrees of sharing. With activity log length greater or equal 512 entries, the benchmark with no sharing fits entirely into a single epoch.

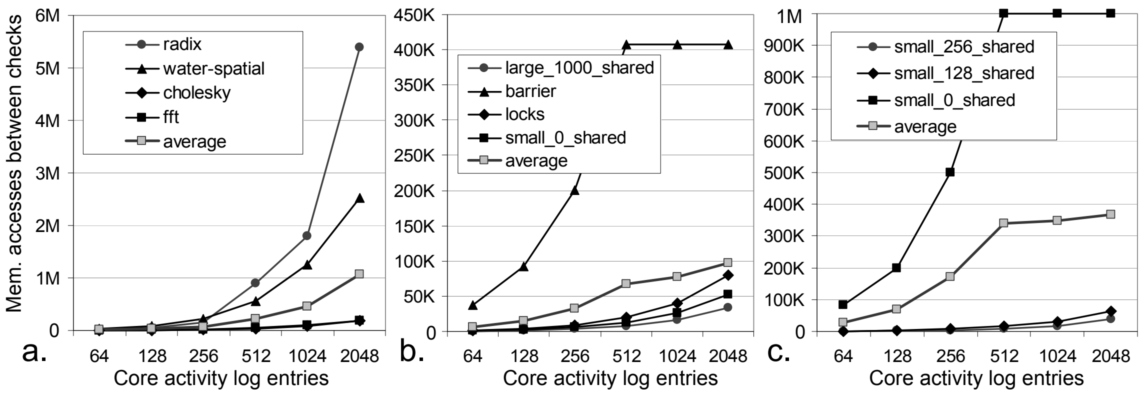


Figure 6.13: Number of memory accesses between checks. **a.** Number of memory accesses between Dacota checks for SPLASH2 benchmarks (4 benchmarks and the average of all 10 benchmarks is shown) **b.** Number of memory accesses between Dacota checks for *barrier, locks* and larger random benchmarks with different degrees of sharing. **c.** Number of memory accesses between Dacota checks for smaller random benchmarks with different degrees of sharing.

for each check (Figure 6.14). Our results demonstrate that these parameters increase in a similar fashion as the length of the activity log grows.

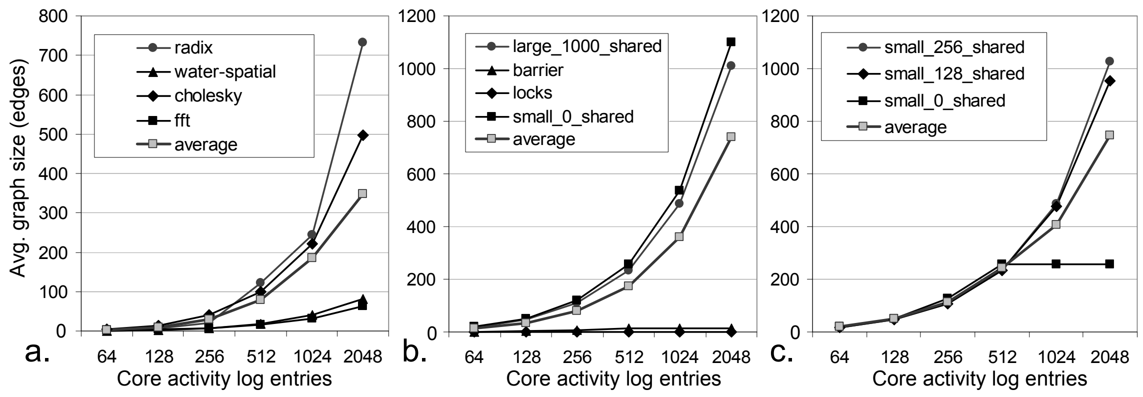


Figure 6.14: Average size of consistency graph. a. Average size of the consistency graph (edge count) for SPLASH2 benchmarks (4 benchmarks and the average of all 10 benchmarks is shown) **b.** Average size of the consistency graph for *barrier, locks* and larger random benchmarks with different degrees of sharing. **c.** Average size of the consistency graph for smaller random benchmarks with different degrees of sharing.

6.5.4 Area Evaluation

To analyze the area impact of Dacota, we implemented the additional hardware required by our solution in Verilog HDL. The module included a block for updating the counter and the access vector, a state machine for activity log management and an index table for conversion between program order and performance order. The module was synthesized with Synopsys Design Compiler targeting a TSMC 90nm library. The area for a control module, one of which is added to each processor core in a system, is 5216μm^2. For comparison, an OpenSPARC T1 [46] chip occupies approximately 378mm^2. Placing a Dacota module on each of the 8 cores in this design results in an overhead of 0.01%. This low overhead is largely due to Dacota's reuse of existing hardware structures, such as cache storage.

6.6 Summary

This chapter presents Dacota, a novel solution for high-coverage, post-silicon validation of memory coherence and consistency in multi-core chips. When enabled by the verification team, Dacota stores sequencing information about issued memory operations, periodically aggregating this information to perform a software-based policy validation. The validation algorithm is implemented purely in software to minimize the area impact of our solution and executes on existing processor resources. Leveraging approximately 6 orders of magnitude speed advantage over pre-silicon simulation, Dacota's post-silicon approach is able to offer significantly higher coverage compared to pre-silicon approaches. The average performance overhead of our solution (compared to operation with error detection disabled) is 26% for SPLASH2 benchmarks. Moreover, through reuse of preexisting hardware resources of the multi-core processor, Dacota is able to incur an area penalty of less than 0.01% for a commercial design, such as the OpenSPARC T1 system. We found that Dacota is effective in detecting subtle memory consistency and cache coherence bugs, showing its promise as a solution to the problem of multi-core processor memory system validation. Furthermore, Dacota enables post-silicon debugging support, which constitutes invaluable information for the post-silicon validation team. Indeed, post-silicon diagnosis and debugging is one of the most time-consuming activities in the entire verification effort. When the product is ready for customer shipment all the Dacota logging and checking activity can be disabled, leaving the system completed unhampered by any performance degradation or memory limitation. The only remaining indication of Dacota is in the inactive checker hardware components, which, as we mentioned earlier, occupy a tiny fraction of the system area.

CHAPTER VII

Hardware Patching of Multi-Cores

Ensuring correctness of execution in complex multi-core processor systems deployed in the field remains to this day an extremely challenging task. The major part of this effort is concentrated on the design's verification, where a range of pre- and post-silicon techniques are used to guarantee that devices behave precisely as stated in the specification. Unfortunately, the performance of even state-of-the-art validation tools lags behind the exponentially growing complexity of multi-core designs. Therefore, subtle bugs still slip into released components, causing incorrect computational results and possibly compromising the safety and security of the end-user systems.

This chapter of the thesis presents Caspar (CAche Subsystem PAtching and Repair) – a novel patching-based runtime validation solution designed specifically for multi-core processors. Caspar relies on a checkpointing system to take snapshots of the state of processor's cores and caches. In addition, the technique incorporates on-die cache-event detectors to identify erroneous situations. Each detector is programmed at system startup with *error-condition patterns* that are compared to the state of the cache line each time a transition occurs in a local or shared cache. The patterns themselves are created by the manufacturer's support team and are distributed to end-customers when new errors are discovered and debugged. When a memory-related error occurs in a system augmented with Caspar, a detector triggers a recovery and a bypass sequence to avoid it. The recovery mechanism itself relies on a reduced-complexity operation mode of the system, where only one memory access is performed at a time globally in the entire multi-core. This style of execution ensures that there are no interactions between processor cores, therefore, all coherence race conditions are eliminated and consistency can be guaranteed. Finally, after running the recovery and bypass for a pre-defined period of time, the system resumes regular operation.

One of the key advantages of Caspar is its small performance impact in the absence of bugs. On the other hand, when the processor contains errors, the performance of the system depends on the frequency of error occurrence. Moreover, the event detectors in Caspar occupy less than 0.01% of the die area (when deployed in an OpenSPARC T1 architecture), and the majority of the overhead is due to the storage space for system checkpointing. Note that, checkpointing can be leveraged for purposes beyond error recovery (*e.g.* post-silicon debugging), potentially reducing the cost of implementing Caspar.

7.1 Caspar Overview

In this section we overview the structure and the operation of Caspar, when implemented in a typical multi-core processor system, similar to the one in Figure 7.1. As shown in the figure, the CPU usually consists of multiple processing elements (cores), each connected to a local L1 cache. Controllers manage the caches, satisfying local requests, as well as requests from other cores received through on-chip interconnect. In addition to the local data storage, the cores have access to a larger second-level cache. To deploy Caspar in such a system we require the addition of a few hardware modules to the baseline system (hashed blocks in the figure). The majority of the area overhead in this case is due to system-level recovery, which requires storage space as we implemented it through checkpointing. In Caspar we create a checkpoint of each core's state by logging the values of its architectural registers. For caches we implement a copy-on-write policy and record state and value for a line upon change. Note that checkpoints in Caspar are stored locally with each core/cache array, eliminating the need to transfer logged snapshots over the interconnect. In addition to checkpoint storage, Caspar augments each cache with a programmable *cache-event detector* – a specialized circuit that compares transitions experienced by individual lines with a pre-defined pattern, thereby identifying memory subsystem errors. If a cache transition matches a detector pattern, Caspar alerts the cache controller, which, in its turn, signals the second level cache to stop program execution and initiate recovery.

The timeline for Caspar operation is shown in Figure 7.2. We assume that at system startup Caspar event detectors are loaded with appropriate patterns of erroneous events. As illustrated, program execution with Caspar is divided into *checkpointing epochs* and, at the end of each one, the system is synchronized and its state is recorded. *Synchronization* in our framework is initiated by one of the cache controllers (L1 or L2) and is required for completion of all cache transactions in flight. During the synchronization phase no new memory accesses are allowed to enter the system, thus by the end of this stage all cache lines reach stable protocol states. At this point, a distributed checkpoint is taken and execution may resume, initiating the next epoch. In the second epoch, shown in the figure, one of the detectors identifies a cache transition corresponding to a known coher-

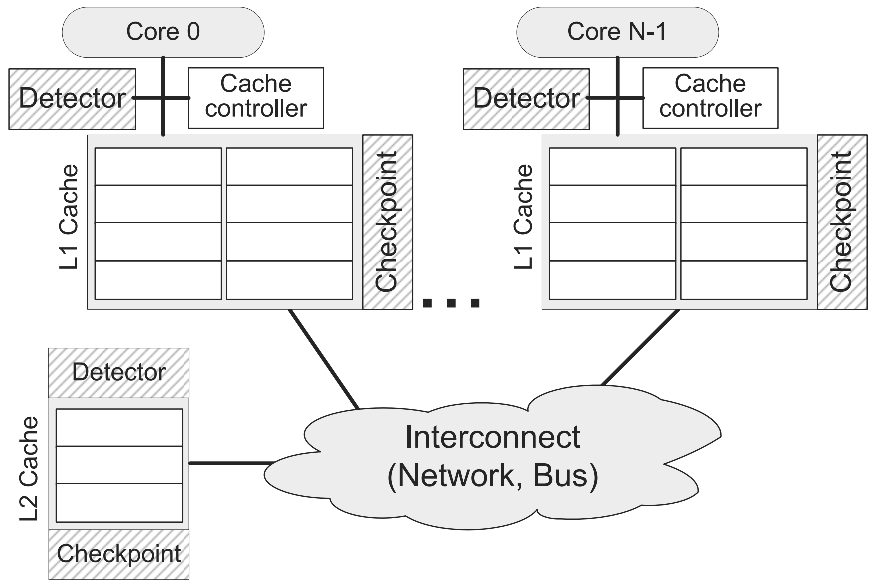


Figure 7.1: Architecture of a multi-core system augmented with Caspar hardware. Multiple processing elements (cores), each having a private L1 cache, are connected through on-chip interconnect to each other and to an L2 cache. The additional hardware required to implement Caspar is shown in hashed blocks and includes checkpoint storage space and cache-event detectors.

ence issue and triggers the recovery protocol orchestrated by the L2 cache controller. To recover, Caspar stops program execution and restores the last checkpoint, returning the system to a known-correct state before the occurrence of the error. The L2 cache then starts polling individual cores, allowing only one memory access at a time over the entire system. This ensures that no race conditions exist in the accesses to individual cache lines, thereby eliminating the possibility of coherence errors. In addition, this mode of execution enforces strict ordering of memory accesses, thus bypassing memory consistency issues. When a pre-determined number of accesses complete, Caspar stores a new checkpoint and re-enables the cores to continue regular program execution at full performance. Note that after completion of the last memory operation in bypass, the system is again in a stable state, since during recovery only one memory operation was executed at a time.

As discussed above, Caspar can be decomposed into three major components: the *checkpointing system* that periodically records the system's state, the *event detectors*, which identify bugs, and the *recovery and bypass mechanism* that allows Caspar to restore valid state and circumvent the error. In the rest of the section we will discuss each of these components and detail their implementation in Caspar.

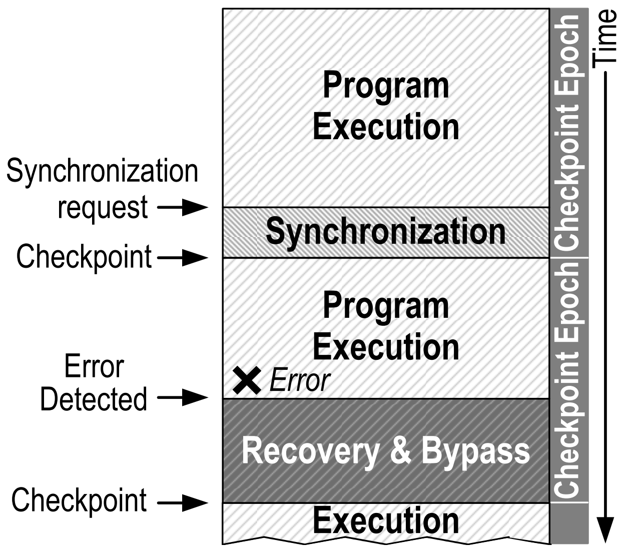


Figure 7.2: Timeline of Caspar operation. Cache-event detectors are initially loaded with patterns of cache transitions corresponding to errors. In the example no errors occur during the first epoch, which completes with system synchronization and checkpointing. However, during the second epoch an error occurs and is detected, triggering the recovery mechanism. The previously taken checkpoint is restored and then the system attempts to bypass the error by forcing strict ordering of accesses. When forward progress is made a new checkpoint is taken and the next epoch commences.

7.1.1 Checkpointing

Checkpointing in Caspar is designed to allow fast and efficient system recovery if an erroneous event is detected. As mentioned above, the storage for checkpoints is distributed throughout the system to reside locally with the module whose state it records. The implementation of the core checkpoints is straightforward, since in pipelines only values of architectural registers must be recorded. This includes, in particular, the register file, the program counter and machine status registers. In Caspar we implement the core checkpoint storage as shadow registers that store or overwrite the architectural state when a checkpoint is taken or when a restore operation is performed, respectively. Caches and memories, on the other hand, are too large to be checkpointed in such a way, so for them Caspar adopts a copy-on-write policy, recording the previous value or the coherence state of a cache line only when it changes for the first time after the beginning of a new epoch. To this end we augment each cache line with two *modification bits* indicating the modification of a line's data and state during the current epoch. We also create a log, residing next to the cache and organized as a hardware queue that records the addresses, state and data of altered lines. The queue and the modification bits are accessed in parallel with the main cache operation and are cleared when the checkpoint is taken. Then, if a line's state is altered during an epoch, we set the state modification bit in the cache and append the line's address and previous state to the log queue. It is important to note that if only the state is modified (for example from "Exclusive" to "Shared") the line's data is not copied to the log, allowing for better utilization of the queue storage space. On the other hand, if both the data and the state are modified during an epoch, Caspar logs them both, together with the line's address. If the block is subsequently altered again, the new change will not be logged, because the modification bits in the cache are already set. To restore the checkpoint, Caspar suspends the core's operation and performs the following two tasks in parallel. First, the core's architectural state is restored from the shadow registers. Simultaneously, Caspar traverses the log queue in reverse order, restoring the state and/or data of the lines from the log and resetting the modification bits. When the traversal is completed, the information in the cache is fully recovered and the log queue can be cleared.

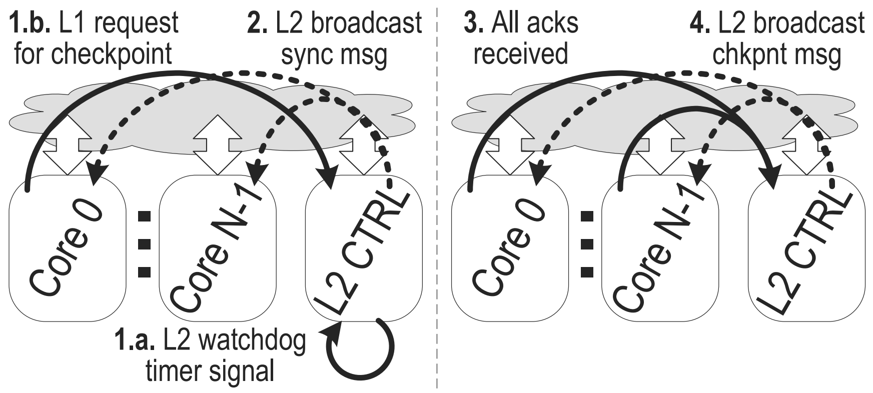

1.b. L1 request for checkpoint
2. L2 broadcast sync msg
3. All acks received
4. L2 broadcast chkpnt msg
Core 0
Core N-1
L2 CTRL
Core 0
Core N-1
L2 CTRL
1.a. L2 watchdog timer signal

Figure 7.3: Synchronization for checkpointing in Caspar. Synchronization requests are sent either by the watchdog timer in the L2 controller (1.a) or a local cache controller that exhausted its log space (1.b). Following the request, the L2 controller broadcasts a synchronization message to all cores (2), who stop issuing new memory accesses and acknowledge completion of all pending operations (3). As the final step, L2 sends the checkpoint message causing architectural state of the cores and local caches to be saved.

Although cache checkpoints in the system are distributed, their recording must be synchronized, ensuring a consistent system state after recovery. In designing our solution we decided to avoid special physical clocks, required by approaches such as SafetyNet [67], to synchronize the cores for checkpointing. Such clock networks would require significant routing effort to be implemented in a large multi-core CPU, and incur additional power and area overhead. Instead, for synchronization of the checkpointing process, Caspar relies on a four-step procedure shown in Figure 7.3. The synchronization is invoked by either a signal from the watchdog timer in the second-level cache (1.a in the figure) or a message from a cache controller that exhausts its logging resources (1.b). Subsequently, the second-level cache controller broadcasts an explicit synchronization message (2) stopping program execution in all cores. Once all cores have acknowledged that all pending operations have completed, the L2 broadcasts a checkpoint message and the state of the chip is recorded as discussed above.

7.1.2 Detection

Caspar is designed specifically for patching errors in the memory subsystem by comparing events (cache state machine transitions) to pre-loaded patterns describing known bugs. We implement this feature with event detectors – small programmable hardware blocks residing at each of the system's caches. Since each coherence transition of a cache line is uniquely defined by the line's state and the input trigger, the error patterns contained in the detectors consist of two fields, matching the state and the trigger, respectively. Each field is stored as a bit vector, where a 1 in a particular position matches a certain trigger or a state. Note that multiple bits in a pattern can be set to encode combinations of erroneous cache events.

The diagram of the hardware event detector is shown in Figure 7.4. During every coherence transition the line's actual state and trigger are passed to one-hot encoders, which create bit-vectors describing the event. The vectors are then separately compared with stored patterns on a bit-by-bit basis. The error detection signal, however, is asserted only if the match occurs in both fields. Caspar event detectors can also be expanded to contain several entries, each identical to the one in the figure. At the cost of additional area this feature can increase the power of our approach by allowing multiple error matchings to be performed in parallel.

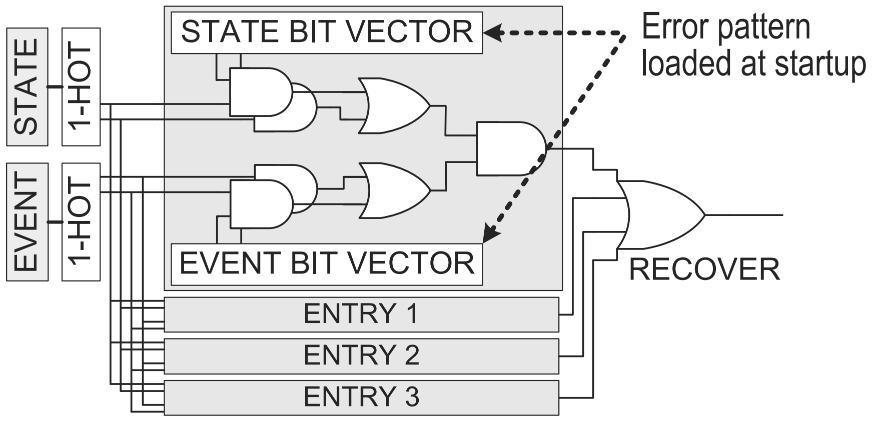


Figure 7.4: Structure of Caspar hardware event detector. Bug patterns are divided into a state and trigger fields and stored as bit-vectors, where a set bit indicates an erroneous state or transition trigger. During each cache access, the detector encodes the current event as two one-hot vectors and compares them with the stored pattern, asserting an error detection signal only if both match.

We illustrate Caspar's detection in the following example. Assume that a core is performing a store and the line is in intermediate state "SM" (half-way between "Shared" and “Modified”) when a store (invalidation request) arrives from another core. Due to an incorrectly implemented cache state machine this race of stores would result in a deadlock or coherence violation in a non-Caspar system. However, by programming Caspar with a pattern describing this state and trigger, we enable the detection and avoidance of this error at runtime. Note that such errors are associated with state machine transitions, and occur regardless of the address of the line or its prior history of transitions. Therefore, address-specific detectors are unnecessary, simplifying Caspar hardware without the loss of efficiency.

In addition to coherence errors, Caspar framework enables the detection of more complex and subtle issues related to memory consistency. As it was demonstrated in [57], consistency can be ensured if three system-wide invariants are guaranteed: i) uniprocessor ordering, ii) absence of illegal re-orderings and iii) cache coherence. As discussed above, event detectors in Caspar recognize dangerous transitions in the L1 and L2 caches, thus ensuring that the third invariant is upheld. The checkers for the former two properties, on the other hand, can be implemented in efficient small non-programmable logic blocks that can signal errors similarly to event detectors.

7.1.3 Recovery and Bypass

Once an error in the program execution is detected, Caspar initiates a system-wide mechanism to recover from the bug and circumvent it. To return the system to a know-correct state Caspar forces all cores, as well as the local and shared caches, to restore the last checkpoint. After rollback, the system attempts to bypass the error to ensure forward progress. Since errors in the memory subsystem are triggered by interactions of simultaneous accesses, we designed an efficient bypass system to eliminate such race conditions, at a cost of reduced performance. During this phase of CPU operation, we only allow one outstanding memory access in the system at a time, reconfiguring the L2 cache controller to poll cores directly for requests. Note that, in this case, coherence transitions in the system are never interrupted in intermediate states and cache operation is essentially simplified to the level of the underlying coherence protocol. This allows designers to validate the correctness of the bypass operation using powerful formal techniques. Moreover, since the L2 explicitly orders the accesses from cores, memory consistency can be ensured while in bypass mode.

After a pre-programmed number of accesses complete following a recovery, the L2 controller broadcasts a checkpointing request, so the stable state reached in bypass is saved, and normal execution resumes. Note that many more accesses are usually performed during an epoch’s normal execution than during bypass, therefore, it is possible that the offending sequence of instructions is executed and triggers an error again. However, in this case the system rolls back to the checkpoint reached after the previous recovery, thus guaranteeing forward progress, so the error is eventually circumvented. Moreover, since the network and message buffers are drained during recovery, the interaction of accesses after bypass is often different than the one that triggered the recovery originally. Therefore, it is possible that when normal operation is resumed after bypass, the execution of the same sequence does not produce an erroneous event again.

7.1.4 Post-silicon debugging with Caspar

In addition to providing runtime protection from subtle cache coherence and memory consistency bugs, Caspar can be used effectively in a variety of post-silicon debugging and software profiling frameworks. To this end, the checkpointing and the recovery capabilities are disabled, allowing the system to run at full speed. In addition, the detectors are reconfigured to count the number of occurrences of each event, or issue a system trap on a match. The patterns loaded in detectors in this case do not describe erroneous events, but rather transitions that are of particular interest for the designer or the validation engineer.

7.2 Experimental Evaluation

In this section we first introduce the experimental platform used for Caspar's evaluation, and then present detailed analyses of bug-finding ability and overheads of our solution. We investigate both the area and the performance overheads due to checkpointing, as well as the recovery penalty for thirteen complex bugs inserted into our baseline design. Finally, we analyze the area impact of Caspar's event detectors.

7.2.1 Experimental Platform

We simulated operation of Caspar in a 16-node multi-core processor using the Wisconsin Multifacet GEMS architectural simulator [53]. The local L1 caches were 64KB each, while the shared L2 cache had a size of 4MB. The cores were interconnected with a mesh network and used MOESI directory protocol for coherence. Caspar was implemented inside the GEMS memory tester module (Ruby) as a collection of functions to manage the event detectors and initiate checkpointing, recovery and bypass procedures. Event detectors in our studies were placed in the L1 and L2 caches, as well as into the directory controller on the chip.

Caspar was tested using a set of twenty benchmarks that included ten traces of SPLASH2 benchmarks [81] and randomly generated stimuli. The real-life benchmark traces were each 10M instructions long, while the random benchmarks contained 1M memory accesses and varied in degree of data sharing and total address space used. The final two benchmarks were created using Ruby's built in generator in "barrier" and "locks" modes.

7.2.2 Checkpointing Overheads

In our first study we analyzed the overhead of Caspar's checkpointing process. Recall that, to take a checkpoint, the system must synchronize and execution of some memory accesses must be delayed. This results in a slowdown of the system, which is inversely proportional to the length of an epoch. In other words, longer epochs result in synchronization algorithm being invoked less often, lowering the performance overhead. On the other hand, due to the copy-on-write policy for cache checkpointing, longer epochs can only be achieved with larger log size, which imposes a higher area overhead. We investigate this tradeoff in Figure 7.5, where we plot the slowdown and the area penalty for a range of epoch lengths. The area overhead is calculated as a percentage of cache lines that were modified during Ruby simulation and thus need to be logged. Each data point is computed as an average for the twenty benchmarks described above. As the figure demonstrates, longer epochs lead to better performance, however, it is important to remember that in the presence of errors this will cause more time to be spent in bypass and thus will incur greater slowdowns. For the following experiments we set the length of the checkpointing epoch to 8000 cycles, which translates to 2.4% performance and 5.8% area overhead.

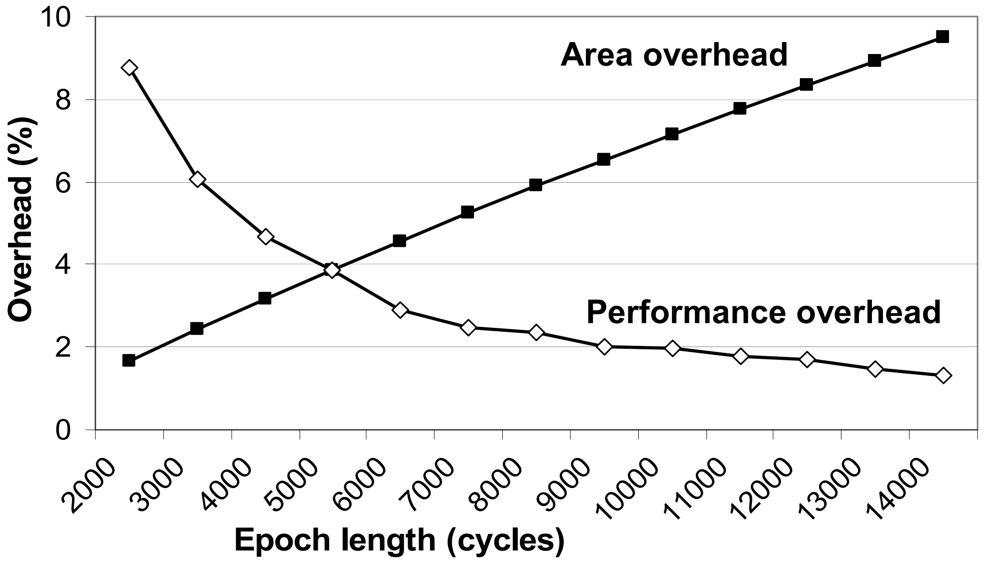


Figure 7.5: Overhead of Caspar checkpointing. Percent slowdown and area overhead (percent of the logged cache lines) required to sustain checkpointing over varying epoch lengths.

Table 7.1: Design errors for Caspar resiliency analysis study. Names, descriptions and occurrence frequencies (instances/1M cycles) of design errors in the experiment.

Bug name	Description of the error	Freq.
L1 Store race	L1 transitions from S to M and another cache issues a store	0.32
L1 WBack+Load	L1 evicting dirty data and another cache issues a load	0.72
L1 Own. Evict+Load	L1 owner evicting dirty data and another cache issues a load	0.04
L2 Block+Unblock	L2 blocked, waiting for ack and requester issues unblock	34.3
L2 Block+Load	L2 blocked, waiting on directory and L1 issues a load	0.03
L2 Load race	L2 satisfying load and another load arrives	0.12
L2 WB	L2 satisfying WB and dirty data arrives	2.72
Dir Block+Unblock	Directory blocked on a load and L2 issues unblock	35.1
Dir Clean WB mem	Directory blocked on WB and clean WB forwarded to mem	12.3
Dir Dirty WB mem	Directory blocked on WB and dirty WB forwarded to mem	21.7
Dir Clean WB L2	Directory blocked on WB and clean WB forwarded to L2	0.04
Dir Clean WB L2	Directory blocked on WB and dirty WB forwarded to L2	0.09
Dir Store+Unblock	Directory blocked on a store and L2 issues unblock	34.2

7.2.3 Error Resiliency Analysis

In our second study, we inserted thirteen bugs listed in Table 7.2.3 into various modules of the multi-core processor under test. The errors varied in frequency of occurrence from rare (0.03 instances per 1M cycles) to rather frequent (35.1 instances per 1M cycles). Each bug was associated with a particular state machine transition and described uniquely by the error patterns loaded by Caspar. Overall, these errors represent complex corner cases of operation of the L1, L2 and directory, when multiple accesses race for resources or an unexpected request arrives when the system is in an intermediate state. Without Caspar enabled, these errors could lead to cache state corruption or deadlock, however, when patterns identifying the bugs were loaded into the event detectors, all simulations completed correctly.

7.2.4 Caspar Recovery Performance

In addition to the resiliency analysis we investigated the overhead associated with different bugs. In the study we fixed the epoch length to 8000 cycles and bypass length to 2000 instructions, and recorded the fraction of the execution time that the system spent in each of the four phases: normal operation, synchronization for checkpoint, recovery and bypass. The results of this study are shown in Figure 7.6, where we also plot the average of the thirteen bugs and provide the bug occurrence rates for reference. As the experiment demonstrates, for more frequent bugs a significant portion of execution is spent in recovery and bypass, while for rare errors this fraction is negligible. Note that all errors were identified by Caspar precisely, since they were associated with unique transitions of the state machines, therefore no false-positive and false-negative detections occurred. In this study we observed that sometimes several detections were required to bypass a single bug instance. This was the case when the error occurred later in the epoch and a 2000 instruction bypass was insufficient to circumvent the problem after the first recovery. In addition, we observed cases where several potential errors, clustered together in program execution, were circumvented in one pass. Therefore, longer bypass sequences may be beneficial for bugs that are frequent or tend to occur in groups.

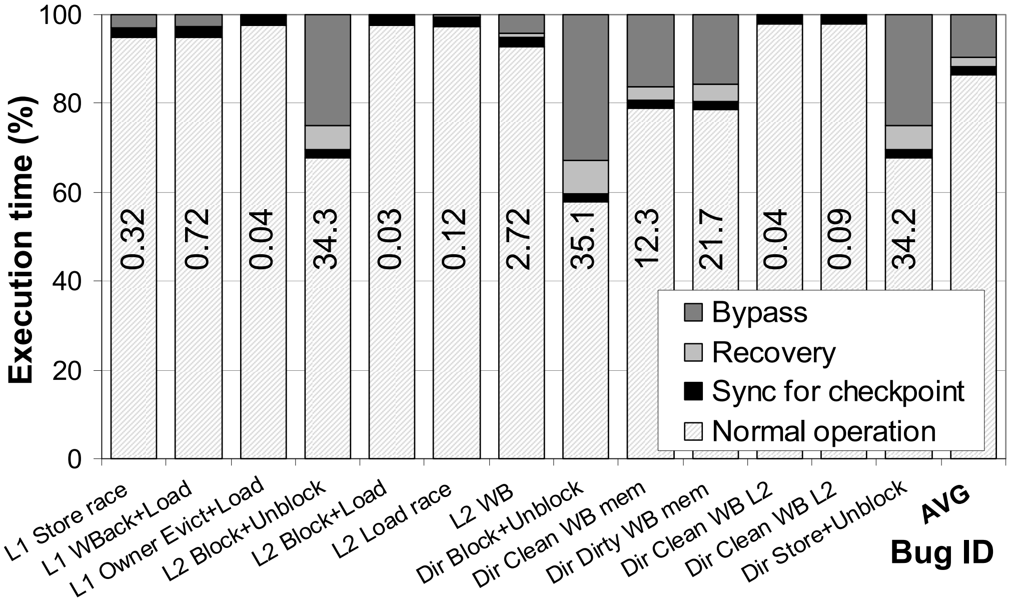


Figure 7.6: Execution time breakdown. Percentage of time the system spends in the four execution phases (normal operation, checkpointing, recovery and bypass) for each of the thirteen errors. We also plot the average of all bugs and provide frequency of occurrence (instances/1M cycles) for each error.

To analyze the impact of the bypass sequence length on the system's performance we conducted a further study on five errors with very different frequencies of occurrence. In this experiment the bypass length varied from 500 to 5000 instructions and the overall performance slowdown of the system was recorded (see Figure 7.7). As you can see, a longer bypass was beneficial for frequent errors, primarily because it guaranteed that the error would be circumvented on the first attempt and there was a higher chance that multiple errors would be covered by a single bypass. On the other hand, we noticed the opposite trend for rare bugs (*L1 WBack+Load* and *L2 Block+Load*). In these cases errors instances were significantly further from each other, thus a longer bypass did not reduce the number of recoveries. Moreover, the amount of time spent operating at lower performance in bypass increased, causing an overall slowdown. From this study we conclude that Caspar's fixed-length bypass strategy is not optimal for all bugs. Indeed, the system could be improved to load the number of instructions to run in bypass together with the bug pattern, or calculate the bypass length dynamically based on where the error was detected in the current epoch.

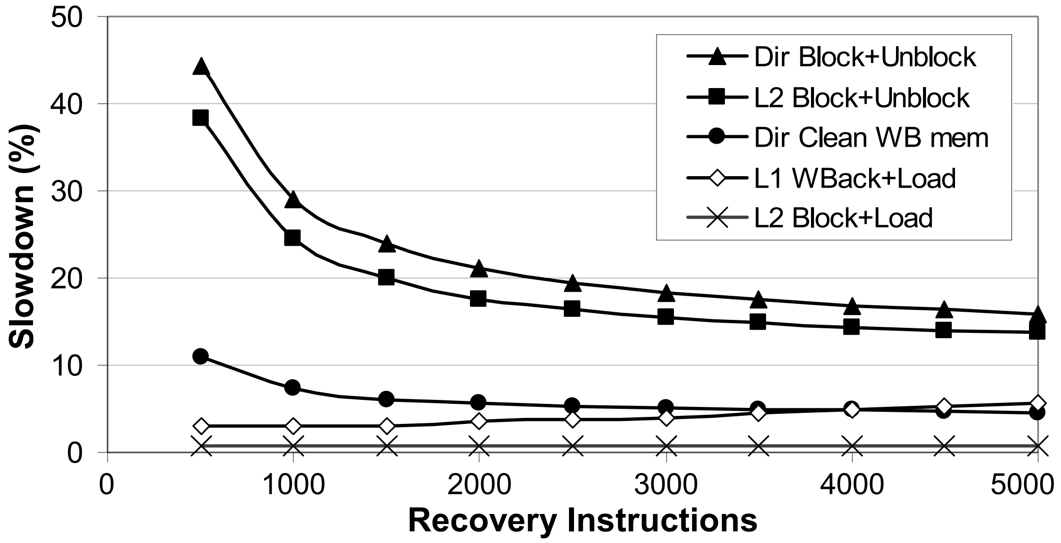


Figure 7.7: Performance vs. bypass length. For frequently occurring errors, a longer bypass is more beneficial, for it allows multiple bug instances to be covered together. For rare bugs, on the other hand, a longer bypass results in larger penalties due to longer execution at lower performance.

7.2.5 Event Detector Area Overhead

Finally, we investigated the area overhead of Caspar event detectors. To this end we implemented the detectors for the L1 and L2 caches in Verilog HDL and synthesized with in TSMC 90nm technology. The area of the L1 cache detector in this case was 984.2μm^2, while the additional hardware for the second-level cache occupied 2422μm^2. In comparison, an OpenSPARC T1 chip, which has eight cores with private caches and a shared 4-bank L2 cache, has an approximate area of 378mm^2 [46]. Thus, placing event detectors in each private cache and in each L2 bank of this chip incurs an area overhead of 0.005%, a negligible penalty, compared to the area of the checkpointing storage analyzed in Section 7.2.2.

7.3 Summary

In this chapter we have presented Caspar – an effective solution for in-the-field patching and repair of the memory sub-system in modern multi-core processors. Caspar relies on periodic system checkpointing and hardware event detectors programmed with descriptions of known erroneous transitions of the system's state machines. When an error is detected at runtime, Caspar stops the execution, recovers the correct state, and attempts to bypass the bug by enforcing race-free operation of the caches. Our experimental results demonstrate that Caspar's checkpointing scheme incurs little performance and area penalty (2.4% and 5.8%, respectively, for the target system in our experiments). Thus, with no errors present in the design, the impact of our solution is kept quite small, especially if the checkpointing is amortized by uses beyond Caspar. When errors do occur, recovery can potentially become expensive; however, we found experimentally that with low error frequencies (which is expected given that the system has undergone heavy design-time verification) the overall performance penalty is also small. These features allow Caspar to work as an effective patching solution, protecting even the most complex of today's designs. Finally, in addition to runtime verification, we suggested other uses of Caspar, including software and hardware profiling, as well as post-silicon debugging of modern multi-cores.

CHAPTER VIII

Conclusions and Future Work

This thesis has presented a novel and comprehensive framework to address the issues of verification of today's microprocessors. An important aspect of this work is that it is designed to fit directly into traditional industrial verification flows, incur minimal performance and die area overhead, and provide fast and high-quality validation with minimal involvement of engineering team. Moreover, the solutions detailed here are designed specifically for multi-core microprocessors, an increasingly popular processor architecture today. The framework we outlined consists of several novel and powerful techniques that can be deployed across all stages of the verification process: from verification of submodules and individual blocks, to verification of multi-core chips and on-die interconnect. Acknowledging that the verification flow may not be exhaustive (due to time-to-market pressure or insufficient resources), this work also details several efficient solutions to combat bugs that do escape into the final silicon product.

Today, the majority of processors that appear in end-user systems are multi-core architecture. These architectures divide the chip into two types of blocks: computational cores that execute individual instructions, and a multi-core interconnect system, which is typically implemented as a shared memory system. Cores in such designs feature complex control logic that governs the interaction of multiple instructions in the processor's pipeline. As a result, the control logic is the hardest part to verify of the processor core and frequently errors in modern processors occur precisely in these blocks. On the other hand, the shared memory communication system is used to propagate data between computing cores, and it features complex coherence protocols and consistency models that ensure proper execution of software programs. Therefore, we developed two groups of solutions to attack both of these two verify diverse challenges: solutions that verify individual cores and ones that address multi-core communication at all levels of a processor's life-cycle, from pre-silicon, to post-silicon, to runtime. Solutions for the verification of individual cores include:

1. **StressTest**: A closed-feedback constrained-random generator for testing individual

design blocks and single core processors. StressTest is a highly portable and scalable solution that uses dynamically adjusted Markov models to learn dynamically about quality of individual testing sequences and improve the subsequently generated tests. StressTest can be further extended with hierarchical Markov models to efficiently test components with multiple parallel interfaces *e.g.,* on-chip routers and switches. Experimental results demonstrate that StressTest can find significantly more errors in processor cores faster than traditional constrained-random approaches.

2. **Reversi**: A post-silicon technique that dramatically increases the speed of functional validation with constrained-random tests, compared to traditional post-silicon methodologies. This solution creates tests with known final correct state, eliminating the need for costly simulation and unlocking the full potential of post-silicon validation. Our evaluation shows that Reversi is capable of speeding up the post-silicon validation process by twenty times and can find more bugs than a traditional post-silicon verification flow.
3. **FRCL**: Field-repairable control logic is a patching technique, that allows to correct escaped bugs in components deployed in the field, by means of a reliable safe mode and a programmable matcher that is loaded with descriptions of known bugs at start up. The technique can be further enhanced with a **Semantic Guardians**, which ensure correctness of execution even in presence of unknown hardware bugs. Semantic guardians are created based on the information about verified and un-verified control states of a microprocessor, and, similarly to FRCL, use safe mode to bypass un-verified (and thus potentially buggy) scenarios. Both, FRCL and semantic guardian approach, occupy a tiny fraction of a processor die (2% and 3.5% respectively in our experiments), and incur no performance penalty during error-free operation of the chip. In presence of bugs, the overall slowdown of the processor is dependent on the error frequency, however, the correctness of execution is maintained by our approach at all times.

To target the communication sub-system in multi-core designs we developed a set of specialized solutions, again encompassing all levels of the verification effort:

1. **MCjammer** is a pre-silicon technique that verifies shared-memory communication and on-chip interconnect in multi-core designs. It features multiple distributed agents, each having a simplified view of the system's cache coherence protocol. The agents attempt to satisfy their own verification goals and/or help other agents to reach their objectives as well. Similarly to StressTest, which targets single cores, MCjammer uses a feedback loop to determine the quality and the coverage of pre-

viously generated tests and direct the agents towards unexplored system behaviors. We show experimentally, that the MCjammer approach significantly outperforms traditional constrained-random testing, and can verify orders of magnitude more transitions and states in the system finite-state machine within the same amount of time.

2. **Dacota** is an approach to post-silicon validation of memory consistency and cache coherence in shared memory multi-core processors. During the verification process Dacota reclaims a portion of the systems' caches to store side-band data, which is periodically aggregated and analyzed by a software algorithm running on the device-under-test itself. When validation completes, Dacota is disabled, restoring the system's cache configuration. As our analysis shows, Dacota can identify a wide range of coherence and consistency errors, while incurring minimal (less than 0.01%) area overhead for the end-user.
3. Finally, **Caspar** is a hardware patching solution for shared-memory communication protocols in multi-core chips. By adding small programmable matchers to the finite state-machines of the coherence protocol Caspar allows for escaped errors in the memory functional protocols to be detected and bypassed efficiently at runtime. Our experimental estimations show the area and performance overheads introduced by Caspar to be 5.8% and 2.4%, respectively.

The techniques presented in this work allow a designer to effectively attack the problem of processor verification at several levels, maximizing the advantages that can be gathered from each phase of verification. For example, in early design stages, the internals of the processor model has perfect visibility, thus StressTest and MCjammer use this information for high-quality feedback and improvement of verification coverage. Moreover, these procedures are automated and require minimal human involvement, which is vital, since today pre-silicon verification of complex microprocessors demand thousands of person-years. On the other hand, a hardware prototype of a processor can execute tests significantly faster than it is possible with pre-silicon simulation, however, much of the design's internal state becomes very hard to observe. Therefore, in our post-silicon techniques, namely Reversi and Dacota, we eliminate the costly feedback and allow verification to run at full hardware speed, which results in overall improvement of design coverage.

In addition to tackling the challenge of verification at individual stages, the set of techniques presented in our work allows designers to efficiently *balance* the verification effort throughout a processor's life-cycle and trade off verification costs for performance of the final product, while still maintaining correctness. In this case, pre-release verification activity (pre- and post-silicon) can be used to ensure just a basic level of functionality sufficient for most applications, while patching hardware monitors are generated automatically to ensure correctness of operation of the end product. Thus, the design timeline can be shortened and verification costs can be reduced. The verification can then be continued *after* the product release, so that newly developed patches fix escaped errors or augment on-die semantic guardians, reducing the time spent in safe mode, and thereby increasing processor performance. Therefore, this framework presents a novel paradigm for microprocessor design, yet it still fits smoothly into traditional design flows, making it attractive for adoption by processor design houses.

Our work also opens the door to several future research directions. First, the work demonstrates the importance of machine-learning techniques in pre-silicon verification, thereby suggesting that more sophisticated AI-based techniques could provide even better coverage and performance. This may include, for instance, solutions that combine learning from different design modules or, perhaps, even from different processor designs, and that may use this knowledge to produce higher-quality tests. In the post-silicon domain, on the other hand, further research is needed to develop methodologies whereby on-chip resources are re-used through reconfiguration to support the post-silicon validation phase. While there is a strong demand for integrating verification and testing resources directly onto chip hardware, yet this trend is still very limited today because design houses prioritize performance over correctness in allocating the silicon real estate on die. However, as the Dacota solution suggests, it is possible to devise solutions that fulfill both these contrasting demands. Finally, the runtime solutions that we presented here, assume that patches are delivered from the processor's design house to the end-user and installed on the system securely. Thus, it is crucial for new research works to provide effective solutions for such distribution, as well as mechanisms to prevent reverse-engineering of patches.

Altogether, the verification solutions developed in this work enables processor vendors to dramatically shorten their design timelines, while simultaneously lowering development costs and improving the quality and reliability of their products. We demonstrated that the growing challenge of verification of a modern multi-core processor cannot be tackled with ad-hoc, expedient measures, but must be addressed comprehensively. We hope to see some of these techniques adopted in the industry in the near future, but most of all, we hope that this thesis work could inspire new ways of thinking about verification and a new type of research in this arena, where verification is seen as a very practical challenge that can be addressed throughout the development of a system, beyond the search for formal proofs.

BIBLIOGRAPHY

[1] D. Abts, S. Scott, and D. J. Lilja. So many states, so little time: Verifying memory coherence in the cray x1. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 11.2, Washington, DC, USA, 2003. IEEE Computer Society.

[2] Accellera. *Property Specification Language Reference Manual, Rev 1.1*, June 2004. `http://www.eda.org/vfv/docs/PSL-v1.1.pdf`.

[3] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test of Computers*, 21(2):84–93, 2004.

[4] A. Adir and G. Shurek. Generating concurrent test-programs with collisions for multi-processor verification. In *HLDVT '02: Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop (HLDVT'02)*, pages 77–82, Washington, DC, USA, 2002. IEEE Computer Society.

[5] Advanced Micro Devices, Inc. *AMD Athlon (TM) Processor Model 10 Revision Guide*, Oct. 2003. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/27532.pdf`.

[6] Advanced Micro Devices, Inc. *Revision Guide for AMD Athlon(TM) 64 and AMD Opteron(TM) Processors*, Aug. 2005. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25759.pdf`.

[7] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Lavander, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification language. In *TACAS, Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Apr. 2002.

[8] T. M. Austin. DIVA: A dynamic approach to microprocessor verification. *Journal of Instruction-Level Parallelism*, 2000.

[9] H. Baker and C. Parker. High level language programs run ten times faster in microstore. In *Proceedings of the 13th annual workshop on Microprogramming*, pages 171–177, 1980.

[10] P. T. Barch, C. J. Ellingham, F. L. Wagner, and J. R. Larkin. *U.S. Patent no. 5923836: Testing integrated circuit designs on a computer simulation using modified serialized scan patterns*. Texas Instruments, Inc., Nov. 2006.

[11] B. Bentley. Validating the Intel Pentium 4 microprocessor. In *DAC, Proceedings of Design Automation Conference*, pages 224–228, 2001.

[12] B. Bentley and R. Gray. Validating the Intel Pentium 4 Microprocessor. *Intel Technology Journal*, pages 1–8, Feb. 2001.

[13] K. H. Bierman, D. R. Emberson, and L. T. Chen. *U.S. Patent no. 7133818: Method and apparatus for accelerated post-silicon testing and random number generation*. Sun Microsystems, Inc., Nov. 2006.

[14] The open source IA-32 emulation project, Sept. 2007. `http://bochs.sourceforge.net/`.

[15] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[16] E. B. Brett, D. P. Hunter, and S. L. Smith. Moving atom to windows nt for alpha. *Compaq DIGITAL Technical Journal*, 10(2), Jan. 1999.

[17] B. Brock and W. A. Hunt. Report on the Formal Specification and Partial Verification of the VIPER Microprocessor. In *Compass '91: 6th Annual Conference on Computer Assurance*, pages 91–98, Gaithersburg, Maryland, 1991. National Institute of Standards and Technology.

[18] D. Burger and T. Austin. The SimpleScalar toolset, version 3.0. `http://simplescalar.com`.

[19] D. V. Campenhout, T. Mudge, and J. P. Hayes. Collection and analysis of microprocessor design errors. *IEEE Design & Test*, 17(4):51–60, 2000.

[20] A. Carbine. *U.S. Patent no. 5253255: Scan mechanism for monitoring the state of internal signals of a VLSI microprocessor chip*. Intel Corporation, Nov. 1990.

[21] X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou. Reducing verification complexity of a multicore coherence protocol using assume/guarantee. In *Formal Methods in Computer Aided Design*, pages 81–88, 2006.

[22] B. Cohen, S. Venkataramanan, and A. Kumari. *Using PSL/Sugar for Formal and Dynamic Verification*. VhdlCohen Publishing, 2004.

[23] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, Aug. 1998.

[24] DDJ Microprocessor Center. `http://www.x86.org/`.

[25] A. DeOrio, A. Bauserman, and V. Bertacco. Post-silicon verification for cache coherence. In *ICCD, Proceedings of the International Conference on Computer Design*, Oct. 2008.

[26] A. DeOrio, I. Wagner, and V. Bertacco. Dacota: Post-silicon validation of the memory subsystem in multi-core designs. 2009. Submitted to HPCA, International Symposium on High-Performance Computer Architecture.

[27] E. A. Emerson and V. Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 247–262, 2003.

[28] S. Fine and A. Ziv. Coverage directed test generation for functional verification using bayesian networks. In *DAC, Proceedings of Design Automation Conference*, pages 286–281, June 2003.

[29] S. German. Formal design of cache memory protocols in IBM. *Formal Methods in System Design*, 22(2):133–141, 2003.

[30] M. D. Goddard and D. S. Christie. *U.S. Patent no. 5796974: Microcode patching apparatus and method*. Advanced Micro Devices, Inc., Nov. 1995.

[31] F. I. Haque, K. A. Khan, and J. Michelson. *The Art of Verification with Vera*. Verification Central, 2001.

[32] P. H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long. Smart simulation using collaborative formal and simulation engines. In *ICCAD '00: Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 120–126, Piscataway, NJ, USA, 2000. IEEE Press.

[33] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. Architecture validation for processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 404–413. ACM Press, 1995.

[34] Y. Hollander, M. Morley, and A. Noy. The *e* language: A fresh separation of concerns. In *Technology of Object-Oriented Languages and Systems*, volume TOOLS-38, pages 41–50, Mar. 2001.

[35] K. hui Chang, V. Bertacco, and I. Markov. Simulation-based bug trace minimization with bmc-based refinement. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 1045–1051, Nov. 2005.

[36] Intel Corporation. *Intel(R) StrongARM(R) SA-1100 Microprocessor Specification Update*, Feb. 2000.

[37] Intel Corporation. *Intel(R) Celeron(R) Processor Specification Update*, Sept. 2002. `http://developer.intel.com/design/celeron/specupdt/24374847.pdf`.

[38] Intel Corporation. *Intel(R) Pentium(R) II Processor Invalid Instruction Erratum Overview*, July 2002. `http://developer.intel.com/design/pentiumii/specupdt/24333749.pdf`.

[39] Intel Corporation. *Intel(R) Pentium(R) Processor Invalid Instruction Erratum Overview*, July 2004. `http://www.intel.com/support/processors/pentium/sb/cs-013151.htm`.

[40] Intel Corporation. *Intel(R) Pentium(R) III Processor Specification Update*, May 2005. `http://download.intel.com/design/PentiumIII/specupdt/24445355.pdf`.

[41] International Business Machines Corporation. *IBM PowerPC 750GX and 750GL RISC Microprocessor Errata Notice*, July 2005.

[42] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu. Checking cache-coherence protocols with TLA+. *Formal Methods in System Design*, 22(2):125–131, Mar. 2003.

[43] D. E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1968.

[44] D. Koncaliev. Bugs in the Intel Microprocessors. `http://www.cs.earlham.edu/~dusko/cs63/`.

[45] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1997.

[46] A. Leon, K. Tam, J. Shin, D. Weisner, and F. Schumacher. A power-efficient high-throughput 32-thread SPARC processor. *IEEE Journal of Solid-State Circuits*, 42(1):7–16, Jan. 2007.

[47] Y. Levhari. Verification of the PalmDSPCore using pseudo random techniques. Technical report, VeriSure Consulting, Ltd., 2002.

[48] T. Litt. Support for debugging in the Alpha 21364 microprocessor. In *International Test Conference*, pages 584–589, Oct. 2002.

[49] J. M. Ludden, W. Roesner, G. M. Heiling, J. R. Reysa, J. R. Jackson, B.-L. Chu, M. L. Behm, J. R. Baumgartner, R. D. Peterson, J. Abdulhafiz, W. E. Bucy, J. H. Klaus, D. J. Klema, T. N. Le, F. D. Lewis, P. E. Milling, L. A. McConville, B. S. Nelson, V. Paruthi, T. W. Pouarz, A. D. Romonosky, J. Stuecheli, K. D. Thompson, D. W. Victor, and B. Wile. Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems. *IBM Journal of Research and Development*, 46:53–76, Jan. 2002.

[50] The M5 simulator system, Nov. 2007. `http://www.m5sim.org`.

[51] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[52] N. Malik, S. Roberts, A. Pita, and R. Dobson. Automaton: An autonomous coverage-based multiprocessor system verification environment. In *RSP '97: Proceedings of the 8th International Workshop on Rapid System Prototyping (RSP '97) Shortening the Path from Specification to Prototype*, pages 168–172, Washington, DC, USA, 1997. IEEE Computer Society.

[53] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[54] E. J. McCluskey. Minimization of boolean functions. *Bell Systems Technical Journal*, 6(35):1417–1444, Nov. 1956.

[55] K. J. McGrath and J. K. Pickett. *U.S. Patent no. 6438664: Microcode patch device and method for patching microcode using match registers and patch routines*. Advanced Micro Devices, Inc., Oct. 1999.

[56] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. *IEEE Micro*, 28(1):52–59, 2008.

[57] A. Meixner and D. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. In *Proc. DSN*, pages 73–82, 2006.

[58] M. Melani, F. D'Ascoli, C. Marino, L. Fanucci, A. Giambastiani, A. Rochhi, M. D. Marinis, and A. Monterastelli. An integrated flow from pre-silicon simulation to post-silicon verification. In *Research in Microelectronics and Electronics 2006, Ph. D.*, pages 205–208, June 2006.

[59] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface, Second Edition*. Morgan Kaufmann, 1997.

[60] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. *Lecture Notes in Computer Science*, pages 82–97, 2001.

[61] /QIfdiv (Enable Pentium FDIV Fix). `http://msdn2.microsoft.com/en-us/library/ms856573.aspx`.

[62] H. Rotithor. Post-silicon validation methodology for microprocessors. *IEEE Design*

& Test of Computers, 17(4):77–88, Oct. 2000.

[63] A. Saha, N. Malik, B. O'Krafka, J. Lin, R. Raghavan, and U. Shamsi. A simulation-based approach to architectural verification of multiprocessor systems. In *Computers and Communications, 1995. Conference Proceedings of the 1995 IEEE Fourteenth Annual International Phoenix Conference on*, pages 34–37, Mar. 1995.

[64] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas. Patching processor design errors with programmable hardware. *IEEE Micro*, 27(1):12–25, 2007.

[65] J. Siek and L.-Q. Lee. BOOST graph library. `http://www.boost.org/doc/libs/release/libs/graph`.

[66] I. Silas, I. Frumkin, E. Hazan, E. Mor, and G. Zobin. System-level validation of the Intel Pentium M processor. *Intel Technology Journal*, 07:38–43, May 2003.

[67] D. Sorin, M. Martin, M. Hill, and D. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proc. ISCA*, pages 123–134, 2002.

[68] G. Spirakis. Opportunities and challenges in building silicon products in 65nm and beyond. In *Design and Test in Europe (DATE-2004)*, 2004.

[69] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, Apr. 2004.

[70] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, and K. Keutzer. A functional validation technique: Biased-random simulation guided by observability-based coverage. *ICCD, Proceedings of the International Conference on Computer Design*, pages 82–88, 2001.

[71] S. Taylor, M. Quinn, D. Brown, N. Dohm, S. Hildebrandt, J. Huggins, and C. Ramey. Functional verification of a multiple-issue, out-of-order, superscalar Alpha processor: The DEC Alpha 21264 microprocessor. In *DAC, Proceedings of Design Automation Conference*, pages 638–644, 1998.

[72] I. Wagner and V. Bertacco. Engineering trust with semantic guardians. In *DATE, Proceedings of Design, Automation and Test in Europe Conference*, pages 743–748, Apr. 2007.

[73] I. Wagner and V. Bertacco. MCjammer: Adaptive verification for multi-core designs. In *DATE, Proceedings of Design, Automation and Test in Europe Conference*, pages 670–675, May 2008.

[74] I. Wagner and V. Bertacco. Reversi:. In *ICCD, Proceedings of the International Conference on Computer Design*, 2008.

[75] I. Wagner and V. Bertacco. Caspar: Hardware patching for multi-core processors. 2009. Submitted to DATE, Design, Automation and Test in Europe.

[76] I. Wagner, V. Bertacco, and T. Austin. StressTest: An automatic approach to test generation via activity monitors. In *DAC, Proceedings of Design Automation Conference*, pages 783–788, 2005.

[77] I. Wagner, V. Bertacco, and T. Austin. Depth-driven verification of simultaneous interfaces. In *ASP-DAC, Proceedings of Asian-South Pacific Design Automation Conference*, pages 442 – 447, 2006.

[78] I. Wagner, V. Bertacco, and T. Austin. Shielding against design flaws with field

repairable control logic. In *DAC, Proceedings of Design Automation Conference*, pages 344–347, 2006.

[79] I. Wagner, V. Bertacco, and T. Austin. Microprocessor verification via feedback-adjusted markov models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(6):1126–1138, June 2007.

[80] I. Wagner, V. Bertacco, and T. Austin. Using field-repairable control logic to correct design errors in microprocessors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(2):380–393, Feb. 2008.

[81] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proc. ISCA*, pages 24–36, 1995.

[82] D. A. Wood, G. A. Gibson, and R. H. Katz. Verifying a multiprocessor cache controller using random test generation. *IEEE Des. Test*, 7(4):13–25, 1990.