GOLDMINE: AN INTEGRATION OF DATA MINING AND STATIC ANALYSIS FOR AUTOMATIC GENERATION OF HARDWARE ASSERTIONS

BY

DAVID SHERIDAN

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Assistant Professor Shobha Vasudevan
ABSTRACT

We present GoldMine, a methodology for generating assertions automatically. Our method involves a combination of data mining and static analysis of the Register Transfer Level (RTL) design. The RTL design is first simulated to generate data about the design’s dynamic behavior. The generated data is then mined for “candidate assertions” that are likely to be invariants. We present both a decision tree supervised learning algorithm as well as a coverage guided mining algorithm for generating high-quality assertions. These candidate assertions are then passed through a formal verification engine to filter out the spurious candidates. The assertions that are attested as true by the formal engine are system invariants. These are then evaluated by a process of designer ranking that is provided as feedback to the data mining engine. We present results of using GoldMine for assertion generation of the RTL of Sun’s OpenSparc T2 many-threaded processor. Our results show that GoldMine can generate complex, high-coverage assertions in RTL, thereby minimizing human effort in this process.
Dedicated to my family and my fiancée, Kelsey
ACKNOWLEDGMENTS

First, I would like to thank Lingyi Liu and Viraj Athavale for contributing to GoldMine and providing ideas when needed. I would also like to thank Sam Hertz for crafting an extremely efficient and powerful GoldMine in C++, vastly improving the original Java implementation. Thanks also to Sanjay Patel and Daniel Johnson, who allowed us to test GoldMine on the developing Rigel 1000-core CPU design. Thanks to Bill Touhy who provided excellent insight due to his experience in the verification field, which kept us focused on practical applications for GoldMine.

Big thanks to David Tcheng, who helped us craft an appropriate data mining algorithm for GoldMine. It was David’s help which really got the project off the ground. Thanks also to Hyungsul Kim, who took the initiative to provide us with ideas on how to improve GoldMine’s data mining algorithm. Hyungsul helped us design the coverage guided mining algorithm, which allowed us to generate much higher quality assertions. Both David and Hyungsul provided their data mining expertise so that we could utilize the power of data mining to the fullest extent.

Lastly, I would like to thank Assistant Professor Shobha Vasudevan, my adviser. I would not have attended graduate school had Shobha not opened my eyes to the world of research. Her insight is invaluable and her ideas are always creative and interesting. She is an inspiring individual, and I expect to one day see her mold GoldMine into a powerful tool that changes the industry.
# TABLE OF CONTENTS

LIST OF ABBREVIATIONS ........................................... vii

CHAPTER 1 INTRODUCTION ........................................... 1

CHAPTER 2 RELATED WORK ........................................ 4
  2.1 Software Invariant Generation .................................. 4
  2.2 Assertion Generation for Hardware ............................. 5

CHAPTER 3 BACKGROUND .......................................... 7
  3.1 The Hardware Design Cycle .................................... 7
  3.2 Assertions .................................................... 8
  3.3 Data Mining .................................................. 9

CHAPTER 4 GOLDMINE: ASSERTION GENERATION METHODOLOGY ........................................... 11
  4.1 Data Generator ................................................ 11
  4.2 Lightweight Static Analyzer ................................... 12
  4.3 A-Miner ....................................................... 12
  4.4 Decision Tree Based Supervised Learning Algorithms ....... 13
  4.5 Formal Verifier ............................................... 14
  4.6 A-Val: Evaluation and Ranking ................................ 15
  4.7 Temporal Assertions ......................................... 17
  4.8 An Example Run through GoldMine ............................ 18
  4.9 Applications of GoldMine .................................... 20

CHAPTER 5 A CASE STUDY OF A MULTI-CORE RTL .......... 21
  5.1 Subjective Ranking of Assertions by a Designer .......... 21
  5.2 Complex Assertions in GoldMine ............................. 23
  5.3 Outputs Covered by GoldMine ................................ 23
  5.4 The Acid Test: Regression Test Experiments ................. 24

CHAPTER 6 SCALING GOLDMINE TO INDUSTRIAL DESIGNS:
  THE OPENSPARC T2 ............................................. 26
  6.1 Evaluation of True Assertion Success Rate ................. 26
  6.2 Evaluation of Assertion Input Space Coverage .............. 27
# LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>FV</td>
<td>Formal Verification</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Design Language</td>
</tr>
<tr>
<td>HVL</td>
<td>Hardware Verification Language</td>
</tr>
<tr>
<td>ABV</td>
<td>Assertion Based Verification</td>
</tr>
<tr>
<td>TLB</td>
<td>Translation Lookaside Buffer</td>
</tr>
<tr>
<td>IQ</td>
<td>Input Queue</td>
</tr>
<tr>
<td>GB</td>
<td>Gigabyte</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>ITC</td>
<td>International Test Conference</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>SoC</td>
<td>System on a Chip</td>
</tr>
<tr>
<td>GHz</td>
<td>Gigahertz</td>
</tr>
<tr>
<td>FPGA</td>
<td>Fully Programmable Gate Array</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High-Speed Integrated Circuit</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Design Language</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

In the hardware design industry, having a design error can be disastrous. In the Intel Pentium P5 chip, a floating point division bug caused Intel to lose up to $475 million in 1995. More recently in 2007, AMD encountered a virtualization bug in its Phenom line of CPUs requiring them to revise the silicon - a costly procedure. Unlike software bugs, hardware bugs cannot always be fixed with a simple patch. These bugs cost hardware manufacturers millions of dollars and precious time in a quickly moving industry.

Assertions or invariants provide a mechanism to express desirable properties that should be true in the system. Assertions are used for validating hardware designs at different stages through their life-cycle, such as pre-Silicon formal verification, dynamic validation, runtime monitoring, and emulation [2, 3, 4]. Assertions are also synthesized into hardware for post-Silicon debug and validation and in-field diagnosis [2, 5].

Among all the solutions for ensuring robustness of hardware systems, assertion based verification has emerged as the most popular candidate [6] solution for “pre-Silicon” design functionality checking. Assertions are used for static (formal) verification as well as dynamic verification of the Register Transfer Level (RTL) design in the pre-Silicon phase.

The key question then is: How are these assertions generated? Assertion generation is an entirely manual effort in the hardware system design cycle. Placing too many assertions can result in an unreasonable performance overhead. Placing too few assertions, on the other hand, results in insufficient coverage of behavior. The trade-off point for crafting minimal, but effective (high coverage) assertions takes multiple iterations and man-months to achieve [3, 7, 8]. Another challenge with assertion generation is due to the modular nature of system development. A module developer would write local assertions that pertain to his/her module. Maintaining consistency of inter-modular global assertions as the system evolves.

This chapter includes work previously published in [1].
in this fragmented framework is very tedious. In sequential hardware, *temporal properties* that cut across time cycles are usually the source of subtle, but serious bugs. It is difficult for the human mind to express and reason with temporal relations, making temporal assertion generation very challenging.

We integrate two solution spaces, statistical, dynamic techniques (data mining) and deterministic, static techniques (lightweight static analysis and formal verification), to provide a solution to the assertion generation problem. Static analysis can make excellent generalizations and abstractions, but its algorithms are limited by computational capacity. Data mining, on the other hand, is computationally efficient with dynamic behavioral data, but lacks perspective and domain context.

We present GoldMine, a tool for automatically generating RTL assertions. An RTL design is simulated using random vectors to produce dynamic behavioral data for the system. This data is mined by advanced data mining algorithms to produce rules that are *candidate assertions*, since they are inferred from the simulation data, but not for all possible inputs. These candidate assertions are then passed through a formal verification engine along with the RTL design to filter out spurious assertions and retain the system invariants. Static behavioral analysis techniques are employed to guide the data mining process. A designer evaluation and ranking process is facilitated in GoldMine to provide useful feedback to the iterative data mining process.

GoldMine proposes a radical, but powerful validation paradigm. It uses two high impact technologies, data mining and static analysis, symbiotically to assimilate the design space. It then reports its findings in a human digestible form (assertions) early on and with minimal manual effort. This technique is intended to replace the traditional method of the engineer deducing all possible correct behaviors, capturing them in assertions, testing assertions, creating directed tests to observe behavior and finally applying random stimulus.

Random stimulus is applied late in the validation phase, when the design and assertion-based verification environment are mature enough to withstand and interpret random behavior. GoldMine explores the random stimulus space and distills it into assertions that a human can review. GoldMine’s data mining, then, gains knowledge about design spaces that are as yet unexplored by a human-directed validation phase. Eventually, the manual, iterative process of validation will arrive at a point of high coverage. Using GoldMine, however, this step can be done very early in the design, making a quantum leap in the validation cycle. If an unintended invariant behavior is observed, a bug is detected. Otherwise, an
assertion that can be used for all future versions of the design has been generated. GoldMine is best utilized in the regression test suite of an RTL design.

GoldMine is completely automatic. It is able to generate many assertions per output for a large percentage of module outputs in very reasonable runtimes (see case study). It has the ability to minimize human effort, time and resources in the long-drawn assertion generation process and increase validation productivity. Along with input/output or propositional assertions, GoldMine can also generate temporal assertions in Linear Temporal Logic [9].

GoldMine can generate assertions that are complex or span multiple logic levels in the RTL.

The contributions in this work are as follows.

- We present a tool which can be used to generate assertions automatically.
- Our tool can produce complex assertions for both combinational and sequential designs, a feat which has not been accomplished by any other known tools.
- We enable the discovery of design knowledge otherwise unattainable early in the design logic.
- With GoldMine, we propose a validation paradigm that can significantly reduce the time and effort of assertion creation.
- We explore both a decision tree based supervised learning algorithm and an alternate coverage guided algorithm which can produce high coverage assertions.

We demonstrate that GoldMine produces excellent results on real RTL designs in the form of complex, high coverage assertions.

---

1 At this time, we can generate assertions with the X operator.
CHAPTER 2
RELATED WORK

GoldMine is not the first tool to attempt assertion generation. Many techniques have been used since the 1970s for both hardware and software. In this section, we review work related to GoldMine.

2.1 Software Invariant Generation

The roots of automatic assertion generation can be traced back to the automatic generation of invariants for proving loop behavior. Both Wegbreit [10] and Katz [11] attempted to do this using two methods. The first method was a top-down approach, which attempts to generalize behavior by observing the entry and exit conditions. A second bottom-up approach involved directly observing the behavior of the loop body. Caplain [12] observed that for complex programs, it was not possible to necessarily derive the entry and exit conditions of non-trivial loops such as nested loops. He suggested a technique which involved decomposing the general loop body into individual expressions at each iteration of the loop and tried to match these cases with common loop behavior. Misra [13] also extended this work by solving the loop invariant generation problem for while loops.

When model checking became common in the 1990s, many researchers used invariant generation as a technique for verifying properties. This technique involved generating an invariant that was stronger than the given property, proving that the property was true. Bensalem [14] presented various techniques to improve invariant generation including generalized reaffirmed invariance, propagation of invariants, refined strengthening, and combining invariants. Bjorner [15] noted that, while invariant generation was complete, it was not practical for real world application. He suggested a technique involving “assertion graphs”, which break down the property into individual assertions. Stark [16] worked on coupling the heuristic and theorem proving component together, which allowed the use of
failed proofs to help direct the invariant search using successive approximations. Tiwari [17] used a technique that involved generating an upper and lower bound for the reachable state set and iterating until the upper bound provided the best invariant. Pasareanu [18] also extended this technique by using symbolic execution for checking that the generated invariant is inductive. This research included an implementation for property checking of Java programs.

It became apparent that using static analysis for generation of invariants was not scalable for complex programs. This invited the use of dynamic analysis as an alternative. Cheng [19] uses data mining to produce a set of invariants based on the execution of the software. These invariants are then used to limit the search space, making model checking much more tractable. Ammons [20] has explored data mining as a method for generating program specifications. This tool analyzes program behavior and summarizes frequent patterns as state machines. This specification can also be used to identify bugs if a designer encounters incorrect behavior in the generated specification.

DAIKON [21, 22, 23] is a tool that has been developed for generation of assertions in software. This tool performs dynamic analysis of a simulation trace to try and match behavior to a set of pre-defined property templates. These templates specify that a variable is constant, non-zero, or within some range. In addition, it can determine some more complex properties such as linear relationships or library functions.

### 2.2 Assertion Generation for Hardware

One of the first attempts at assertion generation for hardware designs was developed by Wang [24] who observed that manually generating assertions would not only cut down on the manual effort, but increase the efficiency and quality of verification. This technique relies on abstracting out the sequential elements of the design and considering only the combinational elements. The tool then uses symbolic values to define the logic of the design. This methodology was used for verification of the memory arrays of the PowerPC.

Hekmatpour [25] proposed a schema-driven assertion generation strategy based on the block-level structure of the design. First, a schema is defined or selected from a pre-existing library which defines the actions that need to be taken. This schema is then used for constructing assertions based on the information collected
from the design. This system supports both interface assertions and interconnect assertions.

Hangal [26] presents the IODINE tool, which has a striking similarity to DAIKON. IODINE uses a set of pre-defined property templates and attempts to match these templates with actual behavior that is observed during simulation. This tool supports one-hot, mutual exclusion, state traversal, req-ack protocol, and scoreboard-related assertions.

Rogin [27] presents the Dianosis tool which is not restricted by pre-defined property templates like IODINE and DAIKON. The main idea behind this technique is to combine existing basic assertions to generate new, complex assertions. If no basic properties exist, an approach similar to IODINE is used to generate an initial set of basic properties. The combined assertions are checked using the simulation trace. Any combined assertions which do not comply with the simulation trace are discarded.
CHAPTER 3

BACKGROUND

3.1 The Hardware Design Cycle

To understand why verification is important and what methods are used for testing circuits, it is important to understand the hardware development cycle. The first step in the hardware development cycle is the specification stage, where architects will specify the behavior of a circuit. This may include creating system-level models to simulate this behavior using tools like SystemC [28]. The next step is to specify the Register Transfer Level (RTL) implementation using a hardware design language (HDL) such as Verilog [29] or VHDL [30] which describes the flow of data in a circuit and how that data is manipulated to achieve the desired behavior. The RTL implementation is then synthesized into a gate-level implementation, which specifies how the circuit must be constructed out of individual logic gates. This gate-level implementation is then mapped out to determine where the transistors and wiring will be physically located on a chip. This physical layout is then manufactured at a fabrication plant where the circuits are printed onto silicon. This silicon is placed into a package which can interface with other systems.

Since there is so much work and cost that goes into each step of this cycle, hardware designers put an extremely large effort into making sure that each step is done correctly. Making a mistake in one of the steps means that all of the following steps will be wrong, costing even more time and money. In this thesis, we focus on the testing of the RTL design. There are many strategies used in the testing of the RTL design. The first testing strategy is known as a directed test, which involves biasing the inputs in a certain way to create expected behavior. The directed tests are often paired with mechanisms which check the outputs and internal state to ensure that the expected behavior and the actual behavior match. Another strategy is to randomize the input stimuli to create completely random behavior. This random simulation is paired with many checkers that ensure that
circuit behavior is legal for the system. The last strategy is called assertion based verification.

3.2 Assertions

The idea of an assertion was first proposed by Alan Turning, who suggested breaking down a large software routine into a set of properties which could each be checked [3]. Later, Robert Floyd developed a formal system for reasoning about flowcharts [31] which was then refined by C. A. R. Hoare [32]. The system was adapted for use in software verification which allowed a programmer to check that certain conditions did not occur [3]. Hardware design and verification was a largely manual process until the VHDL became a standard in 1987 [30]. VHDL supports the ‘assert’ keyword, which allows a designer to specify a condition that must always evaluate to true. Around this same time, formal verification of assertions was also introduced which allowed assertions to be formally proved [33]. However, the power of assertions was limited until hardware verification languages (HVLs) were developed which introduced the concept of assertion based verification (ABV) [34]. Today, there are many different HVLs which enable ABV such as SystemVerilog [35], OpenVera [36], and Property Specification Language [37].

Assertion based verification [3, 2] involves defining desired properties of the hardware design and asserting that those properties are never violated. These assertions can be paired with a dynamic method, such as directed tests or random simulation, and will give an error if the property is violated. In addition, a tool called formal verification is a static method that creates a model of the design and checks if the assertion can ever be violated. Formal verification either guarantees that the property can never be violated or gives a counterexample that shows how the assertion is violated. In addition to RTL testing, assertions can be physically synthesized into silicon and used for checking after the chip has been fabricated [2, 5]. Because of their power and versatility, assertions have become the most popular method of verifying an RTL design [6].

However, assertion based verification has a significant drawback. Assertion generation up until this point has been a manual effort. Assertions must be specified by the designer or the verification engineer. This can be easy enough for simple combinational properties, but for complex temporal properties, it can be
very time consuming. In addition, it is difficult to reason between module boundaries. Even if the assertion is correctly specified, certain constraints must also be specified for the assertion to be true. It can also be difficult coming up with the right number of assertions. If the set of assertions is too small, it will not provide very good coverage of the design, leading to a large number of bugs. It can be easy to provide high coverage if there are a very large number of assertions, but this can take a very long time to produce. Additionally, a large set of assertions can also make simulation very slow and synthesis for post-silicon verification impossible if the area is too large. This means that it is up to the designers to produce a minimal set of assertions that also provides high coverage of the design. This process can take up a large percentage of the design cycle [3, 7, 8], resulting in many lost months of productivity. The solution to this problem is taking the manual effort out of assertion generation.

3.3 Data Mining

Data mining is a relatively young field that developed as a means for organizing and analyzing the information stored in databases [38]. There are many forms of data mining such as frequent pattern mining, sequential mining, and clustering. However, we will focus on frequent pattern mining since this is the type of mining we currently use in GoldMine. In general, frequent pattern mining involves finding correlations, or patterns, between items.

3.3.1 Decision Tree Based Learning

The decision tree [39, 40] algorithm works by making successive recursive splits on a database in relation to a target item. Each split implies that a new item from that database has been added to the set of items, referred to as the itemset. These splits are based on statistics referred to as mean and error. Mean refers to the average value of the target item in the database. The error refers to how well the items in the pattern correlate with the target item. The goal is to find a correlation between the target item and the items in the pattern by reducing the error.

For example, consider a database that contains the items which were purchased by customers at a supermarket. Each transaction has a Boolean value associated with each item indicating if that item was purchased (1) or not (0). We want to
see what items are frequently purchased along with the target item, “milk”. The
decision tree observes that splitting on the item “bread” reduces the error more
than splitting on any other item. This means that the decision tree will partition
the database into entries where bread = 1 and entries where bread = 0. Bread is
added to the itemset and the recursive process continues for each set of database
entries. The result is a tree structure that predicts whether milk is likely to be
purchased depending on the other items that are purchased.

3.3.2 Association Rule Learning

Association rule mining [41] is a data mining method that attempts to generate all
possible correlations between items. This is done by recursively adding items to
an itemset until that itemset is frequently correlated with some target item. Though
this algorithm has an exponential complexity in the worst case, high efficiency is
achieved by applying constraints and using pruning techniques.

Considering the example above, we want to check what items are purchased
along with milk. The algorithm attempts to match each single item with milk to
determine if there are a significant number of transactions to consider this a valid
pattern. After this step, all possible sets of two items are checked for correlation
with the target item. This process continues until all possible combinations of
items are tested for correlation with milk. This algorithm gives all likely correla-
tions with milk, though the runtime may make it intractable. Significant effort is
put into pruning the search space to make this algorithm reasonable to use.
CHAPTER 4

GOLDMINE: ASSERTION GENERATION METHODOLOGY

We propose GoldMine, a methodology to automatically generate assertions using data mining and static analysis. There are five main parts in GoldMine, as shown in Figure 4.1.

4.1 Data Generator

The Data Generator simulates a given design (or a “module” of the design). If regression tests or workloads for the design are available, they can be used to obtain the simulation traces. GoldMine also generates its own set of simulation traces using random input vectors.

Typically, simulating with randomized inputs produces the largest number of true assertions. We used a script to generate a testbench for each verilog design that we wanted to test. In the testbench, each input bit is assigned with a completely random value for each cycle by using the verilog \$random function. We have the ability to expand this method in the future by constraining the random input values using background information where certain input combinations may not be allowed. For most of our tests, we simulate for 10,000 cycles, though we can increase this number for extremely large or complex designs.

---

This chapter includes work previously published in [1].

---

Figure 4.1: Goldmine Tool Suite
4.2 Lightweight Static Analyzer

The static analyzer extracts domain-specific information about the design that can be passed to A-Miner. It can include cone-of-influence, localization reductions [42], topographical variable ordering, and other behavioral analysis techniques.

The current version of the tool only uses static analysis for logic cone information. The logic cone of a signal consists of all of the inputs which can influence the value of a given output. Since data mining methods can only use statistical methods to infer relationships between signals, it is possible that an unrelated input may be correlated to an output. The logic cone prevents this problem by restricting the searched inputs to only those which are related to the output. This static analysis is also advantageous in that it decreases the runtime in many data mining algorithms since there are fewer inputs to consider.

We have developed a script for generating the logic cone of an output. This script first synthesizes the target RTL into gate-level RTL and flattens the hierarchy, making it easier to parse. Then the script analyzes each gate and records which input signals influence the output of the gate to generate a one-level-deep logic cone for each internal signal and primary output. Based on these one-level logic cones, the script recursively adds the logic cones of the signals in each primary output’s logic cone until a full logic cone has been produced.

4.3 A-Miner

The A-Miner phase derives knowledge and information from the simulation trace data. This is done by searching for correlations between the inputs and a target output. For example, in a simulation trace, whenever inputs $A$ and $B$ are both 1, the output $C$ is also 1. A data mining algorithm can quickly and efficiently recognize this pattern. Data mining algorithms use statistics such as support and confidence to determine whether there is actually a relationship between the inputs and target output. Given a rule $A \implies B$ (henceforth of the form if $a$ then $b$), support($A$) is the proportion of instances in the data that contain $A$. Confidence can be interpreted as an estimate of the conditional probability $P(B|A)$. If a rule has 100 percent confidence, it means that within the data set, there is complete coincidence between $A$ and $B$. A high support for this rule means that $A$ occurs frequently in the data set. In GoldMine, we must guarantee that the confidence
is 100% if we want to generate an assertion that is likely to be true. The reason for this is that if a given antecedent is correlated with an output that has multiple different values, then that cannot be an assertion since the antecedent does not imply a single value.

A-Miner also provides hooks for incorporating domain specific information from the lightweight static analyzer into the mining algorithms. The data mining algorithm allows specification of which inputs have a relationship with the target output as determined by the logic cone. In addition, this phase of GoldMine can have multiple feedback loops from different parts of the tool. Using the information provided to it, the A-Miner produces a set of candidate assertions which are likely to be true. Objective measures of interestingness [43] can be used to rank this set of candidate assertions, such as the support as specified above.

4.4 Decision Tree Based Supervised Learning Algorithms

Association rule based data mining algorithms find all possible associations between sets of predicates and rank them according to support/confidence. For sequential blocks that might have temporal properties, exhaustive search is an inefficient option in our experience (see case study).

We primarily use decision tree based supervised learning algorithms [39] in A-Miner. In a decision tree, the data space is locally divided into a sequence of recursive splits in a small number of steps. A decision tree is composed of internal nodes and terminal leaves. Each decision node implements a “splitting function” with discrete outcomes labeling the branches. This hierarchical decision process that divides the input data space into local regions continues recursively until it reaches a leaf.

We require only Boolean splits (for Boolean variables) at every decision node. The error function implemented to select the best splitting variable at each node is the variance between the target output values and the values predicted by a candidate antecedent. The winner is the one whose error is minimum, which then forms the next level of the decision tree. Each leaf in the decision tree becomes a candidate assertion where the variable and value at each split represents a proposition in the antecedent and the mean of the output represents its predicted value in the consequent.
The decision tree algorithm used in GoldMine is shown in Algorithm 1. The decision_tree function has three inputs: $F$ represents the set of inputs that are available to split on, $P$ represents the set of propositions in the antecedent of an assertion, and $E$ represents the set of simulation trace samples. In addition, $A_c$ represents the set of candidate assertions and $z$ represents the output for which assertions are being mined.

The mean function calculates the mean of the values for $z$ in each sample and represents the expected value of $z$. The error is a function that calculates the absolute deviation of the output value in each sample from the expected value. Other functions, such as variance, can be used as an error function. The error function will be high when there is a lot of deviation in the output’s value in each sample and it will be zero when the output’s value is the same in each sample.

The algorithm first checks if the error of the simulation trace is zero. If so, a candidate assertion is added to $A_c$ where $P$ represents the set of propositions in the antecedent and the output is equal to the mean in the consequent. If the error is zero, it indicates that all values of the output are the same, meaning that the mean is equivalent to the value of the output in all samples.

If the error is not zero and an assertion cannot be created, the algorithm looks for a suitable input in $F$ to split on. The potential error is calculated based on partitioning the simulation data into only the samples where $f_i = 0$ and only the samples where $f_i = 1$. The potential error of each set of samples is summed and subtracted from the error of the unpartitioned data set. This is the potential error reduction for splitting on $F_i$. The $f_i$ that results in the best error reduction is chosen as the splitting variable. The algorithm recurses with the splitting variable removed from $F$. One instance of decision_tree will add $f_{best} = 0$ to $P$ and have $E$ partitioned with respect to 0 while the other instance will have $f_{best} = 1$ in $P$ and $E$ partitioned with respect to 1.

4.5 Formal Verifier

In order to check if the likely invariants generated by A-Miner are system invariants, the design and candidate assertions are passed through a formal verification engine. If a candidate assertion fails formal verification, a counterexample can be generated for feedback to the A-Miner. We use SMV [44] and Incisive Formal Verifier as our formal verification engines. The candidate assertions are attached
Algorithm 1 Decision Tree Supervised Learning Algorithm

\[ \text{decision_tree}(F, P, E) \]

1: \( \text{mean} = \text{mean}(E, z) \)
2: \( \text{err} = \text{error}(E, z) \)
3: \( \text{if} \ \text{err} \leq 0 \ \text{then} \)
4: \( A_c = A_c \cup P \implies z = \text{mean} \)
5: \( \text{return} \)
6: \( \text{end if} \)
7: \( \text{best_reduction} = 0 \)
8: \( \text{for each} \ \text{input in} \ F, f_i \ \text{do} \)
9: \( \text{reduction} = \text{err} - \text{error}(E_{f_i=0}, z) - \text{error}(E_{f_i=1}, z) \)
10: \( \text{if} \ \text{reduction} > \text{best_reduction} \ \text{then} \)
11: \( \text{best_reduction} = \text{reduction} \)
12: \( f_{\text{best}} = f_i \)
13: \( \text{end if} \)
14: \( \text{end for} \)
15: \( \text{if} \ \text{best_reduction} > 0 \ \text{then} \)
16: \( \text{decision_tree}(F - f_{\text{best}}, P \cup f_{\text{best}} = 0, E_{f_{\text{best}}=0}) \)
17: \( \text{decision_tree}(F - f_{\text{best}}, P \cup f_{\text{best}} = 1, E_{f_{\text{best}}=1}) \)
18: \( \text{end if} \)

to the design for verification and checked at the positive edge of the clock cycle. The reset signal of the design is constrained to “off” so to prevent spurious counterexamples. Although the attempt in GoldMine is to minimize the human effort in the assertion generation process, we need human intervention to differentiate between a spurious candidate assertion that fails the formal verification and a genuine system invariant whose failure reports the existence of a bug.

4.6 A-Val: Evaluation and Ranking

Once the assertions have been generated through GoldMine, their evaluation is extremely important to the process. This is because assertion generation has been a completely manual process thus far in the system design cycle.

There are several ways for us to evaluate A-Miner’s performance. One basic metric is the hit rate of true assertions. The hit rate of a run in GoldMine is the ratio of true assertions to candidate assertions. This provides a very crude indicator of performance. In addition, we can consider output hit rate, which is the number of outputs for which GoldMine could generate a true assertion over
the total number of inputs.

Since there are no commercially used metrics for evaluating the coverage of an assertion, we have devised a method to evaluate assertion coverage. It should be noted that this metric has no relation to standard coverage metrics such as code, branch, or path coverage. The reason for this is that those metrics are used for judging the quality of a directed test suite, which means that they cannot be applied to a set of individual assertions. We can evaluate the coverage of an assertion by considering the input space that is covered by the antecedent of the assertion. If we consider the truth table with respect to some output, each entry that corresponds to the propositions in the antecedent of an assertion is defined as covered by that assertion. For example, if there is an assertion \((a = 1 \& b = 1 \implies c = 1)\), we can consider the input space coverage to be 25% since we know that 25% of the truth table entries contain \(a = 1, b = 1\). The reasoning behind this thinking is that if there is a set of assertions that covers each entry in the truth table of an output, that output is well covered by the set of assertions. This metric is simple to calculate since we can determine the percentage of the input space that an antecedent of an assertion covers without knowing every single input combination. The input space coverage is defined as \(1/2^{|P|}\) where \(|P|\) is the number of propositions in the antecedent. Based on this definition, it can be seen that the input space coverage is relative to the number of propositions in the antecedent. It should also be noted that this notion of coverage can be extended to sequential designs. If we consider an unrolled circuit where each signal, \(s\), in the truth table is represented at the current time, \(s[t]\), we can consider the signal at each time cycle before it, \(s[t-1], s[t-2], ..., s[t-n]\). Given that \(n\) is large enough, we will always be able to represent this coverage accurately in these terms.

In order to bridge the gap between the human and the machine generated assertions, human judgment can also be made a part of the GoldMine process where the designer ranks the true assertions according to some pre-defined ranks. This provides an objectification of an inherently subjective decision and can be used as feedback into A-Miner, with a view to predict the ranking of a generated assertion and optimize the process for achieving higher ranks.
4.7 Temporal Assertions

There are some single-cycle assertions which are interesting, but it can be even more interesting to see assertions which span several cycles. These multi-cycle assertions can be found without having to change the data mining algorithm. When the simulation trace is produced, each signal in a sample refers to the value of that signal at the current time, \( t \). The maximum length of a temporal assertion is user-specified as \( l \). We want to represent the signals at previous time cycles \( t - 1, t - 2, \ldots, t - l \) in this sample. For each signal, we can determine the previous values of that signal by checking the samples representing a previous time before the current sample. Now that there is data representing each signal over a number of cycles, the data mining algorithm can proceed as normal to look for relationships.

For example, consider a protocol that asserts \( \text{ack} = 1 \) two cycles after \( \text{req} = 1 \). The simulation data for this module is shown in Table 4.1.

<table>
<thead>
<tr>
<th>time</th>
<th>req</th>
<th>ack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1: The simulation data for a req/ack protocol

We set our maximum assertion length \( l = 2 \) and perform the necessary data transformation. The resulting simulation data are shown in Table 4.2. Since there is no information for \( [t - 1] \) in cycle 0 or \( [t - 2] \) in cycle 0 or 1, we must discard the data in cycle 0 and 1.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.2: The previous cycle information is added to enable temporal assertion mining

The data mining algorithm used for single-cycle assertions cannot be applied. In cycle 3, there is a clear relationship between \( \text{ack}[t] \) and \( \text{req}[t - 2] \) which results in the assertion \( \text{req} = 1 \times \times \implies \text{ack} = 1 \). This assertion represents the expected behavior for the protocol.
4.8 An Example Run through GoldMine

always @ *
    if (int.valid && int.has_dreg) 
        wb_valid0 = 1;
    else 
        wb_valid0 = 0;
always @ * 
    int.L1_hit = int.has_dreg

<table>
<thead>
<tr>
<th>int.valid</th>
<th>int.L1_hit</th>
<th>int.has_dreg</th>
<th>wb_valid0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Consider the fragment of the Rigel processor RTL source code shown in Figure 4.2. This code implies that writeback on port 0, \(wb\text{\_valid0}\), is valid if the integer writeback signal \(int\text{\_valid}\) is set and a register is available \(int\text{\_has_dreg}\). This event updates the L1 cache hit rate. We now illustrate the GoldMine assertion generation process on this code. The Data Generator runs a few simulations and produces the simulation results shown in the table in Figure 4.2.

In the absence of any guidance from the static analyzer, A-Miner forms a decision tree for the data. Figure 4.3 shows this process. The mean of \(wb\text{\_valid0}\) is set to 0.25 (average of its values) and the error is set to the absolute difference from the mean, 0.375. The decision tree now tries to split based on the maximum error reduction among all the input values. The values of error for the 0/1 values of \(int\text{\_valid}\), \(int\text{\_L1\_hit}\) and \(int\text{\_has_dreg}\) are (0,0.5). Since all values (0/1) of all inputs produce equal error, and in the absence of any guidance from the static analyzer, the decision tree uses the simple heuristic of splitting on the first variable in the list, \(int\text{\_valid}\). On the \(int\text{\_valid} = 0\) branch, error is reduced to 0, making it a leaf node. \(A0: if (int\text{\_valid} = 0) then (wb\text{\_valid0} = 0)\) is the candidate assertion generated. Since the error value has not yet reached 0 on the \(int\text{\_valid} = 1\) branch, the decision tree tries to split again. Although the value of \(int\text{\_has_dreg}\) is the variable that affects the output of interest, the splitting variable is \(int\text{\_L1\_hit}\) since the error reduction for all variable values are equal, and it is first in the list. Since both branches of the tree at this level reach error = 0, the leaves produce \(A1: if (int\text{\_valid} = 0) and (int\text{\_L1\_hit} = 0) then (wb\text{\_valid0} = 0)\) and \(A2: if (int\text{\_valid} = 1) and (int\text{\_L1\_hit} = 1) then (wb\text{\_valid0} = 1)\) as candidate assertions.

All candidate assertions \(A0, A1, A2\) are passed to a formal verification engine, that passes \(A0\) and \(A1\), but fails \(A2\). Hit rate is 2/3 in this case. \(A3\) fails due to the
false causality that is established by simulation data.

In the presence of the lightweight static analyzer, the logic cone-of-influence information would suffice in this case. The logic cone establishes the part of the design that is causal to \text{int.valid}, providing a list of variables to the decision tree that excludes \text{int.in.L1.hit}. The corresponding decision tree is shown in Figure 4.3. The candidate assertions produced now are \( A0 \) (same as in previous case), \( A1: \text{if} (\text{int.valid} = 0) \text{ and } (\text{int.has_dreg} = 0) \text{ then } (\text{wb_valid} = 0) \) and \( A2: \text{if} (\text{int.valid} = 1) \text{ and } (\text{int.has_dreg} = 1) \text{ then } (\text{wb_valid} = 1) \). All these candidate assertions are passed by the formal verifier, with a consequent hit rate of 1.

There are three disadvantages of the temporal assertion mining method. The first is that there must be a user-specified bound on the maximum number of cycles in an assertion, \( l \). The second is that as \( l \) increases, the runtime of the algorithm increases since the number of signals that the data mining algorithm needs to search has increased. The third disadvantage is that as \( l \) increases, the quality of the generated assertions can decrease since the number of inputs can get so large that making a good splitting decision is difficult. These disadvantages can be mitigated by using background knowledge of the design to choose a good maximum cycle length, \( l \), or testing several different values for \( l \) to optimize results.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{decision_tree.png}
\caption{Example Decision Tree Output with and without Static Analysis}
\end{figure}
4.9 Applications of GoldMine

Though GoldMine is an interesting tool, it can be difficult to see how it can be used in a realistic verification environment. Since GoldMine produces assertions based on RTL which are then verified using formal verification, it is trivial that generated assertions will pass on the given RTL. The beauty of this tool is that it can actually be applied in a number of ways, included applications that have not even been developed yet.

One way to use GoldMine effectively is to use the assertions as a regression test throughout development. The assertions that are true in one revision may fail in a later revision. This can indicate that the assertions are no longer relevant, which indicates that those assertions must be updated. However, it can also indicate that a revision of the design introduced a bug which the assertion can help to locate. For example, GoldMine is used on an ALU unit and produces a set of assertions. The ALU is then revised to make a certain function faster. If there are any assertions that fail, it likely indicates that there is a bug in the revised code.

When using random testing to verify a design, it can be difficult to determine the number of cycles to simulate before declaring a unit fully verified. One way to measure testing completeness is to use standard coverage metrics, but this method only gives a very general idea of the coverage. GoldMine can also be used in addition to standard coverage metrics to increase confidence of a design. The trace from the random test simulation can be mined for assertions using GoldMine. Any assertion mined from this trace indicates behavior that is covered in the simulation trace. This means that if the assertions generated in GoldMine have a high coverage, it is likely that a high percentage of design behavior has been covered in the random test. If the assertions generated do not have high coverage, the simulation likely needs to run for more cycles.
We first present the results of applying GoldMine to the 1000+ core Rigel RTL design. Our intention is to use assertions from GoldMine to provide a regression test suite for the Rigel RTL that is in the later stages of its evolution. We generated assertions for three principal modules in Rigel: the writeback stage, the decode stage and the fetch stage. The writeback stage is a combinational module with interesting propositional properties. The decode and fetch stages are sequential modules with many interesting temporal properties.

5.1 Subjective Ranking of Assertions by a Designer

We performed some experiments to help evaluate GoldMine’s assertions. We performed an extensive designer ranking session for every phase of assertion generation of each module. Also, since the Rigel RTL does not have manual target assertions to compare against, we performed a subjective, but intensive evaluation strategy. Rankings were from 1 to 4, calibrated as below:

1. Trivial assertion that the designer would not write
2. Designer would write the assertion
3. Designer would write, captures subtle design intent
4. Complex assertion that designer would not write

The results presented in Figure 5.1 show the distribution of these ranks for a sample of representative assertions over all the modules. The algorithmic knobs that produced the highest hit rate as well as the highest number of assertions were

This chapter includes work previously published in [1].
Figure 5.1: GoldMine assertion complexity; ranking by designers

turned on for this experiment. Most assertions in this analysis rank at 2. The writeback module has some assertions ranked 3. The absence of 3 in the sequential modules, according to the designers, is due to the fact that intra module behavior is not complicated enough to have many subtle relationships. For example, an assertion ranked 1 is: *If the halt signal in the integer, floating point, and memory unit is set to 0, the halt signal is 0.* In the RTL, the halt signal is a logical OR between the integer, floating, and memory units. GoldMine found a true, but over-constraining rule. The designers ranked it 1, since they would not have written this rule. Now, consider this RTL code:

```plaintext
decode2mem.valid <= valid_mem &
!issue_halt & !branch_mispredict &
fetch2decode.valid & !follows_vld_branch
```

An assertion ranked 2 is *if branch_mispredict is high, decode2memvalid will be high in the next cycle.* An assertion ranked 3 is *If an integer unit does not want to use the first port, and the floating point unit does not want to use the second port, then the second port remains unused.*
5.2 Complex Assertions in GoldMine

Despite the small size of the modules, GoldMine achieved rank 4, *i.e.* it produced assertions that capture complex relationships in the design. This is an advantage of mechanically derived assertions: they are able to capture unintentional, but true, relationships that can be excellent counter checks and can be brought to the designer’s attention. We assessed complexity by the number of levels (depth) of the design captured by assertions. In a few cases, the assertions capture temporal relationships that are more than 6 logic levels deep in the design. This provides a different perspective on the RTL, outside of the expectation, but may provide avenues for optimizing or analyzing the RTL. For example, the RTL has the following relationship:

\[
\text{if( choice_mem)} \\
\quad \text{decode\_packet} \leq \text{decode\_packet1};
\]

An assertion ranked 4 is: \(\text{if (reset=0) and (issue0=0) and (decode\_packet\_dreg=0), and in the next cycle if (instr0\_issued = 0), then decode\_packet\_dreg = 0.}\) This assertion relates a single field in the \text{decode\_packet} variable to \text{reset} and \text{instr0\_issued}, both of which are related to \text{choice\_mem} when the code is traversed beyond 6 levels of (sequential) logic. Such a relationship would have been extremely hard to decipher through static analysis and code traversal. To the best of our knowledge, there is no state-of-the-art tool/technique that can claim to decipher such complex assertions. Figure 5.1 shows the distribution of assertions with respect to complexity.

5.3 Outputs Covered by GoldMine

| Table 5.1: Percentage of outputs covered by GoldMine for Rigel |
|-------------------|-----------------|
| time              | Outputs Covered |
| Decode Stage      | 46.76%          |
| Fetch Stage       | 35.71%          |
| Writeback Stage   | 87.50%          |

Table 5.1 shows the number of outputs per module for which assertions were generated by GoldMine. Although candidate assertions were generated for all the module outputs, the assertions that passed formal verification covered a percentage of them. Figure 5.2 shows the probability distribution of true assertions per
output. At the 50% mark, there will be approximately 4 to 5 unique assertions per output in the decode module. Although we are not able to get a precise notion of path coverage per output signal, the unique assertions per output are indicative of high path coverage.

![Cumulative Distribution Function](image)

**Figure 5.2: Distribution of unique assertions per output in all modules**

5.4 The Acid Test: Regression Test Experiments

As a final evaluation of the entire regression suite of GoldMine assertions, we appended them in the RTL and ran a new set of directed Rigel tests.

We will analyze the results for the writeback module, since the fetch and decode are very similar. We used Synopsys VCS with RTL conditional coverage for procuring coverage of the directed tests. We used the conditional coverage metric since unique assertions in GoldMine pertain to different paths. This metric is meaningful for us since it examines individual path conditions in generating an output.

The writeback module directed tests achieved 76% conditional coverage, while the random tests used to generate the GoldMine assertions achieved 100% con-
ditional coverage and generated 200 unique assertions. When the GoldMine assertions were included in the directed test runs, 110 (55%) of the assertions were triggered\(^1\) by the directed tests. Therefore, 90 assertions, or 45%, refer to design behavior as yet untested by the directed tests. Figure 5.3 shows the overlap of assertions with directed tests. This highlights the value of GoldMine, since it provides significant coverage of the unexplored regions of the design at this early stage.

The overlapping assertions that coincide with the designer-crafted directed tests can be used for static checking, formal verification, etc. However, the untouched assertions can be used to improve the quality of the directed tests. They can be used as regression checks as the test patterns mature and the regression test suite evolves. It is probable that the manual assertion generation process would eventually get to this point after multiple iterations. In contrast, GoldMine, a mechanical assertion generator, could explore the design space far beyond the human generated tests. The designers of Rigel have evaluated GoldMine’s contribution as “covering a wide design space much earlier in the design cycle than typically achievable” [45].

---

\(^1\)An assertion is triggered if the antecedent condition evaluates to true.
It is difficult to properly assess the utility of GoldMine without testing the tool on an industrial size design. This can be difficult since most companies are protective of their IP and will not distribute their HDL designs. However, Sun provides a few open source designs for the UltraSparc series of CPUs. Sun’s OpenSparc T2 CPU [46] is a many-threaded, open source design which makes it an optimal example to demonstrate GoldMine.

For our initial tests, we have isolated the memory management unit (MMU) of the core for GoldMine assertion generation. This unit reads the TLB for the data and instruction caches and performs a page table walk in the case of a miss. This unit has 59 inputs, 54 outputs, and 313 internal signals. We searched for assertions for the 16 outputs for which we could generate a significant number of samples using random input vectors. The A-Miner searched for correlations between each output and all of the inputs and internal signals. If no logic cone information is used, the total number of bits that the decision tree can split on is nearly 3000, making this a complex test for the A-Miner. We performed tests with 10,000 and 1 million cycles worth of simulation data for assertion generation.

6.1 Evaluation of True Assertion Success Rate

The first metric for gauging the success of GoldMine is to determine what percentage of outputs had at least one valid assertion generated. We compare several different configurations of GoldMine for this statistic. Our first configuration uses 10,000 cycles of simulation data and no logic cone of influence. The second also uses the same 10,000 cycles of simulation data, but includes the logic cone information. The third and fourth configurations use 1,000,000 cycles of simulation data. Similarly, the third configuration does not use logic cone information and the fourth configuration does. In Figure 6.1, we can see that both increasing the
number of cycles of simulation data and using logic information can increase the
number of outputs with at least one true assertion. Also interesting to note is that
only using the logic cone can result in more outputs being covered than increasing
the number of cycles of simulation data by two orders of magnitude.

![Figure 6.1: Percentage of outputs for which at least one true assertion was generated](image)

6.2 Evaluation of Assertion Input Space Coverage

In the process of evaluating the assertions generated for the Rigel design, we had
the liberty of having the generated assertions ranked by the actual designers of the
modules. However, we do not have the same ability to get subjective rankings on
the OpenSparc CPU. Because of this, we have to use an objective way to assess
the quality of the assertions. To do this, we use input space coverage as defined in
Section 4.6. For this experiment, we use simulation data generated using 10,000
and 1,000,000 cycles of random input stimulus. As shown in Figure 6.2, we can
see that GoldMine is able to produce assertions with good input space coverage
with only 10,000 cycles of simulation data. The input space coverage when using
a large amount of data produces a set of assertions with even greater input space
coverage. However, as we can see in Figure 6.3, the total number of assertions
increases greatly to account for the new coverage.
6.3 Evaluation of the Percentage of Complex Assertions

To assess the complexity of the assertion sets, we again must use an objective measure since we do not have a designer to review the assertions. We can consider the complexity of an assertion to be relative to the number of propositions in the antecedent of an assertion. For this experiment, we made a statistic of the number of assertions that had more than 10 propositions based on the intuition that it would be difficult for a verification engineer to develop an assertion with this complexity. The percentage of complex true assertions out of the total number of true assertions is shown in Figure 6.4. There are certain outputs which have a higher percentage of complex assertions which can be attributed to the complexity of the logic corresponding to that circuit. This figure also shows that the experiment with 1,000,000 cycles tends to produce more complex assertions since complex behavior is more likely to appear in a larger random input simulation trace.
6.4 Comparing the Generated Assertions with the OpenSparc Specification

To further judge the quality of the assertions generated by GoldMine, we observe some candidate assertions generated for the L2 cache controller (L2T) of the OpenSparc SoC. These candidate assertions are generated with respect to the L2 pipeline stall signal. To understand these assertions, the circuit behavior must first be understood. A stall (l2t_pcx_stall_pq) is signaled when the input queue (IQ), which contains requests for the L2 cache, is full. There are two signals which control whether a request is added or removed from the queue. The data ready signal (pcx_l2t_data_rdy_px2_d1) indicates that a request will be added to the queue, causing an increase in queue size. The input queue select signal (arb_iqsel_px2_d1) indicates if a request can be removed and processed by the L2 cache, causing a decrease in queue size.

A complication to this is that the data ready and IQ select signals are passed through a series of flops before they are evaluated to determine whether or not there is a stall. The input signals to the L2 cache for the data ready signal is pcx_l2t_data_rdy_px1, while the input signal for IQ select is arb_iqsel_px2. The chart in Table 6.1 shows that the input for IQ select arrives one cycle before being evaluated, while the input for data ready arrives three cycles before.
Figure 6.4: Percentage of true assertions which have greater than 10 propositions in the antecedent.

Table 6.1: The temporal relationship between signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Cycle t-3</th>
<th>Cycle t-2</th>
<th>Cycle t-1</th>
<th>Cycle t</th>
</tr>
</thead>
<tbody>
<tr>
<td>IQ Select</td>
<td>–</td>
<td>–</td>
<td>arb</td>
<td>arb</td>
</tr>
<tr>
<td>Data Ready</td>
<td>data</td>
<td>data</td>
<td>data</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>rdy_px1</td>
<td>rdy_px1_fnl</td>
<td>rdy_px2</td>
<td>rdy_px2_d1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>px2</td>
<td>d1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The candidate assertions generated by GoldMine are shown below.

**Candidate Assertions:**

@ (posedge gclk)

gm1: l2t_pcx_stall_pq=1 |=> #1 l2t_pcx_stall_pq=1;

@ (posedge gclk)

gm2: pcx_l2t_data_rdy_px1==1 #2 arb_iqsel_px2==0 && l2t_pcx_stall_pq==0 |=> #1 l2t_pcx_stall_pq==0;

@ (posedge gclk)

gm3: pcx_l2t_data_rdy_px1==1 #2 arb_iqsel_px2==1 && l2t_pcx_stall_pq==0 |=> #1 l2t_pcx_stall_pq==0;

@ (posedge gclk)

gm4: pcx_l2t_data_rdy_px1==0 #2 l2t_pcx_stall_pq==0 |=> #1 l2t_pcx_stall_pq==0;

Based on the behavior described, we can now determine the validity and usefulness of the given assertions. In assertion gm1, the assertion indicates that if there
was a stall in the previous cycle, there will also be a stall in the current cycle. This is clearly a spurious assertion. If the queue is currently full, an instruction may be processed causing the queue to no longer be full. This means that in the simulation trace, there were many instances where a stall was followed by a stall, leading to a correlation. However, the circuit does not behave in this way, meaning \( \text{gm}_1 \) is false.

What assertion \( \text{gm}_2 \) indicates is that if stall is currently not active and a request is added, the pipeline will not become stalled. If stall is inactive, the size of the queue is currently below capacity. Based on the inputs, the size of the queue must increase since data ready is true and IQ select is false. This assertion will be true most of the time, but if the queue is one below capacity, an added request will cause it to become full. This will result in a stall. This means that \( \text{gm}_2 \) is also false.

Assertion \( \text{gm}_3 \) states that if there is no stall and the queue size does not change, the queue will remain unstalled. The queue size remains the same because data ready is true, indicating that a request is added, but IQ select is active, meaning that a request is also processed. If the queue is below capacity and stays the same size, it will remain below capacity, meaning that there is no stall. Assertion \( \text{gm}_3 \) is true.

In assertion \( \text{gm}_4 \), if there is currently no stall and there are no more requests added to the queue, the pipeline will remain unstalled. Since data ready is false, the queue size must stay the same or decrease (if IQ select is active). This means that if the queue is below capacity, it must remain below capacity. This means that assertion \( \text{gm}_4 \) is also true.

These results show that the assertions that GoldMine generates are interesting and complex, making them good choices for including in an RTL design. It also shows that, while it can take humans a long time to reason about circuit behavior, GoldMine is able to do it much more quickly and efficiently.

### 6.5 Evaluation of the Runtime and Memory Usage of GoldMine

Our last experiment evaluates the performance of GoldMine. In this experiment, we compare Rigel and OpenSparc modules in terms of runtime and memory. First, we present a comparison of the characteristics of our test modules as shown in
Table 6.2. We will later show how each of these factors affects the runtime and memory consumption.

<table>
<thead>
<tr>
<th>Module</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rigel - Decode Stage</td>
<td>2195</td>
<td>79</td>
<td>32735</td>
</tr>
<tr>
<td>Rigel - Fetch Stage</td>
<td>458</td>
<td>6</td>
<td>4165</td>
</tr>
<tr>
<td>Rigel - Writeback Stage</td>
<td>963</td>
<td>3</td>
<td>269</td>
</tr>
<tr>
<td>OpenSparc - MMU</td>
<td>3393</td>
<td>16</td>
<td>66395</td>
</tr>
</tbody>
</table>

We will first look at the runtime of GoldMine. We use a simulation trace of 10,000 cycles as well as a trace containing 1,000,000 cycles. These tests are performed on a 2.66GHz Intel Core 2 Quad CPU with 4GB RAM. Figure 6.5 shows the runtimes without formal verification. This figure shows that, even on a common desktop processor, GoldMine is able to produce candidate assertions in a very short time. GoldMine runs in just minutes for both the 10,000 and 1,000,000 cycle simulation trace. It can also be observed that the runtime has no relationship with the circuit size. Instead, it is the number of inputs, outputs, and cycles of simulation data which affects the runtime. This means that GoldMine has extremely good scalability.

![GoldMine runtime; no formal verification](image)

The runtime is also evaluated for GoldMine when formal verification is used. For this test, we use a cluster of four six-core AMD Operton 8435 CPUs and enable parallel formal verification of assertions. Figure 6.6 shows the runtimes when...
formal verification is used. The runtime is expectedly much higher, but even for
the complex OpenSparc MMU module, the 10,000 cycle test completes in only
one hour and the 1,000,000 cycle test completes in just over 2 hours. One of the
largest factors influencing the runtime when formal verification is enabled is the
number of candidate assertions that are generated, since each one must be verified.
A solution for the reduction of runtime would be to limit the number of candidate
assertions produced. Though this may limit the number of true assertions gener-
ated, it may be a viable choice when runtime is limited.

Our last performance experiment is to record the maximum memory usage of
GoldMine. This test is performed on the Intel Core 2 Quad CPU. Since formal
verification does not affect the memory usage of GoldMine, it is disabled in this
test. Figure 6.7 shows the results of the test. From this figure, it is clear that
GoldMine is very efficient in terms of memory usage. Even in the worst case,
GoldMine does not exceed 1GB of memory usage in these tests. It can also be
observed that the memory usage is again not related to the area of the circuit. The
memory usage is actually related to the size of the simulation trace that must be
stored in memory, meaning that both the number of inputs and simulation cycles
affect the memory usage. The memory usage is also affected by the size of the
decision tree data structure which, in the worst case, can be exponential with the
maximum height of the tree. If we limit the height of the tree, we do not have
to worry about the tree size ever becoming a problem. Since memory usage is

Figure 6.6: GoldMine runtime; formal verification enabled
not relative to the area or complexity of a circuit, GoldMine has great memory scalability as well as runtime scalability.

**Figure 6.7:** GoldMine maximum memory usage
CHAPTER 7

THE EVOLUTION OF GOLDMINE

GoldMine has evolved continuously since the original concept for the tool was developed.

7.1 Shaping GoldMine: Early Changes to the GoldMine Methodology

In the initial phase of GoldMine development, we used an FP-Growth algorithm. This took an unreasonable time (>10 hours) for reaching rules with just 3 predicates for the decode module. We therefore resorted to the decision tree algorithm for our purposes. The decision tree is a very fast data mining algorithm that does not suffer from an exponential runtime like the FP-Growth algorithm does.

In the first iteration of the Data Generator, we used directed test simulations. This data was insufficient (approximately 15 tests of 1000 samples each), producing a very low hit rate. We then used random input vector generation on the RTL for the target modules. Even when using only 10,000 samples of simulation data, this drastically increased the hit rate as well as number of true assertions, demonstrating that the type and amount of data can greatly affect the results of GoldMine. For the writeback module, we achieved a 100 percent hit rate with this step alone.

Another aspect that had been changed is the stopping criterion of the decision tree splitting. Our initial experiments continued the splitting process beyond the point where the minimum error reduction was reached. This process gave us an extremely high number of candidate assertions (>80,000) with many duplicates (289 out of 300 in one test). In the later stages, we elected to end the decision tree splitting when error was numerically equal to “0”, i.e. at the point of 100% confidence, since nothing can be gained past this point.

Originally, GoldMine only worked with combinational circuits, which are inter-
interesting, but not very useful to the average verification engineer. The reason for this is that pattern recognition algorithms used in data mining look for correlations that hold true in all samples, which is consistent with combinational behavior since outputs change immediately. However, in sequential circuits, outputs do not change until the positive edge of the clock. This means that if a sample is taken before the clock edge, the output will contain the value determined by the inputs at the previous clock edge, and not the inputs at the current time. This means that no relationship can be found since the current inputs have not influenced the output yet. If a sample is taken after the clock edge, the inputs have already changed from the values that determined the current output, meaning that there is still no relationship that can be inferred from the samples. This problem can be solved without having to change the data mining algorithm. The data is only sampled once per positive clock edge since that is when the interesting behavior happens. The exact time at which the signal is sampled depends on the type of signal. If the signal is an input, it is sampled right before the positive clock edge, and if the signal is an output, it is sampled right after the clock edge. This makes it seem as if the inputs and outputs have changed at the same time and the data mining algorithm is able to find relationships between the inputs and outputs.

In the next phase of GoldMine, we added **lightweight static analyzer information** that was specific to the domain, such as logic cone-of-influence generation. Although this increased the hit rate only marginally, it increased the number of true assertions significantly. This shows that the static analysis information was very useful in helping A-Miner focus on the relevant neighborhood of variables to generate candidate assertions.

### 7.2 Performance Enhancements

The decision tree algorithm is very quick, but the formal verification in GoldMine can take a long time when there are many assertions to verify. By using a commercial tool for formal verification instead of SMV, we were able to achieve a significant speedup. We have also used parallelism to increase the speed of the formal verification step. Since each assertion can be verified concurrently, several formal verification threads can be used for a significant speedup. Because there is some overhead in creating the model in formal verification, a small batch of assertions is verified in each thread.
Since memory conservation is important for large problems, we have ported our code from Java to C++. Since Java has dynamic memory management, it is difficult to control the memory usage and it can be difficult to debug memory leaks. Since C++ requires manual memory management, it is easier to keep the memory usage low and controlled.

7.3 Improving the Core GoldMine Algorithms

Though we made many changes to the core of GoldMine, we have also extended the GoldMine tool in several ways. The first extension to GoldMine was the addition of counter-example feedback. When the formal verification step determines that a candidate assertion is false, it also produces a counter-example to prove that the assertion can be violated. We can use this information to our advantage and give feedback to the data mining engine. What we do is convert the counter-example into a data sample, as if that sample were included in the original simulation trace. This forces the data miner to reconsider the confidence of this assertion. Since a counter-example is added, the confidence can no longer be 100%, meaning that the decision tree can continue to split. This method allows the decision tree to continue to produce new candidate assertions until all assertions are true. This methodology is discussed in detail in [47].

We have also explored alternative data mining algorithms to the decision tree based supervised learning algorithm detailed in Section 4.3. A coverage guided algorithm for generation of high-quality assertions is described and evaluated in Chapters 8, 9, and 10.
CHAPTER 8

MOTIVATION FOR A COVERAGE GUIDED APPROACH

While the decision tree supervised learning algorithms produces excellent results and provides an excellent jumping-off point for the A-Miner in GoldMine, this data mining algorithm has several flaws.

8.1 Why Decision Tree Assertions Need Improvement

In addition to the lack of assertion quality awareness, the decision tree has other shortcomings. Due to its faithfulness to a (binary) tree structure, it explores every value of each splitting variable. An assertion generated at a leaf node will necessarily have all the splitting variables of the previous levels of the tree. This leads to assertions that are over-constrained, or contain too many propositions (variable, value pairs) in the antecedent. Intermittent poor splitting choices during tree construction can result in irrelevant variables being added in the assertions as well.

For instance, a decision tree would create the assertion \((\text{request} \land \text{we} \land \text{rd} \land \text{branch}) \implies (\text{gnt})\), where the dependencies on the write enable and read signals are coincidental, but not causal. The desired assertion would be \((\text{request}) \implies (\text{gnt})\). \(^1\) Over-constraining restricts behavior and reduces the input space behavioral coverage of assertions. It also decreases the readability of the assertions. Since individual decision tree assertions have low input space coverage, a large number of assertions is required to cover the design behavior. An increase in the number of assertions is an undesirable side effect, since it implies overhead, whether in pre-Silicon runtimes or in post-Silicon cost.

A subjective ranking distribution by the designers of the Rigel \([48]\) processor for decision tree generated GoldMine assertions is shown in Figure 5.1. Rank 1 represents a trivial assertion that would not be used in verification of the design.

\(^1\)GoldMine produces assertions using Linear Temporal Logic \([9]\). The proposition on the left-hand side of the implication operator is the antecedent and the right-hand side is the consequent.
8.2 Coverage Guided Mining

We present the coverage guided mining algorithm, which is intended to increase the number of rank 3 assertions and decrease the number of rank 2 assertions produced by GoldMine. This coverage guided association miner replaces the decision tree in the A-miner phase of the GoldMine algorithm. It uses a combination of association rule learning, greedy set covering, and formal verification. In each iteration of the coverage guided mining algorithm, the association rule learning finds each assertion that has higher coverage than a specified minimum coverage. In successive iterations, the minimum coverage for each assertion is lowered. This guarantees that the highest coverage assertions are added to the candidate assertion set in a greedy manner at every iteration. In addition, a formal verifier is used to verify that candidate assertions added to the solution set are true.

Algorithms based on association rule learning are typically not scalable due to their nature of finding all relations between all variables exhaustively. However, in our algorithm, we constrain the solution space of the association learning by considering only those candidates that fulfill a coverage criterion. We also require that the candidates should be true as attested by formal verification. We also use a heuristic of having minimal propositions in an antecedent for our greedy selection of high-coverage candidate assertions. These restrictions sidestep the exhaustive nature of the association learning and result in an efficient, scalable approach.

Our approach produces succinct assertions, with higher expressiveness per assertion. This upgrades the value added by an assertion. Since the value added by an assertion can be quantitatively expressed as input space coverage, this al-
algorithm iteratively refines the set of assertions until it maximizes the coverage achieved by them. The coverage guided mining algorithm, therefore, converges to a set of assertions that are few in number, but high in coverage. A graphical representation of these two methods is shown in Figure 8.1.

Figure 8.1: Comparison between assertions in decision tree and coverage guided mining over time for a design output. The dots represent behavior points in the design. Decision tree generated assertions are unaware of behavior coverage and do not optimize the design points covered. Coverage guided mining is coverage conscious when generating assertions and greedily picks the highest coverage ones.

Our experimental results are shown on the OpenSparc T2 [46], OR1200 [49], SpaceWire [50], ITC benchmarks [51], and Rigel [48] processor RTL modules. We show that coverage guided association mining performs competitively against the decision tree method in terms of overall input space coverage and far better than decision trees with respect to input space coverage per assertion, number of propositions per assertion, and subjective designer rankings.
CHAPTER 9

THE COVERAGE GUIDED MINING ALGORITHM

9.1 Background Concepts

Association rule mining [41] is a data mining method that attempts to generate all possible correlations between items. Though this algorithm has an exponential complexity in the worst case, high efficiency is achieved by applying constraints and using pruning techniques.

The set covering problem refers to a case where there are many sets that each cover several elements and one wishes to find the minimal number of sets that cover all possible elements. The complexity of finding the optimally minimal set cover is NP-Complete [52]. However, there are many approximation algorithms which can find a near-optimal solution efficiently. The greedy set covering algorithm works by choosing the set that covers the largest number of uncovered element until all elements have been covered.

Gain is a data mining concept that refers to the value of adding some rule to the solution set of rules. In data mining, we only want to add a rule to our solution set if its gain is higher than any other potential rules. This concept fits well with our concept of input space coverage since we can define a notion of coverage gain. The coverage gain of a rule (assertion) refers to the change in total coverage of a set given that the rule is added to that set. For example, if a set of assertions has a total input space coverage of 75% and an assertion with a coverage gain of 12.5% is added, the new total coverage of that set will be 82.5%.

Typically, an association mining algorithm will try to exhaustively produce all possible rules relating all input variables to all output variables. To restrict the number of rules, we apply several constraints. Our first constraint, as in [1], is that only rules with 100% confidence can be considered as candidate assertions for association rule mining. We now include coverage feedback as a constraint. We impose a minimum coverage gain to drastically limit the number of candidate
assertions. We then gradually relax this constraint until we have reached a desired coverage value. The greedy set covering algorithm will always choose the highest coverage assertions in each iteration.

As defined in Section 4.6, input space (or truth table) coverage is a metric which has been adopted for the purpose of evaluating a set of assertions in relation to some output. Because no alternative metric exists for evaluating the quality of an assertion, we use this definition for the coverage guided mining algorithm. It should be noted that if coverage is mentioned, it is assumed that it is input space coverage.

9.2 Algorithm Explanation

We run this algorithm to generate assertions for a specified output in a design, z. The assertions will be in the format where a set of propositions describing input variables and their respectively assigned values imply that the output, z, will be a certain value.

$A_s$ is defined as the solution set of assertions. The expected total input space coverage of $A_s$ is defined as $c(A_s)$. We define $g(A_s, A'_s)$ as the input space coverage gain between two sets of assertions where $A'_s = A_s + a$ and $a$ is an assertion. We also define $g_{min}$ as the minimum coverage gain. The minimum coverage gain ensures that any assertion that is mined must raise the total coverage of $A_s$ by $g_{min}$. The total coverage of $A_s$ is defined as $c(A_s)$. We set a minimum coverage gain threshold $g_{threshold}$ and a maximize total coverage threshold $c_{threshold}$ which result in algorithm termination when reached. Our goal is to maximum the expected total input space coverage $c(A_s)$ by maximizing the $g(A_s, A'_s)$ in each iteration while minimizing the total number of assertions and propositions in the antecedent of each assertion.

The basic flow of the algorithm is shown in Figure 9.1. We will apply the algorithm as it is explained to the simulation trace in Figure ??.

We set the maximum total coverage threshold to 99% and the minimum coverage gain threshold to 1%.

The algorithm starts by initializing the $g_{min} = 50\%, A_s = \{\}$, and $c(A_s) = 0\%$. We know that at least one proposition must be in the antecedent of the assertion, which means that the maximum coverage gain must be 50%. We do not consider assertions without any propositions in the antecedent since those assertions are trivial.
In the next step, \textit{gen\_candidates}, the algorithm described in Algorithm 2, is invoked. In the \textit{gen\_candidates}, \( P \) refers to a set of \{input variable, value\} pairs representing the antecedent of a potential assertion \( a \). \( F \) refers to the set of \{input variable, value\} pairs not in \( P \), since we do not want to add the same \{input variable, value\} pair to an antecedent twice. \( E \) refers to the simulation trace and is represented as a set of signal values at each cycle. In our example \( F = \{\{a, 0\}, \{a, 1\}, \{b, 0\}, \{b, 1\}, \{c, 0\}, \{c, 1\}\} \), \( P = \{\} \), and \( E \) is the data in Table 9.1.

\begin{table}[h]
\centering
\begin{tabular}{cccc}
\hline
\textbf{a} & \textbf{b} & \textbf{c} & \textbf{z} \\
\hline
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 \\
\hline
\end{tabular}
\caption{The dataset for the example function \( z = (a \neg c) \& b \) }
\end{table}

Essentially what \textit{gen\_candidates} does is recursively add \{input variable, value\} pairs to \( P \). If all pairs in \( P \) are 100% correlated with the output pair \{\( z, 0 \)\} or \{\( z, 1 \)\} in all cycles of the simulation trace represented by \( E \), a candidate assertion
Algorithm 2 Association Miner

gens candidates(F, P, E)

1: for each {input variable, value} pair in F, fi do
2: if \( g(A_s, A_s \cup \text{assertion}(P \cup f_i \implies \{z, X\})) \geq g_{\text{min}} \) then
3: \( A_c = A_c \cup \text{assertion}(P \cup f_i \implies \{z, 0\}) \)
4: else if \( \forall e_j \in E, P \cup f_i \implies \{z, 1\} \) then
5: \( A_c = A_c \cup \text{assertion}(P \cup f_i \implies \{z, 1\}) \)
6: else
7: \( \text{mine}(F - f_i, P \cup f_i) \)
8: end if
9: end if
10: end for

is generated based on that correlation and the algorithm returns. The algorithm also returns when the coverage gain falls below the minimum coverage gain because adding more propositions to the antecedent can only decrease the coverage gain.

In line 1, \( f_i = \{a, 0\} \). The coverage gain of the assertion \( (a = 0) \implies (z = X) \) \(^1\) is calculated to 50% in line 2, which is equal to \( g_{\text{min}} \). At line 3, we can see that for the data in every cycle, \( e_j, (a = 0) \implies (z = 0) \), which means that there is a correlation between \( a = 0 \) and \( z = 0 \) which indicates a candidate assertion. The candidate assertion \( a1 : (a = 0) \implies (z = 0) \) is added to \( A_c \), the set of candidate assertions, in line 4.

Now, back at line 1, \( f_i = \{a, 1\} \). Even though the coverage gain of assertion \( (a = 1) \implies (z = X) \) is also 50%, neither the rule \( (a = 1) \implies (z = 0) \) nor \( (a = 1) \implies (z = 1) \) is true for each cycle of data, \( e_j \). This means that the conditions in lines 3 and 5 are not satisfied. The algorithm recurses at line 8 with \( P = \{\{a, 1\}\} \) and \( F = \{\{b, 0\}, \{b, 1\}, \{c, 0\}, \{c, 1\}\} \).

Now the coverage gains of assertions \( (a = 1 \land b = 0) \implies (z = X) \), \( (a = 1 \land b = 1) \implies (z = X) \), \( (a = 1 \land c = 0) \implies (z = X) \), and \( (a = 1 \land c = 1) \implies (z = X) \) are each 25% since each has two propositions in the antecedent. The minimum coverage gain is never satisfied in lines 2, and the algorithm returns.

The algorithm is continued from line 1 for the remaining \{input variable, value\} pairs resulting in the candidates \( a2 : (b = 0) \implies (z = 0) \) and \( a3 : (c = 1) \implies (z = 0) \) being added to \( A_c \). The assertions in \( A_c \) are sorted by the number of

\(^1\)X refers to a “don’t care” value since the output does not affect the input space coverage
propositions to keep the number of propositions per assertion to a minimum. In
the example, the list remains unchanged since each candidate has the same number
of propositions.

Algorithm 3 recalibrate_add

```
recalibrate_add(A_c, A_s)
1: for all a ∈ A_c do
2:     if g(A_s, A_s ∪ a) ≥ g_min then
3:         A_s = A_s ∪ a
4:     end if
5: end for
```

In the next step, recalibrate_add adds candidate assertions with coverage gain
greater than or equal to $g_{\text{min}}$ to the solution set as shown in Algorithm 3. Because
coverage gain, $g(A_s, A'_s)$ is relative to the solution set $A_s$, as soon as the solution
set changes, the coverage gain of all assertions must be recalculated based on
the new solution set. For this reason, even though all assertions in $A_c$ must have
coverage gain greater than or equal to $g_{\text{min}}$ with respect to the $A_s$ before this
function is called, the coverage gain of any assertion may decrease below $g_{\text{min}}$ as
other assertions are added to $A_s$. Because of this, $A_c$ must be recalibrated with
regards to coverage gain of each assertion before an assertion may be added to $A_s$.

In our example, $a_3$ is added to the solution set, $A_s$, since $A_s$ remains the same as
before the function was called. After adding that candidate to the solution set, the
coverage gain of next candidate, $a_2$, is recalculated based on the new $A_s$. Since
$A_s$ contains assertion $a_1$ with the antecedent ($a = 0$), it should be noted that the
truth table entries where $a = 0$ and $b = 0$ are already covered. Therefore, the
assertion $a_2$ with antecedent ($b = 0$) can only cover the truth table entries where
$a = 1$ and $b = 0$, resulting in decreased coverage gain of only 25%. By the same
logic, the coverage gain of assertion $a_3$ with antecedent ($c = 1$) is also reduced
to 25%. Since both candidates have coverage gain less than $g_{\text{min}}$, they are both
discarded.

In the final step of the first iteration, $A_c$ is cleared and the minimum coverage
gain, $g_{\text{min}}$, is reduced by half. In the example, $g_{\text{min}}$ is reduced from 50% to 25%,
which is still greater than the minimum gain threshold. The total coverage of $A_s$
is 50%, which is less than the maximum total coverage threshold, $c_{\text{threshold}}$. Since
neither threshold is passed, the algorithm continues to the second iteration.

In the second iteration, gen_candidates is performed again with the reduced
$g_{\text{min}}$. This generates the following candidate assertions which are added to $A_c$:
\[a_4 : (a = 1 \land b = 0) \implies (z = 0), a_5 : (a = 1 \land b = 1) \implies (z = 1),
\]
\[a_6 : (a = 1 \land c = 0) \implies (z = 1), a_7 : (a = 1 \land c = 1) \implies (z = 0),
\]
\[a_8 : (b = 0) \implies (z = 0), \text{ and } a_9 : (c = 1) \implies (z = 0).\] These candidate assertions are added to \( A_c \) and then sorted by the number of propositions per assertion with resulting order of \( a_8, a_9, a_4, a_5, a_6, a_7 \).

Assertion \( a_8 : (b = 0) \implies (z = 0) \) is added to \( A_s \). The coverage gain of the remaining candidate assertions is recalculated, causing \( a_4, a_6, a_7, \) and \( a_9 \) to each drop to 12.5%. This leaves only the assertion \( a_5 : (a = 1 \land b = 1) \implies (z = 1) \) that remains at 25% which is also added to \( A_s \).

It should now be noted that the expected total input space coverage of \( A_s \) has reached 100%, which is above the total coverage gain threshold. This means that the algorithm can exit, producing the following assertions: \( a_1 : (a = 0) \implies (z = 0), a_8 : (b = 0) \implies (z = 0), \) and \( a_5 : (a = 1 \land b = 1) \implies (z = 1). \)

It should be noted that this algorithm can be applied to temporal assertions much like in the decision tree algorithm [1]. For temporal assertions, the circuit is unrolled a user-specified number of times. The number of times the circuit is unrolled is known as the lookback amount. A separate set of inputs is created for each clock cycle that the circuit is unrolled where each new set of inputs represents the value of that signal relative to the current time. For example, \( a[t] \) represents signal \( a \) in the current cycle and \( a[t-1] \) represents the value of \( a \) in the previous cycle. With this data transformation, the data mining algorithm can treat the newly added signals as separate from the signals in the current time and use the same algorithm as is used on combinational signals.

### 9.3 Integration of Formal Verification

In our greedy set covering approach, we only choose candidate assertions based on coverage. Because these candidate assertions are only necessarily true with respect to a simulation trace, it is possible that a spurious assertion may be added to the solution set. Additionally, adding this spurious assertion to the solution set will prevent true assertions that cover the same input space from being added to the solution set, which adversely affect overall coverage.

Consider the example presented in Section 9.2. While \( a_5 \) and \( a_8 \) are true, \( a_1 \) is not. Even though the expected input space coverage of the solution set is 100%, the actual coverage is reduced to 75% since the \( a_1 \) is untrue. We want to be able
to check whether any assertions are true before ever adding them to the solution set.

Algorithm 4 recalibrate_add with Formal Verification

\begin{algorithm}
\caption{recalibrate_add with Formal Verification}
\begin{algorithmic}
\Function{recalibrate_add}{$A_c, A_s$}
\ForAll{$a \in A_c$}
\If{$g(A_s, A_s \cup a) \geq g_{\text{min}}$}
\If{$\text{FormalVerify}(a) == \text{True}$}
\State{$A_s = A_s \cup a$}
\EndIf
\EndIf
\EndFor
\EndFunction
\end{algorithmic}
\end{algorithm}

The solution to this problem is to integrate the formal verifier into the algorithm to validate candidate assertion choice. We modify the \textit{recalibrate_add} function to include a formal verification check as shown in Algorithm 4. After the association rule miner produces the set of candidate assertions, the formal verifier is used to prune the false candidates while retaining the true assertions. This guarantees that any assertion that is added to the solution set is going to be true. If we use this modified algorithm on our example presented in the previous subsection, we produce the assertions \((b = 0) \Rightarrow (z = 0)\), \((a = 1 \land b = 1) \Rightarrow (z = 1)\), \((b = 1 \land c = 0) \Rightarrow (z = 1)\), and \((a = 0 \land c = 1) \Rightarrow (z = 0)\) which results in 100\% input space coverage.

It should be noted that the use of formal verification does present a scalability concern. Large designs can result in a state space explosion, making verification slow or even impossible. Though formal verification does have these disadvantages, it does not mean that the coverage guided algorithm is crippled by them. To date, we have discovered only one module that was so large that it was not possible to verify (OpenSparc L2 cache). In this case, there are several options. One option is to individually verify the submodules of the limiting module. Another option is to disable formal verification of candidate assertions. The candidate assertions can then be simulated and manually checked by humans to determine if they are valid.

9.3.1 Scalability

For \(N\) input variables in a given simulation trace, searching through the space of all antecedents \(3^N\) is not scalable. In our algorithm, however, the minimum cov-
verage gain helps guide and focus our antecedent search on important assertions. By definition of coverage gain, an assertion with \( k \) propositions in its antecedent covers at most \( \frac{1}{2^k} \) of the whole input space. In general, the number of antecedents with \( k \) propositions is \( 2^k \binom{N}{k} \) and their coverage gains are at most \( \frac{1}{2^k} \). Thus, if the minimum coverage gain is \( \frac{1}{2^k} \), the maximum number of possible antecedents in the search space is \( O((2N)^k) \). For a fixed \( k \), each iteration runs in polynomial time in terms of \( N \). In our algorithm, we iteratively increase \( k \) by 1, decreasing the minimum coverage gain \( g_{\text{min}} \), until that minimum coverage gain threshold, \( g_{\text{threshold}} \), is reached. The maximum iteration of \( k \), \( k_{\text{max}} \), is defined as the iteration when \( g_{\text{threshold}} \) is reached. This helps to limit the search space. The algorithm only increases the search space if necessary. This results in the overall complexity of the algorithm being \( O((2N)^{k_{\text{max}}}) \), which is polynomial for a fixed \( g_{\text{threshold}} \). Moreover, because of the search space pruning, the actual number of antecedents searched in practice is much smaller than this theoretical bound.

Our algorithm’s scalability is only restricted by formal verification. Although formal verification technology is sensitive to state space, we find that in practice, we are able to effectively verify many modules of large designs, like the OpenSparc MMU. So far, the only module that was too large to verify is the OpenSparc L2 cache. The reason for this is that the L2 cache contains very many RAM elements, which are difficult for the formal verifier to model. In these infrequent cases, there are several options. One option is to individually verify the submodules of the limiting module. Another option is to disable formal verification of candidate assertions. The candidate assertions can then be simulated to determine if they are valid. The rest of the algorithm will proceed in the same manner.
CHAPTER 10

A COMPARISON BETWEEN THE COVERAGE GUIDED AND DECISION TREE APPROACHES IN GOLDMINE

We compare the decision tree and coverage guided methods for multiple designs. The designs used for testing include fetch\_stage and wb\_stage from Rigel [48], b10, b13, and b15 from the International Test Conference Benchmark Suite [51], b100, b101, b102, and b103 from the OpenRisc1200 CPU [49], and Transmitter, Receiver, and SPW\_FSM from the European Space Agency SpaceWire codec[50]. We have included results for the OpenSparc T2, which is an open source industrial size design. The number of inputs bits, outputs, and area can be seen in Table 10.1.

Table 10.1: Characteristics of each module used for experiments

<table>
<thead>
<tr>
<th>Module</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Area ((\mu m^2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR1200 - b100</td>
<td>122</td>
<td>9</td>
<td>788</td>
</tr>
<tr>
<td>OR1200 - b101</td>
<td>163</td>
<td>11</td>
<td>1178</td>
</tr>
<tr>
<td>OR1200 - b102</td>
<td>234</td>
<td>9</td>
<td>1223</td>
</tr>
<tr>
<td>OR1200 - b103</td>
<td>596</td>
<td>9</td>
<td>3324</td>
</tr>
<tr>
<td>Rigel - fetch_stage</td>
<td>458</td>
<td>6</td>
<td>4165</td>
</tr>
<tr>
<td>Rigel - wb_stage</td>
<td>963</td>
<td>3</td>
<td>269</td>
</tr>
<tr>
<td>Spacewire - SPW_FSM</td>
<td>46</td>
<td>7</td>
<td>342</td>
</tr>
<tr>
<td>Spacewire - Receiver</td>
<td>75</td>
<td>15</td>
<td>979</td>
</tr>
<tr>
<td>Spacewire - Transmitter</td>
<td>96</td>
<td>5</td>
<td>896</td>
</tr>
<tr>
<td>ITC - b10</td>
<td>27</td>
<td>2</td>
<td>282</td>
</tr>
<tr>
<td>ITC - b13</td>
<td>55</td>
<td>6</td>
<td>720</td>
</tr>
<tr>
<td>ITC - b15</td>
<td>534</td>
<td>4</td>
<td>9947</td>
</tr>
<tr>
<td>OpenSparc - MMU</td>
<td>3393</td>
<td>16</td>
<td>66395</td>
</tr>
</tbody>
</table>

All tests were run on an Intel Core 2 Q6600 with 4GB of RAM. Each simulation trace contains 10,000 cycles of data. The parameters are configured such that the minimum support is set to 0.1%, the minimum coverage gain threshold is 0.2%, and the coverage threshold is set to 99%.
10.1 Input Space Coverage as a Function of Iterations

In the first experiment, we show the number of iterations the algorithm takes to converge. The results for this experiment are taken from the OR1200 data cache controller module. The results are shown in Figure 10.1. It is clear that there is a logarithmic increase in input space coverage at each iteration since the minimum gain is decreased in each cycle.

![Figure 10.1: Graph showing the number of iterations taken for each design to reach 100% input space coverage using the coverage guided mining algorithm](image)

10.2 Runtime and Memory Requirements of Our Algorithm

We applied the algorithm to several outputs from the OR1200 data cache controller. For runtime, we recorded the time when the algorithm starts to the time that the algorithm exits as defined in Figure 9.1. Formal verification is enabled in this test. To record the maximum memory usage, we used the Massif tool in the Valgrind [53]. The runtime is shown in Figure 10.2 and the maximum memory usage is shown in Figure 10.3.

Though the runtime of the coverage guided mining algorithm is not as fast as the decision tree (as shown in Figure 6.6), the tool is still very scalable, even with formal verification enabled. If runtime is a concern, the formal verification can be disabled. This produces assertions much more quickly although there will
Figure 10.2: The runtime of the coverage guided mining method. The highly complex OpenSparc MMU module completes in a total of five hours.

be no feedback on the validity of the candidate assertions. Maximum memory usage is also very low. This is due to memory usage scaling with the size of the simulation trace \((\text{inputs} \times \text{number of cycles})\). If a bigger simulation trace is used, the maximum memory usage will increase linearly with the number of cycles.

10.3 Comparison of Input Space Coverage

We compare the total input space coverage of the assertions generated by the coverage guided and decision tree algorithms. The input space coverage of a primary output is defined as the sum of the input space coverage of each assertion generated with respect to that primary output. The average input space coverage is calculated as an average of the input space coverage of each primary output in the design. The results are shown in Figure 10.4. In every module, the coverage guided algorithm produces an input space coverage comparable to the decision tree method. In many cases, the coverage guided algorithm outperforms the decision tree algorithm. This indicates that in those tests, the decision tree made poor splitting decisions while the coverage guided algorithm did not suffer from the same problem.
Figure 10.3: The maximum memory usage of coverage guided mining. The memory usage of this algorithm is negligible.

10.4 Comparison of Succinctness of Assertions

Since a primary intent of the coverage guided mining algorithm is to improve assertion quality, we compare the average number of propositions in the antecedent between the two algorithms. A low number of propositions in the antecedent indicates a high input space coverage and also means that the assertion is more concise and thus easier to read by a human. The results of the test are shown in Figure 10.5. These results show that the coverage guided mining algorithm produces a lower average number of propositions in every module tested.

10.5 Comparison of Conciseness of Generated Assertions

In this experiment, the total number of assertions generated for all primary outputs of each design is recorded for each algorithm. A lower number of assertions in the final set when the input space coverage is the same indicates that the set of assertions will occupy less time and area overhead for synthesis as well as simulation. The results are in Figure 10.6. The set covering technique in the coverage guided mining algorithm outperforms the decision tree. For the b10 module in particular, the decision tree generates almost ten times more assertions than the coverage guided method even though the coverage guided method has a higher
input space coverage. It should be noted that while the coverage guided method generates more assertions for the SpaceWire modules (SPW_FSM, Transmitter, Receiver), it also achieves a significantly higher input space coverage.

10.6 Comparison of Information per Unit: Average Input Space Coverage per Assertion

It is interesting to see what the average input space coverage per assertion is. This metric is based on the total input space coverage divided by the number of assertions in the set. The results in Figure 10.7 show that the coverage guided algorithm produces higher coverage assertions than the decision tree method.

10.7 Comparison of Number of Assertions Triggered in Directed Tests

In this experiment, we produce a set of assertions for the fetch_stage and wb_stage of Rigel. We then run the directed test suite created by the designers to determine how many assertions are triggered. If an assertion is triggered, it indicates that the assertion is checking behavior that would be likely to occur in a realistic environment. The results of this test are shown in Figure 10.8.
10.8 The Final Test: Subjective Designer Rankings

For this experiment, we generated assertions for the fetch_stage and wb_stage of Rigel and then asked a designer to rank a set of assertions generated by the decision tree method and a set generated by the coverage guided mining method. The designer was not informed of the difference between the two sets. The rankings were assigned from 1 to 3 as described below.

1. Trivial assertion that the designer would not write
2. Designer would write the assertion
3. Designer would write, captures subtle design intent

The results in Figure 10.9 show that the coverage guided algorithm produces a much higher percentage of rank 3 assertions than the decision tree algorithm. Any assertions that were good, but included more propositions in the antecedent than necessary, were reduced from a rank 3 to a rank 2, which was the case for many decision tree assertions. **Overall, the designer commented that he would use the set of assertions generated by the coverage guided method over the assertions generated by the decision tree method.**
Figure 10.6: Comparison of total number of assertions generated using each algorithm. Coverage guided mining often produces a much smaller set of assertions while retaining high input space coverage.

Figure 10.7: Comparison of the average input space coverage per assertion using each algorithm. High input space coverage shows more information per assertion. In some cases, the coverage guided mining algorithm assertions have average coverage per assertion up to 20-30% more than the decision tree algorithm.
Figure 10.8: Comparison of both algorithms in terms of the percentage of assertions triggered in the Rigel directed test suite. Assertions generated by coverage guided mining are triggered at least once, meaning that they are more likely to be triggered in a realistic environment than those generated by the decision tree algorithm.

Figure 10.9: Subjective ranking by a designer of the set of assertions generated by each algorithm. All datapath assertions were considered a rank 1 by this designer because he did not consider them valuable. The coverage guided mining algorithm produces a significantly higher percentage of assertions which are at rank 3, which was the original motivation of the technique.
CHAPTER 11
RESOURCES

This chapter contains resources on using obtaining and using GoldMine.

11.1 Obtaining GoldMine

Currently, GoldMine is only available within the University of Illinois. However, the GoldMine binary will soon be available for research purposes at

http://faculty.ece.illinois.edu/shobhav/

A subversion repository is maintained on the Coordinated Science Laboratory AFS network. This repository can downloaded using the command:

svn co file:///afs/crhc.illinois.edu/project/goldmine/common/SVNROOT/goldmine

The repository truck is organized as follows. The java directory contains the source for the original Java implementation of GoldMine. This Java version is now depreciated and has been replaced with a C++ implementation. The cpp directory contains the C++ implementation of both the decision tree based src and coverage guided mining cgm versions of GoldMine. The ruby folder contains the various scripts that are used in conjunction with GoldMine. The tex directory contains the LaTeX source files for various conference papers and articles. The public directory contains the current distributable package of GoldMine.
11.2 Using the Decision Tree Based GoldMine Implementation

To quickly see GoldMine in action, there are some examples included in the problem directory of the public distribution. GoldMine takes in the problem directory as its only argument. GoldMine can be run on the OpenRisc1200 b100 module by executing:

```
./bin/goldmine problem/b100
```

GoldMine’s output (including tree printouts) can be found in

`work/(module name)/(output name)`

The generated assertions can be found in the (output name).true file while the tree printout can be found in (output name).tree.

11.2.1 The Problem Directory

GoldMine uses a problem directory which contains the input that GoldMine requires to be run. These directories consist of the following components.

Background Knowledge

For each output, there should be a .bk (background knowledge) file which contains the following parameters. Multiple .bk files are allowed.

- MODULE: Name of the module
- INPUTS: The primary inputs to use as antecedents in assertions
- REG: Registers or wires that are used as antecedents.
- OUTPUTS: The outputs to use as consequents for assertions
- LOOKBACK: Number of cycles to unroll a circuit for temporal assertions
- ASSERTION_FORMAT: The format for the assertions. Supports SVA.
• FV: The formal verifier to use. Only supports Cadence IFV.

• RTL: The location of the design file.

Example Behavior

In the problem directory, there should be a directory named “csv” which contains the example behavior of the design in comma-separated value format. Each column represents a signal and each row represents one cycle at the positive clock edge.

Using Ruby to Automatically Create the Problem Directory

For automatic creation of the problem directory, both Ruby [54] and Synopsys VCS [55] are required. In the public distribution, run

```
./bin/create_problem.sh (verilog module)
```

Where (verilog module) represents the name of a module in the verilog folder with all necessary files (submodules and library files) included using the “include” directive. This script creates a random testbench for the specified module which applies a random value to each input signal, then simulates the design with the created testbench producing a VCD dump, converts that VCD file into CSV format, and then produces a background knowledge (.bk) file for each primary output. To create bk files that support formal verification and counterexample refinement, change $enable_formal_verification = true in ./lib/full_signal_generator.rb.

11.3 Using the Coverage Guided Mining GoldMine Implementation

The coverage guided algorithm is based on the decision tree implementation, but some conversion has to be done to prepare the problem directory for the coverage guided algorithm. In the cgm subfolder of ruby, there is a script named do.rb. To use this script, create a text file containing the module(s) that you would like to run GoldMine on (located in the verilog directory) separated by line feeds. To
run this program, use

`./do.rb (text file containing module names)`

It should be noted that before the program is run the first time, one will need to edit `quick_parse.rb` and change HOME to the folder containing the verilog directory and edit `do.rb` and change `$my_dir` to the directory containing the verilog and problem directories (likely the same as HOME).

This script automatically instantiates the coverage guided algorithm. The output of the GoldMine is piped to (module name).out.
CHAPTER 12

CONCLUSIONS

While the work done on GoldMine has been extensive, it only touches the surface of what can be done. Because of its speed, it can be deployed where static analysis fails. Because of its power, it is able to generate complex assertions where competing tools can only generate trivial assertions. GoldMine also has the versatility to be extended in a variety of ways and can serve many different uses for the verification of hardware designs. GoldMine has a very good chance of being developed into a tool that designers and verification engineers cannot live without by maximizing productivity and minimizing human resources and cost in the assertion generation process.
REFERENCES


