Reining in the Functional Verification of Complex Processor Designs with Automation, Prioritization, and Approximation

by

Biruk Wendimagegn Mammo

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
2017

Doctoral Committee:
Professor Valeria M. Bertacco, Chair
Professor Todd M. Austin
Associate Professor Heath Hofmann
Professor Scott Mahlke

© Biruk Wendimagegn Mammo

All Rights Reserved
2017

To my family

Acknowledgments

I would first like to thank my advisor, Professor Valeria Bertacco, for her constant support and mentorship, which started even before I joined the University of Michigan. This thesis would not have been possible without her guidance and hard work. I would also like to thank my committee members, Professors Todd Austin, Heath Hofmann, and Scott Mahlke, for their support and valuable feedback.

I have benefited greatly from the excellent faculty at the University of Michigan. I am very fortunate to have worked closely with Doctor Mark Brehob, Professors Todd Austin, Scott Mahlke, and Tom Wenisch, who are all excellent teachers, mentors, and collaborators. I would like to give special thanks to Professor Austin for being a source of inspiration and good humor. Thank you for always making time to discuss ideas with me and to help me improve my papers and presentations.

I am fortunate to have been part of a dynamic group of graduate students. I would like to thank Andrew DeOrio, Joseph Greathouse, Jason Clemons, Andrea Pellegrini, and Debapriya Chatterjee for showing me the ropes and guiding me through life in graduate school. I am grateful for the innumerable discussions I have had with Ritesh Parikh and Rawan Abdel-Khalek that helped me make progress in my research. This work would not have been possible without the contributions of my collaborators: Doowon Lee, Harrison Davis, Yijun Hou, Milind Furia, Daya Khudia, Andrew DeOrio, and Debapriya Chatterjee. I would also like to thank the many friends and colleagues that provided an intellectually stimulating and enjoyable environment. This includes Aasheesh Kolli, Ankit Sethia, Gaurav Chada, Misiker Aga, Abraham Addisie, Zelalem Aweke, Shamik Ganguly, Ricardo Rodriguez, and many others I do not have space to mention here. I am especially grateful to my dear friends and colleagues Salessawi Ferede, William Arthur, and Animesh Jain, for supporting me through all the trials and tribulations of graduate school.

I would also like to express my gratitude to the people who worked behind the scenes to make this work possible. Thank you Laura Fink Coutcher, Kyle Banas, Don Winsor, and other DCO staff for keeping the computers running. I will never forget how Don once went beyond the call of duty to come to work on a Sunday evening to fix a problem so that I can meet a deadline. Thank you Dawn Freysinger, Denise Duprie, Karen Liska, Anne Rhoades, Lauri Johnson-Rafalski, Stephen Reger, Ashley Andreae, Tracie Straub, and Christine Boltz for your support and for being there when I needed help with administrative issues. I would also like to thank the National Science Foundation (NSF) and the Center for Future Computer Architecture Research (CFAR) for funding my research.

Finally, I would like to thank my family. I am indebted to my wife, Yodit Beyene, for her friendship and love during the challenging last years of graduate school. I would like to express my deepest gratitude to my parents and my siblings, for their support and encouragement. Even though they live half a world away, they have been with me in spirit through every moment of this journey.

Preface

Modern processor designs are among humankind's most complex creations. A processor design today is so complex that it is impossible to guarantee its complete correctness. Processor companies spend a significant amount of time and money to obtain what can only be described as *an acceptable risk of inaccuracy*. Add a highly competitive market with ever-shrinking time-to-market demands, and it is common for processors to be released with design defects that may have devastating consequences to consumers and companies. These challenges put undue pressure on the functional verification effort to ensure the correctness of a design, which today surpasses the effort to actually implement the design.

Today's designs cram several interacting components into small spaces. The functional verification burden is only expected to worsen as more and more custom designs with higher levels of integration emerge. We are witnessing the proliferation of custom computing systems, ranging from custom systems-on-chip (SoCs) for servers to custom nodes in an internet-of-things (IoT) environment. Emerging applications, such as machine learning, big data, and virtual reality, have massive computing demands that are going to drive even more complexity into processor designs. For these and other unforeseen design efforts to be successful, it is imperative that the cost of design and, ultimately, the time and resources spent on functional verification go down.

Simulation-based methods carry a bulk of the functional verification effort. In these methods, tests are executed on an implementation of the design while the design's behavior is monitored and checked for correctness. The simulation-based functional verification effort today is mostly a best-effort endeavor limited by design complexity and the amount of time available to execute tests. A typical simulation-based functional verification effort today is focused on executing tests to meet a set of verification goals defined at the beginning of the design project. Even though the goals are an incomplete set of design behaviors to target for verification, the verification process is practiced with an expectation of completeness: *engineers strive to accurately and completely meet an inherently incomplete set of goals.*

In this dissertation, we recognize an opportunity to reduce the cost of verification by

removing the expectation of completeness, not just at the beginning of a design project, but across all functional verification activities. To this end, we introduce a new perspective that we refer to as the automation, prioritization, and approximation (APA) approach. In this approach, we perceive functional verification as multiple endeavors that may sometimes be at odds with each other. These endeavors pertain to handling design complexity, developing mechanisms to detect functional errors, undertaking tasks to execute a verification plan, and managing engineer effort. Our APA approach gives us the means to strategically leverage incompleteness for these endeavors. We first prioritize certain aspects of an endeavor, and then automate activities pertaining to the prioritized aspects while exploiting approximations that trade-off accuracy for large gains in efficiency.

In this dissertation, we partition the functional verification effort into three main activities – planning & test generation, test execution & bug detection, and bug diagnosis – and present our solutions in the context of these activities. We present an automatic mechanism for prioritizing design behaviors while planning the verification of modern SoCs. This automatic prioritization is achieved by analyzing the execution of software packages being developed for the SoC. Our mechanism is also able to generate regression suites that run much faster than the original software packages. We also develop automatic bug detection mechanisms that unshackle high-performance validation platforms, allowing them to execute tests at high speeds while looking for high-priority bugs in the design. Finally, we develop an automatic solution for diagnosing bugs, which can shave off weeks of engineer effort.

Table of Contents

List of Figures

List of Tables

Abstract

Our quest for faster and efficient computing devices has led us to processor designs with enormous complexity. As a result, functional verification, which is the process of ascertaining the correctness of a processor design, takes up a lion's share of the time and cost spent on making processors. Unfortunately, functional verification is only a best-effort process that cannot completely guarantee the correctness of a design, often resulting in defective products that may have devastating consequences. Functional verification, as practiced today, is unable to cope with the complexity of current and future processor designs.

In this dissertation, we identify extensive automation as the essential step towards scalable functional verification of complex processor designs. Moreover, recognizing that a complete guarantee of design correctness is impossible, we argue for systematic prioritization and prudent approximation to realize fast and far-reaching functional verification solutions. We partition the functional verification effort into three major activities: planning and test generation, test execution and bug detection, and bug diagnosis. Employing a perspective we refer to as the automation, prioritization, and approximation (APA) approach, we develop solutions that tackle challenges across these three major activities.

In pursuit of efficient planning and test generation for modern systems-on-chips, we develop an automated process for identifying high-priority design aspects for verification. In addition, we enable the creation of compact test programs, which, in our experiments, were up to 11 times smaller than what would otherwise be available at the beginning of the verification effort. To tackle challenges in test execution and bug detection, we develop a group of solutions that enable the deployment of automatic and robust mechanisms for catching design flaws during high-speed functional verification. By trading accuracy for speed, these solutions allow us to unleash functional verification platforms that are over three orders of magnitude faster than traditional platforms, unearthing design flaws that are otherwise impossible to reach. Finally, we address challenges in bug diagnosis through a solution that fully automates the process of pinpointing flawed design components after detecting an error. Our solution, which identifies flawed design units with over 70% accuracy, eliminates weeks of diagnosis effort for every detected error.

Chapter 1
Introduction

Over the past six decades, we have witnessed exponential advances in semiconductor fabrication technology, as foretold by Moore's law [88]. These advances in turn have enabled the design of powerful processors residing in the computing devices that pervade our lives today. Our quest for fast and energy-efficient devices has led us to processor designs that are among humankind's most complex creations.

One of the major undertakings in a processor design project, known as functional verification, is the task of looking for functional errors (bugs) in the design and fixing them. In fact, functional verification takes up the lion's share of the time and resources of a processor design project today [46]. The primacy of functional verification is mainly due to the difficulty of adequately putting a complex design through its paces within the grueling schedules of today's competitive processor markets. The functional verification challenge is only expected to grow, driven by more competition and increasing complexity heralded by emerging applications, such as machine learning, big data, and virtual reality.

Despite the resources poured into functional verification, severe bugs still escape detection, only to surface after the processor has been manufactured and put in operation. Such bugs tarnish reputations and may incur massive financial damages. In 2007, a processor bug forced AMD to temporarily halt production and issue a workaround [108]. The workaround had the unwanted effect of reducing processor performance, damaging AMD's competitiveness and reputation. In 2011, Intel projected a total loss of $1 billion due to recalls from a bug in a chipset accompanying its latest processor [106]. More recently, a number of high-profile bugs were found in Intel's and AMD's processors, at least one of which required a new feature to be completely disabled [37].

Complex designs, short times-to-market, and escaped bugs are all functional verification burdens that are crippling processor design projects. The industry has not been able to scale functional verification techniques to cope with these challenges. We believe the main reason for this predicament is the expectation of completeness prevalent in the practice of functional verification, when in reality, the limited time and the enormous complexity merely allow a best-effort process that cannot completely guarantee the correctness of most aspects of a design. At the core of this dissertation is our recognition that fully embracing the incompleteness of functional verification provides valuable opportunities for improving both the quality and efficiency of verification.

In this dissertation, we regard functional verification as an interplay between multiple endeavors at odds with each other: surmounting design functional complexity, unearthing functional bugs, executing a verification plan, and allocating engineer effort. We introduce a novel way of looking at these endeavors through what we call the *automation, prioritization, and approximation (APA)* approach. Central to the APA approach is the development of new solutions and revisiting of existing ones to efficiently automate functional verification efforts. We systematically select entities to automate by prioritizing the more frequent, critical, and time-consuming aspects of a design's functionality, detection mechanisms for classes of bugs, and effort spent on performing verification tasks. In the quest for automation, we actively look for opportunities to approximate certain functionalities to achieve significant gains in efficiency with modest impacts on the design's quality of validation. The solutions we developed through our APA approach cut down the time required for functional verification while extending its reach to rarely occurring design behaviors that are extremely difficult to exercise and to corner-case bugs that are difficult to detect. Our solutions reduce the sizes of frequently executed testbench suites by up to 11 times, may eliminate several weeks of effort that engineers spend diagnosing each bug they find, and enable efficient bug detection methods so that powerful validation platforms can be effectively included within typical validation methodologies employed in companies today. We also enable the detection of important but complex bugs that would otherwise escape functional verification.

1.1 Design complexity and the verification challenge

Over the years, processor architects have been incorporating more and more features and operating modes into their designs. The prevailing processor core design trends today utilize complicated execution structures for out-of-order execution and advanced power management techniques. In addition, designers are incorporating more and more application-specific accelerators into their designs to squeeze out more performance under always tighter energy/power constraints. These accelerators reside on a chip alongside the processor cores, a memory subsystem, and peripheral components, all connected via an on-chip interconnect. Each of these individual components are complex on their own,

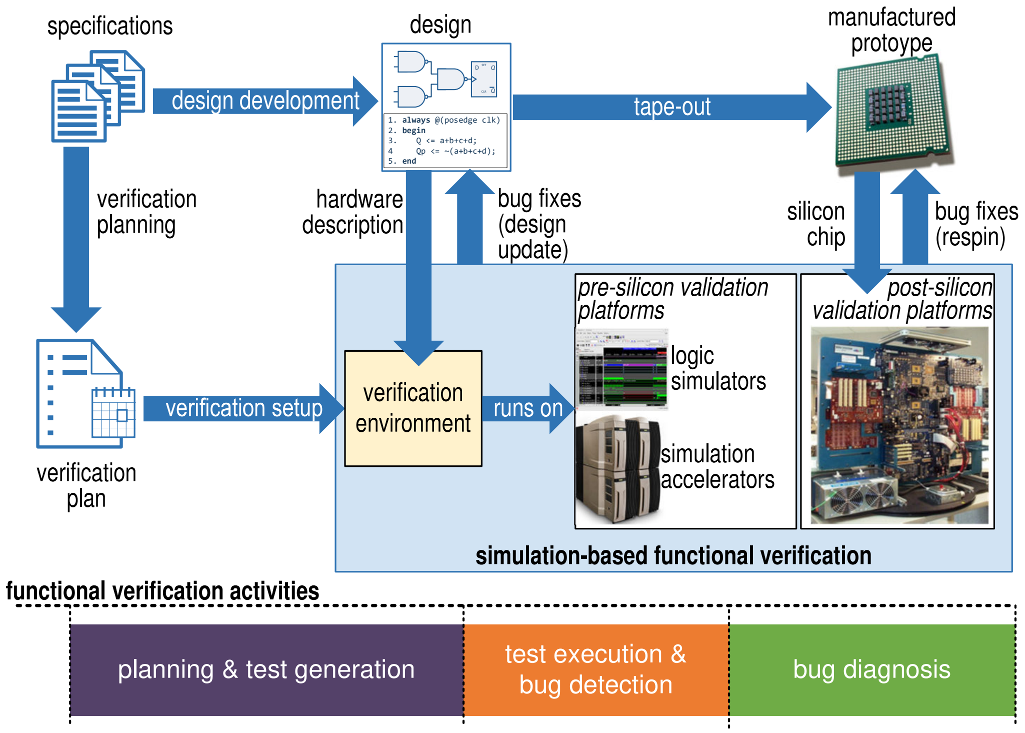


Figure 1.1 The functional verification process. A design project begins with a set of design specifications. A verification plan is drafted and implemented into a verification environment. As the design matures, it is dropped into the verification environment and tested to detect and fix bugs. Verification continues even after the design has been taped out into silicon. Planning & test generation, validation, and diagnosis are the main functional verification activities that we identify in this dissertation.

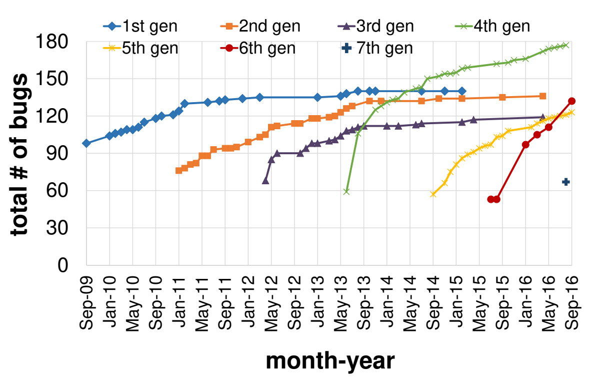
with several design blocks, power management features, and operating modes. Not only do we have to verify that each component operates correctly, but we also must ensure their correct interoperability. Be it at the component-level or the system-level, as the number of interacting entities increases, the sheer number of aspects that need to be verified also increases.

In order to reasonably ensure the correctness of all this complexity, designs go through an extensive, best-effort functional verification process as summarized in Figure 1.1. A design project starts with a set of specifications that are prepared after identifying market needs and analyzing innovative features to incorporate into the design. The specifications describe the capabilities that the design should implement and set the standard for correct operation. Based on the specifications, engineers draft a verification plan outlining the design aspects that must be tested for the implementation to be considered correct. The functional verification effort commences in parallel with the design development effort. A verification team implements a verification environment, which encases the design and runs tests on it. Different validation platforms are utilized to run tests at different stages of the verification process. Software simulators are replaced by simulation accelerators as the design matures. Automatic behavior checkers built into the environment detect bugs, which the engineers then diagnose and fix. Once the hardware description of the design has been sufficiently verified, it is manufactured into silicon in a process known as *tape-out*. Functional verification continues on the silicon prototype in a stage referred to as post-silicon validation. Bugs detected during the post-silicon validation stage trigger a new tape-out, which is referred to as a *respin*. In this dissertation, we partition the functional verification process into three major activities: planning & test generation, test execution & bug detection, and diagnosis. These activities will be discussed in Section 1.3.

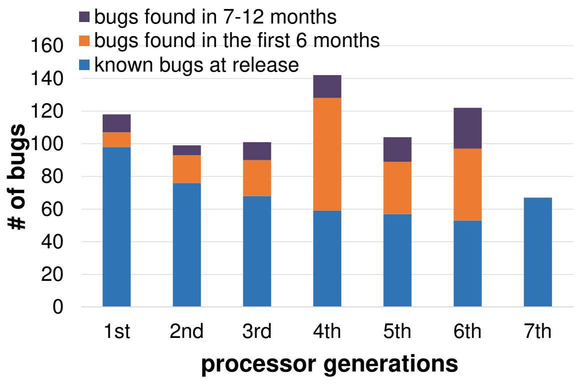
A significant amount of verification effort entails running tests on a simulated model of the design while observing its behavior. Simulation speed is of paramount importance as each change in a design requires a regression suite of tests to be simulated to verify that nothing was broken. Software-based simulation today is too slow to attain all goals in the verification plan within the demanding times-to-market of modern designs. Accelerated simulation, which uses special purpose hardware to either accelerate software simulation or emulate the design, is now taking on a growing portion of the verification responsibility. Even with the added speed of accelerated simulation, modern designs are not sufficiently verified by the time the first silicon is taped out. Hence, post-silicon validation is today a critical part of the functional verification process.

Despite the significant amount of resources and time poured into functional verification, no design today is guaranteed to work correctly all the time. And in practice, none does. Even for companies with several years of design experience, every new design generation brings its own set of verification challenges. Aggressive product schedules lead to bugs that are only discovered after products have been sold to customers. Figure 1.2 illustrates analyses of the bugs listed in Intel's processor specification updates for the last seven generations of mobile processor products [57, 58, 59, 60, 61, 63, 64]. The number of bugs discovered after a product has been released is increasing, with a big spike observed in the 4th generation. Some bugs are discovered so late that the features they affect are already incorporated into subsequent generations and can not be fixed even in the newer products. The bug that affected the transactional memory support introduced in the 4th generation of Intel's processors, for example, was discovered a year after release. Intel responded by disabling this new feature in the 4th and 5th generations of its mobile products [37]. The specification update for the 6th generation of mobile products still contains a description of this bug.

Note that the analysis from Figure 1.2 focused on Intel's products only because of the availability of detailed, post-release bug data for multiple generations of Intel's products.



(**a**) Number of bugs found in 7 generations of products



(**b**) Number of bugs found within 1 year from release

Figure 1.2 Number of bugs found in Intel's recent products. Data was obtained from specification updates for 7 generations of Intel® mobile processor products since September 2009. Some data was interpolated.

Intel is not the only company that is suffering from the functional verification burden. Studies [46] and personal discussions with experts in the industry have revealed that functional verification is a thorn in the side for many design projects. The best-effort nature of functional verification always leaves room for bugs to escape into products. The decision to ship a product is made with some trepidation since an escaped bug has the potential to tarnish the reputation of a company and cost billions of dollars to rectify [108, 106, 78].

1.2 The APA approach

In an ideal design project, every task in a verification process would always be accurate and a design would be completely verified before release. Due to increasing design complexity and shrinking times-to-market, functional verification today is only a best-effort process to meet the goals in the verification plan. These goals are not a complete enumeration of the entire capabilities of the design; they are only best estimates of the design behaviors that are likely to be exercised in the field, derived from the needs and experiences of architects, software developers, and users. As such, the entire verification process stands on the foundation of an approximate verification plan. Even if the verification environment is implemented accurately, the test runs hit the verification goals throughly and completely, and the debugging attempts eliminate all encountered bugs, the entire verification process is inherently incomplete and the outcome is only a reasonably verified design.

In this dissertation, we fully recognize the uncertainty inherent in the verification process. We embrace this uncertainty so as to identify and exploit opportunities for improving the efficiency and quality of functional verification. We introduce the automation, prioritization, and approximation (APA) approach as a way to make conscious and reasonable accuracy trade-offs for the purpose of increasing productivity. At the core of the APA approach is a shift in the way we perceive functional verification. In contrast to the traditional perception of functional verification as a process oriented towards meeting intuitively defined verification goals, we view it as a progression of multiple, interdependent endeavors: overcoming design complexity, implementing mechanisms to discover bugs, performing tasks to execute a verification plan, and wisely allocating engineers' effort.

We start with the recognition that not all design behaviors are equally important, not all bugs are equally destructive, not all tasks are equally critical, and not all effort is equally costly. We systematically prioritize those that are of highest value. In doing so, we expose opportunities for automation. We then explore approximation techniques that trade-off a small loss in accuracy for a much bigger gain in efficiency. In some cases, the trade-offs we make open up opportunities for efficient completion of tasks that would otherwise have not been possible. In other cases, we obtain significant speedups that justify the trade-offs we make. For instance, a solution that can automatically pinpoint the location of a bug significantly cuts down the amount of time spent diagnosing a bug. Even if this solution points to the wrong location once in a while, the time it saves when it is correct more than makes up for the extra effort engineers have spend when it is wrong.

1.3 Functional verification activities

In a typical processor design project, the functional verification process starts before design development begins and continues even after the first silicon prototypes are manufactured. In this dissertation, we partition the functional verification process into three activities as discussed below. For each activity, we provide a brief overview and present how our solutions tackle the challenges pertaining to the activity.

1.3.1 Planning & test generation

Based on the design specifications, the verification team creates a plan that outlines the verification goals to be met. These goals are intended to encompass all foreseeable usage scenarios that the design may be subjected to in the field and more. As the design matures and features are added, updated or removed, the verification plan and its set of goals also evolve to keep up with the changes. The goals in the verification plan are encoded into a *functional coverage model* that is used for measuring verification progress. Engineers also develop input stimuli that exercise the design aspects outlined in the verification plan. A verification team is said to achieve verification closure when all the goals in the verification plan have been met.

Traditionally, verification planning is mostly manual with engineers intuitively identifying the important design aspects to verify. The modern SoC design process presents unique challenges for verification planning. The large number of complex, interacting components create opportunities for several interaction scenarios, which can not all be verified before shipment. Moreover, SoC design projects today mostly involve integrating non-trivial design blocks, referred to as intellectual property (IP) blocks, which are built by parties other than the engineers integrating the SoC. The processor cores, the graphics processors, and the interconnect in a modern SoC are all instances of IP blocks that are designed by 3rd parties. This practice of integrating 3rd party IP blocks creates a knowledge gap that makes it difficult to identify and prioritize important design behaviors for verification.

In this dissertation, we apply the APA approach to tackle planning and test generation for verifying complex SoC designs. We recognize that the design behaviors that should be of high priority during SoC-level verification are those that arise from interactions between the IP blocks. Furthermore, some inter-block interactions are likely to be more commonly encountered than others. We develop an automated approach that identifies these high-priority inter-block interaction scenarios, automatically extracts functional coverage models for them, and automatically generates test programs that exercise them. Our approach forms an approximate view of the actual design behaviors arising from these interactions by analyzing the execution of software packages developed for the SoC on a high-level model of the design. In addition to eliminating the traditionally manual process of planning for which design behaviors to verify, our solution generates a test suite that is representative of the analyzed software packages but is much more compact. In our experiments, we found that the generated test suite can execute up to 11 times faster than the original software package. A fast test suite is very desirable for regression suites that have to be executed after each design modification to ensure that changes do not break functionality.

1.3.2 Test execution & bug detection

Once a verification plan is detailed, engineers implement a verification environment to realize the goals in the plan. During the rest of the verification process, the components of the verification environment execute tests on the implementation of the design and monitor its behavior for erroneous operation. A verification environment typically consists of functional coverage models and test suites from the planning & test generation activity, a testbench to connect the hardware description of the design with the verification environment, and code in the testbench to check the design behavior for correctness when tests are executed on it.

A bulk of the test execution & bug detection activity is performed by simulating the hardware description of the design during a phase known as pre-silicon verification. As design complexity increases, it becomes increasingly difficult to sensitize bugs during simulation. The input test sequences necessary to trigger buggy conditions require several cycles of simulation. The verification effort matures along with the design development effort. In a typical multi-core design project, functional verification initially focuses on unit-level verification, where the correctness of small design units is checked. After the design units are integrated into cores and other bigger subsystems. verification focuses on independently verifying these subsystems. Finally, all subsystems are integrated and verified during system-level verification. Unfortunately, the performance of software-based simulation is too slow even for subsystem-level verification. Today, hardware-assisted simulation accelerators and emulators play a major role in speeding up subsystem- and system-level simulation. However, even with acceleration, pre-silicon verification alone is not sufficient. Engineers perform more functional verification on the actual silicon prototype in a phase known as post-silicon validation. Unfortunately, going from pure software simulation to acceleration and then to post-silicon validation for speed comes with two major side effects: lower visibility into the internal mechanisms of the design, and reduced bug detection capability. Software-based simulation provides full visibility to all internal signals of the design's hardware description, allowing for robust checkers. Accelerated simulation affords a more restricted visibility than software-based simulation and also reduced checking capability. The silicon prototype offers the least amount visibility and is the most challenging platform for bug detection.

In this dissertation, we apply the APA approach to develop fast bug detection mechanisms that overcome the limitations of accelerated simulation and post-silicon validation. First, we recognize that most of the subsystem- and system-level functional verification effort is conducted on accelerated simulation and post-silicon validation platforms. Thus, we prioritize improving the quality and efficiency of bug detection capabilities on these platforms. Secondly, we assert that during system-level functional verification we should prioritize the detection of system-level bugs that are impossible to detect with the earlier subsystem- and unit-level verification efforts. We develop automated bug detection mechanisms for the processor core subsystem and whole system verification on accelerated simulation and post-silicon validation platforms. Our core-level mechanisms take the type of robust software checkers available during software simulation, and approximate their functionalities to deploy them on accelerated platforms. At the system level we implement a mechanism that includes an automatic, hardware-based information collection mechanism and automated software analysis for detecting bugs in system-level interactions. This solution can be deployed during both simulation acceleration and post-silicon validation. The class of bugs we prioritize for this solution, which are hidden in the interaction between processor cores, memory elements, and the interconnect, are extremely challenging to detect with traditional methods. In order to implement our mechanism efficiently, we identified opportunities to approximate away certain properties that are redundantly checked by earlier efforts or other properties that do not affect the quality of detection.

1.3.3 Bug diagnosis

Once a bug is detected, it goes through a diagnosis process to locate the source of the bug in the design and fix it. Unless a detected bug is severe enough to stop progress, design development and verification typically continue while the bug is being diagnosed. Initially, engineers go through a triaging process to assign the bug to engineers that are most familiar with the part of the design that the bug is likely to be in. The verification team then works with the design engineers responsible for the problematic components to locate and and fix the bug. The bug diagnosis process relies heavily on information gathered during simulation to identify the design components that are likely to be the source of the bug. In accelerated simulation and post-silicon validation environments where it is hard to collect information during test execution, engineers may have re-execute tests several times to reproduce and isolate the bug.

For today's complex processor design projects, engineers spend most of their time diagnosing bugs [46]. This burden stems from the difficulty of identifying the source of a bug in a complex design with several components. In this dissertation, we apply the APA approach to develop a solution that automatically pinpoints likely bug locations. We recognize and prioritize the most difficult diagnosis effort where engineers must sift through several components to identify the source of a bug with low visibility into the internal mechanisms of the design. We develop a methodology that uses machine learning to learn the approximate correlation between the location of a bug and the way it appears in low-visibility environments. Our solution, which automatically pinpoints the location of a bug with high accuracy given information only from the detection mechanism, can eliminate weeks of diagnosis effort per bug.

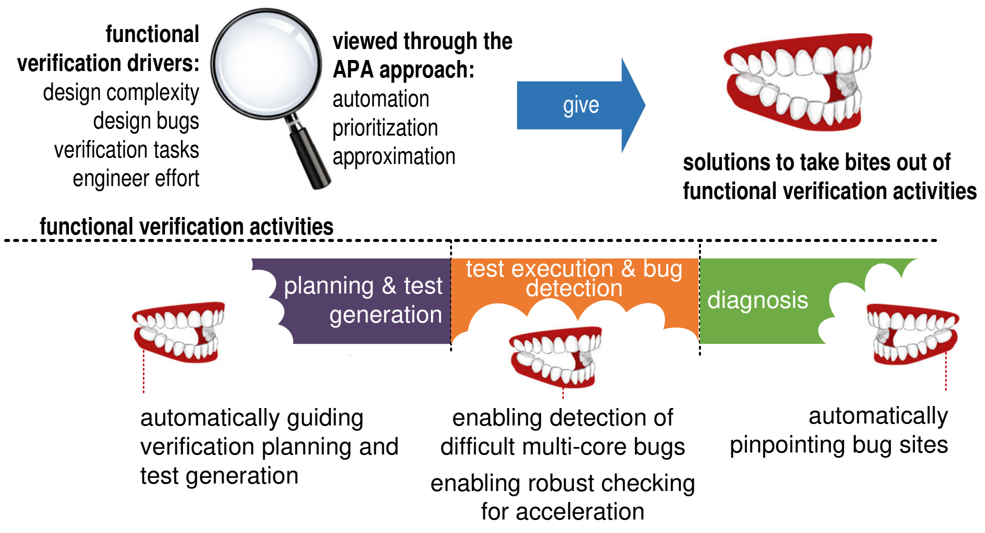

functional verification drivers:
design complexity
design bugs
verification tasks
engineer effort
viewed through the APA approach:
automation
prioritization
approximation
give
solutions to take bites out of functional verification activities
functional verification activities
planning & test generation
test execution & bug detection
diagnosis
automatically guiding verification planning and test generation
enabling detection of difficult multi-core bugs
enabling robust checking for acceleration
automatically pinpointing bug sites

Figure 1.3 Dissertation overview. The solutions proposed in this dissertation take multiple bites out of the verification process through the APA approach.

1.4 Dissertation overview

The functional verification process has not been able to scale with the increasing design complexity witnessed in modern processor designs. In this dissertation, we introduce the APA approach as a lens through which we evaluate the different drivers that contribute to the challenges of functional verification: increasing design complexity, difficult-to-detect design bugs, labor-intensive verification tasks, and the effort engineers spend to perform manual tasks. We develop several solutions that employ the APA approach to take bites out of the functional verification challenges grouped into the functional verification activities discussed in Section 1.3. Figure 1.3 provides an overview of the major contributions in this dissertation, which are also summarized below.

We automate planning and test generation for modern SoC designs. We present a novel, automated mechanism that analyzes the behavior of software executing on a high-level model of a design to produce an abstract representation of high-priority design behaviors to target for verification. In addition, we contribute a novel mechanism that generates compact test programs from the abstract representations that we produce. Our experiments show that our mechanisms can effectively automate the tedious process of prioritizing design behaviors for verification. In addition, our compact test programs cut down the simulation time required for tests that reproduce these behaviors by up to 11 times.

We enable efficient test execution and bug detection on acceleration platforms. We present a novel mechanism for approximating the functionality of traditionally software-only checkers so as to implement them in hardware with low overhead. In addition, we discuss a novel, synergistic checking strategy that deploys low-overhead, hardware checkers and on-platform compression logic to minimize the coupling between an acceleration platform and software checkers running on a host computer. Our experiments show that for some of our approximation techniques, we can reduce the sizes of our hardware checkers by over 56% with little impact on accuracy. For our synergistic checking strategy, our experiments showed up to a 92% decrease in communication traffic with only a 20% reduction in checking accuracy.

We execute tests and detect memory consistency bugs efficiently during acceleration and post-silicon validation. We develop a novel mechanism for observing, recording, and analyzing shared-memory interactions. Our solution adds low-overhead hardware mechanisms into the design for recording and flexibly deploys a software analyzer for bug detection. In our experiments, our solution achieved an 83% bug detection rate across a variety of memory consistency models, while performing analysis in under 20 seconds.

We localize bugs automatically in low-visibility validation environments. We present a novel mechanism for running tests and collecting bug information in low-visibility validation environments so as to pinpoint the likely sources of bugs using a machine learning classifier. We develop a feature engineering approach to convert the collected data into feature vectors to be processed by classification algorithms. We also discuss a novel bug injection strategy for generating synthetic bugs to train the classifier model. Our experimental results show that our solution is able to correctly identify the source of a bug 72% of the time in an out-of-order processor model.

1.5 Dissertation organization

The remainder of the dissertation is organized by the functional verification activities discussed in Section 1.3. Chapter 2 discusses the state-of-the-art in functional verification. It lays the background for the rest of the dissertation by discussing the different practices, methodologies, and tools used in modern industrial functional verification processes. Chapter 3 presents our solutions that tackle planning & test generation. Our solutions that address test execution & bug detection are presented in Chapter 4. Chapter 5 presents our solution for automatic bug diagnosis. Finally, Chapter 6 summarizes the major contributions of this dissertation and hints at possibilities for future work.

Chapter 2

An Introduction to Simulation-based Functional Verification

In this dissertation, we look at the entire gamut of the functional verification process through our APA approach. This chapter presents a primer on industrial functional verification, particularly focusing on simulation-based functional verification, that is useful for the rest of the dissertation. We begin with an overview of the functional verification process as practiced in the industry today. We then dive deeper into the components of a simulation-based functional verification environment that are relevant to this dissertation. Lastly, we discuss high-level design models and their role in the simulation-based functional verification process.

2.1 Overview of the functional verification process

After design specifications are drafted for a processor design project, hardware designers build a detailed, *behavioral model* of the design using a *hardware description language (HDL)*, such as verilog or VHDL. Meanwhile, verification engineers build a verification environment to achieve the goals set in a verification plan. The many design blocks are distributed among teams of design engineers who build to specifications and verification engineers who perform unit-level functional verification. As design blocks become available, they are integrated into subsystems and verified. The process continues until all the design blocks are completed and integrated at the system level. Design and verification engineers diagnose and fix bugs detected during the verification process. The functional verification effort that occurs before a silicon prototype is built is termed as *pre-silicon verification*. During pre-silicon verification, the HDL description of the design is automatically checked against specifications by the verification environment. *Formal verification* and, to a greater extent, *simulation-based verification* are the primary modes of pre-silicon verification.

Formal verification tools convert the HDL description of the design to Boolean functions that are compared against properties derived from the specifications. As long as engineers are able to represent design specifications as precise Boolean properties, the tools are able to mathematically prove or disprove the correctness of the design. The strong guarantee that formal verification provides when the design is correct, and the counter-examples it generates when the design is incorrect, makes it a very desirable verification tool. Unfortunately, completely representing a design's specifications as formally verifiable properties is a challenging task. In addition, the computational demands of formal verification tools are prohibitive for everything but small design units and subsystems. As a result, the usefulness of formal verification is limited to small design blocks. Since the goal of this dissertation is the verification of complex designs, we do not discuss formal verification further.

Simulation-based functional verification assumes most of the pre-silicon verification effort. Software-based simulation platforms compile the hardware description of the design into an executable that can run on any computer. The verification testbench, which itself is software that executes alongside the simulated design, directly interfaces with the simulation and has access to any design signal. The flexibility of software simulation and the high observability into design signals enables robust checking and debugging capabilities. Engineers can develop sophisticated checkers, step through the design using debuggers, record waveforms, *etc.*Unfortunately, software simulators are too slow for complex designs, simulating at best 100's of design cycles per second. At these speeds, it would take months to simulate a second of a design's operation.

To overcome the slow speed of software simulation, companies today depend on special-purpose hardware platforms that accelerate logic simulation. These platforms are built from either regular field-programmable gate arrays (FPGAs) or custom application-specific integrated circuits (ASICs). Not long ago, companies used to build their own FPGA-based prototypes for their designs. Today, there are EDA vendors that offer robust acceleration/emulation platforms [82, 26, 105]. These platforms can simulate hundreds of thousands of design cycles per second and can link with a verification testbench running on a host computer. However, engineers have to trade-off performance for increased visibility into the design's signals. Therefore, checking and debugging capabilities on these platforms are more restricted than on software platforms.

Once the pre-silicon, hardware description of the design is considered sufficiently verified, the physical design process commences. *Logic synthesis* tools convert the HDL description into a gate-level netlist, which is manufactured into silicon prototypes. These prototypes come back to the design and verification engineers who run tests to discover any bugs that escaped into the prototypes in an effort known as *post-silicon validation*. When it comes to speed, nothing beats the actual silicon prototype. Extending the simulation-based approach of executing tests on the design, post-silicon tests run at actual speeds, with billions of cycles executing in a second. Unfortunately, signal visibility is limited to the processor pins, making it extremely difficult to check and debug the design. Post-silicon design-for-debug (DFD) techniques add elements into the design that enable mechanisms for observing selected design signals at the processor pins. Bugs found at this stage may require changes in the behavioral model or the netlist, resulting in expensive silicon re-fabrications, known as *respins*.

2.2 Simulation-based verification testbench

Modern designs undergo trillions of cycles of simulation during pre-silicon verification [102]. The simulated design interacts with a verification testbench, which typically includes *stimulus generators* that provide input stimuli to the designs, *coverage models* that keep track of the design aspects that have been tested, and *checkers* that check for correctness. Typically, each block and subsystem in the design will have its own testbench. Simulation is typically performed until a checker fires to signal a bug in the design. Engineers then embark on a process to diagnose the root cause of the bug and fix it. For most bugs, problems start a number of cycles before the checkers fire and involve several design signals in addition to those being monitored by the checkers. Hence, a majority of debugging time is spent trying to *spatially* and *temporally* localize the root cause of a bug. Localization is a tedious process that involves analyzing signals and events over a number of cycles before the time of checker firing, to identify instances of anomalous behavior. To assist with this process, logic simulators typically come with the ability to generate signal waveforms during simulation. In addition, engineers can add testbench components that generate more meaningful traces of events during simulation.

In this section, we discuss in detail the three major components of a simulation-based verification testbench. For each component, we present how it fits into the rest of the dissertation.

2.2.1 Stimulus generators

The goal of logic simulation is to explore as much of the design state space as possible. Stimulus generators are designed to provide input sequences that steer the design through its state space. On one end of the spectrum, engineers can create simple stimulus generators that supply manually-crafted sequences of input stimuli. Such input stimuli are referred to as *directed stimuli* and are usually designed to test how the design responds to common-case input stimuli or to guide it to specific corner cases. On the other end, engineers can design random generators that produce sequences of *random stimuli*, obeying a number of constraints. Such stimuli use the power of randomization to reach vast swathes of the design's state space. A well-designed suite of random stimuli should reach a wide range of interesting design states as quickly as possible.

Block-level testbenches rely more on directed than random stimuli. The complexity of block-level designs is manageable enough that engineers can enumerate several important conditions that can be tested with directed input stimuli. For instance, when testing the ALU, engineers conceive sequences of input vectors that exercise all ALU operations and corner cases such as overflow and carry. As more and more blocks are brought together, constrained-random stimuli take prominent roles. Instead of crafting sequences of vectors, engineers specify a set of directives to constrain the vectors generated by a random stimuli generator. For instance, when generating instruction sequences to test the memory subsystem, engineers can add constraints to restrict the generated instructions to loads and stores and to limit the addresses that these instructions are accessing.

The constrained-random generator can be a separate tool that generates traces of input stimuli that are loaded by the testbench. This can be achieved by developing custom generator scripts or by using commercial constrained-random generators such as Raven [90] and Genesys-Pro [6]. Alternatively, hardware verification languages, such as SystemVerilog and Specman E, provide mechanisms for constructing constrained-random generators inside the testbench. SystemVerilog provides the `rand` and `randc` qualifiers for class members that will be assigned random values during simulation, when the built-in `randomize()` method is called on objects of the class. Constraints can be specified using the `with` construct at the time of the call to `randomize()` or using `constraint` blocks in the class definition. These SystemVerilog features are shown in the example in Figure 2.1.

Randomization offers a statistical confidence of verification reach after a large number of simulated cycles. But, increasing design complexity raises the need for more simulated cycles while decreasing the speed of simulation. Today, verification engineers have to be frugal with the amount of cycles their test suites can consume, demanding high-quality tests that reach interesting scenarios quickly. In Chapter 3, we present a solution for automatically identifying interesting scenarios for complex accelerator-rich SoC designs and generating compact tests that target them.

```
 1  program Generator;
 2
 3  class Instruction;
 4    rand bit [5:0] opcode;
 5    rand bit [4:0] base_reg;
 6    rand bit [11:0] offset;
 7    rand bit [4:0] data_reg;
 8    // ...
 9    constraint ld_st { opcode inside {`OPLD, `OPST}; }
10    constraint addr_hi {offset[11] == 1; }
11    //...
12  endclass
13
14  Instruction inst = new;
15
16  initial
17  repeat (100)
18    inst.randomize() with {base_reg == 4; data_reg == 5; };
19
20  //...
21  endprogram
```

Figure 2.1 Example of a SystemVerilog constrained-random stimuli generator. Instruction fields are qualified with `rand`. Line 9 constrains the opcode to be either that of a load or a store instruction. Line 10 constrains the most significant bit of the address offset to 1. Line 18 populates the instruction fields with random values obeying the constraints specified above and the additional constraints specified using the `with` construct.

2.2.2 Checkers

Verification testbenches are equipped with checkers that automatically validate design correctness. There are several ways of implementing checkers, ranging from low-level assertions embedded in the design to complex offline analysis scripts operating on simulation traces. To implement checkers, verification engineers need a model of correct behavior to check against. They typically derive this model from the specification, independently from the design. Some correctness conditions can be expressed as assertions that must always hold true. For instance, an arbiter for a shared resource must grant only one requester at a time. This property can easily be expressed by an assertion statement that checks if the grant signals coming from the arbiter are always one-hot encoded. Assertions can be more sophisticated than this simple example, spanning signal values over multiple cycles and allowing for the construction of non-trivial logic statements. Other correctness conditions might be too complicated to capture with assertions. Such conditions may necessitate the verification testbench to dynamically compute the correct outcomes, typically through high-level golden models developed independently from the design.

```
1 // immediate assertion
2 assert (dest_reg > 0 && dest_reg <= 15)
3   $display("Valid destination register");
4 else $display("Invalid destination register");
5
6 // concurrent assertion
7 assert property (@(posedge clk) reg_wr_en |-> #1 dest_reg != 0)
8 else $display("Register write enabled for read-only r0");
```

Figure 2.2 Examples of SystemVerilog assertions. Line 2 contains the condition for an immediate assertion that checks for the validity of the destination register. The statement in line 3 is executed when the assertion passes and that in line 4 when the assertion fails. Lines 7-8 express a concurrent assertion that is evaluated on the rising edge of the clock. It checks if the destination register is valid one cycle after the register-write-enable signal is asserted.

High-level verification languages provide powerful constructs for expressing assertions. SystemVerilog provides two types of assertions: immediate assertions, which are procedural statements that behave like assertions in high-level programming languages, and concurrent assertions, which can describe properties that span multiple cycles. Figure 2.2 shows examples of these two types of assertions. SystemVerilog assertions (SVA) are powerful tools that are also utilized for formal verification, stimulus generation, and functional coverage modeling. The latter will be discussed in Section 2.2.3.

Assertions and golden-model-based checkers are widely used in software-based verification testbenches. However, adapting these powerful, automated checkers for acceleration- or emulation-based validation environments may sometimes lead to prohibitive overheads. In Chapter 4, we present solutions that exploit accuracy trade-offs to efficiently deploy checkers on hardware-based validation platforms. We also present a mechanism for realizing complex memory consistency checkers tailored for hardware-based validation platforms. Once a checker detects a bug, engineers embark on diagnosing and fixing the bug. In Chapter 5, we present a solution that automates the initial laborious effort to diagnose the root cause of a bug.

2.2.3 Coverage

In modern simulation-based verification efforts, coverage models capture and tally the aspects of the design that were exercised during verification, acting as metrics for measuring verification progress. Types of coverage models, known collectively as *code coverage*, measure which parts of the HDL code have been exercised and can be automatically inferred by the electronic design automation (EDA) tools. *Functional coverage models*, however, are typically designed by the verification team to capture design-specific goals outlined in the verification plan. Functional coverage planning and model development start early, along with verification planning, and evolve as the design and verification effort progresses.

Most modern design projects have adopted a *coverage-driven verification methodology* where coverage models play a central role in driving the verification process [46]. When setting a schedule for the verification process, milestones are set from expected coverage targets. For instance, it maybe desired to execute all instruction types at least once in a given time-frame. This goal then targets 100% coverage of executed instructions. In addition, in constrained-random verification, coverage metrics are used to guide the test generation process. Early in the verification process, the constraints for the stimuli generator may guide the verification effort towards some coverage goals while completely missing others. Using the insights from their coverage models, engineers can adjust the random generator constraints to steer the verification effort to fill these coverage holes. Hence, coverage metrics boost the efficiency of the verification process by giving quick feedback to direct the testing effort and minimize wasted simulated cycles.

High-level verification languages offer support for expressing functional coverage models. SystemVerilog provides the `covergroup` construct to define coverage models. A SystemVerilog `covergroup` may contain several `coverpoints` that sample design behaviors. Figure 2.3 shows an example of a coverage model that samples the types of instructions executed by a design. In this example, the coverage model is defined inside the `Instruction` class from Figure 2.1. Modern EDA tools provide interfaces for navigating coverage metrics collected during simulation.

```
 1  class Instruction;
 2    rand bit [5:0] opcode;
 3    //...
 4    covergroup opcodes;
 5        coverpoint opcode;
 6    endgroup
 7
 8    function new();
 9      this.opcodes = new();
10    endfunction
11    //...
12  endclass
```

Figure 2.3 Example coverage model in SystemVerilog. This simple coverage model is defined inside the *Instruction* class from Figure 2.1. The `coverpoint` on line 5 keeps track of the different values generated for the `opcode` field. The coverage model is instantiated inside the class constructor on line 9.

Coverage models play a crucial role in the verification process by encoding the verification plan into the verification testbench. However, setting verification goals and defining coverage models for them is a mostly manual process that relies on intuition. The solution we present in Chapter 3 can automatically identify high-priority goals for the verification of complex SoCs and can define coverage models that capture these goals.

2.3 The role of high-level design models in verification

The purpose of functional verification is to ensure the correct HDL implementation of a design. In a typical processor design project, designs are modeled at higher abstraction levels before development on the behavioral model begins. A programmable processor implements an *instruction set architecture (ISA)*, which defines a set of instructions, registers, memory semantics, and input/output handling that is visible to programmers. For a new ISA, architects develop a high-level *instruction set simulator (ISS)*. The ISS simulates the correct, programmer visible-outcomes to be expected from executing a program on a processor implementing the ISA. Since the ISS, which is developed in a high-level programming language like C++, does not model the details of a particular implementation, it can quickly simulate the execution of programs developed for the ISA.

Most processor design projects, however, are new implementations of an existing ISA, such as Intel's 64 and IA-32 [62], ARM's ARMv8 [14], and IBM's Power [56] ISAs. An implementation of an ISA is referred to as a *microarchitecture*. For a new design project, architects explore several approaches for designing a microarchitecture with enhanced performance and energy-efficiency. To this end, they develop another high-level *performance model* that simulates the timing of microarchitectural elements and their interactions. Even though a performance model is more detailed than an ISS, it still abstracts away a lot of details that are intrinsic to a real hardware model, and is developed using high-level languages like C++ and SystemC. A performance model allows architects to quickly implement an idea and get estimates for its impact on the overall performance of the design. Through this *(micro)architectural exploration* process, architects identify the beneficial features to incorporate into the microarchitecture. The performance model then serves as the template from which the design specifications are drafted.

These high-level design models are utilized for different purposes during the functional verification process. Firstly, they are used to vet randomly-generated test programs. In constrained-random stimuli generators, certain constraints may be difficult to specify for some types of test scenarios. For example, consider a memory subsystem test scenario that targets a specific range of memory addresses while allowing for any register to be used in the address computation. The contents of the registers, however, could be random values written by earlier instructions. Here, an ISS can be used to simulate already generated instructions and supply register values when required to evaluate a constraint. Secondly, high-level design models are crucial for the creation of self-checking test programs. These programs include instruction sequences that check the correctness of previous computations performed by the same program. The expected outcomes for those computations are obtained by simulating the program on a high-level design model. Thirdly, high-level design models can provide expected outcomes that are required by complex checkers in the verification testbench. The availability of high-level behaviors to compare with is especially useful for checking end-to-end design properties that can not be described succinctly with assertions. Finally, high-level models are used to verify software being developed for the design that is under development. Modern SoC design projects verify hardware and software together in a process referred to as *hardware/software (HW/SW) co-verification*. In HW/SW co-verification flows, software developers test and debug their programs on high-level models of the design while the behavioral model of the design is in development. Typically, the software consists of boot and initialization code, operating systems, compilers, device drivers, and user applications that will be shipped together with the SoC.

The solutions we discuss in this dissertation utilize high-level design models in various capacities. Chapter 3 presents a solution that leverages HW/SW verification environments to extract high-priority targets for verification by analyzing software executing on a high-level design model. The experimental framework used for one of the solutions presented in Chapter 4 utilizes a high-level design model to vet randomly-generated test programs. Central to the automatic diagnosis approach described in Chapter 5 is a bug-detection strategy that compares the execution of a design with the execution on a high-level model.

2.4 Summary

In this Chapter, we presented an introduction to a modern simulation-based functional verification process. We discussed the various tools, methods, verification platforms, and testbench components used for functional verification today. We looked at the verification process, starting from early, unit-level pre-silicon verification, through subsystem-level accelerated simulation, to system-level post-silicon validation. We discussed the three major components of a simulation-based verification testbench – stimulus generators, checkers, and coverage models – while presenting SystemVerilog code examples. We also saw the different levels of design models and how they are utilized in the verification process. In our discussions, we touched upon the challenges of the simulation-based verification process, briefly previewing how we tackle them in this dissertation. The topics discussed in this chapter are helpful for understanding the material presented in this dissertation. The rest of this dissertation dives deep into our solutions.

Chapter 3
Addressing Planning and Test Generation

In this chapter, we present our application of the APA approach to aspects of verification planning and test generation for modern SoC design projects. In such design projects, the many complex components integrated at the system level create several possible interaction scenarios, which can not all be verified in the time alloted for system-level verification. One of the major challenges in the verification of modern SoCs is prioritizing and testing the system-level interaction scenarios that are likely to be critical for the consumer. By automatically analyzing the execution of software on a high-level model of the design, we discovered that it is possible to prioritize the system-level interaction scenarios that are most likely to be exercised by the software expected to run on the final product. Since the high-level design model does not implement the exact timing of events, we recognized a need for approximations that estimate the timing of actual interactions based on the inexact timing information obtained from the high-level model. Moreover, we discovered that the results of our analysis can also be used to drive the generation of coverage models and test programs targeting the prioritized interaction scenarios.

Based on these insights, we developed a solution that we call **AGARSoC** (**A**utomated Test and Coverage-Model **G**eneration for Verification of **A**ccelerator-**R**ich **S**ystems-**o**n-**C**hip). Illustrated in Figure 3.1, AGARSoC leverages HW/SW co-verification environments to automatically discover high-priority interaction scenarios. These scenarios are identified by analyzing memory traces obtained from the execution of software on a high-level design model. In addition, AGARSoC also generates functional coverage models and compact test programs that represent the identified scenarios. The generated programs are much more compact than the original software suite they are derived from. As a result, they require fewer simulation cycles, which is a desirable trait for frequently executed regression suites. Our experimental results show that our generated tests are able to exercise all the prioritized scenarios up to 11 times faster than the original software suites while meeting all of the verification goals identified by AGARSoC.

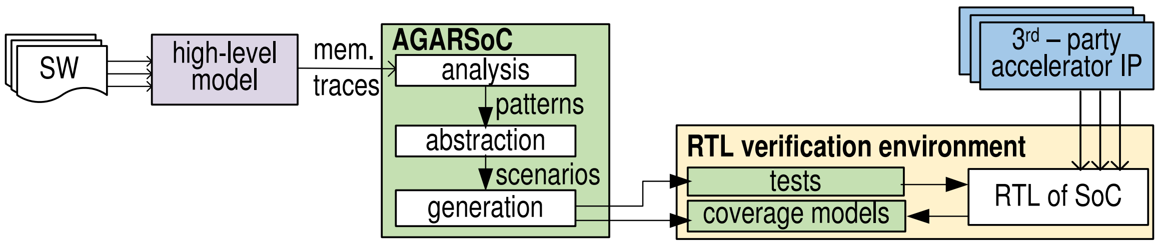


Figure 3.1 AGARSoC overview. We analyze the execution of software on a high-level model of an accelerator-rich SoC to identify high-priority accelerator interaction scenarios for verification. We then generate corresponding coverage models and compact test programs, which are used to guide the verification of the RTL for the SoC. AGARSoC is mostly automated and can be flexibly controlled by parameters supplied at runtime.

The remainder of this chapter is organized as follows. Section 3.1 presents an overview of accelerator-rich SoCs and their verification challenges. Section 3.2 discusses the accuracy tradeoffs of our APA approach. Sections 3.3 - 3.6 present the details of our mechanisms and our experimental results. We discuss related work in Section 3.7 and end by summarizing the chapter in Section 3.8.

3.1 Background on accelerator-rich SoCs

Driven by stagnating performance gains and energy-efficiency constraints, multicore systems-on-chip (SoCs) that integrate several general purpose cores and highly-specialized accelerators are being widely adopted [1, 2, 86, 33]. These components are connected via an on-chip interconnect as illustrated in Figure 3.2. Each accelerator in the system is designed to execute a specialized task quickly and efficiently. Communication between the components in these multi-core SoCs is achieved through reads and writes to shared-memory locations. A shared-memory system presents a single logical memory that all components can read from and write to. Even though the actual physical implementation involves a hierarchy of memory elements, some of which are private to components, the shared-memory abstraction allows components to observe each other's reads and writes. Some of these logical memory locations are configuration registers inside the accelerators. If a thread running on a core wants to invoke a task on a particular accelerator, it writes the commands and parameters for launching the task to the accelerator's configuration registers through writes to the memory locations mapped to the accelerator's configuration registers. A launched accelerator performs its task, possibly bringing a chunk of data from shared memory to its scratchpad memory, operating on it, and finally writing the results back to shared memory. Upon completion of its task, an accelerator notifies the software, often

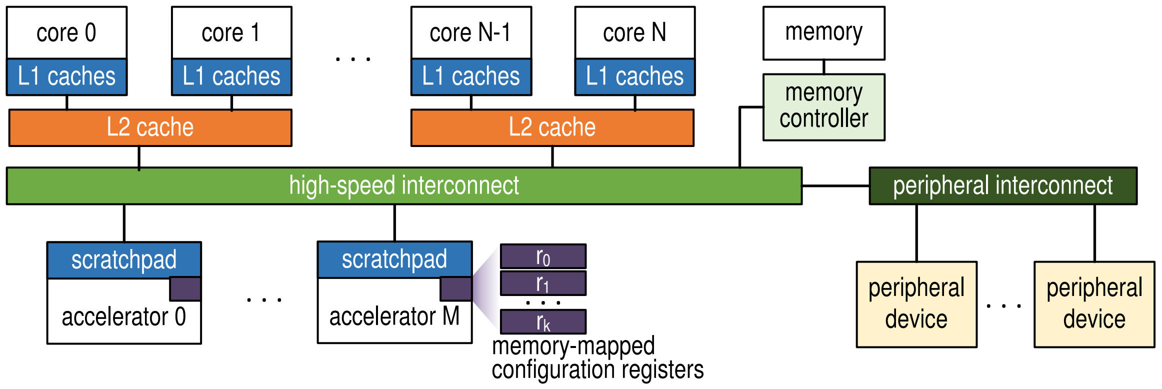


Figure 3.2 Accelerator-rich multiprocessor SoC architecture. Multiple cores, accelerators, memories, and peripheral components are integrated in a chip through an interconnect. These components interact via memory operations, which pass through multiple levels of memory and interconnects to arrive at their destinations. Memory-mapped registers in the accelerators are used to configure accelerators for execution.

by writing its status to a memory location specified during configuration. The accelerators in the SoC can execute concurrently, possibly stressing the interconnect and the shared memory implementation.

Often, the components that comprise these SoCs are designed and verified independently by different teams, possibly from companies other than the one building the SoC. Even though these components are supposedly verified, they may still harbor bugs that only manifest when integrated with other components, in a specific configuration. These bugs are either due to conditions that are very difficult to sensitize during independent verification, or due to incompatible interpretations of specifications and assumptions taken by the designers of the different components. SoC integration engineers bring components progressively together and verify the resulting configurations by running tests that sensitize different interaction scenarios among the components. Each accelerator in the system can be configured in several different ways, leading to several different modes of operation, which, coupled with the large number of components in the system, result in a copious amount of interaction scenarios to be considered for verification. This makes it challenging for verification engineers to prioritize scenarios, develop coverage models, and generate tests for the many interaction scenarios that arise.

3.2 APA: benefits and accuracy tradeoffs

AGARSoC automatically analyzes the execution of software on high-level models of the design to identify the design behaviors to prioritize for verification. Our analysis mechanism operates on architectural memory traces, which contain dynamic, non-speculative, memory operations and their associated data. For shared-memory multicore SoCs, memory traces collected from different models, or even from different executions on the same model, are not guaranteed to be identical. Micro-architectural features that vary among models / implementations, such as caches, affect the order in which different components in the system observe memory accesses. Multiple correct program executions, each with a different interleaving of memory accesses observed by the components of the SoC, are legal in this context.

In addition to the discrepancy between traces, certain design behaviors may only be observable at the microarchitectural-level. For example, in an implementation of the ARM ISA [13], barriers, atomic accesses through exclusive loads and stores, and cache and TLB maintenance instructions may all trigger microarchitectural events that affect multiple components. Not all of these events are readily visible architecturally: i) a microarchitectural (μ-arch) event may be completely hidden, ii) others may manifest out-of-order, iii) a μ-arch event may have a delayed manifestation, iv) some may manifest coalesced, and v) non-concurrent μ-arch events may manifest concurrently.

For these reasons, the insights derived from analyzing multicore SoC memory traces are not completely accurate. Yet, by factoring the uncertainty of microarchitectural interactions into the analysis mechanisms, the impacts on accuracy can be minimized. For instance, an architectural level analysis can relax certain observed orderings among operations because of the understanding that those operations may be reordered microarchitecturally. In addition, any sacrifice in accuracy comes with the following benefits:

Early availability: Architectural-level design models are available early and serve as the golden reference for the RTL design process. This is what enables AGARSoC to analyze system-level behaviors early in the verification process, to identify the important scenarios on which to focus verification, and to guide test generation.

Portable automated analysis: Analysis performed on architectural memory traces can be ported to different models, implementations, or generations of the same design.

Access to software: In modern design flows, software is co-designed with the SoC hardware. Software developers test their software on high-level models of the design. By analyzing actual software to deployed on the SoC, AGARSoC gathers insight into the aspects of a design that are of high priority to software.

3.3 Prioritizing system-level interactions with AGARSoC

AGARSoC discovers high-priority system-level interactions triggered by software. To capture software behavior, we analyze memory operation traces collected from software running on a high-level model. These traces include regular read and write accesses to memory from processor cores and accelerators, write operations to accelerator configuration registers, and special atomic memory operations for synchronization. Each access in a memory trace includes the cycle, memory address, type, size, and data for the access.

3.3.1 Analyzing memory traces

We scan the memory traces from the processor cores to detect patterns of memory accesses belonging to the invocation and execution of tasks on accelerators. Our analysis algorithm first identifies patterns of accesses to accelerator configuration registers. The mapping of configuration registers to memory locations is a design property that is provided as input to our algorithm. We then identify memory operations originating from the just-configured accelerator, occurring after the configuration event. Finally, we look for special memory operations that indicate the completion of a task. We bring together these patterns into what we refer to as an *accelerator execution pattern*.

It is common for accelerators to be configured for the same type of task multiple times. In such cases, the multiple accelerator executions may be configured to operate in the same mode but fetch data from and store results to different memory locations. Multiple executions of the same mode of operation are uninteresting from the perspective of verification. For each accelerator execution pattern we detect, we separate configuration accesses into those that define the mode of operation and those that specify where to fetch and store data. Our algorithm then keeps track of unique modes of operation, which we refer to as *execution classes*, and their associated accelerator execution patterns.

AGARSoC's analysis is a mostly automated procedure with minimal engineer input. Engineers specify the memory mappings for the different components, which is obtained from the SoC configuration. We expect accelerator designers to provide descriptions of start and end conditions for detecting accelerator access patterns, as well as specify which configuration writes define a mode of operation for their accelerator. Integration engineers can then encapsulate these conditions as Python functions and easily pass them as arguments to AGARSoC's analysis tools. Below are examples of designer-specified conditions for the multi-writers, multi-readers (MWMR) protocol in the Soclib framework [110], encapsulated in the Python functions shown in Figure 3.3:

- *"An accelerator signals task completion by writing a non-zero value to a memory location written to configuration register 0x4c".*
- *"Writes to configuration registers 0x44, 0x48, 0x60, 0x50, and 0x5c define mode of operation."*

```
def MWMREndDetector(configPat, accOp):
    endLoc = configPat.accesses[0x4c].data
    if accOp.addr != endLoc or not accOp.isWrite():
        return False
    return accOp.data != 0

def MWMRConfigHash(configPat):
    mode = [configPat.target]
    for reg,acc in configPat.accesses.iteritems():
        if reg in [0x44, 0x48, 0x60, 0x50, 0x5c]:
            mode.append(acc.data)
    return tuple(mode)
```

Figure 3.3 User-defined functions. `MWMREndDetector` returns `True` if a particular accelerator memory operation is signaling the end of a task. `MWMRConfigHash` returns a hashable tuple that contains the mode of execution. The `configPat` (accelerator configuration pattern) and `accOp` (accelerator memory operation) data structures are provided by AGARSoC.

3.3.2 Identifying interaction scenarios

We identified two types of interaction that are likely to lead to scenarios that were not observed during independent verification. Firstly, concurrently-executing accelerators interact indirectly through shared resources. Any conflicting assumptions about shared resource usage manifest during concurrent executions. Secondly, the state of the system after one accelerator execution affects subsequent accelerator executions. Any operation performed by one accelerator that leaves the system in a state not expected by another accelerator is likely to cause the second accelerator to malfunction. Our scenario identification algorithm discovers concurrent and ordered accelerator executions that might lead to these two types of interactions among accelerators.

Concurrent accelerator interaction scenarios are groups of accelerator execution classes whose execution instances have been observed to overlap. The overlap is detected by comparing the start and end times of the accelerators' execution patterns. While start and end times observed from a high-level model may not be exactly identical to those observed in RTL, they represent *the absence of ordering constraints* and therefore indicate a possibility of concurrent execution on the SoC. Concurrently-executing accelerators interact indirectly through shared resources (interconnect, memory, *etc.*.) leading to behaviors that may not have been observed during the independent verification of the accelerators.

Ordered accelerator interaction scenarios are either intended by the programmer (program ordered, and synchronized) or coincidentally observed in a particular execution. We infer three types of ordered accelerator executions:

Program-ordered: non-overlapping accelerator execution classes invoked in program order from the same thread, detected by comparing start and end times of accelerator executions invoked from the same thread.

Synchronized: accelerator execution classes whose instances are invoked from multiple threads, but are synchronized by a synchronization mechanism. We detect these by first identifying synchronization scopes (critical sections) from core traces and grouping accelerator invocations that belong in scopes that are protected by the same synchronization primitive. For lock-based synchronization, for instance, we scan the memory traces to identify accelerator invocations that lie between lock-acquire and lock-release operations. We group the accelerator execution classes for these invocations by the lock variables used, giving us groups of synchronized classes.

Observed pairs: pairs of accelerator execution classes whose execution instances were observed not to overlap. Unlike program-ordered scenarios, which can be of any length, observed pairs are only between two execution classes.

In primarily targeting concurrent and ordered interaction scenarios, we are focusing on indirect interactions among accelerators via shared resources. Even though we did not attempt to capture direct interactions between accelerators in our work, we believe it is possible to do so. Our analysis algorithm can be enhanced to detect an accelerator producing data to be consumed by another without the involvement of the processor cores, multiple accelerators racing to write to shared memory locations, and an accelerator invoking a task on another.

3.3.3 Abstract representation

AGARSoC creates an abstract representation of software execution from the interaction scenarios it detects. This representation only retains configuration-specific information, filtering out information such as specific memory locations and actual contents of data manipulated by the accelerators. Our representation contains three major categories: accelerator execution classes, concurrent scenarios, and happens-before (ordered execution) scenarios. Each entry in these categories contains a unique hash for the entry and a count of how many times it was observed in the analyzed execution. The entries for accelerator execution classes and concurrent scenarios are directly obtained from the analysis. The happens-before category contains combined entries for all unique observed pairs and program-ordered sequences. In addition, we generate all possible non-deterministic interleavings of synchronized accelerator executions. The entries in the happens-before sections have counts of how many times they were observed in the execution traces. For any generated ordering that has not been observed, we store a count of 0.

Figure 3.4 illustrates an example of our abstraction process. The SoC in this example has 3 cores and 3 accelerators. The software threads running on these cores trigger multiple accelerator executions, labeled **1** – **6** in the figure. A circle, a diamond, and a triangle are used to represent invocations of A_0, A_1, and A_2, respectively. The different colors indicate different modes of operation. AGARSoC will identify 5 different classes of accelerator execution corresponding to the different accelerators and modes observed during execution. Executions **1** and **4**, **3** and **5**, and **3** and **6** overlap, resulting in the three concurrent scenarios in the abstract representation. Similarly, AGARSoC identifies two sequences of program-ordered execution classes. Lock-acquire and lock-release operations using the same lock variable, indicated by locked and unlocked padlocks, respectively, mark one group of synchronized accelerator executions. AGARSoC generates all 6 possible interleavings of sequence length 2 for the accelerator execution classes in this group. The unobserved interleavings are marked with a count of 0. The immediate observed pairs extracted from the execution are not shown in the Figure because they have all been included in the program-ordered and synchronized interleavings groups shown. Note that the observed pair resulting from executions **4** and **2** is also a possible interleaving of **2** and **5**.

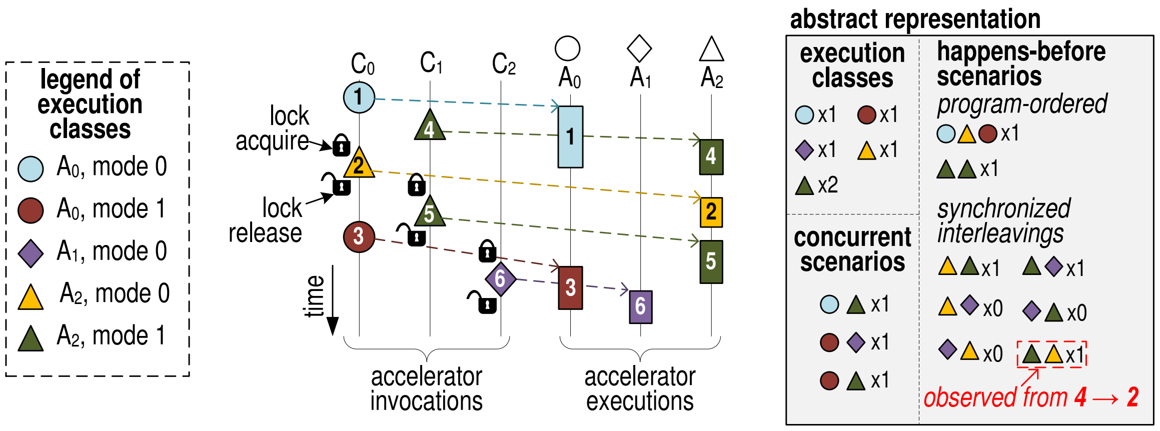


Figure 3.4 Abstracting an execution. Accelerator tasks are launched from the general-purpose cores. We group these task executions into unique execution classes, represented by shape and color. We represent concurrently-executing execution classes and ordered execution classes. The ordering among classes is obtained from program-order, generated interleavings of synchronized invocations, and observed immediate ordered pairs.

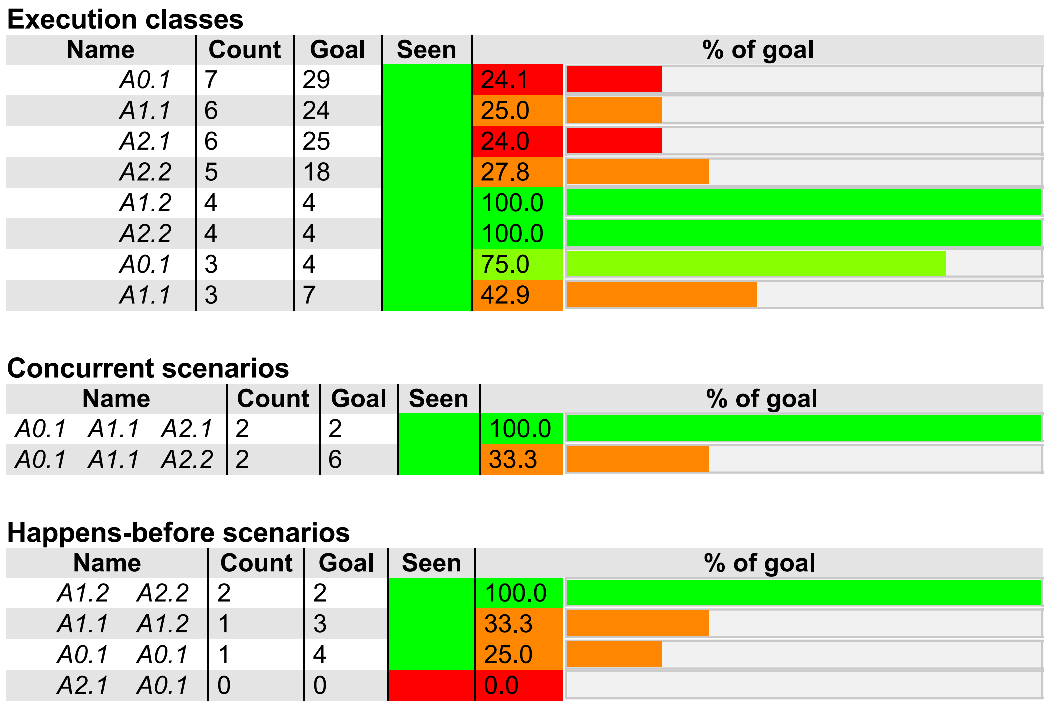
Abstract representations for different software executions can be combined to create a union of their scenarios by simply summing up the total number of occurrences for each unique scenario observed across the multiple executions. Through the process of combining, redundant accelerator interactions that are exhibited across multiple programs are abstracted into a compact set of interaction scenarios. This is one of the key features that allows AGARSoC to generate compact test programs that exhibit the interaction scenarios observed across several real programs.

3.4 Coverage model generation and analysis

AGARSoC generates a coverage model that guides the RTL verification effort towards high-priority accelerator interactions, as identified from software execution analysis. The coverage model generation algorithm performs a direct translation of the abstract representation extracted from the analysis: scenarios are translated into coverage events and scenario categories are translated into event groups. The coverage events in each event group can be sorted by the count associated with each scenario entry in the abstract representation. A higher count indicates a more likely scenario.

In addition to coverage model generation, AGARSoC's coverage engine is capable of evaluating and reporting the functional coverage data from RTL simulation. To take advantage of this capability, the RTL verification environment should output traces of memory operations. We believe this is a feature that most real-world RTL verification environments have, since memory traces are valuable during RTL debugging. AGARSoC's coverage engine is designed to be very flexible to adapt to changing verification requirements. Coverage models that AGARSoC generates can be merged without the need to analyze executions again. In addition, coverage output files are designed such that a coverage analysis output for a particular execution can be used as a coverage model for another execution. These features give engineers the flexibility to incorporate new software behaviors at any stage of verification, to compare the similarity between different test programs, and to incrementally refine verification goals.

Figure 3.5 shows a snippet of a coverage report generated by AGARSoC. This report shows the coverage analysis for an AGARSoC generated program versus a coverage model extracted from a software suite. The *Goal* column shows the counts of each coverage event as observed in the original software suite. AGARSoC currently uses these goals only as indicators of how frequently scenarios are observed, which can guide the choice of scenarios to include for test generation. The *Count* column shows how many times the events



Execution classes

Name	Count	Goal	Seen	% of goal
A0.1	7	29	🟩	24.1
A1.1	6	24	🟩	25.0
A2.1	6	25	🟩	24.0
A2.2	5	18	🟩	27.8
A1.2	4	4	🟩	100.0
A2.2	4	4	🟩	100.0
A0.1	3	4	🟩	75.0
A1.1	3	7	🟩	42.9

Concurrent scenarios

Name	Count	Goal	Seen	% of goal
A0.1 A1.1 A2.1	2	2	🟩	100.0
A0.1 A1.1 A2.2	2	6	🟩	33.3

Happens-before scenarios

Name	Count	Goal	Seen	% of goal
A1.2 A2.2	2	2	🟩	100.0
A1.1 A1.2	1	3	🟩	33.3
A0.1 A0.1	1	4	🟩	25.0
A2.1 A0.1	0	0	🟥	0.0

Figure 3.5 Sample coverage analysis output.

were observed in the generated test program. The *Seen* column indicates the presence of an event occurrence with a green color and its absence with red. The *% of goal* column indicates how many times the synthetic events were observed compared to the events in the original software suite. This column is ignored when all that is required is at least one hit of a coverage event. However, it becomes relevant when comparing the similarity between different test programs.

3.5 Test generation

In addition to coverage models, AGARSoC also generates compact test programs that exercise the accelerator interaction scenarios captured in the abstract representation. By eliminating redundant execution patterns that exist in the original software suite, these generated tests can execute much faster. Their compactness making them desirable for RTL verification where the speed of test execution is critical. AGARSoC allows the user to choose interaction scenarios to target with the generated tests.

3.5.1 Generated test program structure

To ensure that a generated test will always exercise the selected scenarios, we chose a multi-threaded, multi-phased program structure for our generated tests. The generated tests, which are meant to be run on the processor cores, can invoke multiple accelerators concurrently from multiple threads. We group these concurrent accelerator task executions into synchronized phases. All concurrent task executions in a phase are guaranteed to complete before execution of tasks in the next phase begins. Using this program structure, we can simultaneously incorporate concurrent and ordered interaction scenarios. This is achieved by choosing a schedule of task executions, *i.e.*an assignment of task executions to threads and phases, as will be discussed in Section 3.5.2.

For each SoC design we target, we manually write a C function that implements our multi-threaded, multi-phased program structure. This function serves as a reusable dispatcher of accelerator tasks. This function contains generic code that can read accelerator task configurations produced by our generator and invokes the accelerators accordingly. Our program generator produces a queue of task configurations per thread of execution. The specific queue in which a task configuration resides determines the thread that invokes the corresponding task, while its position within the queue determines the phase. For instance, the 4th task configuration in the 2nd queue results in a task that is invoked in the 4th phase by the 2nd thread of execution.

Our dispatcher function first creates as many threads as there are queues and assigns each thread the responsibility of executing the tasks in its respective queue. After executing a task, a thread will wait for other threads to finish executing their tasks before proceeding to the next task in its queue. By synchronizing threads after a task execution, we ensure that they are in the same phase of execution at any given time. The implementation of the waiting mechanism is dependent on the synchronization constructs available on the target SoC. If the generated schedule for a thread does not have a task assigned for a particular phase, then a special null value is placed in the corresponding location in the thread's task configuration queue. In such a case, the thread simply does nothing for that phase and waits for other threads to complete executing their tasks for that phase. The pseudo-code in Algorithm 1 summarizes the operation of our dispatcher function.

3.5.2 Schedule generation algorithm

The goal of AGARSoC's test generator is to produce a program that exhibits all the concurrent and ordered scenarios selected for generation. Since the dispatcher function discussed

```
Function DispatchTasks(taskQueues)
    foreach queue in taskQueues do
        DispatchThread(queue)
    end
    Wait for all threads to complete
end
Function DispatchThread(queue)
    phase ← 0
    while queue is not empty do
        Pop head of queue into taskConfig
        if taskConfig is not the null configuration then
            Invoke an accelerator as per taskConfig
        end
        Wait for all threads to complete this phase
        phase ← phase + 1
    end
end
```

Algorithm 1: Task dispatching algorithm common to all generated tests. Our test generation algorithm produces a schedule of tasks that are grouped into multiple queues. Each entry in a queue contains a task configuration for a particular phase of test program execution. The task dispatcher ensures all tasks in a phase are executed concurrently and phases are ordered.

in Section 3.5.1 is responsible for the actual execution of tasks, the core of the test generator is responsible for creating a schedule of task configurations. A task configuration represents an instantiation of an execution class, which includes the values to be written to the accelerator configuration registers and the layout of the memory that will be accessed by the task. Our test generator can be configured to populate the memory to be read by the accelerators with either random data or data that was observed during analysis.

The schedule generation algorithm attempts to optimize for a compact schedule under two constraints: each concurrent scenario should be scheduled to execute concurrently in a phase, and all ordered scenarios should be observed from the execution of tasks in consecutive phases. For example, consider the following scenarios where A, B, C, and D represent execution classes: A and B are concurrent, C and D are concurrent, and C happens before A. The generated schedule will have two threads and should have at least two phases to accommodate the two concurrent scenarios. Consider a case where the schedule generation algorithm allocates an instance of A to the first phase of thread 1 and an instance of B to the first phase of thread 2. Our test will then execute these instances concurrently. The schedule generation algorithm then allocates C and D to the second phase of the two threads. There are 4 possible ordered executions that will result from these schedule: $A \rightarrow C$, $A \rightarrow D$,

$B \rightarrow C$, and $B \rightarrow D$. However, none of these ordered executions satisfies the ordered scenario $C \rightarrow A$. The schedule generation algorithm will then have to add another phase to satisfy this happens before scenario, resulting in a schedule with three phases. Instead, if the algorithm schedules C and D to the first phase and A and B to the second phase, the $C \rightarrow A$ scenario will be satisfied with a schedule that has only two phases. Our schedule generation algorithm should be designed to prefer the second schedule over the first.

Unfortunately, an algorithm that can guarantee an optimal schedule is computationally prohibitive, especially when there is a large number of scenarios. Moreover, the number of execution classes in a concurrent scenario and the lengths of ordered scenarios can vary based on the characteristics of the SoC, the nature of the analyzed software suite, and user specified configuration parameters. These variations make it difficult to even define the optimality of a schedule. An ideal (not necessarily optimal) schedule will have as many threads as the number of execution classes in the largest concurrent scenario, and as many phases as there are concurrent scenarios, *i.e.*all ordered scenarios are satisfied without extra phases beyond those needed to accommodate the concurrent scenarios.

Our schedule generation algorithm cannot guarantee an optimal schedule but strives to minimize the length of a schedule. Our algorithm builds a schedule in multiple steps. In the first step, it trims the lengths of all concurrent scenarios to the number of available threads. This step is necessary because in our generated program structure, we can only have as many concurrent accelerator executions as we have program execution threads in our SoC. If a length X concurrent scenario has to be trimmed to Y threads, then we synthesize all length Y combinations of the X execution classes to be added to the list of concurrent scenarios.

In the second step, we iteratively build a schedule of all concurrent scenarios. In each iteration, we construct a search tree of depth M to from all possible schedules of M unscheduled concurrent scenarios. We search this tree for the schedule that satisfies the most number of ordered scenarios. We then remove the scheduled concurrent scenarios and satisfied ordered scenarios before proceeding to the next iteration. This step ends when we have scheduled all the concurrent scenarios. For N concurrent scenarios and a small, constant value of $M \ll N$, the computational complexity of our algorithm is $O(N^M)$. An optimal algorithm that attempts to find a complete schedule of concurrent scenarios that satisfies the most number of ordered scenarios will have a complexity of $O(N!)$. Our algorithm is a much less computationally intensive approximation of this optimal algorithm. Indeed, for $M = N$, our algorithm becomes the optimal algorithm.

In our third and final step, we schedule any remaining ordered scenarios. Starting with an empty list, for each ordered scenario, we first see if all or part of the current scenario is contained in the current list. A scenario may be matched fully in the body of the list or in part at the beginning or the end of the list. If the scenario is matched in part, then we add the remaining parts either to the beginning or end of the list. If the scenario is not matched at all, then we append it to the end of the list. Finally, we append the resulting list to the schedule of the first thread. Algorithm 2 summarizes our schedule generation algorithm.

```
Function FindSchedule(concurrent, ordered, numThreads, M)
    concurrent ← Trim(concurrent,numThreads)
    schedule ← empty list
    N ← number of scenarios in concurrent
    for $i = 0$ to $\frac{N}{M}$ do
        if concurrent is empty then
            break
        end
        /* concurrent and ordered are modified below          */
        subSchedule ← FindBestSubSchedule(concurrent,ordered,M)
        Append subSchedule to schedule
    end
    sequential ← empty list
    foreach scenario in ordered do
        if end of scenario partially matches the start of sequential then
            Prepend unmatched parts of scenario to sequential
        else if start of scenario partially matches the end of sequential then
            Append unmatched parts of scenario to sequential
        else if scenario does not match then
            Append scenario to sequential
    end
    Append sequential to schedule [0]
    return schedule
end
Function FindBestSubSchedule(concurrent, ordered, M)
    tmp ← find all possible length M schedules of scenarios in concurrent
    subSchedule ← the schedule in tmp that satisfies the most number of scenarios
      in ordered
    Remove scenarios in subSchedule from concurrent
    Remove satisfied scenarios from ordered
    return subSchedule
end
```

Algorithm 2: Schedule generation algorithm. The algorithm first tries to find a schedule of concurrent scenarios that satisfies several ordered scenarios. It then creates a sequential schedule for the remaining ordered scenarios. The Trim function trims concurrent scenarios that are larger than the number of threads.

3.6 Experimental evaluation

We tested AGARSoC's capabilities on two different SoC platforms. We used Soclib [110] to create a **Soclib ARM platform in SystemC** that has 3 ARMv6k cores with caches disabled, 3 accelerators, 3 memory modules, and 1 peripheral, all connected via a bus. The three accelerators are a quadrature phase shift keying (QPSK) modulator, a QPSK demodulator, and a finite impulse response (FIR) filter. All accelerators communicate with software running on the cores using a common communication middleware. Multi-threaded synchronization is implemented using locks, which are acquired by code patterns that utilize ARM's atomic pair of load-locked and store-conditional instructions (LDREX and STREX). Using Xilinx Vivado® Design Suite [3], we created a **MicroBlaze platform in RTL** that comprises three MicroBlaze processors with caches and local memories, six accelerator IP blocks, one shared memory module, and peripheral devices connected via AXI interconnects. The six IP blocks are a FIR filter, a cascaded integrator-comb (CIC) filter, a CORDIC module, a convolutional encoder, a fast Fourier transfer (FFT) module, and a complex multiplier. Synchronization primitives (*i.e.*, lock, barrier, semaphore) are implemented using a mutex IP.

For each platform, we created software suites that contain several patterns of accelerator execution, as summarized in Table 3.1. We designed our software suites to have a diverse set of accelerator usage patterns. After analyzing all of the Soclib software suites, AGARSoC identified 264 accelerator invocations, 130 observed concurrent scenarios, 315 observed happens-before scenarios. Similarly for MicroBlaze, it identified a total 358 accelerator invocations, 592 observed concurrent scenarios, and 502 observed happens-before scenarios. Figure 3.6 reports average compaction rates from AGARSoC's abstraction process of these observed scenarios. Better compaction rates are achieved for test suites that exhibit redundant interactions. We report the compaction rates for each suite group and the

Table 3.1 Software suites

suite group	# of progs	description
Soclib 1	9	sequential accesses with locks
Soclib 2	9	sequential accesses without locks
Soclib 3	9	concurrent accesses with locks
Soclib 4	9	concurrent accesses without locks
Soclib 5	18	combinations of the four above
μBlaze 1	7	no synchronization
μBlaze 2	8	lock, single-accelerator invocation
μBlaze 3	5	lock, multiple-accelerator invocation
μBlaze 4	7	barrier, redundant computations
μBlaze 5	13	semaphore

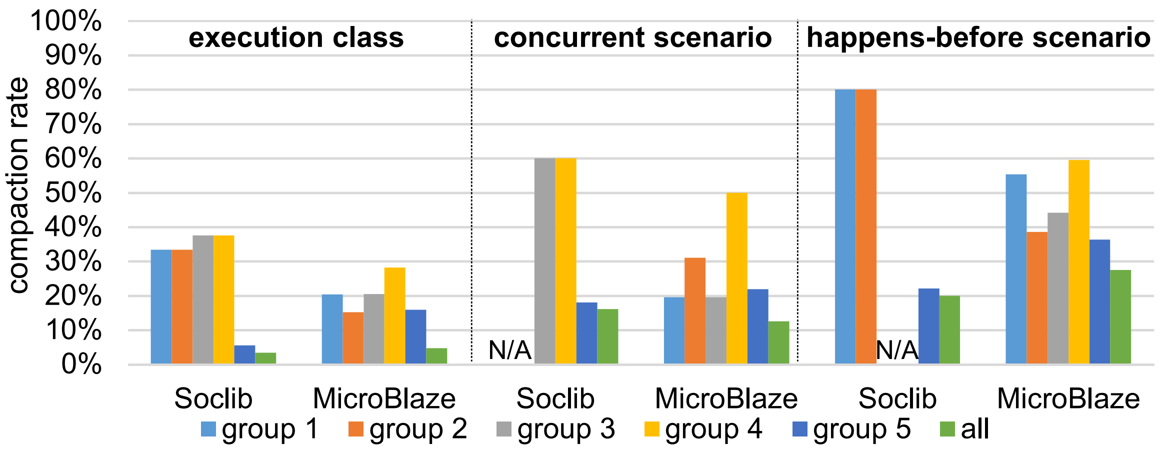


Figure 3.6 Compaction rates from abstract representation. The compaction of accelerator executions into unique instances of accelerator execution classes, concurrent scenarios, and happens-before scenarios for our two software suites. The compaction rate is measured as a ratio of the number of unique instances versus the total number of instances.

three categories in the abstract representation.

Note that by automatically identifying important scenarios, AGARSoC focuses verification to a small, albeit important, portion of the large space of possible interactions. For instance, for the 17 unique classes identified from the MicroBlaze suites, there can potentially be 680 concurrent scenarios. AGARSoC identified only 74 cases. The potential number of unique classes is much larger than 17, and it is only due to AGARSoC's analysis that we were able to narrow it down to 17. If we conservatively assume that the SoC allows 100 unique classes for instance, then 161,700 concurrent scenarios are possible.

We observed that the test programs generated from the compact abstract representation exercise 100% of the scenarios captured in the abstract representations. Figure 3.7 summarizes the reduction in simulation cycles achieved by running these tests instead of the original software suites. The AGARSoC generated tests simulate in only 8.6% of the time taken by the complete software suite for the Soclib platform and 38% for the MicroBlaze platform. However, the tests generated independently from the MicroBlaze software suites 1, 3, and 4 offer no execution time reduction, mainly due to the inability of the test generator to create minimal schedules to satisfy all scenarios. Suite 3 contains a program that is intentionally designed to exhibit accelerator patterns that push AGARSoC's schedule generator to the limit. Also, generated tests potentially exercise more happens-before scenarios than the original suites, due to all unobserved interleavings generated for synchronized accesses. Note that we are able to generate a more efficient schedule when the scenarios observed in all test suites are combined with other suites because we can get more compaction from scenarios that repeat across multiple programs. Compared to the MicroBlaze

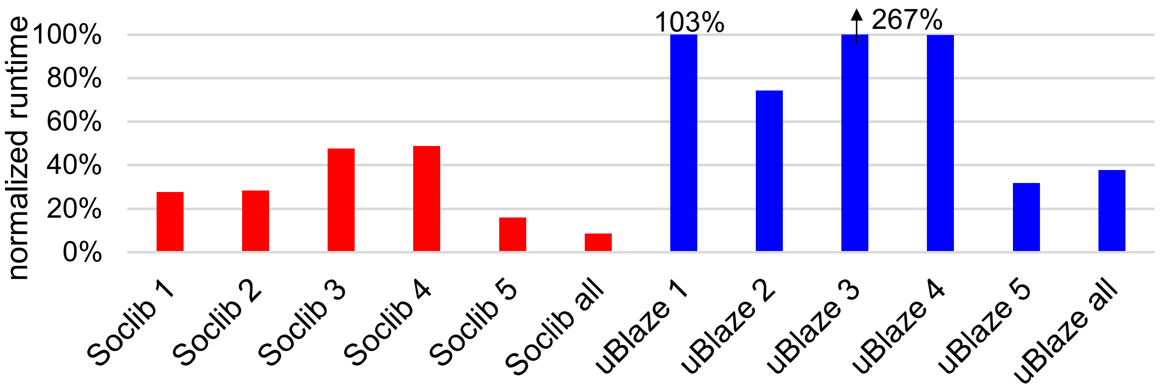


Figure 3.7 Normalized simulation runtime for each test group. While generated tests are usually shorter than the original test suites, MicroBlaze suites 1 and 3 show longer runtime, because of the inability of generating minimal schedules. However, the test program generated from all MicroBlaze suites does not exhibit this property.

suites, the Soclib suites execute more routines that are not related to accelerator execution and hence are not reproduced in the generated tests.

AGARSoC detects synchronized accelerator executions and generates all possible interleavings as discussed in Section 3.3.3. During execution, only a handful of these possible interleavings are observed. We can instruct AGARSoC's test generator to generate tests to exercise the unobserved interleavings. We evaluated this capability by generating separate tests for the unobserved scenarios in the MicroBlaze suite groups 2 and 3. Our generated tests successfully exhibited all of the interleavings that were yet to be observed.

3.7 Related work on the validation of SoCs

While there is a significant body of research into designing accelerator-rich SoCs [34, 99], there are only a handful of works that attempt to address the impending verification challenges. Hardware and software co-design and co-verification has been investigated extensively as a means to shorten time-to-market in designing SoCs [18, 39, 48]. Here, virtual prototyping platforms allow software engineers to develop and verify their software while hardware is still in development. AGARSoC extends beyond the current usage of HW/SW co-verification methodologies to enable early analysis of software with the purpose of guiding hardware verification. We believe that AGARSoC is the first attempt at leveraging software execution on high-level design models for generating coverage models and compact test programs.

Simpoints [51] have been widely used to find representative sections of single-threaded software for use in architectural studies. Ganesan *et al.* [49] propose a mechanism for generating proxy programs to represent multi-threaded benchmarks. These solutions analyze characteristics from execution traces of a program, collect representative behaviors of the program, and find test cases to represent the behaviors. Unlike AGARSoC, these are neither designed for analyzing concurrent interactions in accelerator-rich SoCs nor for guiding the verification process. AGARSoC is the first attempt at generating compact representations of multi-threaded programs for the purpose of functional verification.

Functional coverage metrics [92, 107] are widely used in the verification of processors, providing feedback on the (un)tested aspects of a design. This feedback is often used by verification engineers to direct testing, either by crafting tests or tuning random test generators [90] to "plug" coverage holes. Several solutions, under the name *coverage-directed test generation (CDG)*, have been proposed to automate the feedback process. These solutions mainly focus on ways to transform the coverage information into constraints / biases / directives for the random test generator. These include graph- [83, 84] and machine-learning [45, 22, 101, 65, 66] based approaches. Unlike AGARSoC, these solutions are geared towards single processor systems and do not attempt to automatically generate coverage models by identifying high-priority targets for verification. A few solutions have been proposed for generating coverage models from dynamic simulation traces [76], but they focus on low-level interactions obtained from RTL simulations. AGARSoC's coverage models focus on high-level accelerator interactions and are ready much earlier in the verification process.

3.8 Summary

In this chapter, we presented our APA-based solution to tackle the challenges of planning & test generation for accelerator-rich SoCs. Using the APA approach, we identified opportunities to automate the discovery of high-priority system-level accelerator interaction scenarios for verification. We observed that software for the SoC is developed concurrently with the SoC in a HW/SW co-design/co-verification flow. Such a flow provides an opportunity to automatically analyze the software to identify the interaction scenarios likely to be exercised by the software. The results of software analysis allows us to prioritize interaction scenarios for verification. Moreover, by utilizing a high-level model of the design to execute software for fast analysis, we approximate behavior of the actual SoC with the behavior of its high-level model.

The development of our solution, which we refer to as AGARSoC, was enabled by the synergistic application of automated software analysis, software-guided prioritization, and conservative approximation of design behavior: we analyze the execution of software on a high-level, approximate model of an SoC to automatically prioritize design behaviors for verification. AGARSoC generates coverage models that capture the prioritized design behaviors and test programs that exercise these behaviors. In addition to cutting down the effort required to identify scenarios for verification and representing them with coverage models, AGARSoC makes available fast and representative regression suites by generating tests that represent the original software suite. In our experiments, these generated fast tests could run up to 11x faster than the original software suite while hitting all of the coverage goals that AGARSoC discovered. The tools we developed can support the demands of different design efforts. Capturing design-specific knowledge is simple and easily portable to any verification effort. We were able to adapt AGARSoC to two quite different SoC platforms with minimal effort. In addition, AGARSoC can analyze and incorporate new software into the coverage model as it is developed, and generate tests that hit different coverage goals.

Planning and test generation, which is one of the three major functional verification activities discussed in Chapter 1, traditionally comprises tedious and manual tasks. Developed through our APA approach, AGARSoC slashes the time and effort spent on planning and test generation, bringing us one step closer to efficient functional verification for complex processor designs. In the following chapters, we present solutions that employ the APA approach to address challenges in the other two major functional verification activities.

Chapter 4

Addressing Test Execution and Bug Detection

In this chapter, we discuss our application of the APA approach to tackle the challenges of test execution and bug detection for complex designs. We recognize that acceleration and post-silicon validation are being widely adopted for fast test execution. We identify and prioritize two main bug detection challenges that limit the benefits of fast test execution on acceleration and post-silicon platforms. Firstly, there is a pressing need for efficient mechanisms to deploy existing software checkers on acceleration platforms to eliminate the current inefficient practice of recording signals on the platform and transferring them to a host computer for checking. We realized that by prioritizing and approximating some of the functionality of these software checkers, we can realize low-overhead hardware checkers that can execute on the acceleration platforms along with the designs. Secondly, checkers for some system-level interactions, which are beyond the reach of software-only simulation, have to be designed from the ground up for deployment on simulation acceleration and post-silicon validation platforms. We identify shared-memory interactions as the high-priority system-level verification targets. We discovered opportunities for leveraging existing resources in the design and on the validation platforms to implement efficient shared-memory interaction checkers on acceleration and post-silicon validation platforms.

We designed three APA-based solutions to enable efficient checking on acceleration and post-silicon validation platforms. Our first solution studies the types of software checkers in use today and finds ways to approximate their functionality for automated hardware checking on acceleration platforms. A direct mapping of software checkers to hardware would result in large hardware structures whose implementation on acceleration platforms slows accelerated simulation down. Our approximations tradeoff bug detection accuracy for size reductions in the hardware implementations of the checkers. In our experiments, we have found approximation techniques that can reduce the hardware implementation size by 59% with only a 5% loss in accuracy. Our second solution tries to find a sweet spot between performance gains from porting checkers to hardware and losses in bug detection accuracy. Instead of mapping all complex software checkers into hardware, we develop a synergistic approach that lets some checkers remain in software while minimizing the data transfer between the acceleration platform and the host computer running the checkers. We have found synergistic approaches in our experiments that reduce the data transfer by 32% with a 6% logic overhead and no loss to accuracy. Finally, we developed a solution that enhances a multicore design to enable the detection of bugs in the implementation of the design's memory consistency model. Our enhancements repurpose portions of the design's L1 caches to log memory operations, which are then analyzed by software to detect bugs. In our experiments, we found that we were able to detect a wide range of memory consistency bugs.

The rest of this chapter is organized as follows. Section 4.1 provides a background on test execution and bug detection on acceleration and post-silicon validation platforms. Section 4.2 discusses the benefits and accuracy tradeoffs of the APA approach employed by our solutions. Sections 4.3 - 4.11 discuss our three solutions in detail and associated related works. We end by providing a summary of our work in Section 4.12

4.1 Background on test execution and bug detection

In a software-based test execution environment, a software simulator executes tests on the design while a mix of embedded checkers monitors the correctness of the design's response. In simulation, the ability to observe and control any signal within the design allows close monitoring of the design's activity and provides comprehensive diagnosis capabilities. Unfortunately, this methodology is severely limited by the vast complexity of modern digital designs. Longer and more complex regression suites are required to explore meaningful aspects of a design and, to add insult to injury, simulation speed degrades with increasing design size. As a result, significant portions of a design's state space are left unexplored during software simulation. Alternative test execution platforms, namely accelerated simulation [87, 25], emulation [17, 80] and post-silicon validation [5, 21], are widely used because of the performance boost they can provide.

There is a strong desire for and a trend towards using simulation accelerators as drop-in replacements for software logic simulators. To target these specialized acceleration platforms, a design must first be synthesized into a structural netlist, which is then mapped to the platform. The accelerator exposes the design it is simulating to the complex checkers in the software verification environment, which, presently, execute on separate host computers that interface with the simulation accelerators. Selected signals are logged on the platform itself and periodically off-loaded to the host to be checked for functional correctness. Often the logging and off-loading activities become the performance bottleneck of the entire simulation.

The difficulty of bug detection is even more pronounced in post-silicon validation platforms. Designers often implement custom hardware structures for checking correctness on-chip and for logging design signals to later be offloaded and analyzed on a host computer [89]. However, since these structures compete for transistors and chip pins with actual design features, the number of design signals they can monitor and expose to the software checkers are severely limited. There is a large body of research on selecting critical design signals, which contain the most amount of information about a design's operation, for monitoring [20, 70, 30]. In addition to hardware checking and logging, post-silicon validation efforts also rely on self-checking tests that check for errors in execution outcomes [7].

Both acceleration and post-silicon validation platforms are used to validate system-level interactions that are impossible to reach with software simulation alone. Validating these system-level interactions in modern multi-core processor designs requires the monitoring and analysis of multiple design signals from different subsystems over several cycles. The multiple cores and memory subsystem must work together to correctly preserve system-level properties as defined by the design's memory consistency model [104], *i.e.*, read and write operations to shared-memory by multiple cores must obey the ordering requirements of the consistency model. State-of-the-art out-of-order cores rely on aggressive performance optimizations that reorder instructions, resulting in subtle corner cases for shared-memory operations that manifest over thousands of cycles. Moreover, the shared-memory subsystem is typically realized through a hierarchy of connected memory elements, which themselves might reorder memory operations for increased performance. Checkers that validate the system-level properties of the memory consistency model should monitor the ordering of memory operations and validate them against the memory consistency model.

4.2 APA: benefits and accuracy tradeoffs

The first two solutions presented in this chapter adapt complex software checkers for efficient bug detection on acceleration platforms. The goal of these solutions is to minimize the performance impact of robust checking on acceleration platforms. Due to the hardware-specific nature of acceleration platforms, a straightforward mechanism does not exist for on-platform software checkers. Signals are logged by on-platform recording structures and periodically offloaded to a host computer for checking. Unfortunately, in addition to limited storage available in the recording structures, routing design signals to these structures for recording adds a performance overhead: the more signals recorded, the slower the simulation [29]. As a way around this problem, the software checkers can be implemented as hardware structures that are simulated alongside the design. A direct mapping of software checkers into hardware, however, will result in prohibitively large structures that are sometimes larger than the design units they are supposed to check. Again, due to platform limitations, the larger the hardware to be simulated, the slower the simulation becomes.

In our solutions, we relax the accuracy requirements of our checkers to limit both the number of signals recorded on platform and the sizes of the hardware checker structures adapted from the software checkers. To this end, we explore mechanisms to approximate checker functionality for efficient hardware implementation, identify and implement fully checkers that can be efficiently implemented in hardware, and to implement in-hardware compression to minimize the number of signals that are recorded and offloaded to those checkers that are inefficient to implement in hardware. In our experiments, we found different approaches, each with its own benefits and tradeoffs. Some of our approaches result in significant reductions in both the size of the checking hardware and the number of recorded signals for little or no losses to checking accuracy. Other approaches result in smaller savings for larger losses in accuracy.

Our third solution implements a mechanism for validating memory consistency model implementations. Our mechanism trades accuracy off in order to enable efficient and robust checking that would otherwise not be possible. Our mechanism is designed with other checking mechanisms in mind: it assumes the correctness of certain properties that we know are validated by complementary mechanisms. We therefore miss memory consistency bugs that may arise from violations of those properties not checked by our solution. Our solution may sometimes wrongfully detect a non-existent bug for rare interaction scenarios. The impact of these rare false positives on the overall accuracy of our solution is insignificant.

4.3 Approximating checkers for simulation acceleration

In this section, we introduce the concept of "approximate checkers" illustrated in Figure 4.1. Approximate checkers trade off logic complexity with bug detection accuracy by leveraging insights to approximate complex software checkers into small synthesizable hardware blocks, which can be simulated along with the design on an acceleration plat-

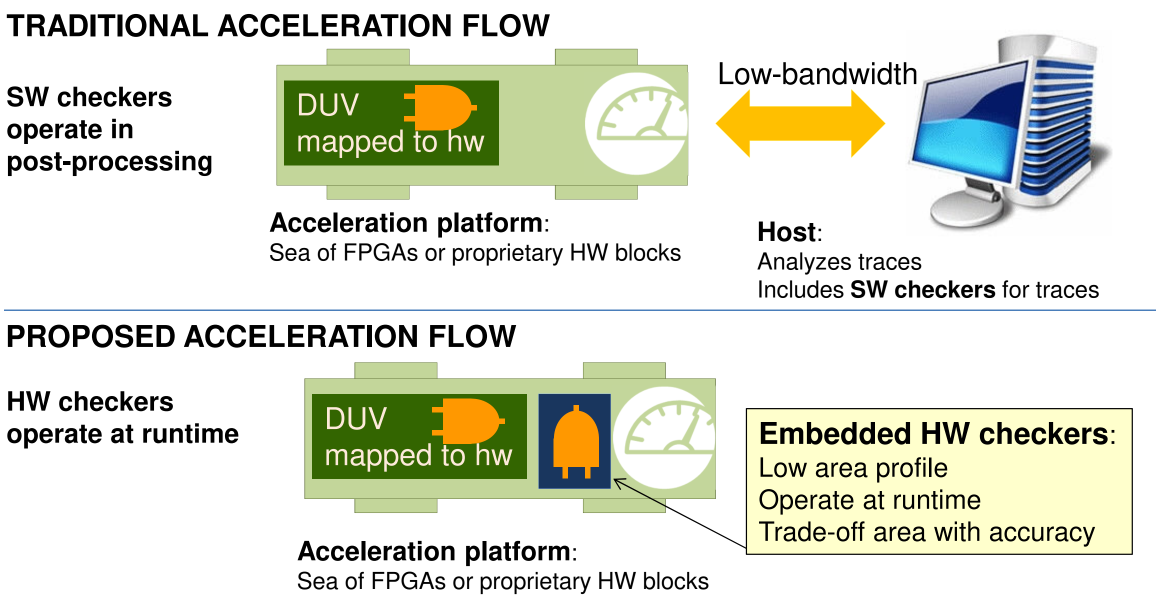

TRADITIONAL ACCELERATION FLOW
SW checkers operate in post-processing
DUV mapped to hw
Low-bandwidth
Acceleration platform: Sea of FPGAs or proprietary HW blocks
Host: Analyzes traces Includes SW checkers for traces
PROPOSED ACCELERATION FLOW
HW checkers operate at runtime
DUV mapped to hw
Embedded HW checkers: Low area profile Operate at runtime Trade-off area with accuracy
Acceleration platform: Sea of FPGAs or proprietary HW blocks

Figure 4.1 Proposed solution overview. Traditional acceleration solutions rely on off-platform checkers running on remote host, requiring the transfer of logged data via a low-bandwidth link. Our solution proposes to embed checkers on platform for high performance checking. Our checkers must have a small logic profile while maintaining detection accuracy comparable to their host-based counterparts.

form. We propose a number of approximation techniques to reduce the logic overhead of the checker while preserving most of the checking capabilities of its software counterpart. We provide a taxonomy for common classes of checkers; we use these classes to reason about the effects of our approximation techniques in the absence of specific design knowledge. The approximation process may lead to the occurrence of false positives, false negatives and/or delays in the detection of a bug. To properly analyze these effects we provide metrics to evaluate the quality of an approximation, and present a case study to demonstrate our proposed solutions.

4.3.1 Checker taxonomy

Our experience with several designs suggests that most checkers are intended to verify a similar set of design properties. Based on this observation, we present the following checker classification scheme:

Protocol Checkers: verify whether the design interfaces adhere to the protocol specification. An example is a checker that monitors the request-grant behavior for a bus arbiter checking that the arbiter sets the grant signal within a fixed number of cycles from receiving a request, or that it never issues a grant when some other requester owns the bus.

Control Path Checkers: verify whether the flow of data within the design progresses as intended. An example control path checker is one that monitors I/O ports of a router to check whether a packet accepted at an input port is eventually transmitted through the correct output port.

Datapath Checkers: verify whether data operations produce expected results. A datapath checker for an ALU, for example, verifies that the result of an addition operation is the sum of its operands.

Persistence Checkers: verify whether or not data items stored in the design remain uncorrupted. For example, a checker for a processor's register file may check that the contents of each register never change except upon a write command.

Priority Checkers: verify whether specified priority rules are held. A priority rule sets the order in which certain operations are to be performed, usually selected from some queue. Consider, for instance, a unified reservation station in an out-of-order processor that must prioritize addition operations over shift operations. A priority checker for this unit verifies that no shift operation is issued when additions are waiting.

Occupancy Checkers: verify that buffers in the system do not experience over- or underflow. For example, an occupancy checker may verify whether a processor dispatches instructions into its reservation station only when there is space available.

Existence Checkers: verify whether an item is present in a storage unit. In a processor cache, for example, an existence checker could verify whether the tag for cache access hit actually exists in the tag array of the cache.

4.3.2 Approximation techniques

Checkers may need to maintain information about the expected internal design states for extended periods. Thus, direct mapping of a software checker to hardware, when even possible, often leads to an extremely complex circuit block, possibly as large or larger than the design itself. Often the checker hardware could be simplified, targeting only functionality rather than also performance or power. For instance, a simple ripple-carry adder is sufficient for a datapath checker that verifies a Kogge-Stone adder. Below, we propose a number of approximation techniques to minimize the complexity of embedded checkers.

Boolean Approximation: can be used to reduce the complexity of any combinational logic block. The "don't care" set of the Boolean function implemented by the block can be augmented by simply changing some outputs from 1 or 0 to don't care (indicated by X). By appropriately selecting which combinations become don't cares, it is possible to

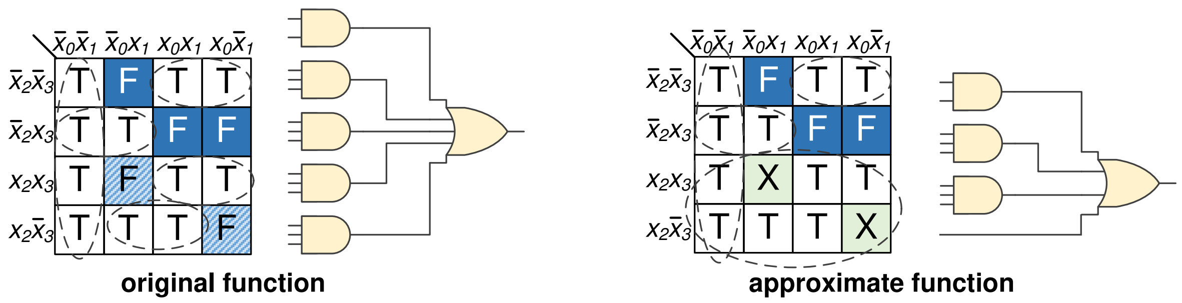


Figure 4.2 Boolean approximation. Replacing some output combinations with don't cares reduces the 2-level function implementation.

greatly reduce the number of gates required for the function. An example is shown in Figure 4.2, where two minterms are set to don't care (highlighted by hashing), reducing the 2-level function's implementation from 6 to 4 gates. Boolean approximation often allows great reductions in circuit complexity with a minimal amount of don't care insertions. The transformation may lead to false positives or negatives for the checker: a 0 approximated to a 1 would lead to a false positive, and vice versa. Note that it is possible to apply the technique to a sequential logic checker by unrolling the combinational portion and then approximating the logic block obtained.

State Reduction: Embedded checkers may include storage elements for a wide variety of purposes. State reduction eliminates some of the non-critical storage to simplify both the sequential state and the corresponding combinational logic. Examples of non-critical storage are counter bits used for checking timing requirements of events at fine granularity and flip-flops for storing intermediate states in a checker's finite state machine (FSM). Figure 4.3 shows a portion of a protocol checker's FSM that verifies whether a signal is set for exactly one cycle. The "DELAY" state can be removed and the check is then performed in all the states following the "NEXT" state.

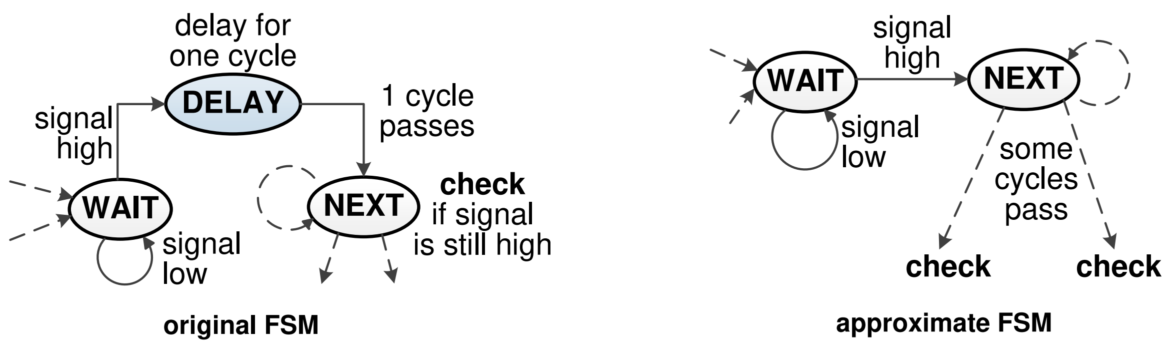


Figure 4.3 State reduction approximation for a protocol checker. The delay state is removed leading to an approximate checking of events timing (duration of signal high).

Even though the checker can no longer account for the delay, it can still verify that the signal is only set for a finite number of cycles. This technique works well for checkers with reference models since the reference model's response can arrive before the design's response. An approximation that removes delays may allow checkers to compute estimates before the design's response. State reduction may also introduce false detections, either positive or negative.

Sampling and Signatures: The width of a datapath affects many aspects of a design, including the width of functional units' operands and of storage elements. To reduce the amount of combinational logic and storage required to handle wide data, an approximate checker can operate either with a subset of the data (sampling) or a smaller size representation (signature) derived from the data. Bit-fields, cryptographic hashes, checksums, and probabilistic data structures are valuable signature-based approximations, trading storage size for signature computation. A checker for an IPv4 router design, for instance, does not need to track all the data bytes of packets entering the system. In most cases, storing the control information and an XOR signature of the data is sufficient for checking purposes (see Figure 4.4). This approximation may result in both false positives and false negatives. As we show in Section 4.5, one can reasonably estimate the impact of these techniques, given the probability distribution of the data payloads and the nature of the design.

The checker classes presented in Section 4.3.1 allow us to evaluate our proposed approximation techniques on a number of checkers without concern of the DUV's implementation details. Table 4.1 summarizes the results of this evaluation. Note that Boolean approximation and state reduction are general techniques applicable to all the classes of checkers. In contrast, sampling and signatures have limited scope as they are only appropriate for situations where monitoring a subset of possible events/combinations enables a detection.

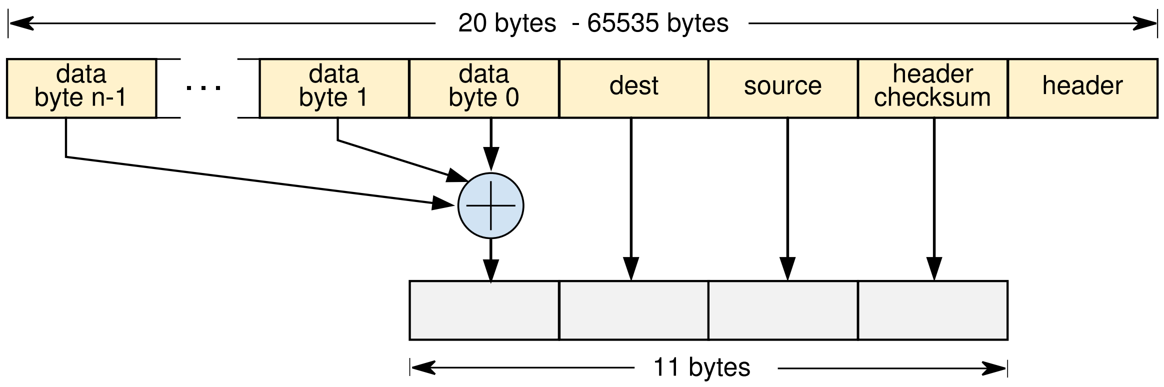


Figure 4.4 **Signature approximation** for an IPv4 packet. A packet can be uniquely identified with high probability by using just a few bytes.

Table 4.1 Approximation to checker matrix. Boolean approximation and state reduction are generic methods applicable to all.

	Boolean	**State Reduction**	**Sampling**	**Signature**
Protocol	✓	✓		
Control Path	✓	✓	✓	✓
Datapath	✓	✓	✓	✓
Persistence	✓	✓	✓	✓
Priority	✓	✓	✓	✓
Occupancy	✓	✓		
Existence	✓	✓		✓

4.4 Measuring approximation quality

An approximate embedded checker may be more relaxed or more restrictive than its original software counterpart. Thus, depending on the time and origins of a bug's manifestation, detection in the approximate checker may occur (or not) as in the original checker – *true positive (TP) or negative (TN)*; the bug may be missed by the approximate checker only – *false negative (FN)*; or the approximate checker may falsely flag the occurrence of a bug – *false positive (FP)*. In this context, it is important to evaluate the relative detection capability of an approximate checker with respect to the original. A good approximate checker should have a small rate of false positives and negatives. If the post-simulation diagnostic methodology is capable of ruling out false positives, then a high false positive rate would not be a critical issue for the approximate checker. We propose to evaluate the quality of an approximate checker with two common statistical metrics deployed in the evaluation of binary classification tests [36]: accuracy and sensitivity. For proper classification, each test has to be run either until a bug is detected (correctly, or wrongly) or until completion.

Accuracy ($\frac{TP+TN}{TP+FP+FN+TN}$) measures how accurate an approximate checker is in reproducing the results of the original checker. A high value exhibits only a few false positives and negatives.

Sensitivity ($\frac{TP}{TP+FN}$) evaluates the ability of a checker in detecting actual bugs (true positive rate). A high value of sensitivity indicates that most bugs that are detected by the original checker are also detected by the approximate checker.

4.4.1 Tuning the approximation tradeoffs

The accuracy and sensitivity of an approximation technique are dependent on several factors, including the input test vectors, the type of DUV, the class of the approximated checker, and the nature of the expected bugs. While we can not offer generic approaches for estimating the tradeoffs involved with an approximation, we can outline some guidelines for making reasonable choices.

Firstly, understanding the distribution of input test vectors goes a long way in tuning approximations. For example, if the changes in the input test vectors are localized to a few bits, then approximations that do not impact those bits offer hardware savings with minimal impacts on accuracy and sensitivity. If information about the input test vectors is not available, then it is best to assume that every bit in the input test vector can change with equal probability. Analysis performed with this assumption is likely to provide an estimate for the average case.

Secondly, Boolean approximation and state reduction can have significant impacts on accuracy and sensitivity for usually smaller savings in hardware size. However, due to the nature of these approximations, their impacts should be obvious to the engineers applying them. These techniques are best reserved for approximating away functionality that has already been thoroughly utilized during software simulation.

Finally, signature approximations that reduce data storage size but still retain some information about the data are likely to have lower impacts on sensitivity and accuracy. For these techniques, their impacts can be estimated by estimating the collision probability for the mechanism that generates signatures.

4.5 Experimental evaluation of checker approximation

We evaluated our approaches on a verification environment for a router design. `Router` is a VHDL implementation of a 4x4 router used internally at IBM for training new employees. It can accept variable-length packets from any of its four input ports and routes them to one of its four output ports based on entries in its statically configurable routing table. This is accomplished through a set of request, grant, ready, ack, nack, and command signals that connect to other routers or a configuration host. All `router` packets are composed of a source address byte, a destination address byte, up to 60 bytes of data, and a parity byte (bitwise XOR of all the bytes in the packet). Up to 16 incoming packets, not exceeding a total of 256 bytes, can be buffered at each of `router`'s input ports. `Router` rejects packets whose destinations do not exist in the routing table, whose parity is bad, or when there is not enough space in the input buffers. In addition, the interface signals must obey timing constraints as provided in the protocol specifications.

`Router` comes with a complete software checker environment designed in *e*. To in-

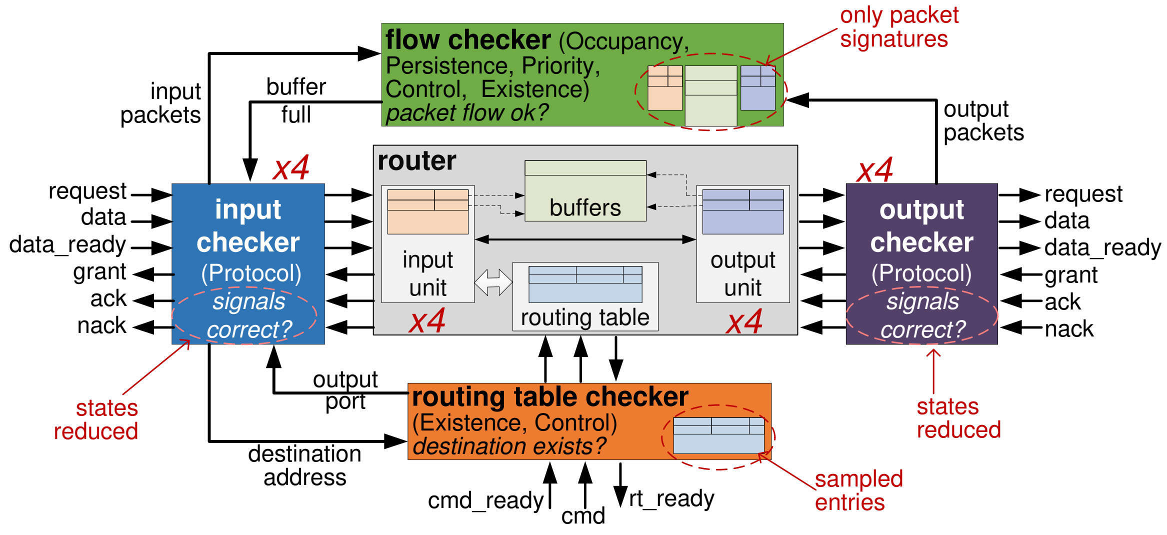


Figure 4.5 Hardware checkers for `router`. The checkers track packets (several classes), I/O signals (protocol), routing table (existence).

vestigate the effects of approximation, we manually developed an optimized but complete hardware version of the *e* environment. The architecture of the checker design is shown in Figure 4.5. The properties verified by the checkers are summarized below, based on the classes presented in Section 4.3.1: a **protocol checker**, monitoring for correct timing and validity of a number of control signals (req, gnt, rdy, ack, nack); a **control path checker**, tracking validity of outgoing packets; a **datapath checker**, checking correct parity computation; a **persistence checker** to detect data corruption in input buffers and routing table; a **priority checker**, monitoring correct ordering of outgoing packets and output port mappings; an **occupancy checker** checking for input buffer overflows; and an **existence checker**, tracking the existence of entries in the input buffers and the routing table.

There could be situations in the `router` verification environment where a single checker implementation verifies multiple properties or vice versa. For example, a checker verifying that a packet transmitted through an output port is valid (control path) also checks that the routing table provides the correct mapping for the destination address (priority and existence). The input and output checkers in Figure 4.5 mainly consist of protocol checks. The flow checker is responsible for maintaining `router`'s buffers and most of the control path, persistence, priority, and occupancy checkers are within this module. It lets the input checker know if the buffer for an input port is full. The routing table checker verifies the existence of output port mappings in the routing table, and predicts whether `router` would accept or reject a packet.

The state reduction, sampling, and signature techniques were investigated for this case study. Using the state reduction approach, a combined total of 40 states and 88 counter bits were eliminated from the input and output protocol checker units. This approximation is unlikely to cause false positives but might produce false negatives. The routing table is approximated with a sampling technique where only the 4 most significant bits (out of 8) of the destination and the mask are stored. This reduces storage requirements of the routing table by 44%. The existence check that utilizes the reference table should still be able to correctly detect when a destination actually exists in the routing table (true positive). However, due to the loss of half the information content, it may wrongly assume that a destination exists in the table when it does not (false positive). For a purely random input pattern, this existence checker will raise false positives half of the time. This fraction could be lower if the test inputs to the routing table were constrained to follow a specific pattern.

The signature approximation applied to the flow checker includes only the length of the packet (6 bits), the source and destination addresses and the parity, and it has a collision probability of 2^{-30}; sufficient for our design, which can have at most 64 packets in flight. This approximation alone corresponds approximately to 31% of the `router` logic. Reducing the packet signature further by removing source and destination addresses saves us about 32 bytes of storage overall while increasing the collision probability to only 2^{-14}.

We conducted multiple evaluations of our solution, by comparing results from our approximate checkers with results from their non-approximated, baseline hardware counterparts. We injected multiple bugs of varying complexity and location into our case study designs one at a time; for each bug, we run separate simulations with the approximate and baseline checkers. Each simulation could terminate either because a bug was detected or because the test ran to completion. Each bug detection (or lack thereof) by an approximate checker was compared to the corresponding detection by its baseline counterpart and then labeled as a true positive, true negative, a false positive or false negative.

Table 4.2 Bugs injected in `router` sorted by decreasing difficulty.

id	**checker**	**description**
under	flow	Input packet list underflow
2sent	flow	Packets sent twice
badin	input	Packets with bad parity accepted
ovr	output	Memory overwrite
ord	routing table	Wrong routing table search order
badout	output	Packets sent with bad parity
tim2	output	Wrong timing for data_ready signal
exist	routing table	Packet accepted when dest. not in routing table
tim1	output	Wrong timing for request signal
lock	input	Routing table not locking itself when searching

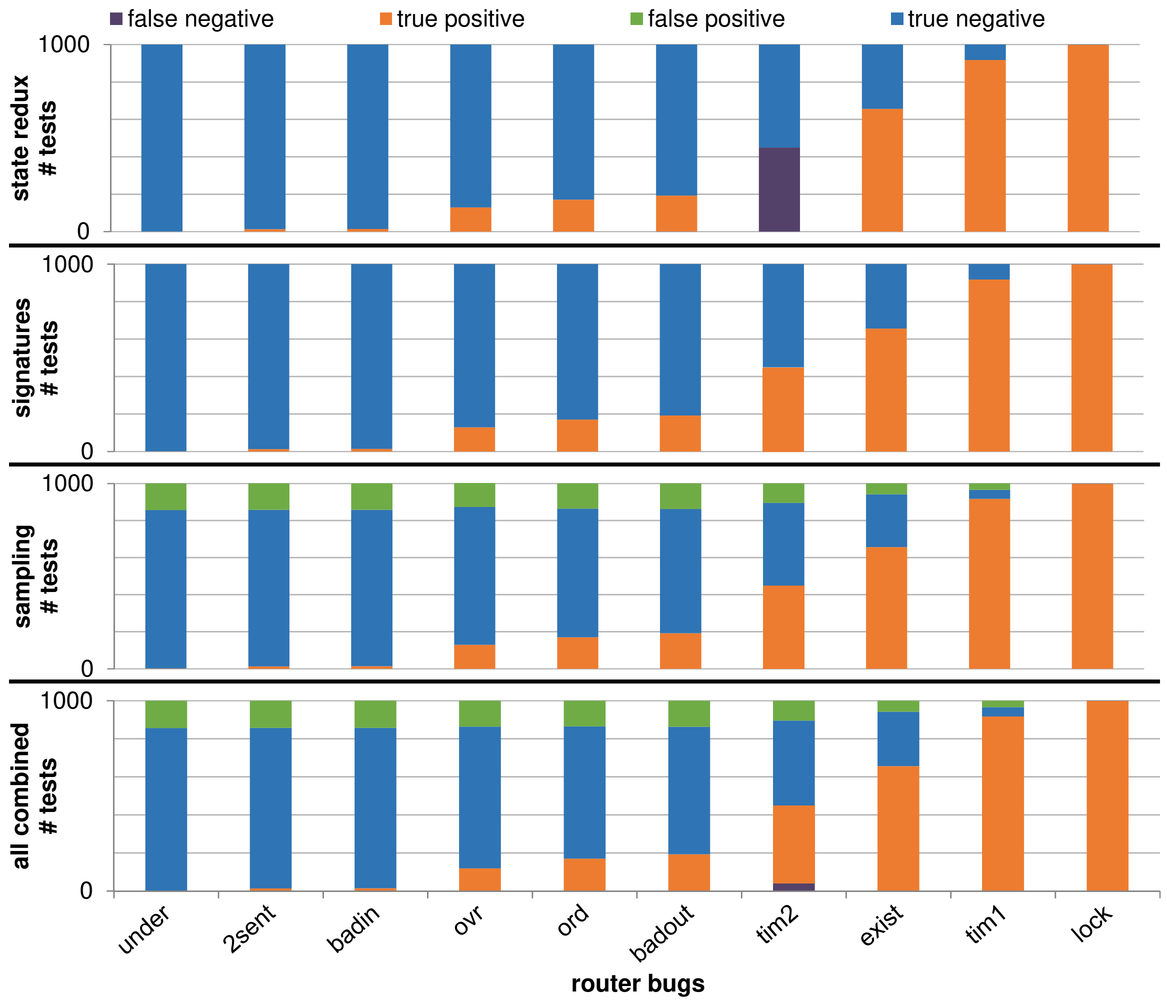


Figure 4.6 Detection of `router` bugs. Different simulations were run for individual approximations and all combined.

Table 4.2 describes the injected bugs. A total of 1,000 tests were executed for each design variant obtained by injecting a different bug. In most cases, test stimuli were randomly generated and uniform for all design variants. For the few hard-to-sensitize bugs, we inserted manually-crafted input sequence snippets at random times in the simulation.

Figure 4.6 shows the breakdown of the test outcomes: for each bug, we report how many tests resulted in each outcome type. The average effects of approximations on accuracy and sensitivity are shown in Table 4.3. The different applied approximations were tested both separately and combined together. Notice that the signature approximation technique produces only true positives and negatives, and thus has a perfect accuracy of 1. The state reduction technique has perfect accuracy for all bugs except *tim2*: this bug is never detected as it requires accurate timing checks, removed by the approximation. However, losing some cycle accuracy does not affect all timing bugs, as *tim1* is still detected.

Table 4.3 Average approximation accuracy and sensitivity.

metric	**`router`**	**`router`**	**`router`**	**`router`**
	st. redux	signature	sample	all
accuracy	95.5%	100%	45.8%	45.3%
sensitivity	80%	100%	100%	89.1%

Uniform degradation is observed for the sampling approximation on the routing table, by introducing false positives for all bugs. Whenever the routing table checker incorrectly assumes that an entry exists in the table, an execution mismatch occurs. To study the impact of input distribution on the approximate routing table checker, we conducted a separate test non-uniform packet address masks to imitate real-world network traffic. We observed that when masks with a higher probability of zeroes in the most significant bits are supplied as inputs, the number of false positives is significantly reduced.

Based on our findings, we make some general observations: i) sampling approximations give poor results if they do not account for the nature of the sampled data. ii) When multiple approximations are combined, the worst-performing one dominates. This is not a characteristic inherent to any approximation technique, but rather the result of the application of a technique to a checker that is not well suited for it. iii) Inaccuracies manifest in different ways for different types of designs and bugs.

Finally, we evaluated the logic complexity of the baseline embedded hardware checkers and compared against our approximate checkers. To this end, we synthesized the router design and the hardware descriptions of the baseline and approximate checkers using Synopsys' Design Complier, targeting the technology-independent GTECH library. Since the process of mapping a digital design onto an acceleration platform is very specific to the platform being used, we simply considered the total number of logic blocks generated as a reasonable indicator of logic size. Table 4.4 shows that the signature-based approximation in the `router`'s flow checker results in quite significant size reduction, at no impact to accuracy (see second block in Figure 4.6).

Table 4.4 Logic complexity of approximate checkers. Overall checker overhead for `router` reduces from 57% to 23%.

unit	**technique**	**original**	**approximate**	**reduction**
		(#blocks)	(#blocks)	(%)
`router` input+output	state redux	3,764	2,664	29.2
`router` flow	signatures	146,910	56,983	61.2
`router` routing table	sampling	2,526	1,835	27.4
`router` checker	**combined**	**153,200**	**61,482**	**59.9**

4.6 Hybrid checking on acceleration platforms

In our checker approximation approach, we strive to map software checkers completely onto acceleration platforms to eliminate platform-to-host communication. Fully-embedded approximate checkers are diametrically opposite to fully-software checkers. We recognize that the spectrum of efficient checking on acceleration platforms also includes synergistic software and hardware checking mechanisms. Thus, we present another solution, illustrated in Figure 4.7, that exploits embedded logic and data tracing for post-simulation checking in a synergistic fashion to limit the associated overhead. Embedded logic can be used for synthesized local checkers that enable checking on acceleration platforms with minimal logic footprint where possible and for compressing the traced data to reduce the number of bits that have to be recorded and offloaded to a software checker. Checkers that have a small logic footprint when synthesized can be embedded and simulated with the design – we call these "local assertion checkers" On the other hand, checkers that must adopt the "log and then check" approach because of their complexity, compress activity logs relevant to the check using on-platform compression logic and then perform the check off-platform – we call these "functionality checkers"

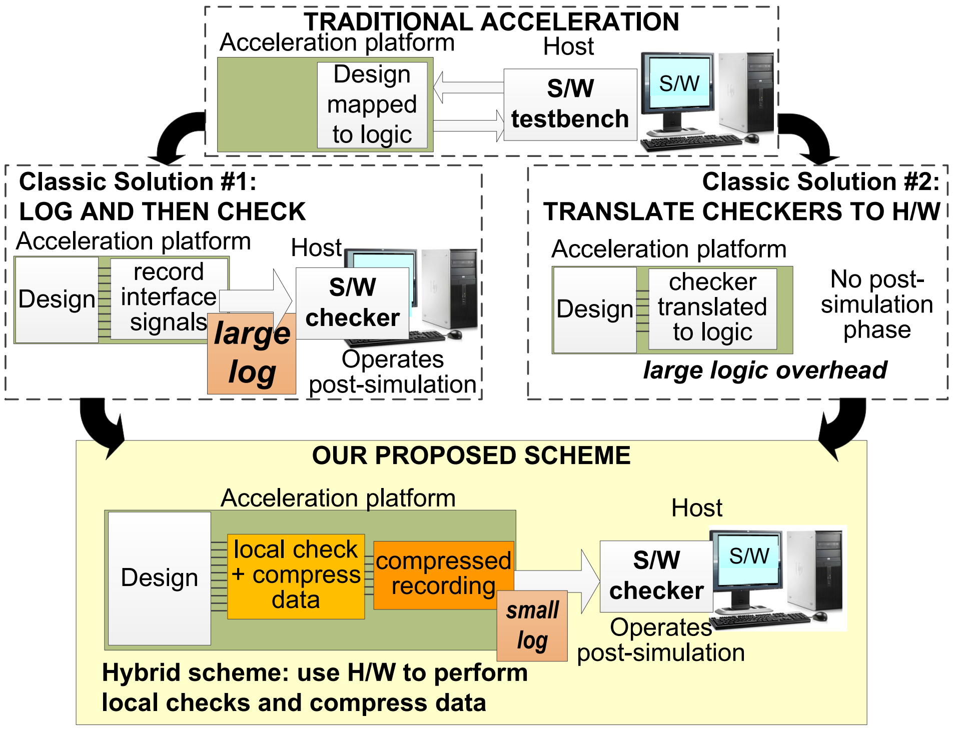


Figure 4.7 Hybrid checker-mapping approach. We use a mix of embedded logic and data-compression techniques for data that must be logged. For the latter, a software checker analyzes the logged data after simulation.

We demonstrate our methodology on the software verification environment for a modern out-of-order superscalar processor design. This environment contains a number of microarchitectural-block checkers and an architectural checker, which together provide a setup representative of real world experiences. Adapting these checkers to an acceleration environment provides a challenge that is representative of those faced by verification engineers working in the field. We provide insights on how the checking activity for a microarchitectural block can be partitioned into local assertion checkers and functionality checkers. We classify essential blocks of an out-of-order processor from a checking perspective. Finally, we demonstrate novel techniques to reduce the amount of recorded data with the aid of lightweight supporting logic units, leading to only a marginal accuracy loss.

4.6.1 Synergestic checking approach

The most common method of checking microarchitectural blocks involves implementing a software reference model for the block. The design block updates a scoreboard during simulation, which, in turn, is checked by the software reference model [113]. This approach is viable in software simulation, but not directly applicable to acceleration platforms. Since acceleration platforms only allow simulation of synthesizable logic, one option is to implement the reference model in hardware; however, this option is often impractical. Another option is to record all signal activity at the microarchitectural blocks' I/O and cross-validate it against a reference model maintained in software after the simulation completes. However, that solution requires recording of a large number of bits in each cycle, leading to an unacceptable slowdown during simulation. Thus, neither solution outlined scales well to complex microarchitectural blocks. We present a novel, two-phase approach that solves this problem by making synergistic use of these two methods while avoiding the unacceptable overheads of both. It may be argued that we do not always need fine-grain checking capabilities on acceleration platforms: engineers may turn off signal tracing during an overnight run and only record the final outcome of the test, while fine granularity checking requiring extensive tracing is enabled only on sighting of a bug. Unfortunately, this approach prevents accelerated simulation to obtain the same-level of coverage as software-based simulation, since a buggy behavior can often be masked in the final test outcome. Moreover, in this approach debugging is still performed at a compromised simulation performance, while the proposed approach achieves both higher coverage as well as debugging support without sacrificing performance.

The first phase performs cycle-by-cycle checking using embedded local assertion checkers on-platform. It focuses on monitoring the correctness of the target block's in-

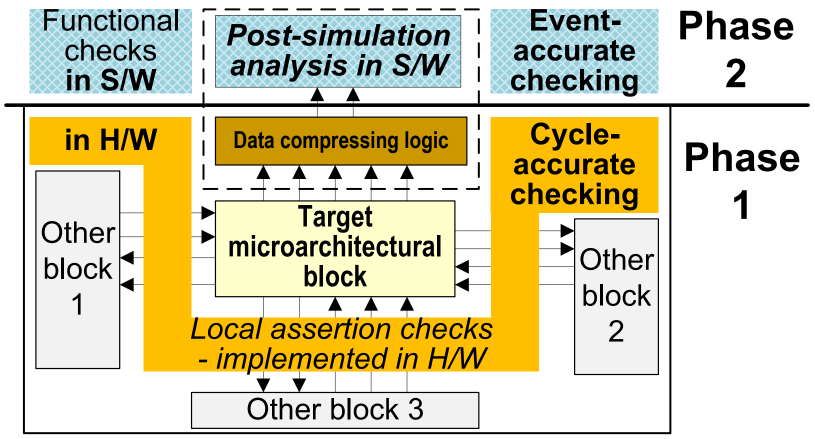


Figure 4.8 Two-phase checking. Local assertion checks are performed by embedded logic in a cycle-accurate fashion, while microarchitectural events are logged and compressed on platform with additional logic, and then evaluated for correctness by an off-platform functionality checker after simulation.

terface activity and local invariants, which can be expressed as local assertions. During this phase, we also log and compress (with embedded logic) relevant microarchitectural events to enable off-platform overall functionality checking. In the second phase, the logged data is transferred off-platform and compared against a software model to validate the functional activity of the block. This approach is illustrated in Figure 4.8. The main idea behind this two-phase approach is the separation of local assertion checking from functionality checking for a block.

Local assertion checking often requires simple but frequent monitoring. Hence it must be performed in a cycle-accurate fashion and can often be achieved via ***low overhead embedded logic***, with minimal platform performance loss. Indeed, most local assertions are specified over a handful of local signals and can be validated in an analysis' windows of a few cycles (*e.g.*, a FIFO queue must flush all of its content upon receiving a flush signal, FIFO head and tail pointers should never cross over, *etc.*). These checkers do not require large storage of intermediate events; rather they must maintain just a few internal states to track the sequential behavior of relevant signals.

In contrast, **functionality checking** can be carried out in an event-accurate fashion. From a functionality perspective, most microarchitectural blocks can be abstracted as data structures accessed and modified through events of read and update operations. The main goal of functionality checking is then to verify the legality and consistence of operations on this data structure. In addition to monitoring the data associated with events, an event-accurate checker also needs to perform bookkeeping of the internal contents of the microarchitectural block, and thus an embedded logic implementation would be grossly

Block	**Local assertion (cycle-accurate, simple, low level)**	**Functionality (event-accurate, complex, high level)**
Reorder Buffer (ROB)	Do ROB head and tail pointers ever cross each other?	Do instructions retire with correct destination register values?
Reservation Station (RS)	Are the contents of the RS flushed 1 cycle after an exception?	Are instructions released with correct operand values?
Load-Store Queue (LSQ)	Are loads whose address has been computed sent to the data cache within a bounded number of cycles (unless flushed)?	Do store operations send correct data to the correct address?
Map Table (MT)	Are all register values marked as in-flight, are in the ROB, or are in the register file?	Are correct tags provided to each dispatched operand field whose value is not ready?

Table 4.5 Examples of local assertion checkers vs. functionality checkers for core blocks of an out-of-order processor with an Intel P6-like microarchitecture. Note that functionality checks are relatively more complex and high-level.

inefficient. Therefore, for functionality checking, the data associated with events should be recorded and transferred off-platform for ***post-simulation analysis in software***, where the validity of the recorded sequence of events is checked. Since events need to be recorded only as they occur, there is no need to record signal values on every simulation cycle. Moreover, we notice that we can further reduce the amount of data recorded by leveraging ***on-platform compression***, while still achieving high-quality functionality checking. Table 4.5 provides examples of local assertion and functionality checks for a few blocks.

4.6.2 Checker partitioning guidelines

It is technically possible, though inefficient, to express any collection of checks entirely as an embedded hardware or entirely as a post-simulation software checker (preserving cycle-accurateness via tracing cycle numbers, if needed). The partitioning of software-based checkers into local assertions and functionality checkers amounts to one of the most critical design decisions for the verification infrastructure and requires the involvement of a verification engineer who can extract the aspects that can be mapped into local assertions. However, there are high-level guidelines that we gained from experience and that can be used to guide and simplify this task. As discussed above, verifying the high-level functionality of a block is naturally a perfect fit for event-accurate functionality checking, whereas verifying simple interface behavior and component-specific invariants with cycle bounds is a better fit for local assertion checking. The primary criterion when making this distinction should be whether event-accuracy is sufficient or cycle-accuracy is needed to implement

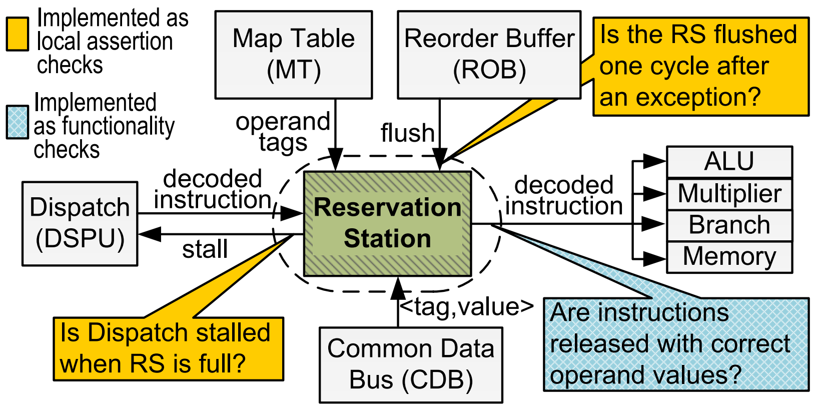

Implemented as local assertion checks
Implemented as functionality checks
Map Table (MT)
Reorder Buffer (ROB)
Is the RS flushed one cycle after an exception?
operand tags
flush
Dispatch (DSPU)
decoded instruction
stall
Reservation Station
decoded instruction
ALU
Multiplier
Branch
Memory
<tag,value>
Is Dispatch stalled when RS is full?
Common Data Bus (CDB)
Are instructions released with correct operand values?

Figure 4.9 **Reservation station block:** interfacing units and example checks.

a check. Another governing principle is that the logic footprint of a synthesized local assertion should be small. Hence, a sufficiently complex interface check that will result in a large logic overhead upon synthesis should be implemented as a post-simulation software checker. Once a checker is selected for local assertion checking, it can be coded as a temporal logic assertion and synthesized with tools such as those in [4, 24]. Note, however, that for our evaluation, we simply coded the assertions directly in synthesizable RTL.

Let us illustrate our approach using the reservation station (RS) block shown in Figure 4.9. This block stores instructions whose source operands are not yet ready and/or are waiting for a functional unit (FU) to become available. It receives input data from: i) a dispatch unit (DSPU) to allocate new instructions, ii) a map table (MT) to get source registers' tags (for not-yet-ready source values), iii) a reorder-buffer (ROB) to receive unique tags for each newly allocated instruction, and iv) a common data bus (CDB) to gather source operand values as they become available on the CDB. The RS unit releases instructions, whose source operands are ready, to multiple functional units. Thus we can abstract the RS to a data structure that *allocates* an entry upon receiving a new instruction from the DSPU, receives *updates* from the CDB, and *releases* ready instructions to the FUs.

One can identify several properties that must be upheld for the correct operation of the RS, three of which are shown in Figure 4.9: i) the DSPU must stall when the RS is full, and resume when a slot is available in the RS, ii) the RS must clear all of its contents on the cycle following an exception, and iii) each instruction must be released by the RS with the correct source operand values. The first two properties are localized to a few interface signals and require cycle-accuracy. Their checkers are also simple to implement, resulting in small logic footprints when synthesized; hence, suitable for implementation as local assertions. The third property pertains to correctness of source operand updates for instructions waiting in the RS block. Its checker must monitor source operand updates from the

CDB, identify the instructions updated, and preserve the operand values for later verification upon release. Note that the checker must take action only during an update event from the CDB and a release event from the RS. Since this checker verifies a complex high-level property spanning the lifetime of an instruction in the RS and can perform its checks in an event-accurate manner, it is best implemented as a functionality checker.

4.6.3 Microarchitectural block classification

We have observed that most microarchitectural blocks can be evaluated as blackboxes that receive control and data inputs from other microarchitectural blocks or shared buses, and either output information after some amount of processing, or perform internal bookkeeping. We label the events associated with input, output, and internal bookkeeping as *allocate*, *release* and *update*, respectively. We present a general classification of microarchitectural blocks' behavioral characteristics based on the types of events they receive and generate. This classification can help us decide the appropriate checker type, as well as the type of compression scheme, that fits best a given unit. We identify three main types of microarchitectural structures discussed below.

Type 1: Structures with allocate and release

This type of structure accepts bundles of information indexed by a unique tag, stores them temporarily, performs some operations on the data in the bundles, and finally releases the transformed bundles. Common examples are processor functional units: they receive operand values indexed by unique ROB-tags, operate on them and release the result to the CDB. For this type of structure, our functionality checker must simply log the allocate and release events during simulation and must find the matching reference events during post-simulation analysis. The unique ROB-tags associated with each bundle are used to identify and match events.

Type 2: Structures with allocate, update and release

This type of structure also targets bundles of information indexed by a unique tag; however, in this case the bundles include placeholders for values to be updated with a final value (by a third party) while the bundle is being processed by the structure. Update events generally relate only to specific entries in the structure, which are identified via a unique tag provided in the update event. A bundle is released (becomes "ready") when all its expected updates have been received. The "ready"-ness criteria is defined by the microarchitectural context. The control information associated with each bundle are the tags for the value updates and the "ready"-ness bits.

Block	**allocate event**	**update event**	**release event**
ROB	dispatch	execution complete	inst. retire
RS	dispatch	register value available on CDB	inst. issue
MT	dispatch	execution complete (tag match)	inst. retire (tag match)
LSQ	dispatch	i) address update on computation ii) value update on cache response	inst. retire

Table 4.6 Allocate, update and release events for essential blocks of an out-of-order processor with Intel P6-like microarchitecture.

This type of structure comprises some of the core microarchitectural blocks of an out-of-order processor. For instance, in an ROB, allocation occurs when a new instruction is dispatched. The update event (in this case just one) occurs when the instruction execution is completed, as indicated by the common data bus producing the corresponding tag. The release event is the instruction's retirement, indicated by the head entry in the ROB buffer. Table 4.6 presents a few of the structures of this type included in an Intel P6-family out-of-order processor: for each structure, we indicate the allocate, update and release events. Several other families of microarchitectures for out-of-order processors (*e.g.*, MIPS R10K) exhibit similar behavior.

Type 3: Combinational/Arbitration structures

The last type of structure consists mainly of combinational blocks, or blocks with a small amount of state information. They are often used for issue, dispatch, or bus-arbitration logic. They must make arbitration decisions based on pending requests in the current cycle: here local assertion checkers generally suffice.

4.6.4 Compression of recorded data

In our solution, our functionality checkers gather all the relevant events for each microarchitectural block, as described above, by logging the associated control and data signals on-platform for later transfer and post-simulation analysis. During the logging process, however, we also compress the collected information so as to reduce transfer time. Our goal is to achieve compression in the recorded information without sacrificing accuracy. From a verification perspective, control signals are more informative than data signals; hence, the guiding principle is to preferentially compress data content over control information. Indeed, the control information is critical in keeping the post-simulation software checker in sync with the design block. Since compression is performed using an embedded logic implementation, we want to leverage low-overhead compression schemes, such as

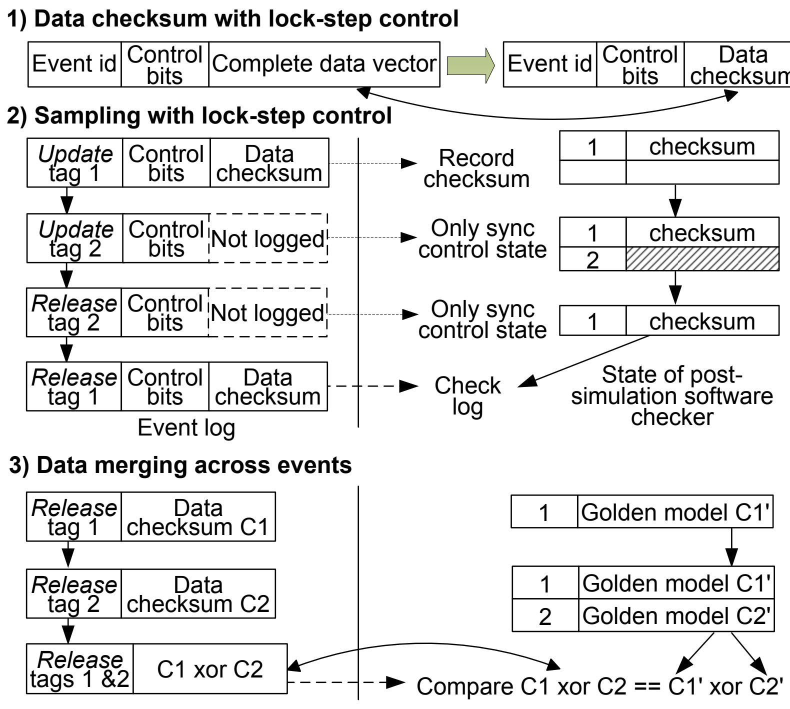


Figure 4.10 Compression techniques for functional events.

parity checksums, which can be computed with just a few XOR gates. In [29] it was shown that blocked parity checksums are generally sufficient to detect value corruptions due to functional bugs in modern complex processor designs. In light of this, we devised three compression techniques, presented below.

Data checksum with lock-step control Often, the post-simulation software must be able to follow the same sequence of control states as the design block to validate its behavior. Based on this observation, we compress the data portion of all events using a checksum (see Figure 4.10.1), while keeping control information intact. In cases where design and test constraints place limits on data range (*e.g.*, when the address field in a data packet is restricted to use only a specific address range), considering only portions of the data may allow for further compression of the recorded information. Moreover, some types of events may undergo additional compression steps as discussed in the next two subsections.

Sampling with lock-step control Taking advantage of the relative importance of control and data signals further, it is sometimes sufficient to record allocate, update and release events with their corresponding control signals, and simply drop the data components of the event (see Figure 4.10.2). In addition, a number of release events contain control signals that do not affect the state of a microarchitectural block. For such events, either sampling intermittent events, or considering only parts of the control signals is a viable approach.

Data merging This technique computes checksums across multiple events. Instead of

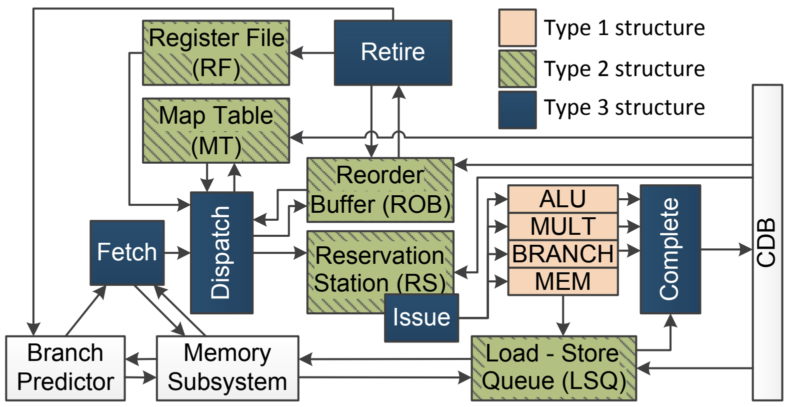


Figure 4.11 **Microarchitectural blocks** in our experimental testbed.

checking individual release events, we can construct their "checksum digest" (see Figure 4.10.3). This is beneficial when only a small fraction of release events may contain errors. Also, note that this approach is complementary to individual event checksums, since the data associated with each event is already compressed.

Choice of technique: The first two techniques are applicable to both type 1 and type 2 structures for compressing data associated with all 3 types of events, while the data merging technique is only applicable to release events for these structures. Some examples of checksum decision choices on a case-study design are presented in 4.7.

4.7 Experimental evaluation of hybrid checking

We applied our checker-mapping solution to a number of representative microarchitectural checkers that are part of the verification environment of a 64-bit, 2-way superscalar, out-of-order processor design, resembling the Intel P6 microarchitecture and implementing a subset of the Alpha ISA. Due to the absence of any open-source out-of-order microprocessor design we used a student project as our design and developed a verification environment around it. Our verification environment consisted of multiple C/C++ microarchitectural block-level checkers, one for each of the blocks reported in Figure 4.11, and an architectural golden model checker (*arch-check*) connected to the design via a SystemVerilog testbench. We also equipped our verification environment with a time-out condition on instruction retirement, indicating whether the processor had hung (μP *hang*).

We synthesized the design (excluding caches) using Synopsys Design Compiler targeting the GTECH library. Table 4.7 reports the contribution of each block to the total logic size: as it can be noted, the lion's share is contributed by structures that are state-heavy,

	ROB	RS	LSQ	MULT	RF	Others
# GTECH blocks	158,163	53,086	46,765	40,894	24,702	30,546
% of total	44.7%	15.0%	13.2%	11.5%	7.0%	8.6%

Table 4.7 Block sizes of our testbed design. The design has a combined 16-entry RS, 64-entry ROB, 32-entry LSQ and an 8-stage multiplier.

such as ROB, RS and LSQ. Developing acceleration-based checkers for such blocks has been traditionally a challenge as: i) if the checker is entirely implemented in hardware, the logic overhead becomes unacceptable – comparable in size to their design counterpart, and ii) these blocks generate many events, thus logging entails lots of storage, data transfer and analysis. Hence, we believe that the validation of these blocks will benefit the most from our solution.

We evaluated the feasibility of our hybrid checking methodology by analyzing several schemes on the checkers in our testbed. The validation stimulus was generated using a constrained-random generator that created a test suite of 1000 assembly regressions. In evaluating the quality of our solution, we considered the three most relevant metrics: **average number of bits recorded per cycle**, **logic overhead** and **checking accuracy**. The first metric reflects the amount of data to be recorded on platform and later transferred; we estimated the second one by using tracing logic similar to [29]; the third one is obtained by comparing our hybrid checkers against the bug detection quality of a software-only checker in a simulation solution. Note that industry experience suggests that the average bits/cycle metric is the most critical for acceleration performance. A recording rate of only 162 bits/-cycle is reported to induce a 50% slowdown for the acceleration platform used in [29]. We use this metric as a proxy for relative performance overhead independent of any specific platform architecture.

We injected a number of functional bugs in each microarchitectural block to evaluate the bug-detection qualities of our solution. To measure the checking accuracy of any compression scheme, the full set of regressions were run with only one bug activated at a time, and this process was repeated for each bug to create an aggregate checking accuracy measure. Each microarchitectural checker was only evaluated over the bugs inserted into its corresponding design block. The quality of each checker is based on its accuracy of detection. We quantified this aspect by computing what percentage of the bug manifestations it detected.

While we investigated our solution on most type 1 and type 2 blocks, and some of the type 3 blocks of Figure 4.11, for reasons of space we report below our findings for only one representative block for each category: a multiplier checker (type 1), a RS checker (type 2)

Name	Compression scheme
csX	compress 64-bit output into X checksum bits
merge	merge results of 5 consecutive release events
samp	record data for only 1 out of 5 instructions going through the block

Table 4.8 Multiplier checker – Compression schemes.

and a dispatch checker (type 3). Note that, as discussed in Section 4.6.3, type 3 checkers can typically be fully mapped to local assertions and do not require a functionality checking component.

Multiplier Checker

Our design contains an 8-stage pipelined multiplier, which *allocates mult* instructions with operand values obtained from the issue buses, and *releases* the result on the CDB after computation. Associated with each instruction are data signals: two 64-bit operand values on the input side, one 64-bit result on the output side and a 6 bit-wide control signal (the tag), on both directions. For this block, the only local assertion was for instruction completion within 8 cycles, excluding stall cycles. Functionality checking was used to verify computation performed by the block. A *data checksum with lock-step control* compression, possibly with *data merging*, is a natural choice for compressing events for the functionality checker; using a checksum scheme on the data output while preserving the whole 6 bits of control. However, to verify the result of the computation, we still need all operands' bits for each passing instruction. Finally, *sampling with lock-step control* can also be used as long as we keep track of all instructions going through the block but record operands and results in a sampled fashion. Table 4.8 details the set of compression schemes for evaluation. We modeled five distinct functional bugs (see Table 4.9) on the multiplier block to evaluate the quality of our checking schemes.

Figure 4.12 explores the trade-off between the checking accuracy of different data compression schemes and their average recording rate: the first bar on the left is for the the full software checker tracing all active signals and leading to an average rate of 25 bits/cycle. In contrast, the hardware-only checker does not entail any logging. Note that the average recording rate is much smaller than the total number of interface signals for the block,

Multiplier's functional bugs	
- Incorrect multiplier pipeline control signal	- Partial result over-written
- Wrong product bit assignment at pipeline stage	- Corruption of result's MSB
- Release of result on the wrong lines of the CDB	

Table 4.9 Multiplier checker – Injected functional bugs.

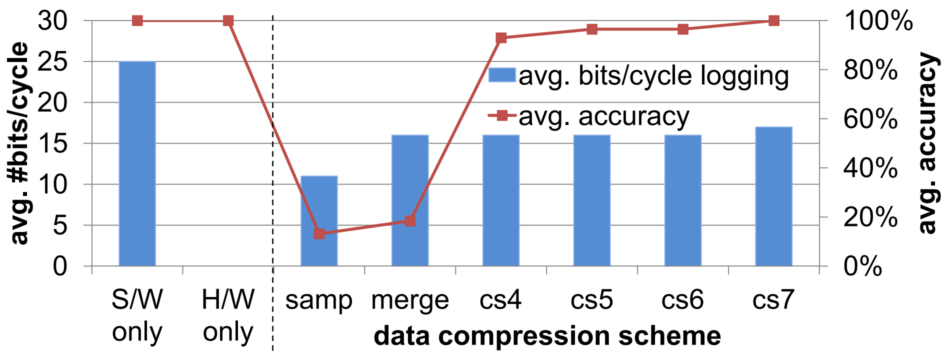


Figure 4.12 Multiplier checker – Accuracy vs. compression. The checksum schemes achieve almost perfect accuracy, logging only about 16 bits/cycle.

since only a fraction of instructions require a multiplier. The other bars represent sampling (low logging rate and low accuracy), merging, and various checksum widths. Note that our checksum compressions provide very high accuracy at minimal logging cost. We believe this is due to i) the ability of checksums to detect most data errors, and ii) the fact that some control flow bugs impact data correctness as well.

Figure 4.13 plots the logic overhead for all the compression schemes evaluated, relative to the multiplier block size. Note that the merging scheme has a slightly higher tracing logic overhead since it needs additional logic to maintain the running checksum.

Reservation Station (RS) Checker

The reservation station (RS) block is especially interesting since it is central to the out-of-order core architecture as it interfaces with almost all other microarchitectural blocks and generates the most events. The task of a RS is to hold instructions whose operand values have not yet been produced. The local assertions of the synergistic RS checker verify

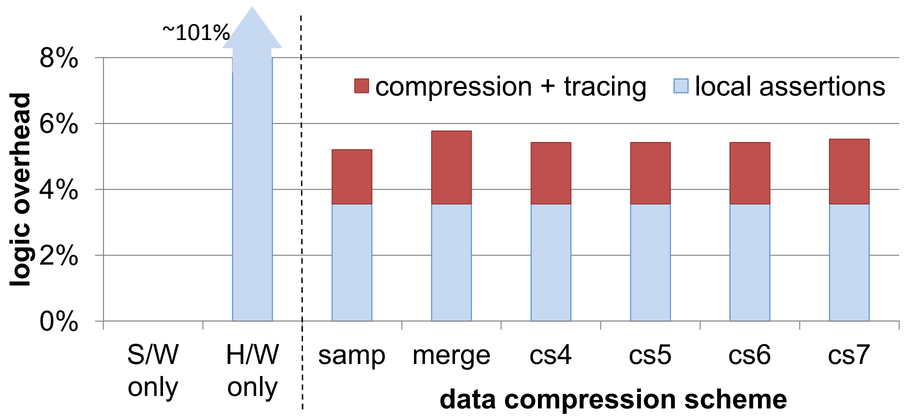


Figure 4.13 Multiplier checker – Logic overhead relative to the multiplier hardware unit for a range of compression schemes. Note that the logic overhead for the local assertion checkers does not vary between schemes since the same local assertion is used for every scheme.

Name	Compression scheme
csX	both 64 bit operand fields are compressed to X bit parity checksum, control and tag stay intact, all other data bits are ignored
twX	Only last X bits of the 6 bit tag are recorded, control bits stay intact, operand fields and all other data bits are ignored

Table 4.10 RS checker – Compression schemes.

the simpler aspects: stall, flush and instruction residency time bounds. From the functionality checking perspective, the dispatch unit allocates instructions to the RS, the common data bus (CDB) updates the RS with ready operands, and the RS releases ready instructions to the functional units. Associated with each instruction are control and data signals: a unique ID tag, operand "ready"-bits, operand values if operands are ready, or corresponding tags otherwise, and decoded instruction data. The control signals for an allocate event (instruction tag, operand tags and readiness bits) require only 20 bits. The data signals for the same event, on the other hand, require a total of 326 bits. We observed this skewed distribution of data and control signals in the *update* and *release* events of most other blocks as well. Thus, we expect our *data checksum with lock-step control* method to be very effective in compressing these events. Table 4.10 lists the compression schemes that we studied for this block. We modeled seven distinct effects of functional bugs (see Table 4.11) for the RS block to evaluate the accuracy of our solution. Note that these bug manifestations capture a large class of design bugs, since a number of functional bugs produce similar effects.

Similarly to what we did for the multiplier checker, Figures 4.14 and 4.15 report the trade-off between accuracy and logging sizes, and logic footprints for several compression variants when applied to the RS checker. In this case, the software checker must log a very high 313 bits/cycle. Note from Figure 4.14, that the "**cs5**" scheme provides almost perfect accuracy at a logging rate of only 23 bits/cycle, a 92.7% reduction. In Figure 4.15 we note that the same scheme can be implemented with a fairly small logic overhead (18%), compared to the extreme H/W only checker (101%). Finally, we observe that the logic footprint of the various tracing schemes decreases with the number of checksum bits calculated, as one would expect.

Reservation station's functional bugs	
- Wrong operand value on issue	- Wrong ROB tag released on issue
- CDB update overwrites ready operand	- Missing a CDB update
- Corruption of decoded instruction within RS	- Ready instruction in RS stalled
- Erroneous ROB tag recorded on allocate	

Table 4.11 RS checker – Modeled functional bugs.

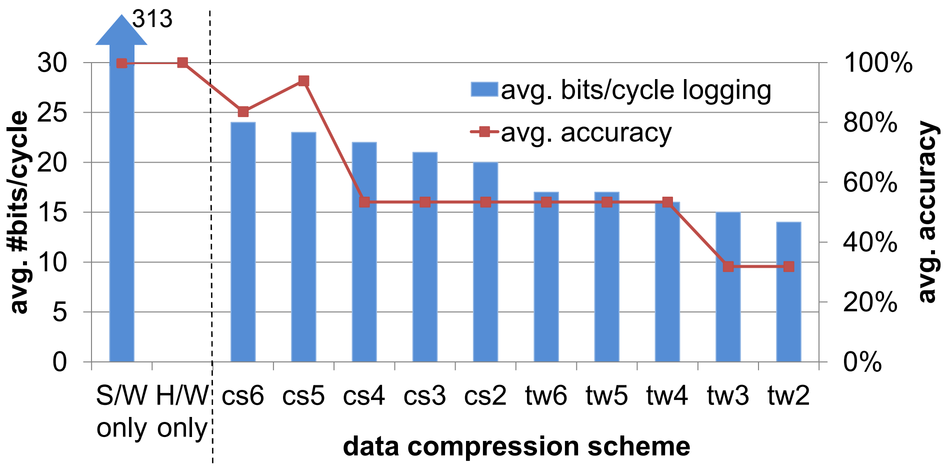


Figure 4.14 RS-checker – Accuracy vs. compression. A cs5 (64-bits operands compressed to 5 bits) compression scheme delivers an almost perfect bug detection accuracy at a recording rate of only 23 bits/cycle.

Finally, in Figure 4.16, we show a distribution of which of the checkers in our verification infrastructure detect each bug first. The portions labeled **RS-local** and **RS-func.** represent the fraction detected by the local assertion and functionality checker of our synergistic RS-checker as described in this paper. **other-check** refers to another microarchitectural block checker, and **arch-check** is the fraction detected by the architectural checker (see the beginning of Section 4.7). Finally, μ**P hang** represents the fraction detected by our timeout monitor. For the S/W-only version, both RS-func. and RS-local checks are implemented as post-simulation software checkers. While some bug manifestations are only detected by the architectural checker, overall, all bugs are detected by at least one of the checkers in place. Among the variants of our solution, the synergistic checker with cs5 compression alone, localizes more than 60% of the bug manifestations, at a logic overhead of only 18%,

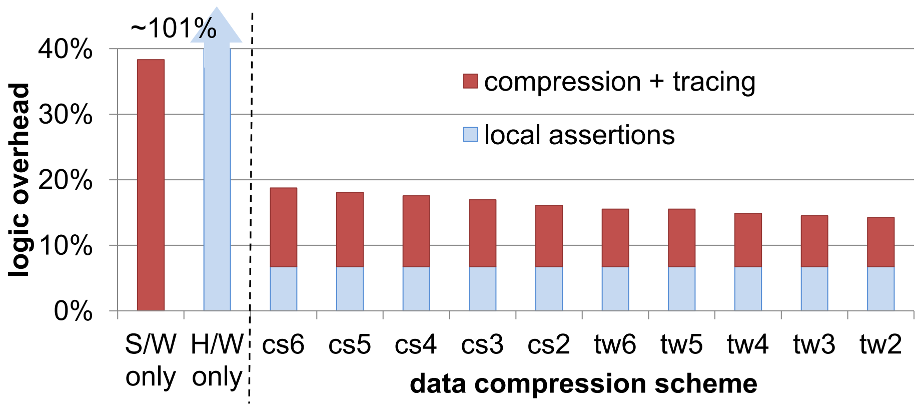


Figure 4.15 RS checker – Logic overhead relative to the RS hardware unit for a range of compression schemes.

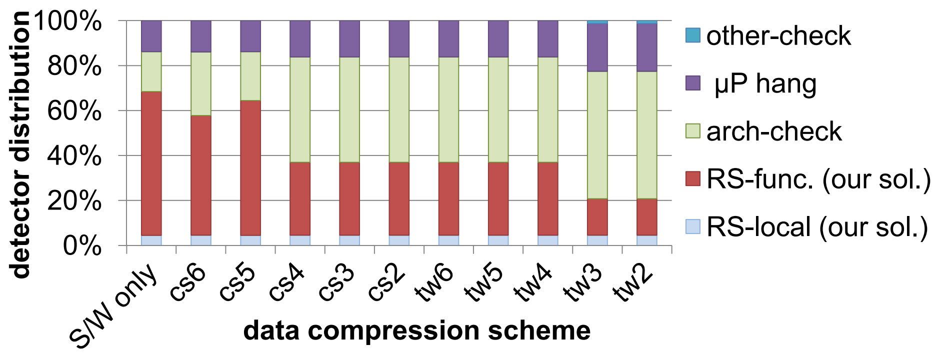


Figure 4.16 RS checker – First detecting checker over various compression schemes. In most cases either the embedded checker or the functionality checker is able to detect a bug before it propagates to the architectural level.

indicating that our solution is effective for a wide range of bugs.

Note that, almost all bugs eventually manifest at the architectural level; hence, the primary reason to validate at the microarchitectural level is to localize the bug in time and space more accurately, easing debugging. To evaluate the success of our solution on this front, we also conducted a study on the latency of bug manifestation in Figure 4.17 , where we plot the difference in cycles between when a bug is detected by our solution and when it is detected at the architectural level. It can be noted that we can flag a bug up to 5,900 cycles earlier, a great benefit for debugging purposes.

Dispatch Checker

The main purpose of the dispatch logic is to dispatch each decoded instruction with correct value or tag (depending on whether the value is available). The block is implemented with combinational logic to select the correct source of tag/value provider based on associ-

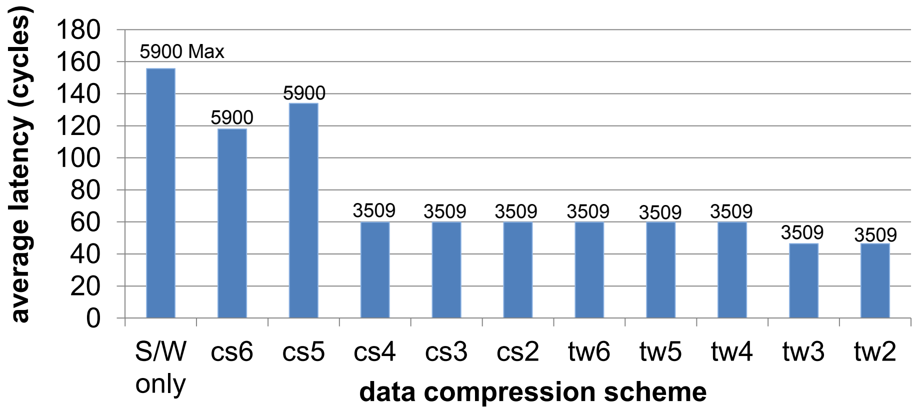


Figure 4.17 RS checker – average latency between detections at microarchitectural (earlier) and architectural (later) level. The maximum bug detection latency is reported above the bars. Our RS checker with cs5 scheme can detect a bug up to 5,900 cycles before it propagates to the architectural level.

ated flags. All checks for this block are implemented via local assertions whose footprint was less than 80% of the size of the block. Even though this is a relatively large footprint, the block itself is very small w.r.t. the whole design, hence the approach is affordable.

4.8 Efficient memory consistency validation

The solutions discussed so far focus on adapting software checkers for checking on acceleration platforms. We complete this chapter by discussing a a solution designed from the ground up for detecting memory consistency bugs during accelerated simulation and post-silicon validation. Our solution offers high bug detection capability, debugging support, low area overhead, and can be deployed on a wide range of systems, spanning multiple memory consistency models. When using our solution, a portion of the first-level data caches in the processor-under-verification are claimed to record memory interactions during test program execution. These memory access logs are then aggregated into main memory and analyzed in software. This approach results in a significantly smaller silicon area overhead than similar solutions proposed in the literature. Furthermore, our solution is transparent to the end user as logging and analysis is disabled and cache space is released upon product shipment. We demonstrate our approach on a CMP with out-of-order cores, each with private L1 caches.

Our solution partitions test program execution into multiple *epochs*, each comprising three distinct phases, as illustrated in Figure 4.18. In the first phase, program execution progresses while memory accesses are tracked in the background and logged into a reserved portion of the processor's L1 data caches. A small amount of additional logic is required to perform this task, but note that the additional hardware is off the critical computation paths. When the system exhausts logging resources, program execution is temporarily suspended and program state is saved, to be restored later at the start of a new epoch. The system then transitions into the second phase, where the memory access logs from all the caches are aggregated into main memory. In the third phase, these logs are analyzed by software that checks whether memory consistency violations have occurred during program execution.

Our analysis algorithm builds a specialized directed graph using the information in the memory access logs. Each vertex in the graph represents a memory access, while directed edges are generated between memory accesses based on the observed order of the memory operations and on the requirements of the memory consistency model. The presence of a cycle in this graph indicates that a memory consistency violation has occurred [31, 100]. The information in the memory access logs and the architectural state at the end of each

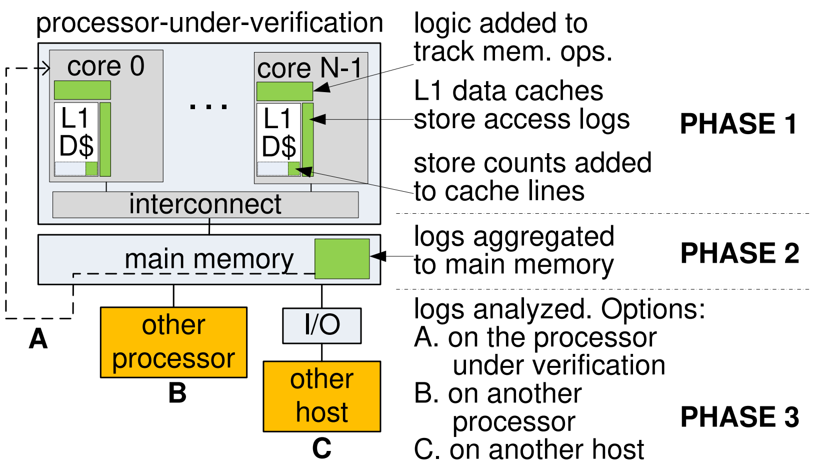

processor-under-verification
core 0
core N-1
L1 D$
L1 D$
interconnect
main memory
other processor
I/O
other host
A
B
C
logic added to track mem. ops.
L1 data caches store access logs
store counts added to cache lines
PHASE 1
logs aggregated to main memory
PHASE 2
logs analyzed. Options:
A. on the processor under verification
B. on another processor
C. on another host
PHASE 3

Figure 4.18 System overview. Each core is modified to track memory accesses. A portion of the L1 caches are temporarily reserved for logging memory accesses. A *store counter* is attached to each cache line to track the order of writes to the line. Finally, the analysis of the logged data is performed in software, which can be carried out on a processor/host available for the task.

epoch provide insight into the activity of the CMP during an epoch, which can be used to find the root cause of a consistency violation. This analysis can be carried out on- or off-chip, based on available resources and validation strategy.

In Figure 4.18, we outline the hardware modifications required to implement our solution. Each core is augmented with a small amount of logic to enable memory access tracking. A *store counter* is added to each cache line for tracking the order of writes to the line. When a cache line is transferred to a different core, its store counter is also transferred with it. In addition, the L1 cache controllers are modified to temporarily reserve and utilize portions of the cores' L1 data caches for storing memory access logs. All of these modifications can be completely disabled after validation, resulting in virtually no performance and energy overheads to the customer.

Our solution allows the log analysis to be performed in a range of time interleavings, as illustrated in Figure 4.19. For instance, if the only goal for a test is the verification of memory interactions, then it is reasonable to analyze the recorded logs right after each execution phase and terminate the test if a violation is detected. Here, the analysis may be serialized as in Figure 4.19a or overlapped with the next epoch's execution as in Figure 4.19b. The scenario in Figure 4.19a would allow for the analysis software to run on the same processor under verification. In cases where logs have to be transferred off the validation platform for analysis on another host, it may be more efficient to adopt the scenario in Figure 4.19c to amortize log transfer overheads. This latter setup is especially useful for emulation-based validation flows, where it may be impractical to execute the analysis software on the processor being emulated due to performance limitations.

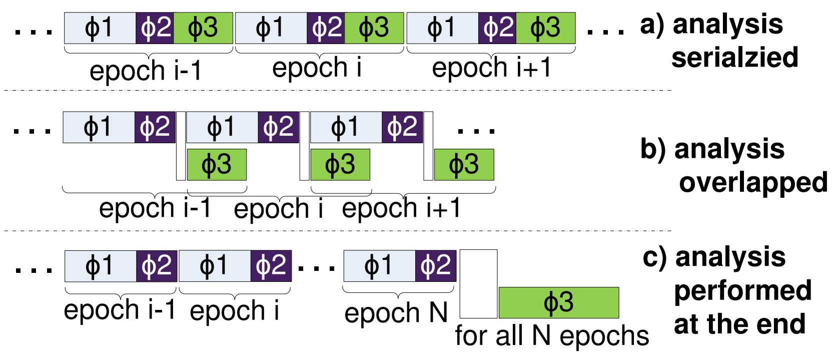


Figure 4.19 Timeline scenarios. When logging resources are exhausted, test program execution is suspended and logs are aggregated for analysis. The analysis may then be done **a.** before beginning the execution phase of the next epoch **b.** overlapped with the execution phase of the next epoch **c.** at the end of the test program execution.

4.8.1 Manifestations of memory consistency bugs

The memory subsystem in a typical modern chip-multi-processor consists of two or three levels of caches and a main memory that service memory requests from load-store units in out-of-order cores. From the perspective of the programs running on the processor, the memory subsystem has to appear as if it were one monolithic structure that preserves the ordering properties specified by the memory consistency model. Several consistency models have been proposed and adopted over the years ([104, 8]), mainly driven by the need to allow high performance CMP implementations. A consistency model for a particular architecture specifies the acceptable orderings of memory accesses that multi-threaded shared-memory programs should expect. Below is a brief description of the ordering requirements for three representative models:

Sequential Consistency (SC) [69]: All memory operations from each core must be performed in their respective program order. In addition, there should exist a unique serialization of all memory operations from all cores.

Total Store Order (TSO) [112]: All memory operations from each core must be performed in their respective program order, except for when a store operation to a memory location is followed by a load operation from another location. In such cases, a core is allowed to execute the load early, before the store completes. Finally, all cores should observe the same unique serialization of all store operations in the system.

Relaxed Memory Order (RMO) [112]: No implicit ordering requirements are enforced amongst memory operations to distinct memory locations. The programmer can explicitly enforce ordering by inserting fence (barrier) instructions.

Weak memory consistency models, such as RMO, rely on fence instructions to explicitly order memory operations. These fence instructions include a field to specify the ordering constraint being enforced by the instruction. For instance, the SPARC V9 ISA [112] includes a `MEMBAR` instruction with two mask fields - *mmask* (4 bits) and *cmask* (3 bits). Each *mmask* bit represents the possible combination of memory operations that ***cannot*** be reordered across the instruction (bit 0: load$\rightarrow$load, bit 1: store$\rightarrow$load, bit 2: load$\rightarrow$store, bit 3: store$\rightarrow$store) – if all four *mmask* bits are set, then no memory operation reordering is allowed across the `MEMBAR`. The *cmask* bits are used to specify ordering requirements between memory operations and other preceding instructions. Here, we limit the scope of our work to consider ordering only with respect to other memory operations. Note that several consistency models can be modeled using RMO with an appropriately masked fence instruction inserted after each memory operation. For example, we could model sequential consistency by inserting `MEMBAR`s, with all *mmask* bits set, after each memory operation. In addition, the most commonly used synchronization instructions (`test-and-set`, `exchange`, `compare-and-swap`, `load-linked` and `store-conditional`) can be modeled as sequences of reads, writes and fences.

Previous work has shown that bugs in an implementation of a memory consistency model can be detected by constructing a directed graph from observed memory operations and then searching for cycles in the graph [42, 31, 54, 27, 15, 96]. The graph should be acyclic if the execution satisfies the ordering rules given by the consistency model. We present an example of such a graph in Figure 4.20 for a multi-threaded execution on a system based on the RMO consistency model. Figure 4.20a shows snippets of memory operations from three different cores for a multi-threaded program. Our mechanism tracks the ordering requirements of the fence instructions and the data dependencies between memory operations. Consider a case where each core performs its memory accesses in program order and accesses from multiple cores are arbitrarily interleaved. Figure 4.20b reports a graph obtained from such an execution. The vertices in the graph represent memory accesses, the red dashed edges are generated from the ordering requirements of the fence instructions, and the solid black edges are generated from the data dependencies observed during execution. A different execution may result in different data dependencies, and hence a different graph. Assume core 0 executes *LD A* before the preceding memory accesses complete, violating the explicit ordering requirements of the two fence instructions in its instruction stream. In this case, core 0 would load the value that is set in its store queue, written by the *ST A* in its instruction stream, instead of the value from core 1. This results in a different graph as reported in Figure 4.20c. Note that violating the ordering requirements of the fence instructions introduced a cycle in the graph.

The main focus of this work is the detection of memory consistency bugs that may occur when multiple cores execute multiple threads. We assume that: i) the system is a

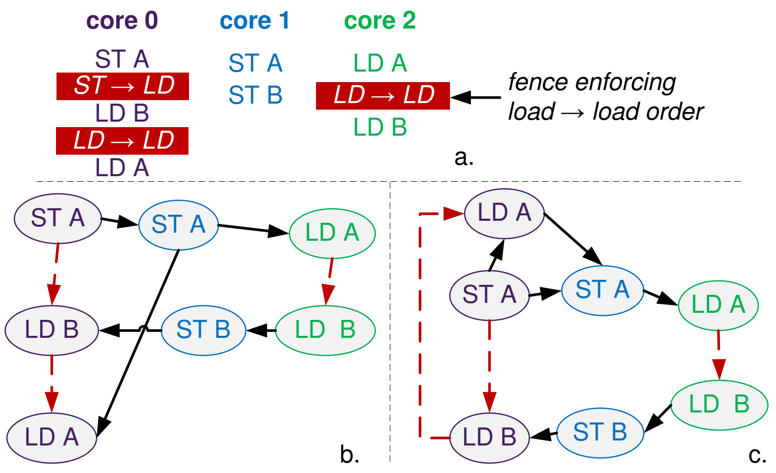


Figure 4.20 Memory access graph examples. a. Sequence of memory operations and fence instructions from three processor cores. **b.** Memory access graph for an execution that abides RMO rules. Solid black edges represent data dependencies, while dashed red edges represent fence-enforced orderings. **c.** Core 0 executes *LD A* out of order, in violation of the fences in its instruction stream. This violation manifests as a cycle in the resulting memory access graph.

cache-coherent CMP that enforces a single writer to a cache line at a time, while allowing multiple readers, ii) the L1 caches implement a write-allocate (fetch-on-write) policy for store misses iii) the threads executing on the CMP interact only through shared memory accesses and iv) intra-thread data dependencies are handled correctly or there is an orthogonal mechanism addressing this type of issues, possibly among those listed in [67]. In developing our solution, we also make use of the following observations:

1. If a thread does not share data with other threads, the core executing it can reorder memory operations compatibly with the correct enforcement of data dependencies. In this case, the thread's execution abides the constraints of any consistency model.
2. If a thread shares data with other threads, it can re-order its memory operations as long as no other thread observes memory access orderings that violate the system's consistency model. If violations do occur, they manifest as cycles in the corresponding memory access graph, involving both inter-thread edges and edges derived from the rules of the consistency model.
3. For a cache-coherent CMP, as we assume, a unique serialization must exist for all stores to an individual cache line.
4. The inter-thread edges in the memory access graph can only result from inter-thread data dependencies.

Observation 2 dictates that we collect information required to construct the required edges to detect violations. In order to construct the edges imposed by the consistency model, our solution collects information about the relative ordering of memory accesses as dictated by the memory consistency model. To construct the inter-thread edges, we collect information about data dependencies, utilizing the property in observation 4. Observation 3 provides us with a mechanism to keep track of data dependencies; by uniquely identifying each store to a cache line, not only are we able to determine the order of write accesses to a cache line, but also which store operation produced a value for a subsequent load. We can capture read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR) data dependency edges with this mechanism. The following sections describe in detail how this information is collected and then used to construct memory access graphs for the subsequent analysis.

4.8.2 Tracking memory access orderings

Our solution tracks two types of ordering information: i) implicit or explicit ordering enforced by the consistency model on memory accesses from the same thread (henceforth referred to as *consistency order*) and ii) ordering observed during execution due to data dependencies between memory accesses (henceforth referred to as *dependence order*). We utilize some of the processor's L1 data cache space to temporarily store this information. We present the details of the information collection mechanism below.

Capturing consistency order

Our solution tags each memory access with a *sequence identifier* that marks its position in consistency order, relative to other memory accesses from the same thread. The generation of sequence identifiers is dependent on the memory consistency model under consideration. Sequential consistency, for instance, requires program order to be enforced between all memory accesses (consistency order is the same as program order). Therefore, unique and monotonically-increasing sequence identifiers are required to capture the ordering requirements. These sequence identifiers can be generated by using a simple per-thread counter that is incremented on every memory access. On the other hand, RMO does not impose any implicit ordering constraints between memory accesses to different addresses. A programmer can explicitly enforce ordering through the `MEMBAR` fence instruction. Depending on the *mmask* field of the instruction, loads and/or stores before the `MEMBAR` are required to be ordered before loads and/or stores after. For such a case, a sequence identifier needs to identify the relative position of memory accesses with respect to fence instructions. Since not all memory accesses are affected by a particular fence instruction, the sequence identifier must also encode the *mmask* fields of the `MEMBAR` instructions. The sequence identifier for a memory access can thus be constructed from a $\{$*count,mask*$\}$ tuple where the *count* is the number of `MEMBAR`s encountered before the memory access instruction and the *mask* is the value in the *mmask* field of the last `MEMBAR` preceding the memory access instruction.

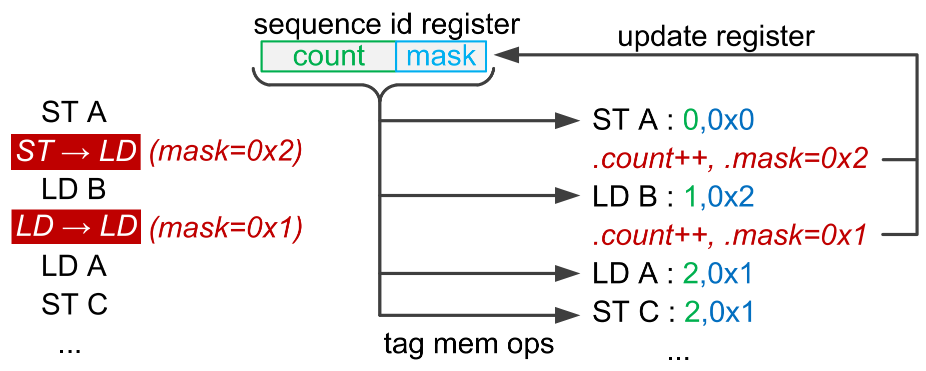


Figure 4.21 Memory access tagging. Upon dispatch, a memory operation is tagged with the value currently stored in the sequence identifier register. A fence instruction causes the *count* field in the sequence identifier register is incremented and its mask is copied to the *mask* field.

We observe that a generic solution based on sequence identifiers for RMO can be extended for use with any other consistency model. For instance, for sequential consistency and TSO, the count field of the sequence identifier tuple is incremented after every memory instruction, while the mask field is kept at a constant value to reflect the restrictions on the types of memory accesses that can be reordered – no reordering is allowed for sequential consistency (*i.e.*, `seq_id.mask = 0xF`) and loads are allowed to be reordered w.r.t. previous stores for TSO, while every other reordering is restricted (*i.e.*, `seq_id.mask = 0xD`). We can then model sequential consistency and TSO using RMO by assuming the existence of `MEMBAR`s with *mmask* `0xF` and `0xD`, respectively, after each memory access instruction. We will use this generic model to discuss our solution without delving into the particulars of any memory consistency model.

We add a special sequence identifier register, with *count* and *mask* fields, to each core. When a fence instruction is dispatched, the count is incremented and the ordering constraints imposed by the fence instruction are stored in the mask field. We also add a "retirement sequence identifier register" which is updated in a similar fashion when a fence instruction is retired. The precise (non-speculative) value in this register is copied into the actual sequence identifier register when the pipeline is flushed. Upon dispatch, all memory accesses are tagged with the value in the sequence identifier register. Figure 4.21 illustrates an example of the sequence identifier generation process. The first *ST A* access is tagged with a $\{$*count, mask*$\}$ tuple of $\{$`0,0x0`$\}$, from the 0-initialized sequence identifier register. The fence instruction following the store causes the sequence identifier register to be updated; the *count* is incremented and the fence instruction's mask is copied to the *mask* field (`mask = 0x2`, *i.e.*, loads can not be re-ordered w.r.t. previous stores). All subsequent memory instructions are then tagged with the new sequence identifier, until a new fence instruction causes a sequence identifier register update.

In an Intel®P6-like microarchitecture, the sequence identifiers for all in-flight memory instructions can be appended to entries in the load and store queues. When a memory instruction is finally performed, the logging mechanism reads its sequence identifier and stores it in the portion of the cache temporarily reserved for our solution's logging.

Capturing dependence order

A unique serialization of all stores to a cache line exists for a program executing on a cache-coherent CMP. We attach a *store counter* to each cache line, which is incremented upon each write to an address in that line. To hold the store counter, we repurpose the ECC bits of the cache line. Note that the counter value is transferred along with the cache data of the line to the cores and through the memory hierarchy. The ECC update mechanisms for all memory elements in the hierarchy are repurposed to preserve the transferred ECC bits. A memory access is tagged with the most recent store counter value for the cache line it accesses. This allows our solution to infer the relative order among shared-memory operations to a given cache line, and thus to derive read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW) dependency edges in the memory access graph. In addition, these counters enable a straight-forward mechanism for verifying the single-writer-at-a-time requirement of a cache-coherent CMP – each write access to a cache line must be tagged with a unique count. A memory operation that accesses multiple cache lines is split into multiple log entries to preserve the store counter values for all the accessed lines. During analysis, such accesses will be merged and all the relative orders inferred from the multiple store counters will be included.

Figure 4.22 illustrates the store counting mechanism and memory access logging for 3 cores executing the program snippet shown in Figure 4.20a. Assume that the cores perform one memory access at a time in the following order: *core 0, core 1, core 2, core 2, core 1, core 0*. This would result in the memory access graph shown in Figure 4.20b. Note that all memory accesses are tagged with appropriate sequence identifiers as discussed before. When core 0's *ST A* writes to core 0's cache, the store count for the written cache line is incremented to 1. Figure 4.22a shows the snapshot of this store count, the sequence identifier tuple, the type of memory access and the address logged in core 0's log storage space. Core 1's *ST A* instruction causes core 0's corresponding cache line to be invalidated and the store count to be shipped to core 1. The store count is then incremented and its snapshot is stored in core 1's log storage space, along with the rest of the log entry, as shown in Figure 4.22b. This updated store count is shipped to core 2, along with the data for the corresponding cache line, when core 2's *LD A* is executed. The log entry for core 2's *LD A* access then contains a store count of 2 as shown in Figure 4.22c. Figure 4.22d shows the final state of the memory access logs after all memory accesses have been performed.

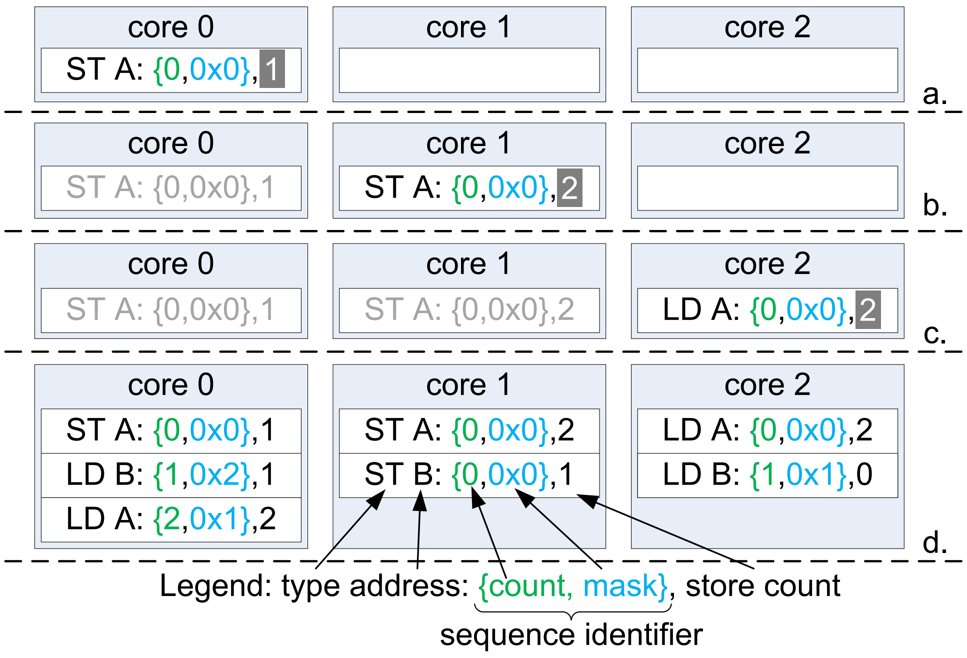


Figure 4.22 Memory access logging example for the program snippet in Figure 4.20a. **a.** Core 0 performs its first store to line A, updating the store count for A to 1 and taking a snapshot. **b.** Core 1 performs a store to A invalidating core 0's cache line. A is transferred to core 1 with an incremented store count of 2. **c.** Core 2 loads A, which is transported along with the store count of 2 from core 1. **d.** Final contents of the logs that generate the graph of Figure 4.20b.

Note that if core 0's *LD A* had received its value forwarded from core 0's store queue, the store count associated with this access would have been 1 and as a result, the memory access graph shown in Figure 4.20c would have been generated.

Logging mechanism

We reserve portions of the cores' L1 caches for log storage at a granularity of cache ways. Groups of blocks in each set are made unavailable for regular memory operations. The cache controllers are modified to use these restricted blocks for log stroage, without requiring changes to the cache addressing scheme. A single log entry holds information about the type of memory access, the memory address, the sequence identifier and a snapshot of the store count for the cache line. For the type information, one bit is used to indicate if the access was a load or a store and a second bit is used to indicate whether it should be merged with the preceding access during analysis. Merging is needed for log entries of memory access that cross cache line boundaries. The data arrays in the reserved cache ways are repurposed to hold log entries, with each entry aligned to byte boundaries. After the execution phase of an epoch, the log entries from all cores must be aggregated for analysis. A section of the address space is allocated for storing the aggregated logs. We utilize the existing cache line eviction mechanism to move the log entries down the mem-

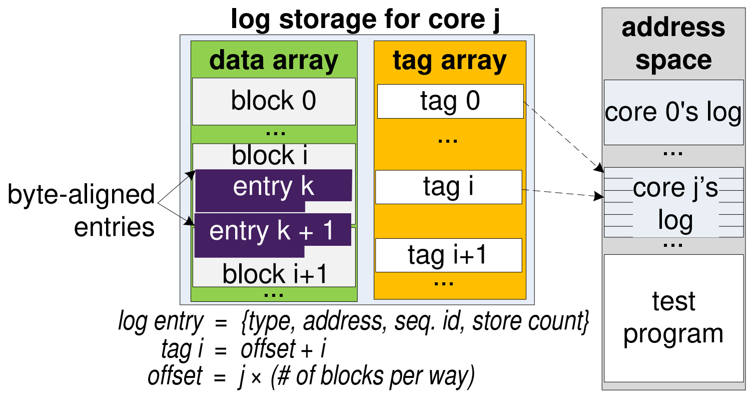


Figure 4.23 Logging mechanism: the data arrays in the reserved cache ways are used to store log entries. The tag arrays store address bits to direct the log entries to their allocated space in memory during log aggregation. In the figure we report the tag computation for the case when only one way of each cache is reserved for log storage.

ory hierarchy. To enable this simple transfer mechanism, the tag arrays in the cache ways reserved for log storage are populated with the appropriate address bits to direct the logs to the proper memory locations. Figure 4.23 shows the details of the logging mechanism.

Log-delay buffer

The logging of a memory access needs to be delayed for certain special cases. First, obtaining the store count values might not be straightforward for load operations that obtain their values forwarded from stores in a load-store queue or a store buffer. For such loads, the store count that needs to be associated with the corresponding log entry can not be obtained until the forwarding store accesses the cache and updates the store count. Second, when multiple stores to a cache line are coalesced in the store buffer, the store count updates corresponding to each store must be applied, even though the cache line is accessed only once. Third, a speculative memory access should not be logged until the instruction that issued the access has retired. Lastly, all available L1 write ports maybe in use when our mechanism has an entry to write into the portion of the L1 cache reserved for log storage. To handle these cases, we incorporate a *log-delay buffer* for saving log entries waiting to be written to the log storage space. Our logging mechanism ensures that entries update their information and leave the log-delay buffer whenever the event they are waiting for occurs.

Logging is performed in the background during program execution until log resources are exhausted. A store count reaching the maximum value, a data block in the reserved log storage filling up, or the log-delay buffer running out of space signal the end of an epoch. The log-delay buffer is designed to handle at least as many entries as the combined sizes of the store buffer and the load-store queue, to reduce the probability of filling up. The fence counter field in the sequence identifier is designed to count at least up to the maxi-

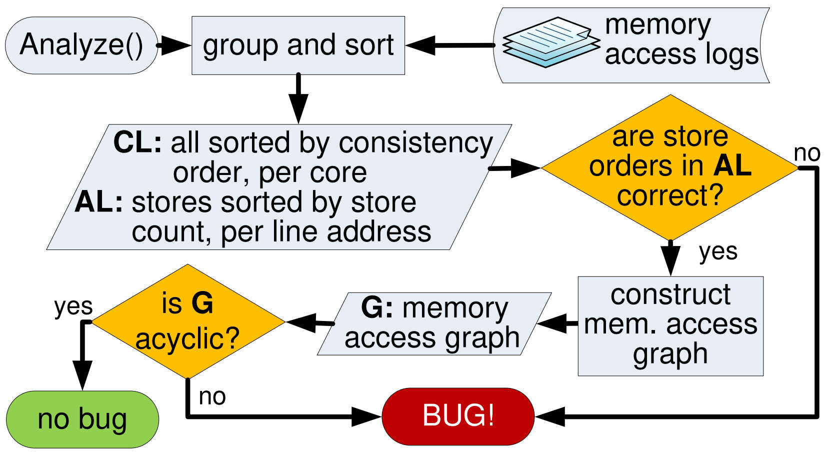

Analyze()
group and sort
memory access logs
CL: all sorted by consistency order, per core
AL: stores sorted by store count, per line address
are store orders in AL correct?
no
yes
construct mem. access graph
G: memory access graph
is G acyclic?
yes
no
no bug
BUG!

Figure 4.24 Log analysis algorithm. The algorithm first creates sorted data structures. The sorted memory accesses for each line address are checked for valid store ordering. The graph construction algorithm infers all ordering edges from the sorted memory access lists. The resulting graph is topologically sorted to detect cycles.

mum number of memory operations that can be in flight at any given time. This allows our analysis software to easily detect when the counter has wrapped around for the log entries so that it updates all entries thereafter with correctly increasing count values.

4.8.3 Log Aggregation and analysis

At the end of the execution phase of an epoch, our system makes a transition from normal program execution mode to log aggregation mode by i) saving the architected state of the processor to memory, much like how it is done during a conventional context switch, ii) disabling the lower level caches and portions of the L1 caches that were used for normal program data and iii) running software that triggers evictions for the cache lines holding the logs. At the end of the log aggregation phase, the logs of memory interactions and the final architected state of the test program reside in main memory. The analysis phase can then resume in one of three ways as described below.

Analysis leveraging the processor under verification: The tracking and logging mechanism is temporarily disabled and the analysis software is loaded and executed.

Analysis on a separate processor connected to the same memory: The analysis software is executed on the separate processor.

Analysis using a separate host: The logs are first transferred from the memory connected to the processor under verification to the memory of the separate host machine. The analysis software is then executed on the host.

Figure 4.24 shows a high-level flowchart of our analysis algorithm. This algorithm first

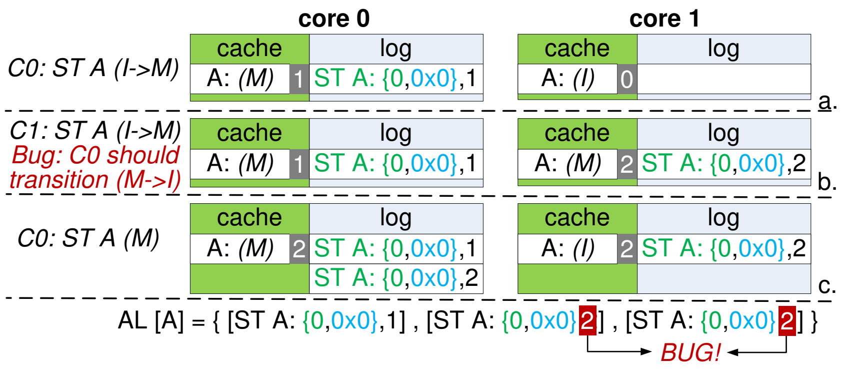


Figure 4.25 Coherence bug example. a. Core 0 writes to line A. **b.** Core 1 gets line A from core 0 to perform a store, however, core 0's cache fails to downgrade its permission to line A. **c.** Core 0's store to line A updates the store count, leading to a conflict detected by the store order checker.

creates two data structures that provide easy access to the memory access logs – the Core List data structure (***CL***), which keeps each core's memory accesses in a list sorted by consistency order, and the Address List data structure (***AL***), which keeps each store to a cache line in a list sorted by store count. These data structures, generated by the "group and sort" process in the flowchart of Figure 4.24, enable efficient implementations of the store order checker, the graph constructor and the cycle detector, which are discussed below.

Checking order of writes to a cache line

For a cache-coherent CMP, there must exist a strictly increasing sequence of store counts, per cache line, if only one writer is allowed to a cache line at a time. Figure 4.25 shows an example for a cache coherence bug that allows two stores to write to a line simultaneously. Core 0 fails to downgrade its permission (transition from Modified to Invalid, for a simple MSI protocol) to cache line A before core 1 upgrades its permission. This violates the single-writer-at-a-time invariant, enabling both cores to write to A at the same time. This violation manifests as two consecutive stores with the same store count in the Address List for line address A (*AL[A]*).

Graph construction and cycle detection

The graph construction subroutine constructs a directed graph from the observed memory accesses, their data dependencies, and the ordering constraints imposed by the memory consistency model. A vertex in the resulting graph represents a memory access and an edge between two vertices represents the order between the two accesses. Dependence order edges are constructed using the store count snapshots. Every memory operation is placed in between two stores – one occurring right before it and the other occurring right after, in dependence order. The sequence identifier information attached with each log entry is used to construct the consistency order edges.

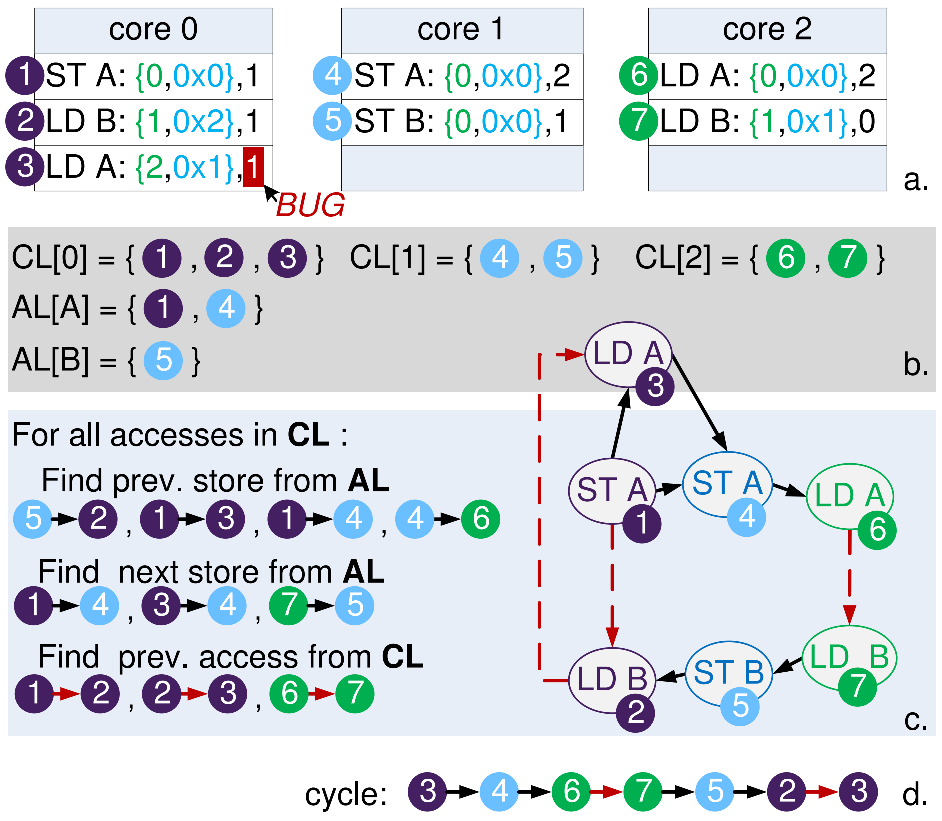


Figure 4.26 Graph construction example. a. Memory access logs from the example introduced in Section 4.8.1. **b.** Contents of CL and AL derived from the logs. **c.** The graph construction algorithm generates edges utilizing the entries in CL and AL. **d.** A cycle is identified in the resulting memory access graph.

Figure 4.26 details our graph construction algorithm using the example introduced in Figure 4.20. Note that the logs shown in Figure 4.26a are from an execution where core 0's *LD A* violates the ordering requirements and obtains its value early from the *ST A* in its instruction stream. The memory access logs are first organized into two lists as presented in Figure 4.26b. For each *core i*, accesses are placed in *CL[i]* in increasing order of their sequence identifiers. Stores to each *cache line x* are placed in *AL[x]* in increasing order of their store count. The graph construction algorithm walks through each access in CL[i] and attempts to construct directed edges to and from the access. First, dependence order edges are constructed by identifying preceding and succeeding stores in dependence order. This is easily achieved by directly indexing AL using indices derived from the address and the store count of the memory access. Next, for each preceding access in CL[i], the algorithm checks if the effective ordering requirement warrants an edge to the current access. The effective ordering requirement is obtained as a bitwise AND of all masks in the sequence identifiers of the memory accesses appearing between the two memory accesses in CL[i].

The edges derived by our algorithm and the resulting memory access graph are shown in Figure 4.26c. The cycle due to the violation of the ordering requirements is presented in Figure 4.26d.

4.8.4 Strengths and limitations

Debugging support: The memory access logs, preserved architectural state and the state of the memory at the end of each epoch can be leveraged to identify the root cause of an error. We can further enhance the debugging capability that our solution provides by adding a slow debug mode where additional useful information can be collected.

Low-overhead store tracking: In our previous work [42], we found that using *access vectors* to track the order of stores to an individual cache line resulted in an efficient analysis algorithm. However, such a mechanism for storing and transferring access vectors with each cache line cannot scale well with increasing core count. Our single store count can be piggy-backed on the ECC bits for a cache line, obviating the need for extra storage and bandwidth. In addition, our improved graph construction algorithm eliminates the inefficiencies that required the use of access vectors in the first place.

Configurable granularity: Our solution can be easily configured to infer memory access ordering at multiple granularities, trading off log storage and analysis complexity for accuracy. Inferring at a cache-line granularity is the natural choice, since we track store counts per cache line. It also lets us handle multi-byte reads and writes that do not cross cache lines. This approach might introduce extra dependence edges, potentially resulting in false positives but no false negatives. Even while tracking store counts per cache line, we can infer memory access ordering at a byte granularity by storing byte addresses in the access logs along with the sizes of the accesses. The analysis algorithm will then infer dependence edges only between overlapping memory accesses.

Multi-threaded cores and multi-level cache hierarchy: Our solution can be easily extended to handle multi-threaded cores by maintaining per-thread sequence identifiers in each core and attaching a thread-identifier with each memory access log entry. If there are multiple levels of private caches per core, any of the caches can be utilized for logging.

No-write-allocate L1 caches: Our solution can be extended with some effort to support L1 caches with a no-write-allocate policy. For such caches, current store counts will not be available at the L1 for stores that miss in the L1. We can work around this issue by moving the store count update mechanism to the L2. This solution will, however, require a specialized, implementation-specific approach to handle loads that forward their values from previous stores that missed in the L1.

Test quality: The detection capability of our solution depends on the quality of the stimulus; hence, our solution can only discover bugs exposed by the test programs running on the CMP. Even though we use constrained random test cases designed to stress the memory subsystem and expose bugs, we do not address the issue of test generation in this work.
Out-of-order core correctness: Even though a memory consistency model allows a core to reorder memory operations from a single thread, the core must enforce data dependence ordering between memory operations. These may be direct dependencies between memory operations to the same address or indirect dependencies between memory operations to different addresses through other instructions in the thread. We notice that this requirement is a subset of the fundamental requirement for an out-of-order core to preserve the illusion of sequential execution and, therefore, we believe its verification is a well-researched problem, orthogonal to the verification of shared-memory multi-processor interactions. Our solution does not attempt to detect violations in these requirements.
Execution perturbation: Theoretically, the search for consistency violations should be performed on a memory access graph constructed from the entire execution of a program. Our solution, however, breaks program execution into multiple epochs due to limited logging resources. The cores in the system drain their pipelines and complete all outstanding memory requests in between epochs. Therefore, RAW data dependencies between an epoch and all subsequent epochs cannot exist. *i.e.*, a load in epoch *i* can never read a value written in epoch *i+1*. This perturbation of program execution may hinder the sensitization of certain functional errors.
Graph size: The analysis phase is dominated by the topological sort algorithm for detecting cycles. The worst-case time complexity of this algorithm is $O(|V|+|E|)$, where $|V|$ represents the number of vertices and $|E|$ the number of edges [35]. Previous works have demonstrated techniques that reduce memory access graph size [31, 42], by recording only those accesses that create inter-thread dependence edges and inferring the transitive closure of intra-thread edges. We can also utilize these optimizations to reduce graph size.
Verifying coherence: While our solution can identify coherence violations that manifest as improper store serializations to a single cache line or cycles in a memory access graph, it is not a complete cache coherence verification solution. However, the mechanisms used in our solution are similar to those utilized by the CoSMa coherence verification solution [40]. Therefore, an enhanced hybrid technique that also logs cache line states and incorporates the CoSMa checking algorithm can provide a more robust memory coherence verification solution, albeit with some degradation in performance.

4.9 Experimental evaluation of memory consistency validation

Experimental Framework

Our experimental framework is built upon the Ruby memory system simulator in the Wisconsin Multifacet GEMS toolset [79]. We configured Ruby to simulate the memory hierarchy for a 16-core CMP using the MOESI directory-based cache coherence protocol with 32kB, 8-way private L1 data caches, a single shared 8MB L2 cache and a 4x4 on-chip mesh interconnect. Cache lines are 64 bytes in size. We augmented the existing trace-based memory operation request driver with support for fences and intra-core memory operation re-ordering windows. This gives us the ability to investigate the effects of memory instruction re-ordering, fence instructions and a range of memory consistency models. We implemented a reordering window of 16 memory operations and a constrained random re-ordering policy that respects the requirements of the memory consistency model. We configured our solution to track memory accesses at a cache-line granularity.

We developed a constrained-random test suite of 10 multi-threaded traces consisting of memory requests and fence instructions. Each test in our suite contained 1.6 million load, store and fence instructions (100,000 per thread) tuned to exhibit various data-sharing characteristics as summarized in Table 4.12. In addition, we modeled and probabilistically

test	sync	%ld	%st	%fc	addr. pool	false sharing
low-sharing	0	50	50	0	100,000	none
few-writes	0.5	60	20	20	10,000	none
few-reads	0.5	20	60	20	10,000	none
synch40	0.4	60	40	0	1,000	none
false-sharing	1	-	-	-	1,000	high
fence40	0	30	30	40	10,000	none
mixed-medium	0.3	40	40	20	10,000	medium
mixed-low	0.2	30	30	40	100,000	low
synch100	1	-	-	-	1,000	none
high-sharing	1	-	-	-	10	none

Table 4.12 Characteristics of the evaluation test suite. Each test in our suite consists of 16 threads. Our constrained-random trace generator produces special synchronization sequences to maximize sharing between the threads. These sequences are generated with the probability shown in the *sync* column. When a synchronization sequence is not generated, load, store and fence instructions are produced with the percentages in the *%ld*, *%st* and *%fc* columns, respectively. The size of the address pool, listed in the *addr. pool* column, is used to control how many unique addresses are generated. We introduce false sharing in some of our tests by generating multiple addresses that map to the same cache line.

id	type	description
bad-order-LD	local	some intra-thread load re-ordering restrictions are violated
bad-order-ST	local	some intra-thread store re-ordering restrictions are violated
bad-order-all	local	some intra-thread load and store re-ordering restrictions are violated
bad-fence-timing	local	fences have no effect on memory operations dispatched on the same cycle
data-dep-violated	local	ordering that violates local data dependency
store-reorder	local	stores reordered in store buffer, regardless of fences
nonatomic-store	global	a store is not visible to all cores at the same time
silent-owner	global	owner of a cache line doesn't respond to GET requests
invisible-store	global	store invisible to other cores
simult.-writer	global	multiple writers to a cache line

Table 4.13 Injected bugs. We injected two types of bugs – those that affect ordering local to a core and those that affect global ordering. The local ordering bugs were injected in the instruction scheduling logic and manifest as violations of the ordering constraints set by the consistency model. Note that the *bad-order-** bugs affect both the implicit ordering constraints in TSO and SC, and the explicit fence constraints in RMO. The global ordering bugs were injected in the memory subsystem and manifest as violations of the data dependencies enforced by a coherent system.

injected a total of 10 bug manifestations, described in Table 4.13, to investigate the ability of our solution to detect bugs.

Evaluating bug detection capability

We investigated the capability of our mechanism to detect a sample set of local and global ordering violations. In Figure 4.27, we show a summary of bug detection results for all the bugs and memory consistency models investigated. In these experiments, our solution was configured to use 16kB per core for log storage. Each bar in the graph indicates the number of tests, out of a total of 10, that exposed a bug manifestation which was detected either as an illegal store ordering (shown in light blue) or as a cycle in the memory access graph (shown in purple). Red cross marks are used for bugs that were not detected by our solution. Note that our solution constructs a graph and performs cycle detection only if a bug is not detected by the store order checker, as illustrated in Figure 4.24.

We observe that, with the exception of two bugs, our solution detects the injected bugs for more than 80% of the test cases. For the *bad-fence-timing* bug to manifest, a memory operation must be dispatched on the same cycle as a fence instruction restricting its ordering. This bug is a rarely occurring event in our RMO configurations. Our TSO and SC configurations are not affected, since ordering rules are implicitly enforced without the need for fence instructions. As discussed in Section 4.8.4, our solution does not address local data dependency violations. Therefore, the *data-dep-violated* bug, which affects intra-

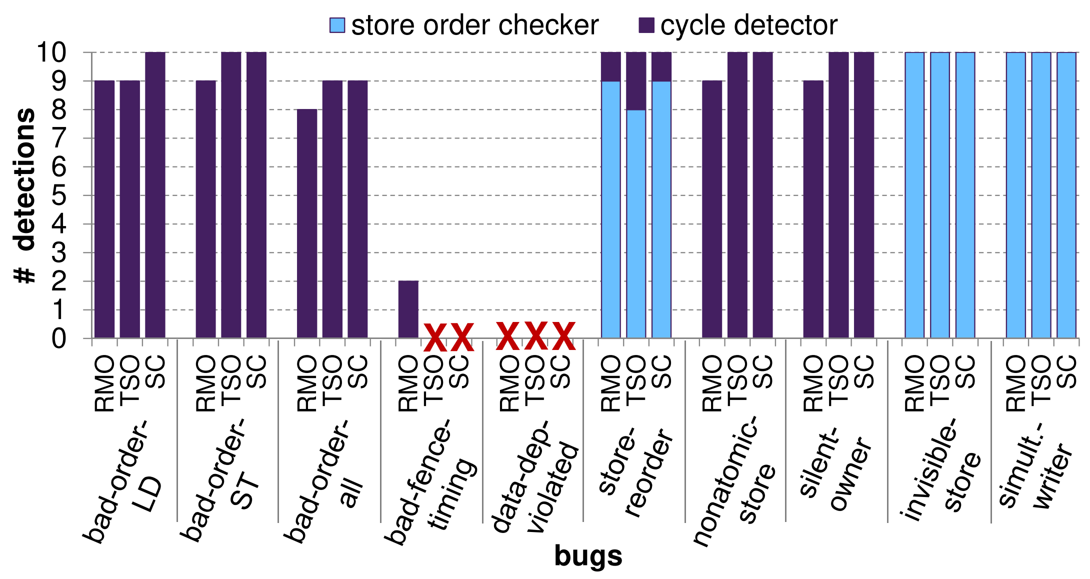


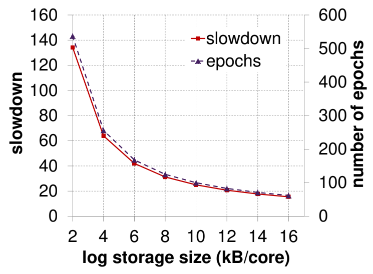
Figure 4.27 Bug detections. We configured CMPs with RMO, TSO and SC consistency models. We ran our experiments by injecting one bug at a time in each configuration and running our 10 tests. We report the number of tests where the injected bug was detected. The store order checker detects bugs that manifest as incorrect store serializations. The cycle detector constructs a memory access graph and detects bugs that manifest as cycles.

thread data dependencies, is never detected by our system. For the remaining cases, the bug manifestation did not create a cycle in the memory access graph for some of the test cases. This situation arises when none of the other cores execute memory instructions that allow them to observe the wrong ordering in the affected core.

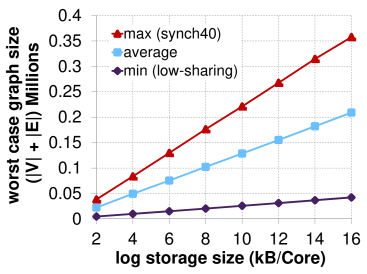
Our store order checker is able to detect global ordering violations that result in illegal serialization of stores to a single cache line. It therefore misses the *nonatomic-store* and *silent-owner* bugs that do not manifest as incorrect store serializations. However, these bugs still produce cycles in the memory access graph.

Evaluating log storage sizes

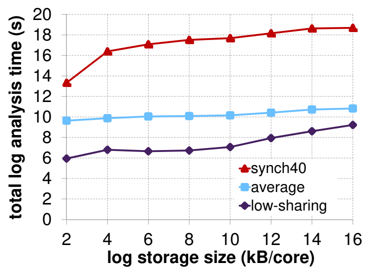
We designed our system with 10 byte log entries (16 bits for the store counter, 20 bits for the sequence identifier, 2 bits for the request type, 42 bits for the physical line address). This configuration was found to be more than sufficient even for our most demanding tests. To study the impact of log storage size, we created 8 L1 data cache configurations, simulating 1 – 8 ways dedicated for log storage. The data arrays for a single cache way provide us with up to 4kB storage, allowing us to store up to 409 entries per cache way. For each configuration, test-case and consistency model combination, we ran two experiments allowing us to investigate log sizes ranging from 2kB to 16kB per core. The logs collected were then analyzed by our single-threaded analysis algorithm running on a host with a 2.80GHz Intel®Core i7-930 CPU and a 1066MHz DDR3 memory.



(a) worst case epoch overheads



(b) worst case graph size per epoch



(c) total graph analysis time

Figure 4.28 The effect of increasing log storage size. **a.** The number of epochs required to complete execution decreases with increasing storage. This also reduces the total execution slowdown caused by the reduced cache capacity and the phase switching overheads. **b.** The sizes of the constructed access graphs appear to increase linearly with increasing log storage size. **c.** Total graph analysis times (sums of analysis times for each epoch) are minimally affected by increasing log size.

Figure 4.28 summarizes the impacts and computation demands of our solution as a function of log storage size for our RMO configurations. In Figure 4.28a , we observe the impact of log storage size on the total test execution time, for the test case exhibiting the worst-case execution time. Note that the total execution time here is the sum of the test program execution times for all epochs and does not account for the time required to aggregate and analyze the logs. We present the slowdown in execution time relative to test execution on a configuration where our solution is completely disabled. This slowdown is due to the following reasons: i) by reserving a portion of the L1 data caches for log storage, we reduce the effective cache capacity that the test program can utilize, potentially resulting in increased L1 capacity misses. We do not observe any significant changes in miss rates for the test programs in our experiments. ii) At the end of each program execution phase in an epoch, the system waits for all pending operations to drain before proceeding to the next phase. This per-epoch drain time is an overhead that is independent of log storage size and thus remains almost constant across all epochs and configurations. The total overhead for an execution is the sum of these drain times and increases with increasing number of epochs. Note that the number of epochs is a function of log storage size – the larger our log storage size, the fewer our epochs.

For each test program, we identified the largest observed graph size under the different log storage configurations; graph size is the sum of the number of vertices and edges. We observed that the worst-case graph size increases roughly linearly with increasing log storage size for all test programs. The *synch40* test program exhibits the maximum worst-case graph size for all log storage sizes while the *low-sharing* test program exhibits the minimum. Figure 4.28b presents the results for the average of all test cases, *synch40* and *low-sharing*. It is worth noting that the per-epoch graph size is also dependent on the consistency model, the frequency of fence instructions (if any), and the amount of data sharing between the multiple interacting threads. These results are from multiple synthetic tests of varying characteristics running on a configuration with the RMO consistency model. Real-world applications would generate far smaller graphs due to limited inter-thread interactions and far fewer fence instructions.

Figure 4.28c summarizes the sum of per-epoch graph analysis times for *synch40*, *low-sharing* and the average of all test programs. Even though a larger log storage size results in larger graphs – hence larger graph analysis times per epoch – there are fewer epochs to analyze. Therefore the total analysis time changes fairly slowly with increasing log storage size. In fact, the maximum increase we observe for an 800% increase in per-core log storage size (from 2kB to 16kB) is 55% for the *low-sharing* test program.

Table 4.14 provides a detailed look into a memory access graph constructed from one execution epoch of the *synch40* test program. The system was configured with the RMO consistency model, 16kB of per-core log storage and had the *bad-order-LD* bug injected. The resulting graph had a total of 26,563 vertices and 316,271 edges. The bug manifested as a cycle in this graph spanning only 4 memory accesses from two cores, reading and writing to two memory locations.

core	**vertices**	**local edges**	**global edges**
0	1620	57,007	5019
1	1624	17,086	2421
2	1624	15,541	2232
3	1621	15,864	2225
4	1621	16,342	2340
5	1622	15,331	2220
6	1623	15,505	2270
7	1614	13,935	2087
8	1623	13,719	2099
9	1620	13,023	2025
10	1618	13,786	2062
11	1619	14,373	2135
12	1618	13,974	2145
13	1617	16,332	2282
14	1622	12,764	1975
15	1618	14,035	2117
Total	**26,563**	**278,617**	**37,654**

Table 4.14 Memory access graph case study. The *bad-order-LD* bug was exposed after the first epoch of the *synch40* test program executing in RMO mode when using 16kB log storage. We present a breakdown of the resulting memory access graph. For each core in the system, we show how many memory accesses, consistency order (local) edges and outgoing dependence order (global) edges were added to the graph. Our algorithm found a cycle involving 4 memory accesses to 2 distinct addresses performed by 2 cores.

Hardware Overheads

Our solution adds a modest amount of hardware overhead. This overhead is mainly due to the extra storage required to track information for in-flight memory operations. A 20 bit sequence identifier register and its associated retirement register are added to each core. Each entry in the load and store queues is increased by 20 bits to hold the sequence identifier for in-flight memory operations. Assuming a size of 32 entries for both the load and store queues, the total overhead due to this augmentation becomes 160 bytes. The log-delay buffer needs to buffer 10 bytes of information per entry. If we design our log-delay buffer to hold as many entries as the load queue, the store queue and the post-retirement store buffer (assumed to be 16 entries), we have an overhead of 800 bytes. The overall per-core storage overhead of our solution is then 965 bytes, which is less than 72% of the size of the Intel®Haswell physical register file [52].

The hardware modifications we introduce are not on the critical computation paths, as all of the control logic we add operates in parallel with the normal processor paths. Our additions interfere with normal processor paths in two places: i) when sampling signals from the normal paths and ii) when driving multiplexers that choose between our control/data and normal processor control/data. For the latter, we would add at most 2 NAND gates (approximately a 50ps delay at 22 nm) to the normal processor path, which is hardly sufficient to make a non-critical path critical. The former can increase capacitive loading on the outputs of some gates. However, we can use sleep transistors to disconnect our solution from the power rails, effectively eliminating the capacitive load during normal operation.

4.10 Related work on validation on acceleration platforms

A plethora of solutions are available for simulation-based validation of digital designs using software-based simulation [113]. A simulation-based validation environment commonly involves checkers that are connected to the design. These checkers are written in high-level languages, such as C/C++, SystemVerilog, and interface with the design via a testbench. Unfortunately such validation schemes cannot leverage the performance offered by hardware-accelerated platforms for validation, namely simulation acceleration, emulation, or silicon debug. Prior research has investigated synthesis of formal temporal logic assertions into synthesizable logic [4, 38], targeting those platforms [23, 24]. Techniques for using reconfigurable structures for assertion checkers, transaction identifiers, triggers and event counters in silicon have also been explored [5]. However, synthesizing all checkers to logic is often not viable for multiple reasons. Software checkers are often developed at a higher level of abstraction for a design block, thus a direct manual translation to logic will run into the challenge of addressing logic implementation details and can be error prone. Though these checkers can be translated into temporal logic assertions and subsequently synthesized with tools such as those described in [4, 24], the size of the resultant logic is often prohibitive for our context. Indeed, the logic implementation of a checker implementing a golden model for a microarchitectural block is often as large as the block itself, and such vast overhead is not tolerable for large blocks. Schemes to share checking logic by multiple checkers via time-multiplexing has been proposed for post-silicon domain [50], however the range of multiplexing is too limited to offer the same degree of flexibility as software checkers. Our checker approximation work is the first attempt at consciously approximating the checking functionality of traditionally software-only checkers so as to realize low-overhead hardware implementations geared towards acceleration platforms.

On the data logging front, acceleration and emulation platforms permit recording the values of a pre-specified group of signals [115], which can be later verified for consistency by a software checker. Recently, a solution was proposed for adapting an architectural checker for a complex microprocessor design to an acceleration platform [29] using this approach: low overhead embedded logic produces a compressed log of architectural events, which is later checked by an off-platform software checker. However, an architectural checker cannot provide the level of insight on design correctness, which a number of local checkers for microarchitectural blocks can. At the architectural level, the information gathered is limited to events modifying the architectural state of the processor; in contrast, microarchitectural checkers track events occurring in individual microarchitectural blocks, generally entailing many more signals. We believe that our synergistic checking approach is the first attempt at adapting several microarchitectural checkers for efficient checking on acceleration platforms.

Finally, approximation of logic functions has been proposed in other related domains, including binary decision diagrams (BDD) [95], timing speculation [43], typical-case optimization [16, 28], and approximate computing [53, 85]. To the best of our knowledge, ours is the first work to leverage approximations for efficient functional verification.

4.11 Related work on memory consistency validation

When verifying a shared-memory multiprocessor, a precise understanding of its consistency model is required. Lamport formalized sequential consistency in [69]. Since then, several consistency models have been proposed with the intention of allowing optimized hardware implementations and/or improving the programmability of shared-memory multiprocessors [8, 77]. Several works attempt to analyze and formalize various memory consistency models implemented by modern shared-memory multiprocessors [15, 98, 97, 9]. Our solution is designed to verify the correctness of memory consistency model implementations. It is designed to be adaptable to different types of memory consistency models.

A directed graph constructed from the dynamic instruction instances in a multi-threaded execution has been used to analyze the performance and correctness of executions [100, 27]. Researchers have proposed verification solutions that leverage the property that an ordering violation manifests as a cycle in such a graph. TSOtool [54] is a software-based approach that constructs a graph from program order and data dependencies captured from pseudo-randomly generated test programs. To capture data dependencies, these programs are generated with unique store values and special code to observe loaded values. Roy *et al.* [96] use an approach similar to TSOtool. Chen, *et al.* [31] augment a shared-memory multiprocessor with hardware to capture and validate memory operation ordering. DACOTA [42] is post-silicon solution that captures memory operation ordering in hardware by repurposing existing resources and performs graph construction and analysis using software running on the multiprocessor under verification. Our solution is similar to this line of research in that we capture memory operations in hardware and construct a graph to check for bugs. However, our consistency model agnostic mechanisms for capturing memory operations are novel contributions.

Other researchers have proposed solutions that do not rely on graph analysis to verify the correctness of shared-memory multiprocessors. Lin, *et al.* [72] insert operations in multi-threaded test programs that check the correctness of values loaded from memory. Meixner and Sorin [81] conquer the memory consistency verification challenge by identifying three invariants and designing hardware checkers for each. Special programs known as "litmus tests" have been used to systematically compare and contrast between multiple memory consistency models [75, 10].

4.12 Summary

In this chapter, we presented three APA-based solutions to tackle the challenges of efficient test execution and bug detection on acceleration and post-silicon validation platforms. These platforms can execute tests at high speeds, allowing functional verification to reach to the system-level design behaviors that are impossible to reach with software simulation alone. Unfortunately, they offer limited visibility into the design's internal signals at any given time. Often engineers have to sacrifice test execution performance and spend extra effort to implement bug detection mechanisms for these platforms. Our solutions offer mechanisms for automated and robust bug detection on these platforms that prioritize the ability to detect system-level bugs efficiently. In addition, they consciously trade detection accuracy for efficiency.

In our first solution, we propose mechanisms that approximate the functionality of robust software checkers to enable their deployment on acceleration platforms. This is the first time intentional approximation of checking functionality was used to reduce the size of hardware checkers during accelerated simulation. Our checker approximation techniques reduce hardware overhead by 59% with only a 5% loss in accuracy. In our second solution, we demonstrated some checking approaches that embed checking and compression hardware while keeping some complex checkers in software to reduce data communication by 32% with only a 6% hardware footprint. To the best of our knowledge, this solution is the first attempt at enabling robust microarchitectural-level checking with reduced communication overhead on acceleration platforms. Finally, we developed a mechanism for uncovering bugs in the implementations of the memory consistency models. Our solution augments a multicore design with minimal extra logic to record memory transactions during test execution. We also developed an algorithm that checks these recorded transactions to identify bugs with high accuracy and within a short time.

Test execution and bug detection are the backbones of modern functional verification. The gamut of solutions discussed in this chapter enable efficient bug detection on fast test execution platforms. These solutions can be deployed in conjunction with the mechanism discussed in Chapter 3, further reducing the amount of time and effort spent on the functional verification of complex designs. The next chapter tackles the last activity in the functional verification process from Figure 1.3: bug diagnosis.

Chapter 5
Addressing Diagnosis

In this chapter, we present a solution for automating bug diagnosis designed through the APA approach. Our automation prioritizes the traditionally manual, time-consuming, and mostly ad-hoc effort to locate the design component that is responsible for a bug detected during system-level validation on acceleration and post-silicon platforms. We recognize that, due to the limited observability afforded by acceleration and post-silicon validation platforms, checking methodologies that compare only the architected state of a design with the state of an ISS executing the same test are widely used. We discover that by automatically analyzing patterns of architectural state mismatches between a design and its ISS, we can approximate the manual root-cause analysis performed by engineers. A successful automatic analysis can save weeks of manual diagnosis effort.

We employed our APA insights to develop a solution we call BugMD (**Bug** **M**ismatch **D**iagnosis), illustrated in Figure 5.1. Traditional design-to-ISS checking solutions stop testing when a bug manifests so as to avoid execution divergence between the design and the ISS that will corrupt subsequent state comparisons. To overcome this challenge, BugMD introduces a state-synchronized ISS co-simulation strategy that allows us to collect multiple independent mismatches beyond the first manifestation of a bug. BugMD utilizes a machine learning classifier to learn patterns to pinpoint the location of a bug from a collection of mismatches, which we refer to as *bug signatures*. Furthermore, we introduce an automated synthetic bug injection framework to approximate real-world bugs for training our classifier model. Our experimental results show that in a processor design with 12 design units, BugMD can correctly localize a bug 72% of the time while narrowing the likely candidates to 3 units with 91% accuracy.

The remainder of this chapter is organized as follows. We present an overview and challenges of bug diagnosis on low-observability validation environments in Section 5.1. We discuss the tradeoffs of our approximations in Section 5.2. We present the details of our solution in Sections 5.3 - 5.6 and our experimental results in Section 5.7. We discuss related work in Section 3.7 and end by summarizing our work in Section 3.8.

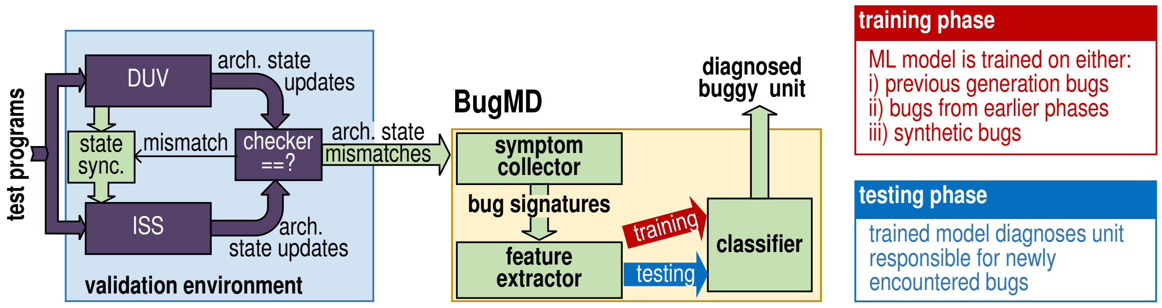


Figure 5.1 BugMD overview. In co-simulation and acceleration methodologies, the design-under-verification (DUV) executes in lockstep with a golden instruction set simulator (ISS) (purple). BugMD (green) augments this methodology with a mechanism to update the ISS state upon a bug manifestation, so that the two models are re-synchronized and subsequent manifestations of the bug can be observed. The complete set of bug manifestations logged by the symptom collector are assembled into a "bug signature", which is then transformed into a "feature vector" by the feature extractor, and transferred to the classifier for diagnosis.

5.1 Background on low-observability bug diagnosis

Bug diagnosis is the most time-consuming component of the debugging process. Engineers sift through several thousands of cycles of test execution data to locate a design component responsible for a bug. The hunt for the design component harboring a bug starts from the design signals that manifested the bug. The nature of the deviation from the expected values for these signals is used to select other design signals to investigate. Engineers then look back in time for any abnormal value changes in these selected signals. This process of finding abnormal behavior, selecting more signals, and inspecting past value changes continues until the bug is localized to signals originating from a single design component or sub-system. The design and verification engineers responsible for that component or subsystem are then tasked with finding the root cause of the bug and implementing a fix.

Diagnosis is particularly challenging for system-level validation on acceleration and post-silicon validation platforms due to the limited observability into a design's internal signals [109] and the large number of components integrated at the system-level. Bugs are usually detected at system-level interfaces many cycles after and many signals away from the actual occurrence of the bug. Engineers do not have enough information about the signals in the design to undertake the localization process. When possible, they re-execute a test with more detailed but slow monitoring capabilities enabled for a selected few signals. Hardware structures designed to assist debugging can be configured to monitor selected signals in the silicon prototype [94, 74, 19, 103] and acceleration platforms can collect signal waveforms at a cost to test execution speed [29].

Bug detection mechanisms that compare the architected state of a program executing on the design with the architected state from an ISS executing the same program are widely used in system-level validation environments. The architectural state of the design is the state of the design that can be observed from the firmware/software layer: memory content, memory mapped registers for accelerators and IP components, and architected state for processors. ISSs are capable of executing tests at very high performance (only 2-3 times slowdown from the manufactured silicon processor) and are considered the golden reference model to determine correct program state. Traditionally, test execution stops when the state of the design differs from the state of the ISS and diagnosis commences. The information obtained from the architectural level bug manifestation is seldom enough to locate the source of a bug, leading to multiple re-executions and an arduous localization process.

5.2 APA: benefits and accuracy tradeoffs

BugMD enhances the design-to-ISS checking framework discussed above to automatically localize bugs. Manual localization of a system-level bug may consume weeks of effort [73]. Correct, automatic localization eliminates this effort altogether. Since BugMD relies on machine learning to automatically localize bugs, it does not always identify the buggy design components correctly. An approximate localization solution like ours is beneficial only if the time savings from correct localization outweigh the extra time spent on recovering from an incorrect localization. We use the formula in Equation 5.1 to estimate the reduction in the overall localization effort afforded by an automatic localization solution.

$$\begin{aligned} &Reduction = 1 - \frac{a \times C + (1-a)(O+E)}{E}, \\ &\quad \text{where:} \\ &\qquad R \ \text{ is average accuracy of automatic localization} \\ &\qquad C \ \text{ is average time spent on correct automatic localization} \\ &\qquad O \ \text{ is average overhead of incorrect automatic localization} \\ &\qquad E \ \text{ is average time spent on manual localization} \end{aligned} \tag{5.1}$$

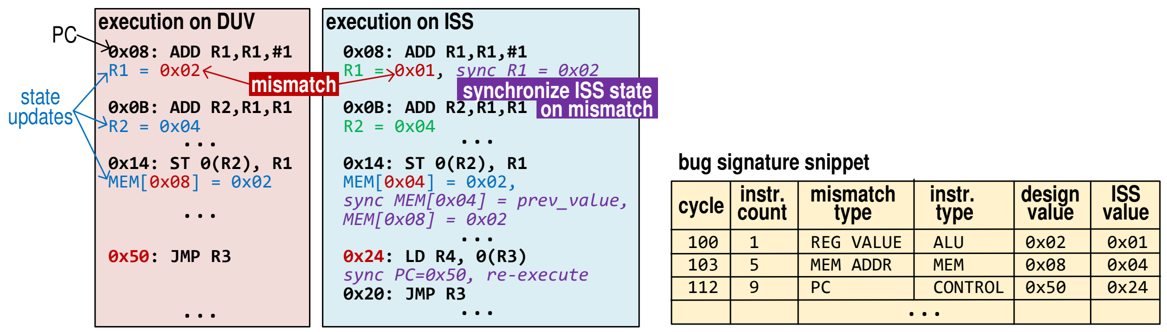
In the best case, where automatic analysis completely eliminates diagnosis effort ($C = 0$) and the overhead of recovering from an incorrect localization is negligible ($O \approx 0$), the reduction in effort is the same as the accuracy of automatic localization ($Reduction = a$). In a setting where automatic localization points to a single design unit, time spent on manual localization is far larger than the time spent on automatic localization ($C \ll E$, $\frac{C}{E} \approx 0$).

If we take the accuracy we obtained from our experiments ($a = 0.72$) and conservatively assume that an incorrect localization results in extra wasted effort that consumes half as much time as a fully manual effort ($O = 0.5E$), we get a 58% reduction in time spent on localizing bugs. Note that the manual localization performed after an incorrect localization, in addition to the incorrectness overhead, is already accounted for in the equation. For an automatic solution that provides a candidate set of design units instead of pinpointing to a single design unit, engineers have to perform additional manual localization to eliminate the unlikely candidates. Assuming that the manual diagnosis effort reduces proportionally to the reduction in candidates, the incorrectness overhead includes the additional manual effort, and using results from our experimental evaluation (12 design units, localized to 3 candidates), we get: $a = 0.91$, $C = 0.25E$, and $O = 0.75E$. These parameters give a reduction of 61.5%. These estimates show that despite the approximate nature of BugMD, it offers significant savings in localization effort.

5.3 Automating bug diagnosis with BugMD

BugMD employs a machine learning classifier to pinpoint buggy design units. It provides a mechanism for collecting multiple bug manifestations from a single test execution. After detecting a bug manifestation by comparing the architected state of a design with that from an ISS, BugMD updates the ISS state with the erroneous design state. This way, both models are pursuing the same incorrect execution, and we can detect any subsequent bug manifestations for the same program execution.

Note that in a traditional validation setting, localization is attempted based on the first, single mismatch between the two models. For each bug manifestation, we log a new *bug symptom*, which we aggregate into a *bug signature*. The collected symptoms are compressed into feature vectors that are transferred to a classifier model [11] for analysis and localization. Our classifier is trained on known bugs, *e.g.*: bugs fixed in previous design generations and bugs fixed in earlier phases of verification. In addition, we created a *systematic and efficient infrastructure to train the BugMD classifier with synthetic bugs* when those from prior design generations or design phases are not available. Our solution can be deployed alongside current methodologies with minimal changes: we simply need to be able to update the ISS state with that of the DUV after a bug manifests, so that we can gather additional symptoms for that same bug.



execution on DUV

```
0x08: ADD R1,R1,#1
R1 = 0x02

0x0B: ADD R2,R1,R1
R2 = 0x04
        ...
0x14: ST 0(R2), R1
MEM[0x08] = 0x02

        ...

0x50: JMP R3


        ...
```

execution on ISS

```
0x08: ADD R1,R1,#1
R1 = 0x01, sync R1 = 0x02

0x0B: ADD R2,R1,R1
R2 = 0x04
        ...
0x14: ST 0(R2), R1
MEM[0x04] = 0x02,
sync MEM[0x04] = prev_value,
MEM[0x08] = 0x02
        ...
0x24: LD R4, 0(R3)
sync PC=0x50, re-execute
0x20: JMP R3
        ...
```

bug signature snippet

cycle	instr. count	mismatch type	instr. type	design value	ISS value
100	1	REG VALUE	ALU	0x02	0x01
103	5	MEM ADDR	MEM	0x08	0x04
112	9	PC	CONTROL	0x50	0x24
		...			

Figure 5.2 Bug signature example. The state updates from execution on the DUV are compared with those from the ISS. We record mismatches as symptoms and aggregate them into bug signatures. We synchronize the ISS state to avoid cascading failures.

5.4 Collecting architectural bug signatures

Once a bug manifests at the architectural level, it corrupts the program state. This state corruption often cascades down to subsequent instructions in the program, leading to a permanent deviation of the DUV execution from that of the ISS. As a result, conventional debugging techniques rely exclusively on the first bug manifestation to diagnose a bug. However, if cascading corruptions could be prevented, and multiple manifestations of the same bug could be observed as several distinct symptoms, interesting patterns could emerge. Some patterns may be identified by investigating multiple manifestations of a bug over several distinct test runs: for instance, a bug may have similar symptoms on different test cases. More interestingly, patterns may exist among the multiple symptoms obtained from a single test execution.

Consider the example in Figure 5.2. A bug in the dispatch logic of a 4-wide, out-of-order core grabs the wrong register value for one of the four instructions it dispatches in a cycle. The first manifestation of the bug is a mismatch in the value being written to a register. For the engineer that observes this mismatch, there could be several reasons for it: the execution logic may have performed the wrong computation, the writeback logic may have corrupted the result, the instruction decoding logic may have provided the wrong operands, *etc.*. In the worst case, the engineer would have to analyze traces of internal design signals for as far back as thousands of cycles (*e.g.*, if the affected instruction was a load operation that missed in the caches) in order to discover what went wrong. However, if we synchronize the ISS state with the state of the DUV and resume execution (and comparisons), patterns emerge that would help narrow down the likely culprits. Here, the 4-instruction periodicity of mismatches, the diversity of affected instruction types, and the differences between the correct and incorrect values give hints that the bug may be located in the dispatch logic.

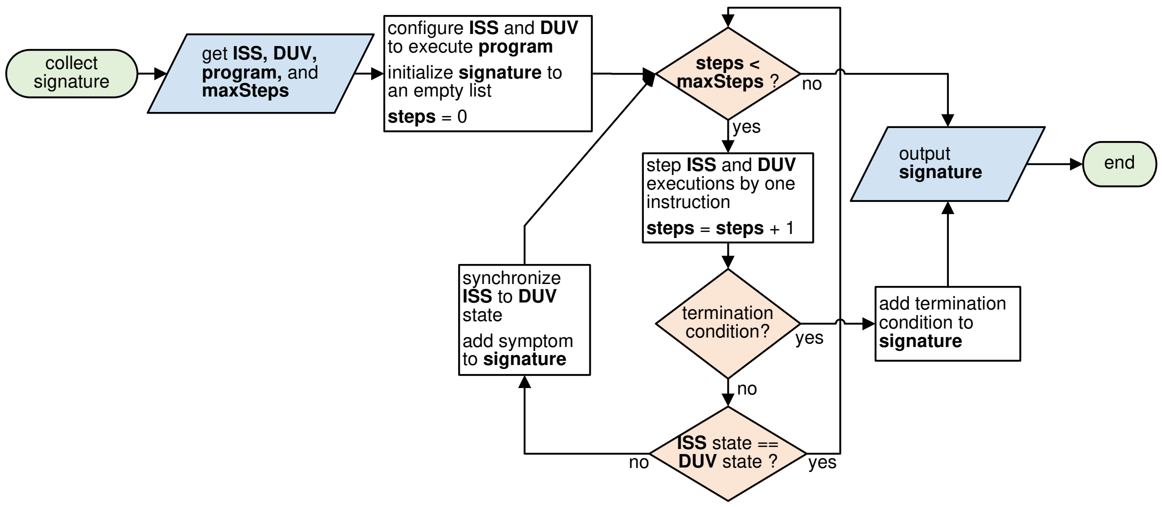

collect signature
get ISS, DUV, program, and maxSteps
configure ISS and DUV to execute program
initialize signature to an empty list
steps = 0
steps < maxSteps ?
no
yes
step ISS and DUV executions by one instruction
steps = steps + 1
termination condition?
yes
no
ISS state == DUV state ?
no
yes
synchronize ISS to DUV state
add symptom to signature
add termination condition to signature
output signature
end

Figure 5.3 Flowchart of BugMD's signature collection mechanism.

BugMD identifies and collects bug symptoms from each mismatch. Mismatches are of the form: i) a write to the wrong register or memory location, ii) a wrong value written to a register or memory location, or iii) a mismatch in the expected program counter. In addition, we consider high-level failures, such as divide-by-zero errors, program hangs, invalid traps, page faults, *etc.*. Previous works have utilized these high-level failures to detect the presence of transient and permanent faults in hardware [68, 71, 111]. A symptom includes the type of mismatch, the current simulation cycle, the instruction count, the state update values from the design and the ISS, and the instruction type. For each test execution, we aggregate the symptoms from multiple mismatches and failures into a bug signature. This aggregation continues until we complete the execution of the test program, the program terminates due to abnormal conditions (*e.g.*: segmentation faults, division by zero, *etc.*.), or until a fixed number of instructions are executed after the first bug manifestation. The flowchart in Figure 5.3 summarizes the signature collection process.

The ISS modifications that BugMD requires are simple enhancements to the ISS data structures used to maintain the architected state. Firstly, all ISS architectural state updates should be made visible to BugMD's symptom collector. Ability to read ISS state updates is a feature that is already available as it is required for comparing DUV and ISS states. Secondly, the ISS should be enhanced to allow modification of its architected state from BugMD's synchronization mechanism. This enhancement should be fairly straightforward to implement for anybody with even moderate familiarity with the inner workings of the ISS. Finally, note that our ISS synchronization approach may trigger unexpected behaviors in the ISS. We expect the ISS designers to ensure that their ISS can execute correctly from any legal architected state. However, a synchronization event may lead the ISS to an illegal state, which may result in side-effects from executions that are incorrect or not defined by the architecture. Examples include the execution of unimplemented instructions, invalid system calls, ISS crashes, *etc.*. BugMD treats these side-effects as bug symptoms.

We expect the effort required to implement ISS modifications to be minimal. Moreover, a single ISS is typically used over multiple design variants and generations; thus, the one-time modification effort is amortized over the lifetime of the ISS. It took one graduate student less than a week's worth of effort to enhance the ISS used in our experimental framework with the modifications described above. It took even less effort to modify another ISS for a subset of the Alpha instruction set architecture (ISA). Note that a new ISS needs to be modified for use with BugMD. The architectural bug signatures generated by the modified ISS, however, are architecture independent. Hence, BugMD's feature extractor and classifier do not require any modifications for use with new ISSs or designs.

5.5 Learning patterns from bug signatures

5.5.1 Extracting features from signatures

For machine learning applications such as our own, raw data has to be converted into fixed-size, real-valued feature vectors for subsequent machine learning. Each feature in the feature vector has to be engineered carefully to provide a meaningful numeric representation for a desired characteristic in the raw data. In addition, the number of features is critical in determining the computation time of the machine learning algorithms as well as the accuracy of their outputs. Unfortunately, there is no standard approach for extracting features from raw data. Feature engineering is an intuitive and experiment-driven process that is typically limited by an application domain. Since BugMD is the first application of its kind, we had to intuitively and experimentally develop our own feature engineering process described below.

BugMD's bug signatures comprise a mix of numeric and non-numeric information. Each symptom is associated with a mismatch type, an instruction type, an expected value, and an observed value. The number of symptoms in a bug signature is only bounded by the number of instructions simulated: these numbers can vary widely among multiple simulations. Our first goal was to ensure that all bug signatures, regardless of the number of symptoms they contain, are converted into feature vectors that have one fixed size. We observed that bug signatures with a large number of symptoms typically have repeating symptom patterns. Thus, we limit the number of symptoms in a bug signature that we consider for feature extraction. Our feature extractor processes symptoms that occur within a fixed number of instructions from the first symptom. In addition to providing a consistent upper bound, this instruction-based window allows us to retain information about the frequency of bug manifestations. For instance, a bug that manifests 2 symptoms within a 10 instruction window is rarer than one that manifests 5 within a similarly sized window.

Our second goal was to capture the extent with which bugs affect architected state. A straightforward way to measure the impact of a bug on architected state is to compute the differences between the observed and expected values of symptoms within our fixed window. We also observed that hardware bugs tend to originate from a few design signals and spread as they propagate to the architected state. The number of architected bits impacted by a bug provides a measure of this spread. We obtain the number of affected bits by computing the Hamming distance between the observed and expected values of each symptom.

In addition to the computed differences and Hamming distances, our feature vectors also need to represent the non-numeric mismatch and instruction types. To this end, we structure our feature vectors by creating up to 5 features for each pair of mismatch and instruction types. For each pair, these features are extracted as follows from corresponding symptoms in the instruction window: i) average of computed differences, ii) average of computed Hamming distances, iii) standard deviation of differences, iv) standard deviation of Hamming distances, and v) a count of the symptoms as a fraction of the total number of symptoms in the window. Averages provide a summary of the effects a bug has on symptoms with similar mismatch and instruction types, while standard deviations indicate how diverse the effects are among these symptoms. By summarizing our differences and Hamming distances as averages and standard deviations per pair of mismatch and instruction types, we limit the length of our feature vectors to a reasonable size. For some combinations of mismatch types and instruction types, some of these features are meaningless and are therefore excluded. Finally we add the total number of symptoms in the instruction window as a feature.

Our feature engineering approach is summarized in Figure 5.4. Applied to our experimental setup, our approach extracted a total of 470 features. For each bug signature, we computed these features from symptoms in a 10,000 instruction window.

5.5.2 Choosing a classifier

There a variety of algorithms available for machine learning applications. The choice of an algorithm for an application is dependent on characteristics of the application and the

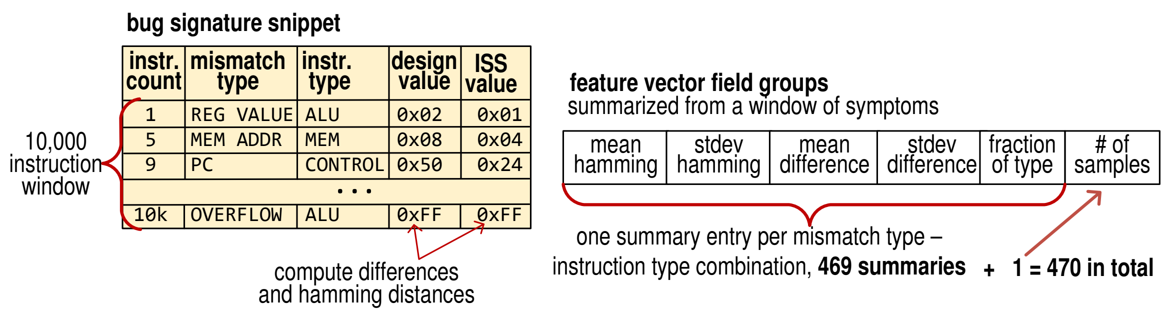


Figure 5.4 Feature vectors. Each feature vector is a summary of symptoms within a user-specified window. For each combination of mismatch and instruction type, the average and standard deviations of the Hamming distances and differences between the DUV and ISS states are computed. The distribution of each combination and the total number of symptoms observed in the window are also computed.

nature of the available data. The task of identifying a buggy unit from a bug signature can be formulated as a classification problem. We investigated several classification algorithms to identify one that is best suited for our application [11]. Based on preliminary analysis of bug signatures and our feature engineering approach, we did not expect our dataset to be linearly separable. In line with our expectations, we observed that non-linear classifiers, such as a decision tree, a k-nearest-neighbor classifier (kNN), and a neural network were more accurate than linear classifiers, such as support vector machines and a naïve Bayes classifier. We found that the random decision forest (RDF) classifier [55] consistently outperformed the other non-linear classifiers that we investigated. Averaged over 10 experiments, our RDF classifier was 8% more accurate than kNN and 20% more accurate than a neural network with one hidden layer.

During training, a RDF classifier partitions the training dataset into multiple random subsets. It then uses these subsets to train multiple decision trees. Each node in a decision tree represents a condition on one feature. The edges originating from the node represent the outcomes of the condition. The leaf nodes represent the categories that feature vectors fall under given the values of their features. During classification, a feature vector is presented to all the decision trees in the the RDF and the predicted categories, which in our case are likely buggy units, are tallied. The RDF chooses the mode of these predictions as its final category.

Note that the accuracy of a classifier is highly dependent on the features used to represent the raw data. In addition, most classification algorithms can be fine-tuned to improve their classification accuracy. Our experiments were not exhaustive enough to draw general conclusions. We may find that other classification algorithms may perform better with another feature engineering approach and different configurations of their parameters. BugMD can be easily adapted to any classification algorithm or feature engineering approach.

5.5.3 Synthetic bug injection framework

Our classifier model is trained with a database of known bugs and their corresponding feature vectors, derived from multiple buggy executions. This training database can be developed from bugs that were fixed in previous design generations or during earlier phases of the verification effort. Engineers label each training input with the design unit found to be responsible for it. Note that only the engineer needs to have the design-specific knowledge to label each input – BugMD's classifier is unaware of the meaning behind each label. Once trained, a classifier can be deployed for use with previously never seen before signatures.

Typically, a larger training database results in better training. In situations where there are not enough known bugs for training, a set of synthetic bugs can be used in their place. We developed a synthetic bug injection framework, inspired by software mutation analysis techniques [114], that randomly injects mutation bugs into gate-level netlists of the design units. Our tools first synthesize each design unit into a technology-independent, gate-level netlist. They then parse the netlist to select random gates and insert a mutation for each gate that takes the same inputs as the original gate but has a different functionality. For example, a 3-input OR gate may be the chosen mutant for a randomly selected 3-input AND gate in the design. For each original-mutation pair, a multiplexer is inserted to select between the two outputs. Each multiplexer represents a synthetic bug in the design; to activate a bug, its associated multiplexer is set to choose the output from the mutant gate. The flowchart in Figure 5.5 summarizes the bug injection process.

Our mechanism for injecting synthetic bugs and collecting training data is extremely low-cost and low-effort. The process of injecting synthetic bugs and collecting their symptoms can be completely automated, allowing it to be performed without taking resources away from other verification efforts. Moreover, the process can start before the availability of the whole design. For instance, it is common practice to develop partial system models to validate specific features: we can thus use these models to gather bug signatures for bugs injected in the units included in these partial models. In addition, we can leverage simulation independence to gather signatures for multiple bugs concurrently. Note that, much like in training with real bugs from prior designs, synthetic bug signatures can also be shared across multiple generations and variations of a design. This further amortizes the cost of generating synthetic bug signatures.

The type of functional bugs we are interested in are usually introduced at the behav-

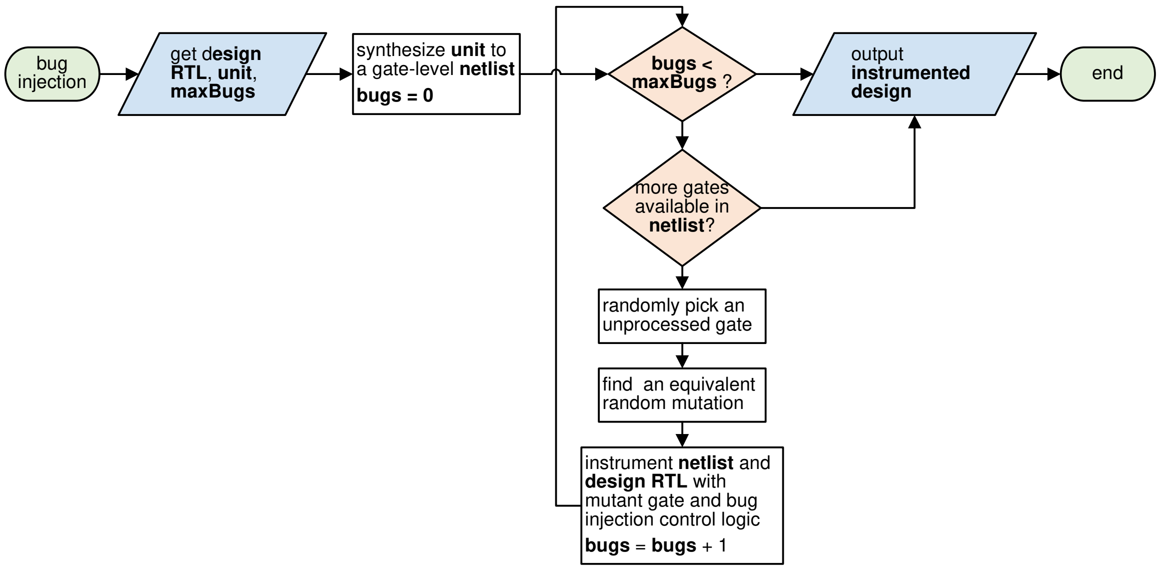

bug injection
get design RTL, unit, maxBugs
synthesize unit to a gate-level netlist
bugs = 0
bugs < maxBugs ?
output instrumented design
end
more gates available in netlist?
randomly pick an unprocessed gate
find an equivalent random mutation
instrument netlist and design RTL with mutant gate and bug injection control logic
bugs = bugs + 1

Figure 5.5 Flowchart of BugMD's bug injection mechanism.

ioral level. Note that our gate-level synthetic bug model is an approximation of bugs at the behavioral level. In addition, synthetic bugs may interact with real, hidden bugs in the system causing unpredictable outcomes. Investigating other synthetic bug models and the interaction with other real bugs is left for future work.

5.6 Discussion of assumptions

Note that for our symptom collection mechanism to work, the bug-free architectural updates from an execution on the design should be identical to those from an execution on the ISS. Similarly, test executions should be deterministic: one execution of a test on the design should be identical to another execution of the same test. For a buggy execution, we expect the bug to manifest consistently for all executions of the same test program. This is a reasonable expectation in typical validation environments.

While our discussion so far presumes the ability to access and compare architectural level state after every committed instruction, this is not strictly required. With a small change in the checking mechanism, BugMD could require much less frequent checking, allowing it to be deployed in post-silicon validation environments where frequent checking may be impractical. This change entails the use of a binary search-and-compare method for identifying mismatches, as detailed below, by running a test program multiple times, either from the beginning or from a last known clean checkpoint.

We first run a test on the DUV, either to completion or abnormal termination (in the presence of fatal bugs), and compare the final architectural state with the final state from the ISS. If there is a mismatch, we re-run the test for half the number of cycles to completion, and repeat the check. Since the ISS is often not cycle-accurate, we can not simply execute the test program on the ISS for half the number of cycles to get the equivalent execution for comparison. We first need to extract the number of instructions completed on the DUV from on-chip performance counters and then execute the test program on the ISS until the same number of instructions are committed. We repeat this process until we find a cycle where there is no mismatch between the DUV and the ISS. We take a checkpoint of the DUV and ISS states and continue the binary search forward. Upon detecting a mismatch, we synchronize the ISS state with that from the DUV and continue the search for more mismatches until the desired number of instructions after the first mismatch has been executed. Even though this approach requires a test to be run multiple times, it reduces the number of comparisons by several orders of magnitude. A similar search based approach has been used to successfully isolate faults in industrial designs [12].

Our discussions so far assume that a bug signature contains symptoms from the manifestations of a single bug in the system. This assumption may not hold in a real-world validation environment: there is never a guarantee that only one bug will manifest during an execution. We believe that BugMD can be easily extended to operate in such scenarios. A bug signature that contains symptoms from multiple bugs can be considered to be the signature for a "composite bug" that is an aggregate of the multiple manifesting bugs. If the manifesting bugs reside in different units, this composite bug can be considered to reside in a "composite unit" that is the union of the multiple units. We can then enhance BugMD with the ability to localize to composite units in addition to regular units by treating the composite units just like any other unit. Engineers can then run other tests to discern among the bugs in a given composite bug.

5.7 Experimental evaluation

We developed our solution on 3 different designs, implementing two ISAs. For lack of space, we report detailed results for only one of our experimental frameworks. This framework comprises a 4-wide, out-of-order processor design described in Verilog and a C++ ISS for the system-level validation flow. BugMD includes a symptom collection and synchronization harness, a synthetic bug injection toolkit, feature extraction scripts, and a machine learning script implemented using Python's scikit-learn [91] library. We only report results obtained using the Random Decision Forest classifier configured to comprise 64 trees. Our

unit	# cells	description
FETCH1	598,671	Instruction fetch logic, stage 1
FETCH2	8,852	Instruction fetch logic, stage 2
DECODE	4,654	Instruction decoding logic
INSTBUF	24,483	Buffer for decoded instructions
RENAME	10,232	Register renaming logic
DISPATCH	998	Instruction dispatch logic
ISSUEQ	39,994	Issue logic and queue
REGREAD	27,205	Physical register file
EXECUTE	24,827	Integer execution logic
LSU	26,129	Load-store-unit
RETIRE	104,069	Reorder buffer and writeback logic
MAPTABLE	3,036	Map table for register renaming

Table 5.1 Design units. The FabScalar design modules were grouped into 12 hardware units. We estimated the size of each unit by counting the number of cells in its synthesized netlist. Units with large storage structures have the largest sizes: *e.g.*, FETCH1 contains the branch target buffer. Engineers label bug signatures used for training with their corresponding buggy unit.

processor design, which implements the SimpleScalar PISA instruction set, was generated using the FabScalar toolset [32]. We modified the SimpleScalar-based ISS that is bundled with FabScalar for use with BugMD. For our test programs, we used several distinct sections of the bzip benchmark that ships with FabScalar. We opted not to include other benchmarks since we found the diversity in the different sections of the bzip benchmark to be sufficient for our experiments. We grouped the design's Verilog modules into the 12 distinct units described in Table 5.1.

We grouped the subset of the PISA instructions that FabScalar implements into 6 types, namely Control, ALU, Multiply/Divide, Load, Store, and Other. Table 5.2 lists the groups of mismatch types that we log in our bug signatures. For certain mismatch types, we observed that some features generated by our feature extraction methodology were not useful (*e.g.*, differences for x-prop group of mismatch types and instruction types for termination mismatch types), and thus were excluded from our feature set. We collected symptom signatures until our tests completed/terminated or until they executed 10,000 instructions after the first mismatch. Our resulting feature vectors were 470 features long.

Using our synthetic bug injection framework, we injected 590 bugs in each of the units in Table 5.1 adding up to 7080 bugs in total. Since we did not have a database of previously fixed bugs for our DUV, we relied exclusively on these synthetic bugs, both to train

group	# of types	mismatched property
value	2	the value of a register or a memory update
address	2	the address of a memory update or the index of a register update
valid	2	the validity of an update
size	1	the size of a memory update
control	2	the program counter and syscalls
bounds	3	out-of-bounds data and instruction accesses
x-prop	10	presence of X bits in updates
abnormal	8	abnormal conditions such as division by zero and ISS anomalies
final	4	termination conditions: normal finish, hang, livelock, and crash
TOTAL	34	

Table 5.2 Mismatch types. Bug symptoms consisted of mismatch types within the groups shown here. In addition to state mismatches, we included erroneous conditions such as division-by-zero and ISS crashes. We had a total of 34 mismatch types.

BugMD and to evaluate its diagnosis capability. To collect bug signatures, we activated one bug in the design at a time and executed test programs, both on the DUV and the ISS. We ran 6 distinct test program executions while activating a single bug per execution and collected over 40,000 bug signatures. We then divided the feature vectors generated from these signatures into disjoint training and testing datasets, ensuring that each unique bug for each unit was exclusively either in the training dataset or the testing dataset. We trained BugMD using the training dataset and evaluated its localization capabilities using the testing dataset. In a real-world scenario, the classifier is trained with data from known bugs or from synthetic bugs. The trained classifier is then used to predict the locations of previously unknown bugs detected during validation.

5.7.1 Localization accuracy

We investigated different approaches to boost the accuracy of classification. Figure 5.6 reports the results of our investigations using a training dataset of 35,352 feature vectors and a testing set of 3,756 feature vectors. We implemented a simple, automated, single-mismatch baseline heuristic that emulates the initial triaging efforts of an engineer who does not have BugMD. This baseline assumes a validation environment where a set of symptoms for only one bug manifestation can be collected and analyzed during testing. We designed our baseline to closely match the initial bug triaging steps that would be taken in the absence of multiple, independent symptoms. Note, however, that in a real-world scenario, a verifica-

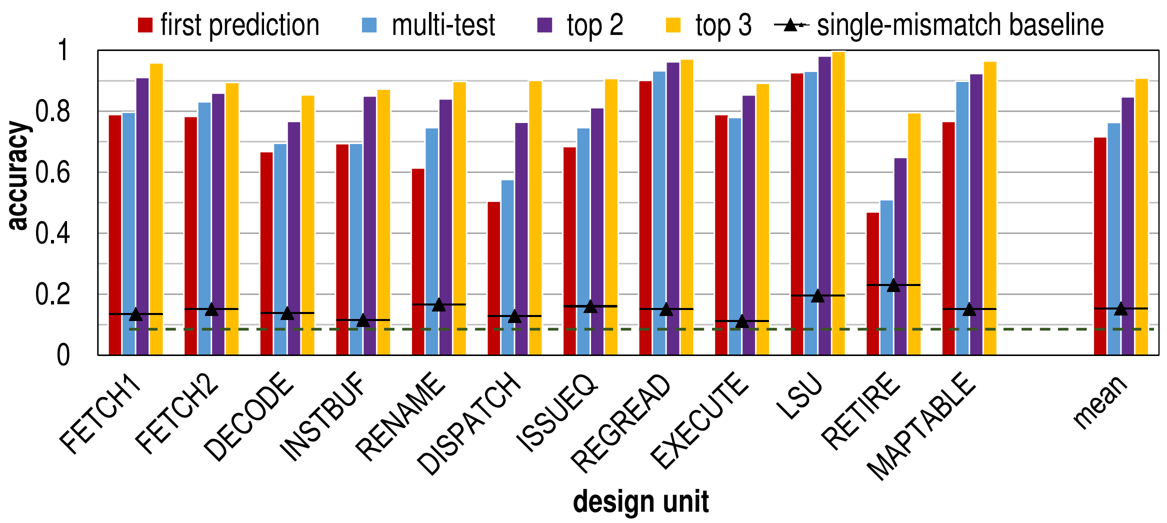
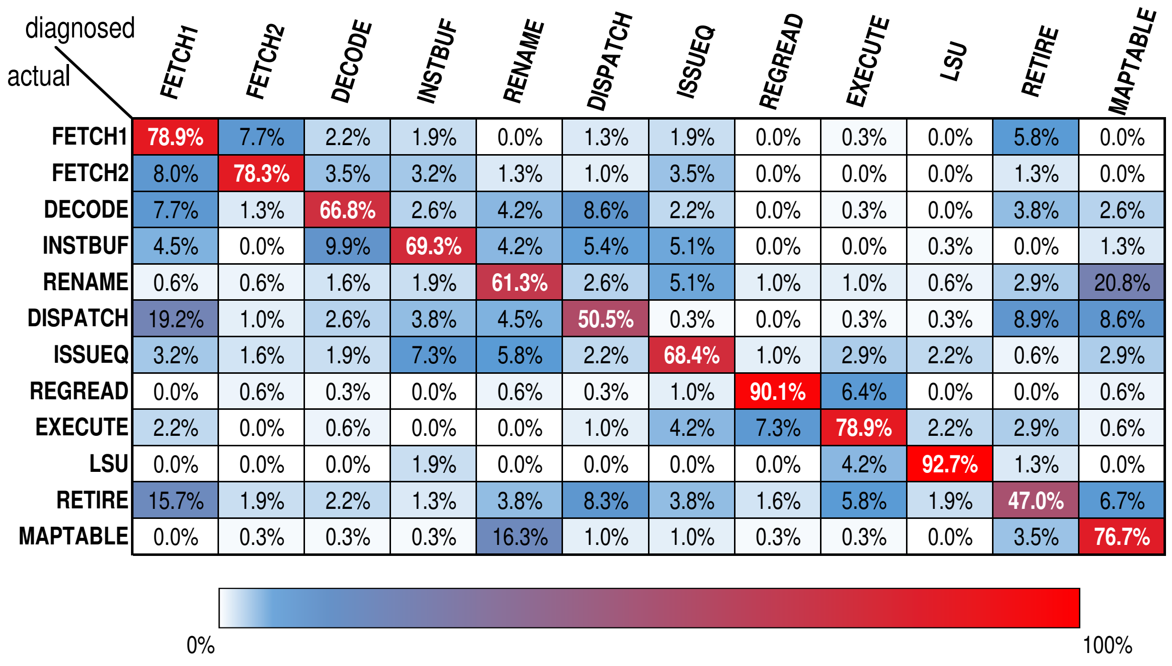


Figure 5.6 Breakdown of localization accuracy for each unit. Our classifier pointed to a single unit with an average accuracy of 70%. This first prediction was further enhanced to 77% if the same bug was exposed by multiple distinct tests and the most common prediction was chosen. BugMD narrowed the search space from 12 units to 3 likely units with a 90% accuracy. The dashed green line indicates the accuracy for a random guess.

tion engineer would run more tests while observing multiple internal design signals and draw from accumulated experience to refine the estimate from the first attempt. Our baseline heuristic, shown with the black horizontal segments, correctly identified a buggy unit on the first try about 15% of the time on average.

Our classifier correctly identified a buggy unit on the first try about 70% of the times, as demonstrated by the red "first-prediction" bars; 90% of the times, the buggy unit was within the top 3 predictions, as shown by the gold "top 3" bars in the figure. When we ran multiple tests for a bug and took the most common prediction as the final prediction, we achieved an overall accuracy of 77%, shown by the blue "multi-test" bars. Note that a purely random guess would only have an 8.3% chance of correctly localizing a bug. We observed that some units, such as the LSU, had bugs that were often correctly localized whereas others, such as the DISPATCH unit, exhibited bugs that were difficult to localize. The LSU bugs mostly affected just memory operations and were easy to discern. The small size of the DISPATCH unit limited the number of options our bug injection framework had when injecting bugs, resulting in low-quality training. The relationship between unit size and localization accuracy is to be studied in future work. For other units, the variation is due to factors including the diversity of bug signatures from difficult-to-localize units, which made their feature vectors appear to be closer to those from other units than to each other. For instance, feature vectors from bugs in the FETCH1 unit demonstrated a lot of PC and instruction overflow mismatches. Thus, the classifier opted to localize bugs with a



actual \ diagnosed	FETCH1	FETCH2	DECODE	INSTBUF	RENAME	DISPATCH	ISSUEQ	REGREAD	EXECUTE	LSU	RETIRE	MAPTABLE
FETCH1	78.9%	7.7%	2.2%	1.9%	0.0%	1.3%	1.9%	0.0%	0.3%	0.0%	5.8%	0.0%
FETCH2	8.0%	78.3%	3.5%	3.2%	1.3%	1.0%	3.5%	0.0%	0.0%	0.0%	1.3%	0.0%
DECODE	7.7%	1.3%	66.8%	2.6%	4.2%	8.6%	2.2%	0.0%	0.3%	0.0%	3.8%	2.6%
INSTBUF	4.5%	0.0%	9.9%	69.3%	4.2%	5.4%	5.1%	0.0%	0.0%	0.3%	0.0%	1.3%
RENAME	0.6%	0.6%	1.6%	1.9%	61.3%	2.6%	5.1%	1.0%	1.0%	0.6%	2.9%	20.8%
DISPATCH	19.2%	1.0%	2.6%	3.8%	4.5%	50.5%	0.3%	0.0%	0.3%	0.3%	8.9%	8.6%
ISSUEQ	3.2%	1.6%	1.9%	7.3%	5.8%	2.2%	68.4%	1.0%	2.9%	2.2%	0.6%	2.9%
REGREAD	0.0%	0.6%	0.3%	0.0%	0.6%	0.3%	1.0%	90.1%	6.4%	0.0%	0.0%	0.6%
EXECUTE	2.2%	0.0%	0.6%	0.0%	0.0%	1.0%	4.2%	7.3%	78.9%	2.2%	2.9%	0.6%
LSU	0.0%	0.0%	0.0%	1.9%	0.0%	0.0%	0.0%	0.0%	4.2%	92.7%	1.3%	0.0%
RETIRE	15.7%	1.9%	2.2%	1.3%	3.8%	8.3%	3.8%	1.6%	5.8%	1.9%	47.0%	6.7%
MAPTABLE	0.0%	0.3%	0.3%	0.3%	16.3%	1.0%	1.0%	0.3%	0.3%	0.0%	3.5%	76.7%

Figure 5.7 Distribution of prediction accuracies. For the bugs in each of our 12 design units, we kept track of the correct outcomes and where bugs were wrongly localized to. Correct outcomes, shown on the diagonal, far exceeded incorrect ones. For some unit pairs, for example FETCH1 and FETCH2, incorrect outcomes could still be considered useful.

large fraction of PC and instruction overflow mismatches to the FETCH1 unit.

BugMD performed significantly better than our single-mismatch heuristic mainly due to the richer information available to it from the multiple symptoms for each bug – our feature sets and the classifier were able to utilize this extra information to discern bug signatures from different units.

5.7.2 Distribution of localizations

We analyzed the diagnosis outputs from BugMD to understand the distribution of correct and incorrect predictions. We report this distribution in Figure 5.7. We observed that for all units, correct outcomes far exceeded incorrect localization to any single other unit. For some units, the other unit that they were most frequently misclassified to is fairly reasonable. For example, a FETCH1 bug was often misclassified as a FETCH2 bug, a RENAME bug was often misclassified as a MAPTABLE bug, and vice versa for both. These misclassifications are close enough to the actual bug site that they can be considered useful. However, bugs in DISPATCH and RETIRE were often undesirably misclassified as FETCH1 bugs. This behavior indicates that a non-trivial portion of DISPATCH and RETIRE feature vectors fell within the FETCH1 classification boundary learned by our

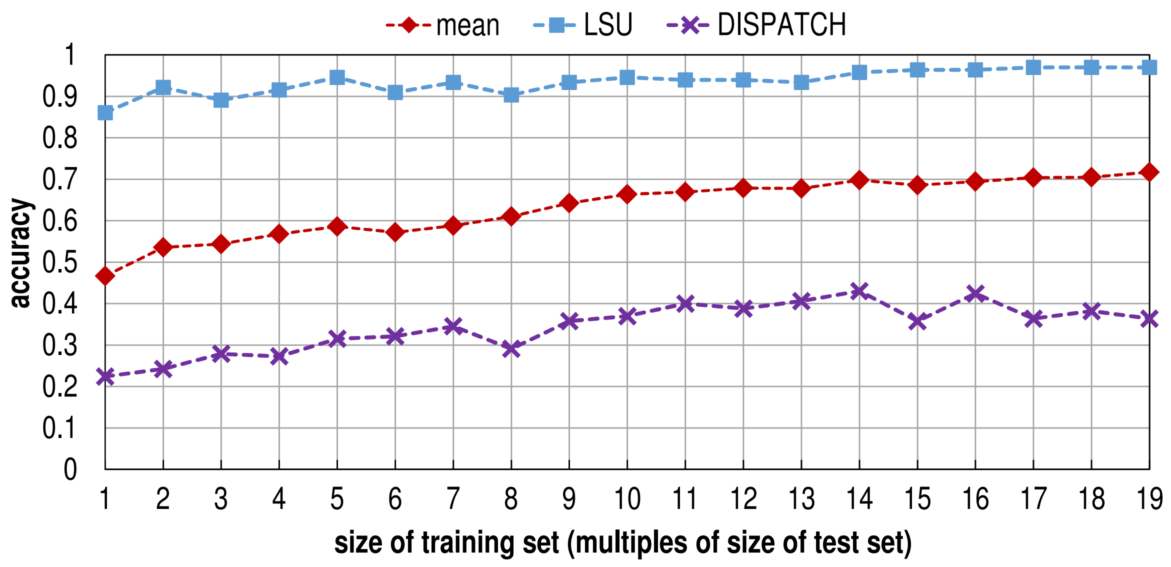


Figure 5.8 Sensitivity to training dataset size. We divided our dataset into disjoint training and test sets. In all cases, the test set contained 1980 feature vectors. The training sets contained feature vectors close to multiples of the test set (1980, 3960, 5940,*etc.*.) as shown. Bugs in LSU were localized correctly even with small training sets, while bugs in DISPATCH were difficult to discern.

classifier. We observed that a large proportion of FETCH1 bugs resulted in a PC mismatch, which then led to early termination either due to invalid instructions fetched from the wrong address or out-of-bounds instruction addresses. A fraction of the bugs in the DISPATCH and RETIRE units affected the targets of branch instructions, which also eventually manifested as PC mismatches and early termination. Some bug signatures with early termination had as few as only two symptoms, which made it difficult to identify discerning patterns, leading BugMD to incorrect diagnoses. However, our classifier still localized to the correct unit at least 3 times more frequently than to an incorrect unit. Note that there is no symmetric relationship in localization accuracies: *e.g.*, the percentage of FETCH1 bugs that appear to be DISPATCH bugs is not the same as the percentage of DISPATCH bugs that appear to be FETCH1 bugs. Thus, Figure 5.7 is not symmetric across the diagonal.

5.7.3 Size of training dataset

Finally, we investigated the impact of training data size on the accuracy of classification. To this end, we partitioned our dataset into several disjoint training and test sets. We trained our classifier using feature vectors generated from the training set and asked it to classify feature vectors generated from the test set. Figure 5.8 summarizes the results from this study. The mean classification accuracy across all 12 design units increased with increasing training data size. Bugs in certain design units were easier to localize than others. We show the individual accuracy trend for the unit with the most correctly localized bugs – the LSU – and one with the least – DISPATCH. We observed that larger training set sizes generally led to better localization accuracies, but the benefits started diminishing after a training size of about 27,000. Highly localizable units, such as the LSU, were not very sensitive to the size of the training set; their bug signatures were quite distinct. The quality of training for smaller units, such as DISPATCH, was limited by the number of gates to inject bugs into; our injection framework had very little room to select and inject high-quality bugs that would result in bug signatures representative of the unit.

5.8 Related work on bug diagnosis

In recent years, a few solutions have been proposed to support bug localization during post-silicon validation. BPS [41] logs measurements of signal activity from multiple executions using a hardware structure added to the design. A clustering algorithm is later used to process these logs to discern among failing and passing tests and identify a candidate set of signals responsible for an intermittent bug. Symbolic QED [73] achieves coarse-grain localization of bugs to processor cores, cache banks, and the crossbar in a multicore SoC by utilizing a combination of bounded model checking, partial instantiations of the design, and test transformations enabled by an extra hardware module added to the fetch stage. We believe that BugMD is the first work that localizes bugs to design units from architecture-level mismatches. Unlike BPS and Symbolic QED, BugMD relies neither on hardware modifications added to the design nor transformations applied to test programs. In addition, it is not limited to post-silicon and can operate on any validation environment that supports DUV-to-ISS state comparisons. Unlike Symbolic QED, BugMD localizes bugs at a much finer granularity, among a larger number of units in out-of-order processor cores.

Several other works have also been proposed that triage bugs by leveraging a combination microarchitectural knowledge [44], microarchitectural or low-level signal traces [44, 93], and SAT-based debugging techniques [93]. Some of these solutions [44, 93] rely on machine-learning algorithms for clustering and/or classification to bin failures and identify their root-causes. To the best of our knowledge, BugMD is the first application of machine learning to pinpoint bug locations without the need for microarchitectural knowledge. BugMD only relies on architectural updates, does not need access to the RTL source code, and is entirely simulation/execution-based.

Finally, Friedler, *et al.* [47] have proposed using information derived from executing a test-case on an ISS to localize the set of instructions responsible for a functional data flow bug in the DUV. Unlike BugMD, this solution focuses on identifying instructions and does not provide any indication of the hardware units responsible for the bugs.

5.9 Summary

In this chapter, we presented our application of the APA approach to design an automatic bug diagnosis solution, which we call BugMD. Due to its manual and time-consuming nature, we prioritized the bug localization effort for automation. We employed a machine learning classifier to learn the correlation between bug sites and patterns of bug manifestations. Our solution is an approximation of the localization process in that it can sometimes give incorrect results. However, we show that even with an accuracy of 72%, it can still shave off 58% of localization time.

Our solution compares a design's architected state with a golden state from an instruction set simulator to collect multiple symptoms for a single bug in a single test run. These multiple manifestations of bugs form bug signatures that are then passed through a machine-learning backend to obtain a prediction of likely bug sites. To train the machine-learning classifier, we developed a synthetic bug injection framework for generating large training datasets when real, previously diagnosed bug signatures are either unavailable or insufficient. BugMD relies only on architectural-level state and does not require any microarchitectural knowledge, making it suitable for low-observability validation environments and portable across design generations.

In this chapter we tackled the last but most time consuming functional verification activity. The solution we presented in this chapter can be deployed jointly with those discussed in previous chapters. The next chapter summarizes the contributions of this dissertation and provides a discussion on future directions.

Chapter 6
Conclusions

This dissertation identifies and addresses functional verification bottlenecks that are crippling modern processor design projects. Functional verification as practiced presently is unable to scale with the increasing complexity and grueling schedules of today's designs. Processor design companies are spending a disproportionate amount of time and resources on ensuring the correctness of an implementation as compared to building the implementation itself: today, verification engineers outnumber design engineers on a processor design project. If this trend continues unchecked, it will soon be infeasible to embark on a new design project. In this chapter, we summarize our contributions towards addressing these functional verification challenges and suggest directions for future work.

6.1 Summary of contributions

Automating planning and test generation for modern SoC designs. Modern SoCs integrate several complex general-purpose cores and special-purpose accelerators that communicate via shared-memory operations. The large number of interacting components in these systems result in a copious amount of design behaviors that can not all be verified in time to meet product release schedules. Traditionally, engineers manually and intuitively identify aspects of the design behaviors to prioritize for verification. We develop an automated solution for identifying high-priority design behaviors for verification. Our solution analyzes the execution of software on a high-level model of a design to identify the design behaviors that are exercised most by software. It can then automatically generate coverage models and compact test programs that capture these important design behaviors. The results of our analysis are approximate because they rely on high-level design behaviors that do not necessarily fully match the timing of actual design behaviors. However, we limit the impact of our approximation by implementing conservative analysis heuristics that take into account the difference in timing between the models. In our experiments, we find that in addition to eliminating the tedious process of manually prioritizing design behaviors for verification, we also cut down the simulation time required for tests that reproduce these behaviors by 11 times. Moreover, by generating coverage models automatically from these prioritized behaviors, we eliminate the need for manually coded coverage models. The compactness of our generated tests make them desirable for regression test suites that are executed repeatedly during the functional verification process.

Enabling efficient test execution and bug detection on acceleration platforms. Simulation acceleration platforms, which offer orders of magnitude faster test execution performance than software simulators, are widely used for functional verification. These platforms are now being used as drop-in replacements for software simulation mainly because they execute a simulation model built from the hardware description of the design, hence providing a fast test execution environment for pre-silicon verification. However, since these platforms do not provide efficient on-platform bug detection capabilities, they are shackled to host computers executing software checkers, which slow performance down. We develop solutions to address the performance penalties associated with robust bug detection on acceleration platforms. Our solutions minimize the communication between an acceleration platform executing a test on a design and a host computer running software checkers. We present an approach that converts software checkers into hardware implementations that can simulate on the acceleration platform along with the design. These hardware implementations approximate the functionality of the original software checkers so as to minimize the sizes of the resulting hardware structures. We also find hybrid approaches that move some low-overhead checkers on-platform along with hardware structures to compress the amount of data that has to be transferred from the acceleration platform to the host computer. Our solutions tradeoff the accuracy of bug detection for reduced overhead of automated checking on acceleration platforms.

Efficient test execution and bug detection for memory consistency verification on acceleration and post-silicon validation platforms. Shared-memory chip-multiprocessor architectures define memory consistency models that establish the ordering rules for memory operations from multiple threads. Validating the correctness of a CMP's implementation of its memory consistency model requires extensive monitoring and analysis of memory accesses while multiple threads are executing on the CMP. Such tests are typically executed on acceleration and post-silicon validation platforms. We developed a low overhead solution from the ground up for observing, recording and analyzing shared-memory interactions for use in acceleration and post-silicon validation environments. Our approach leverages portions of the CMP's own data caches, augmented only with a small amount of hardware logic, to log information relevant to memory accesses. After transferring this information to a central memory location, we deploy our own analysis algorithm to detect any possible memory consistency violations. We build on the property that a violation corresponds to a cycle in an appropriately defined graph representing memory interactions. Our experimental results show an 83% bug detection rate, in our testbed CMP, over three distinct memory consistency models, namely: relaxed-memory order, total-store order, and sequential consistency. Our solution can be disabled in the final product, leading to zero performance overhead and a per-core area overhead that is smaller than the size of a physical integer register file in a modern processor.

Utilizing machine learning for efficient bug diagnosis. System-level validation is the most challenging phase of design verification. A common methodology in this context entails simulating the design under validation in lockstep with a high-level golden model, while comparing the architectural state of the two models at regular intervals. However, if a bug is detected, the diagnosis of the problem with this framework is extremely time and resource consuming. To address this challenge, we developed a bug triaging solution that collects multiple architectural-level mismatches and employs a classifier to pinpoint buggy design units. We also designed and implemented an automated synthetic bug injection framework that enables us to generate large datasets for training our classifier models. Our experimental results show that our solution is able to correctly identify the source of a bug 72% of the time in an out-of-order processor model. Furthermore, our solution can identify the top 3 most likely units with 91% accuracy.

6.2 Future directions

This dissertation introduces an incompleteness-aware perspective for tackling the functional verification of complex processor designs by fully embracing the best-effort nature of the process. While the solutions we presented leverage our APA approach to boost efficiency, there are still several avenues to be explored. Some of these are towards new automations that are enabled by relevant prioritizations and deliberate approximations, whereas others are towards enhancements of the solutions we propose. We summarize these future directions below.

Functional verification for high-level synthesis (HLS) design flows. HLS design flows, where a hardware implementation is automatically synthesized from a design or an algorithm described at a high level of abstraction, are increasingly becoming attractive for the design of special-purpose processors. Even though HLS design flows improve design productivity and minimize the possibility of functional errors, they still need comprehensive functional verification frameworks. This need is especially pertinent for projects where different HLS-designed accelerators are combined into large SoCs. Since the high-level abstractions lack the detailed timing of the RTL models generated by the HLS tools, functional verification performed on the high-level model can never be complete. Design bugs introduced in the high-level model of an accelerator may stay hidden until its RTL model is integrated and tested with the rest of the system. We believe that there are several opportunities for automated verification solutions designed with prioritization and approximation in mind. High-level functional verification solutions should prioritize high-level properties. Tools can then be developed to prioritize and automatically generate tests that target the RTL-level properties that were missed by the high-level verification solutions. In addition, debug frameworks that leverage information from the HLS process can attempt to approximately point to buggy lines of code or flawed synthesis parameters.

Approximate verification for approximate designs. Recently, several researchers have proposed approximate design approaches that deliberately trade off the accuracy of their computations for increased performance or energy efficiency. We believe that the functional verification of approximate hardware will introduce unique challenges that necessitate purposeful prioritization and approximation. Firstly, during verification, engineers should automatically distinguish between an error due to a deliberate approximation and an error due to a functional bug. We expect that the approaches to be used in this context will themselves be approximate statistical analyses. Secondly, the exact aspects of a design are by definition more important from the perspective of functional verification than the approximate aspects. Therefore, automated verification solutions will prioritize the exact aspects of a design with the understanding that consumers are already expecting errors in the approximate aspects of the design. Finally, a bug and an error from an approximation may manifest at the same time during test execution. In such cases, approximation errors may mask design bugs or make their diagnosis very challenging. Diagnosis solutions should "filter" the impact of the approximate error from the manifestation to be able to pinpoint the source of the bug. In addition to the actual diagnosis, we expect this filtering process to be an approximation.

Widening the scope of automated planning and improving the quality of test generation. Further explorations into automated planning and test generation can support a wider range of design approaches and test generation strategies than those discussed in Section 3. Firstly, in addition to capturing concurrency and ordering among accelerator executions, we can also track their data sharing patterns to identify producer-consumer relations, data races, and cases where accelerators invoke other accelerators. For instance, we can support scenarios where a task is executed by a pipeline of multiple accelerators, without the involvement of general-purpose cores in between. These scenarios can be easily incorporated in the abstract representation as extra scenario groups and can be specified as constraints for AGARSoC's test generation algorithm.

Secondly, we can implement mechanisms to detect other multi-threaded, synchronized accelerator executions, in addition to the lock-based and mutex-IP-based mechanism we currently support. Finally, AGARSoC's test generator can be enhanced, with minimal effort, to fully utilize the parallelism available in the SoC-under-verification and also in the verification environment. The schedules we generate to target happens-before scenarios can be multi-threaded. In addition, we can launch multiple concurrent accelerator executions from a single thread. We can also generate multiple, execution-time balanced tests that hit distinct coverage goals to be simulated in parallel.

Automating the process of adapting checkers to acceleration platforms. Two of the solutions that we discussed in Chapter 4 propose approximate approaches for implementing efficient checking mechanisms on acceleration platforms. We believe that the process of exploring the space of possible mechanisms, ranging from fully software checkers on-host to fully embedded, approximate checkers on-platform, can be automated. The design exploration tools can present the verification engineers with a a set of pareto optimal checking mechanisms with various levels of checking accuracy and performance impact. Approximation-aware high-level synthesis tools can be developed to automatically transform software checkers into an optimal checker design point chosen by the verification engineers.

Improving the quality of analysis for automatic bug diagnosis. We believe that there are several directions to be explored for automatic bug diagnosis in low-visibility validation environments. Firstly, the solution we discussed in Chapter 5 considered only bug manifestations in the architected state of a program that appear in a deterministic execution environment. Future work should explore approaches for diagnosing manifestations in a non-deterministic execuiton envirnoment. Secondly, we believe that localization accuracy can improve with a cooperative selection of feature extraction approaches and suitable classifier algorithms. Thirdly, even though a microarchitecture-independent solution is desirable for portability, we believe that localization accuracy can improve significantly with only a marginal addition of microarchitectural visibility.

Future work should explore the right balance of architectural and microarchitectural state to observe for good diagnosis. Finally, our work does not evaluate the closeness of our synthetic bug model to real world design bugs. Future work can evaluate the representativeness of synthetic bug models to real world bug databases.

6.3 Parting thoughts

Despite all the resources and time spent on it, functional verification is still a best-effort attempt at finding and fixing bugs in a design. Yet, current verification practices do not fully embrace the incompleteness inherent in the verification process. In this dissertation, we looked at the various functional verification activities through a perspective that recognizes the incompleteness inherent in the process and leverages it to find efficient solutions. Our perspective, which we call the automation, prioritization, and approximation (APA) approach, strives to prioritize and automate effort pertaining to the most critical design properties and functional bugs. These automations approximate certain functionalities and properties to achieve efficient implementations for modest losses in accuracy. Our APA-based solutions enable efficiency gains across the simulation-based functional verification spectrum. We identify pressing challenges in all the major functional verification activities and employ our APA approach to find automated solutions that trade accuracy for improved efficiency. If deployed jointly, we believe that the solutions presented in this dissertation can herald significant savings in the time, effort, and cost spent on functional verification. More importantly, these solutions lay the foundation for functional verification methodologies that mindfully harness the incompleteness in the functional verification process.

Bibliography

[1] Intel Atom x3 processor series. `http://www.intel.com/content/www/us/en/processors/atom/atom-x3-c3000-brief.html`.

[2] Qualcomm Snapdragon processors. `https://www.qualcomm.com/products/snapdragon`.

[3] Xilinx Vivado design suite. `http://www.xilinx.com/products/design-tools/vivado.html`.

[4] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal. FoCs: Automatic generation of simulation checkers from formal specifications. In *Proc. CAV*, 2000.

[5] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A reconfigurable design-for-debug infrastructure for SoCs. In *Proc. DAC*, 2006.

[6] 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. *Design & Test of Computers, IEEE*, 21(2), mar-apr 2004.

[7] A. Adir, M. Golubev, S. Landa, A. Nahir, G. Shurek, V. Sokhin, and A. Ziv. Threadmill: A post-silicon exerciser for multi-threaded processors. In *Proc. DAC*, 2011.

[8] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12), 1996.

[9] J. Alglave, A. Fox, S. Ishtiaq, M. Myreen, S. Sarkar, P. Sewell, and F. Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proc. DAMP*, 2008.

[10] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: running tests against hardware. In *Proc. TACAS*, 2011.

[11] E. Alpaydin. *Introduction to Machine Learning*. MIT Press, 2014.

[12] M. Amyeen, S. Venkataraman, and M. Mak. Microprocessor system failures debug and fault isolation methodology. In *Proc. ITC*, 2009.

[13] ARM Ltd. ARM architecture reference manual ARMv7-A and ARMv7-R edition, 2012.

[14] ARM Ltd. *ARM® Architecture Reference Manual: ARMv8, for ARMv8-A architecture profile*, June 2016.

[15] Arvind and J. W. Maessen. Memory Model = Instruction Reordering + Store Atomicity. *ACM SIGARCH Computer Architecture News*, 34(2), 2006.

[16] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge. Opportunities and challenges for better than worst-case design. In *Proc. ASP-DAC*, 2005.

[17] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal. Logic emulation with virtual wires. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 16(6), 1997.

[18] B. Bailey and G. Martin. *ESL Models and Their Application: Electronic System Level Design and Verification in Practice*. Springer Publishing Company, Incorporated, 2009.

[19] K. Basu and P. Mishra. Efficient trace signal selection for post silicon validation and debug. In *Proc. VLSI design*, 2011.

[20] K. Basu, P. Mishra, P. Patra, A. Nahir, and A. Adir. Dynamic selection of trace signals for post-silicon debug. 2013.

[21] B. Bentley. Validating the Intel Pentium 4 microprocessor. In *Proc. DAC*, 2001.

[22] M. Bose, J. Shin, E. Rudnick, T. Dukes, and M. Abadir. A genetic approach to automatic bias generation for biased random instruction generation. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, 2001.

[23] M. Boulé, J.-S. Chenard, and Z. Zilic. Adding debug enhancements to assertion checkers for hardware emulation and silicon debug. In *Proc. ICCD*, 2006.

[24] M. Boulé and Z. Zilic. Automata-based assertion-checker synthesis of PSL properties. *ACM TODAES*, 13, 2008.

[25] Cadence. *Palladium*, 2011. `http://www.cadence.com/products/sd/palladium_series`.

[26] Cadence. *Palladium XP Verification Computing Series*, 2016. `http://www.cadence.com/products/sd/palladium_xp_series`.

[27] H. Cain, M. Lipasti, and R. Nair. Constraint graph analysis of multithreaded programs. In *Proc. PACT*, 2003.

[28] K.-H. Chang, V. Bertacco, I. L. Markov, and A. Mishchenko. Logic synthesis and circuit customization using extensive external don't-cares. *ACM TODAES*, 15, 2010.

[29] D. Chatterjee, A. Koyfman, R. Morad, A. Ziv, and V. Bertacco. Checking architectural outputs instruction-by-instruction on acceleration platforms. In *Proc. DAC*, 2012.

[30] D. Chatterjee, C. McCarter, and V. Bertacco. Simulation-based signal selection for state restoration in silicon debug. In *Proc. ICCAD*, 2011.

[31] K. Chen, S. Malik, and P. Patra. Runtime validation of memory ordering using constraint graph checking. In *Proc. HPCA*, 2008.

[32] N. Choudhary, S. Wadhavkar, T. Shah, H. Mayukh, J. Gandhi, B. Dwiel, S. Navada, H. Najaf-abadi, and E. Rotenberg. Fabscalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template. In *Proc. ISCA*, 2011.

[33] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman. Accelerator-rich architectures: Opportunities and progresses. In *Proc. DAC*, 2014.

[34] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman. Accelerator-rich architectures: opportunities and progresses. In *Proc. DAC*, 2014.

[35] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 3 edition, 2009.

[36] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.

[37] I. Cutress. Intel disables TSX instructions: Erratum found in haswell, haswell-e/ep, broadwell-y. *AnandTech*, Aug. 2014. `http://www.anandtech.com/show/8376/`.

[38] S. Das, R. Mohanty, P. Dasgupta, and P. P. Chakrabarti. Synthesis of System Verilog assertions. In *Proc. DATE*, 2006.

[39] G. De Micheli, R. Ernst, and W. Wolf, editors. *Readings in Hardware/Software Co-design*. Kluwer Academic Publishers, 2002.

[40] A. DeOrio, A. Bauserman, and V. Bertacco. Post-silicon verification for cache coherence. In *Proc. ICCD*, 2008.

[41] A. DeOrio, Q. Li, M. Burgess, and V. Bertacco. Machine learning-based anomaly detection for post-silicon bug diagnosis. In *Proc. DATE*, 2013.

[42] A. DeOrio, I. Wagner, and V. Bertacco. Dacota: Post-silicon validation of the memory subsystem in multi-core designs. In *Proc. HPCA*, 2009.

[43] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proc. MICRO*, 2003.

[44] M. Farkash, B. Hickerson, and B. Samynathan. Data mining diagnostics and bug MRIs for HW bug localization. In *Proc. DATE*, 2015.

[45] S. Fine and A. Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Proc. DAC*, 2003.

[46] H. Foster. Trends in functional verification: A 2014 industry study. In *Proc. DAC*, 2015.

[47] O. Friedler, W. Kadry, A. Morgenshtein, A. Nahir, and V. Sokhin. Effective post-silicon failure localization using dynamic program slicing. In *Proc. DATE*, 2014.

[48] M. Fujita, I. Ghosh, and M. Prasad. *Verification Techniques for System-Level Design*. Morgan Kaufmann Publishers Inc., 2008.

[49] K. Ganesan and L. K. John. Automatic generation of miniaturized synthetic proxies for target applications to efficiently design multicore processors. *IEEE Trans. on Computers*, 63(4), 2014.

[50] M. Gao and K.-T. Cheng. A case study of Time-Multiplexed Assertion Checking for post-silicon debugging. In *Proc. HLDVT*, 2010.

[51] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: faster and more flexible program analysis. *Journal of Instruction Level Parallel*, September 2005.

[52] P. Hammarlund, R. Chappell, R. Rajwar, R. Osborne, R. Singhal, M. Dixon, R. D'Sa, D. Hill, S. Chennupaty, E. Hallnor, E. Burton, A. Bajwa, R. Kumar, A. Martinez, S. Gunther, S. Kaushik, S. Jourdan, H. Jiang, T. Piazza, M. Hunsaker, and M. Derr. 4th generation Intel® Core Processor, codenamed Haswell. *IEEE Micro*, 99(PrePrints), 2014.

[53] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *Proc. ETS*, 2013.

[54] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu. TSOtool: A program for verifying memory systems using the memory consistency model. In *Proc. ISCA*, 2004.

[55] T. K. Ho. Random decision forests. In *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, 1995.

[56] IBM. Power ISA version 2.06 revision B, July 2010.

[57] Intel Corporation. *Intel® Core™ i7-900 Mobile Processor Extreme Edition Series, Intel® Core™ i7-800 and i7-700 Mobile Processor Series*, Feb. 2015. Rev. 028.

[58] Intel Corporation. *2nd Generation Intel® Core™ Processor Family Mobile and Intel® Celeron® Processor Family Mobile Specification Update*, Apr. 2016. Rev. 034.

[59] Intel Corporation. *5th Generation Intel® Core™ Processor Family, Intel® Core™ M-Processor Family, Mobile Intel® Pentium® Processor Family, and Mobile Intel® Celeron® Processor Family Specification Update*, Sept. 2016. Rev. 024.

[60] Intel Corporation. *6th Generation Intel® Processor Family Specification Update*, Sept. 2016. Rev. 008.

[61] Intel Corporation. *7th Generation Intel® Processor Family Specification Update*, Aug. 2016. Rev. 001.

[62] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developers Manual*, June 2016. Version 1.0, Rev. 007.

[63] Intel Corporation. *Mobile 3rd Generation Intel® Core™ Processor Family Specification Update*, Apr. 2016. Rev. 022.

[64] Intel Corporation. *Mobile 4th Generation Intel® Core™ Processor Family, Mobile Intel® Pentium® Processor Family, and Mobile Intel® Celeron® Processor Family Specification Update*, Aug. 2016. Rev. 034.

[65] C. Ioannides, G. Barrett, and K. Eder. Introducing xcs to coverage directed test generation. In *Proc. HLDVT*, 2011.

[66] C. Ioannides and K. I. Eder. Coverage-directed test generation automated by machine learning – a review. *ACM TODAES*, 17(1), 2012.

[67] R. Kalayappan and S. Sarangi. A survey of checker architectures. *ACM Comput. Surv.*, 45(4), 2013.

[68] D. S. Khudia and S. Mahlke. Low cost control flow protection using abstract control signatures. 2013.

[69] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9), 1979.

[70] M. Li and A. Davoodi. Multi-mode trace signal selection for post-silicon debug. In *Proc. DAC*, 2014.

[71] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. Trace-based microarchitecture-level diagnosis of permanent hardware faults. In *Proc. DSN*, 2008.

[72] D. Lin, T. Hong, F. Fallah, N. Hakim, and S. Mitra. Quick detection of difficult bugs for effective post-silicon validation. In *Proc. DAC*, 2012.

[73] D. Lin, E. Singh, C. Barrett, and S. Mitra. A structured approach to post-silicon validation and debug using symbolic quick error detection. In *Proc. ITC*, 2015.

[74] X. Liu and Q. Xu. Trace signal selection for visibility enhancement in post-silicon validation. In *Proc. DATE*, 2009.

[75] S. Mador-Haim, R. Alur, and M. M. K. Martin. Generating litmus tests for contrasting memory consistency models. In *Proc. CAV*, 2010.

[76] E. E. Mandouh and A. G. Wassal. CovGen: a framework for automatic extraction of functional coverage models. In *Proc. ISQED*, 2016.

[77] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFX: A simple and efficient memory model for concurrent programming languages. In *Proc. PLDI*, 2010.

[78] J. Markoff. Company news; flaw undermines accuracy of pentium chips. *The New York Times*, Nov. 1994. `http://www.nytimes.com/1994/11/24/business/company-news-flaw-undermines-accuracy-of-pentium-chips.html`.

[79] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4), 2005.

[80] I. Mavroidis and I. Papaefstathiou. Efficient testbench code synthesis for a hardware emulator system. In *Proc. DATE*, 2007.

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

[82] Mentor Graphics. *Veloce2 Emulator*, 2016. `https://www.mentor.com/products/fv/emulation-systems/veloce`.

[83] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. In *Proc. DATE*, 2004.

[84] P. Mishra and N. Dutt. Functional coverage driven test generation for validation of pipelined processors. In *Proc. DATE*, 2005.

[85] S. Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys*, 48(4), Mar. 2016.

[86] S. Mochizuki *et al*. A 197mW 70ms-latency full-HD 12-channel video-processing SoC for car information systems. In *Proc. ISSCC*, 2016.

[87] M. D. Moffitt, M. A. Sustik, and P. G. Villarrubia. Robust partitioning for hardware-accelerated functional verification. In *Proc. DAC*, 2011.

[88] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[89] A. Nahir, M. Dusanapudi, S. Kapoor, K. Reick, W. Roesner, K.-D. Schubert, K. Sharp, and G. Wetli. Post-silicon validation of the IBM POWER8 processor. In *Proc. DAC*, 2014.

[90] Obsidian Software Inc. Raven: Product datasheet.

[91] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12, 2011.

[92] A. Piziali. Coverage-driven verification. In *Functional Verification Coverage Measurement and Analysis*. Springer US, 2004.

[93] Z. Poulos and A. Veneris. Exemplar-based failure triage for regression design debugging. In *Proc. LATS*, 2015.

[94] K. Rahmani, S. Proch, and P. Mishra. Efficient selection of trace and scan signals for post-silicon debug. *IEEE Transactions on VLSI Systems*, 24(1), Jan 2016.

[95] K. Ravi, K. L. McMillan, T. R. Shiple, and F. Somenzi. Approximation and decomposition of binary decision diagrams. In *Proc. DAC*, 1998.

[96] A. Roy, S. Zeisset, C. J. Fleckenstein, and J. C. Huang. Fast and generalized polynomial time memory consistency verification. In *Proc. CAV*, 2006.

[97] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Proc. PLDI*, 2011.

[98] P. Sewell, S. Sarkar, S. Owens, F. Nardelli, and M. Myreen. X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Comm. ACM*, 53(7), 2010.

[99] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks. Aladdin: a pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proc. ISCA*, 2014.

[100] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2), 1988.

[101] H. Shen, W. Wei, Y. Chen, B. Chen, and Q. Guo. Coverage directed test generation: Godson experience. In *Proc. ATS*, 2008.

[102] K. Shimizu, S. Gupta, T. Koyama, T. Omizo, J. Abdulhafiz, L. McConville, and T. Swanson. Verification of the cell broadband engine^{TM}processor. In *Proc. DAC*, 2006.

[103] H. Shojaei and A. Davoodi. Trace signal selection to enhance timing and logic visibility in post-silicon validation. In *Proc. ICCAD*, 2010.

[104] D. J. Sorin, M. D. Hill, and D. A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3), 2011.

[105] Synopsys. *ZeBu Server-3*, 2016. `http://www.synopsys.com/Tools/Verification/hardware-verification/emulation/Pages/zebu-server-asic-emulator.aspx`.

[106] D. Takahashi. Intels billion-dollar mistake: Why chip flaws are so hard to fix. *Venture Beat*, Jan. 2011. `http://venturebeat.com/2011/01/31/intels-billion-dollar-mistake-why-chip-flaws-are-so-hard-to-fix/`.

[107] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *Design & Test of Computers, IEEE*, 18(4), jul/aug 2001.

[108] A. Vance. AMD denies 'stop ship' with barcelona because chip is not shipping. *The Register*, Dec. 2007. `http://www.theregister.co.uk/2007/12/06/opteron_delays_amd/`.

[109] A. Veneris, B. Keng, and S. Safarpour. From RTL to silicon: The case for automated debug. In *Proc. ASP-DAC*, 2011.

[110] E. Viaud, F. Pêcheux, and A. Greiner. An efficient TLM/T modeling and simulation environment based on conservative parallel discrete event principles. In *Proc. DATE*, 2006.

[111] N. J. Wang and S. J. Patel. Restore: Symptom-based soft error detection in microprocessors. *IEEE Trans. on Dependeable and Secure Computing*, 3(3), 2006.

[112] D. L. Weaver and T. Germond, editors. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[113] B. Wile, J. Goss, and W. Roesner. *Comprehensive Functional Verification: the Complete Industry Cycle (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., 2005.

[114] M. Woodward. Mutation testing – its origin and evolution. *Information and Software Technology*, 35(3), 1993.

[115] Xilinx Verification Tool. *ChipScope Pro*, 2006. `http://www.xilinx.com/ise/optional_prod/cspro.html`.