Hole Analysis for Functional Coverage Data

Oded Lachish odedl@il.ibm.com
Eitan Marcus marcus@il.ibm.com
Shmuel Ur ur@il.ibm.com
Avi Ziv aziv@il.ibm.com

IBM Research Laboratory in Haifa
Haifa University
Haifa, 31905
Israel

ABSTRACT

One of the main goals of coverage tools is to provide the user with informative presentation of coverage information. Specifically, information on large, cohesive sets of uncovered tasks with common properties is very useful. This paper describes methods for discovering and reporting large uncovered spaces (*holes*) for cross-product functional coverage models. Hole analysis is a presentation method for coverage data that is both succinct and informative. Using case studies, we show how hole analysis was used to detect large uncovered spaces and improve the quality of verification.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Verification*; D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Verification, Measurement, Algorithms, Experimentation

Keywords

Functional verification, Coverage analysis

1. INTRODUCTION

Functional verification comprises a large portion of the effort in designing a processor [5]. The investment in expert time and computer resources is huge, as is the cost of delivering faulty products [3]. In current industrial practice, most of the verification is done by generating a massive amount of tests using random test generators [1, 2]. The use of advanced random test generators can increase the quality of generated tests, but it cannot detect cases in which some areas of the design are not tested, while others are tested repeatedly.

The main technique for checking and showing that the testing has been thorough is called test coverage analysis [9]. Simply stated, the idea is to create, in a systematic fashion, a large and comprehensive list of tasks and check that each task was covered in the testing phase. Coverage can help monitor the quality of testing and direct the test generators to create tests that cover areas that have not been adequately tested.

Coverage tools collect coverage data and then aid in the analysis of that data. Our experience shows that informative presentation of the coverage information is as important as efficient collection of that information.

Most coverage tools provide two types of reports — status reports and progress reports. Progress reports show how coverage progresses over time. These reports are useful in detecting when the testing is "running out of steam" and predicting its potential coverage value [7]. Status reports show which tasks are covered and which are not, and how many times each task was covered. This information helps the verification team direct the testing to untested, or weakly tested, areas in the design [6].

This paper describes *hole analysis*, a method for discovering and reporting large uncovered spaces for cross-product functional coverage models [6]. In a cross-product functional coverage model, the list of coverage tasks comprises all possible combinations of values for a given set of attributes. A simple example of a coverage model is all the possible combinations of requests and responses sent to a memory subsystem. In this case, a coverage task is the pair <`request, response`>, where `request` is any of the possible requests that can be sent to the memory (e.g., `memory read, memory write, I/O read, I/O write`) and `response` is one of the possible responses (e.g., `ack, nack, retry, reject`).

Hole analysis groups together sets of uncovered tasks that share some common properties, thus allowing the coverage tool to provide shorter and more meaningful coverage reports to the user. For example, it is much more informative to report that a `reject` response never occurred, than it is to include all possible cases of a request with a `reject` response in the list of uncovered tasks. This grouping of uncovered tasks into meaningful sets, in addition to shortening the list of uncovered tasks reported to the user, provides vital information on the cause of the hole. For example, investigation of the hole described above may lead to the simple conclusion that requests are never rejected because they do not arrive fast enough. In this case, increasing the rate of requests to the memory subsystem may be all that is needed to generate rejects.

Our hole analysis technique is based on finding uncovered tasks that have common values in some of the attributes, and finding some common denominator in the attribute values that distinguish them. The simplest case is when all the values of a specific attribute are not covered when each of the other attributes has a particular value. The hole above is an example of such a case. None of the values of the attribute `request` are covered when the `response` attribute equals `reject`. Another possibility is to find a meaningful subset of the values of a certain attribute, for example, I/O requests (including I/O read and I/O write). This type of hole detection leads to the discovery of rectangular holes that are parallel to the attribute axis. This analysis is similar to classification methods used in AI, such as ID3 [11].

Hole analysis has been implemented in a number of coverage tools — Comet [6], its successor Meteor, and FoCus [13] — developed at IBM, and has become an integral part of the verification process for users of these tools. In the paper, we provide two examples of real coverage models and show how hole analysis can provide meaningful reports to the user. In each case, there was a significant improvement in the testing quality. The first example is a coverage model for results of floating point instructions. The second example is a micro-architectural coverage model for register interdependency in the pipelines of a super-scalar PowerPC processor [10].

The rest of the paper is organized as follows. In Section 2, we explain cross-product coverage models and how they should be used during verification. In Section 3, we describe our hole analysis techniques using the floating point coverage model as an example. Section 4 illustrates the usefulness of hole analysis with the interdependency coverage model example. Finally, Section 5 concludes the paper.

2. FUNCTIONAL COVERAGE MODELS

Functional coverage focuses on the functionality of an application. It is used to check that all important aspects of the functionality have been tested. Unlike code coverage models that are program based, functional coverage is design and implementation specific [12].

When using functional coverage, coverage models containing a large set of verification (coverage) tasks are systematically defined by the user. These are used to measure, monitor, and redirect the testing process. Holes in the coverage space expose gaps in the testing, and new tests are created to “plug” these gaps. This is an iterative process in which the quality of the testing is continuously measured, and improvements to the testing are suggested when analyzing the results of the measurement.

Functional coverage models are composed of four components. The main component is a semantic description (story) of the model. The second component is a list of the attributes mentioned in the story. The third component is a set of all the possible values for each attribute. The last component is a list of restrictions on the legal combinations in the cross-product of attribute values.

Table 1 shows the attributes and their values for a functional coverage model taken from the floating point domain. The model consists of four attributes – Instr, Result, Round Mode, and Round Occur – each with the possible values shown. The semantic description of the functional coverage model is: test that all *instructions* produce all possible *target results* in the various *rounding modes* supported by the processor both when *rounding* did and did not occur.

Each attribute may be partitioned into one or more disjoint sets of semantically similar values. This provides a convenient way for users to conceptualize their model and for analysis tools to report on coverage data. In Table 2, which gives several partitioning examples, the instruction attribute is partitioned into arithmetic instructions and non-arithmetic instructions, or according to the number of input operands. The result attribute is partitioned according to the sign of the result, and so on.

Restrictions for this model are shown in Table 3. They describe combinations of attribute values that should never occur, for example, the result of an `fabs` instruction cannot be negative. The coverage model illustrated here has been used to check the coverage of tests generated for the floating point unit of several PowerPC processors.

Attr	Description	Values
Instr	Opcode of the instruction	fadd, fadds, fsub, fmul, fdiv, fmadd, fmsub, fres, frsqrte, fabs, fneg, fsel, ...
Result	Type of result	± 0, $\pm$MinDeNorm, $\pm$DeNorm, $\pm$MaxDeNorm, $\pm$MinNorm, $\pm$Norm, $\pm$MaxNorm, $\pm\infty$, SNaN, QNaN, none
Round Mode	Rounding mode	toward nearest, toward 0, toward $+\infty$, toward $-\infty$
Round Occur	Did rounding occur?	True, False

Table 1: Attributes of the floating-point model

Attribute	Partition	Values
Instr	Arith	fadd, fsub, fmul, fdiv, ...
	Non Arith	fabs, fneg, fmr, fsel, ...
Instr	1 Operand	fabs, fneg, fres, frsqrte, ...
	2 Operands	fadd, fsub, fmul, fdiv, ...
	3 Operands	fmadd, fmsub, fsel, ...
Result	Positive	$+0, \ldots, +$Norm$, \ldots, +\infty$
	Negative	$-0, \ldots, -$Norm$, \ldots, -\infty$
	Not A Number	SNaN, QNaN

Table 2: Attribute partitions (partial list)

Functional coverage models come in many flavors. Models may cover the inputs and outputs of an application (black box), or may look at internal state (white box). Functional coverage models may be either snapshot models, which check the state of the application at a particular point in time, or temporal models, whose tasks correspond to the application’s state as it evolves.

2.1 Creating a Functional Coverage Model

The first step in the functional coverage process is the design of the coverage model. This is typically done by isolating “interesting” attributes of the application and building a story around them. For coverage to impact positively on the testing process, models should cover areas that are considered risky or error prone. The size of the model (number of legal tasks) should be chosen in accordance with testing resources. The model size should not be so large as to make it difficult to cover or impractical to analyze. Constructing a good coverage model requires functional coverage skills and domain specific knowledge.

The part of the model most difficult to define correctly is the model restrictions. Often a deep understanding of the specification, design, and implementation is needed to create correct restrictions.

Name	Description
Arith-SNaN	Instr $\in$ Arith $\Rightarrow$ Result $\neq$ SNaN
NonArith-Round	Instr $\notin$ Arith $\Rightarrow$ Round Occur $= 0$
Abs-Nabs	Instr $=$ fabs $\Rightarrow$ Result $\notin$ Negative AND Instr $=$ fnabs $\Rightarrow$ Result $\notin$ Positive

Table 3: Restrictions for the floating-point model

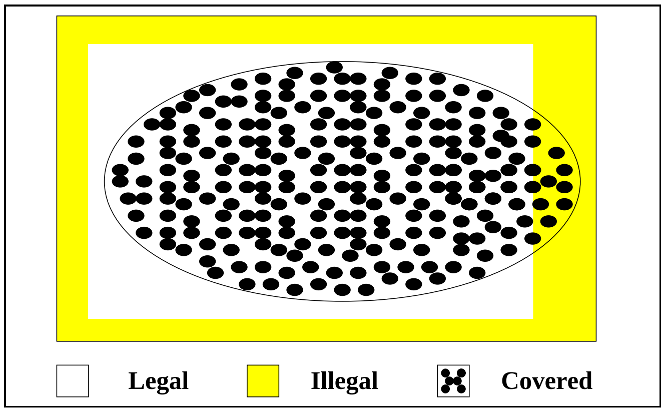


Figure 1: Restrictions are self-correcting

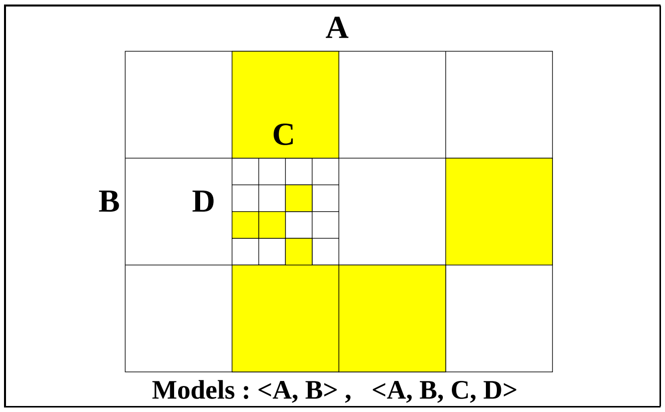
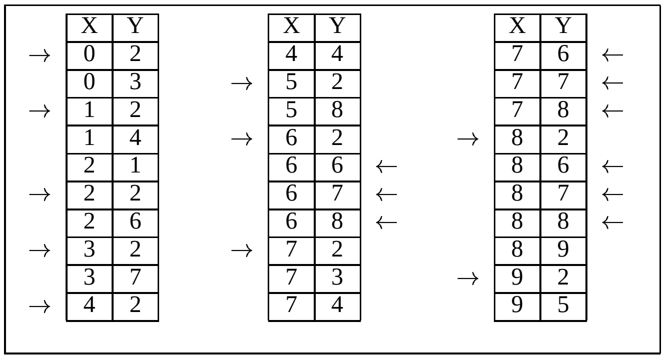


Figure 2: Restrictions are created hierarchically



	X	Y
→	0	2
	0	3
→	1	2
	1	4
	2	1
→	2	2
	2	6
→	3	2
	3	7
→	4	2

	X	Y	
	4	4	
→	5	2	
	5	8	
→	6	2	
	6	6	←
	6	7	←
	6	8	←
→	7	2	
	7	3	
	7	4	

	X	Y	
	7	6	←
	7	7	←
	7	8	←
→	8	2	
	8	6	←
	8	7	←
	8	8	←
	8	9	
→	9	2	
	9	5	

Figure 3: List of uncovered tasks

While creating restrictions, designers may need to think of how the system operates in ways not previously considered. This process turns out to be useful in itself, as it increases their understanding of the application.

2.2 Refining the Functional Coverage Model

Every task in the cross-product of a model's attribute values is either legal or illegal, and has either been covered or remains uncovered, as shown in Figure 1. Uncovered legal tasks are holes in the coverage model. Sometimes after careful analysis, it is determined that a hole is in fact an illegal task, and additional restrictions need to be introduced into the model to eliminate the hole. Conversely, illegal tasks that have been covered are either actual bugs in the application, or reflect a coverage model that is over-restrictive. Thus, the definition of a coverage model is dynamic, with tasks occasionally drifting between being legal and illegal as the application is better understood and the model is refined. In practice, actual bugs in the application are rare, and covered illegal tasks usually reflect problems in the model definition. Holes, in contrast, are more common, and are as likely to result from inadequate testing as from missing restrictions in the model.

The size of a model's cross-product is determined by the number of its attributes and the number of values for each attribute. Consequently, the number of tasks in a functional coverage model can be quite large making the model difficult to cover and analyze. One way to reduce the size of a model is to define restrictions that eliminate illegal tasks.

An alternative approach, suggested by Figure 2, is to start with a coverage model that has fewer attributes. In the floating point domain, for example, an initial model may consider just the two attributes, Instr and Result. Restrictions for such models are simpler both to write and refine. Once these smaller coverage models have been satisfactorily defined, we can apply their restrictions to more comprehensive models. This hierarchical approach to modeling makes it quicker to define restrictions and to adjust the coverage model to the simulation resources.

3. HOLE ANALYSIS

Once the functional coverage model has been defined and coverage of tests has been measured, the next step in the verification process is to analyze the coverage data. A critical aspect of the coverage analysis is to determine those areas of the application that have not been adequately tested. While most coverage tools are able to report on individual uncovered tasks, it remains the function of hole analysis to discover and concisely present information about meaningful holes. For us, holes are considered meaningful either because they represent a relatively large number of uncovered tasks, or because they correspond to a set of tasks that have some semantic coherency. In this section, we present a number of techniques for discovering such quantitatively and conceptually meaningful holes.

Consider a model with just two integer attributes, X and Y, each capable of taking on values between 0 and 9. Figure 3 shows the individual uncovered tasks. This is typical of the results that can be obtained from most tools. Not immediately obvious are the two meaningful holes that exist in the coverage data, shown with arrows in the figure. One hole occurs whenever Y equals 2, and a second hole exists when both attributes have values 6, 7, or 8. This is, however, readily seen in Figure 4. Such graphs, which can be obtained using our tools, provide a convenient way to present holes that are clustered along ordered values in models with a small number of attributes. The challenge for hole analysis is to discover more complex holes in arbitrarily large models, and to present these holes in such a way that their root cause may be more easily discerned.

3.1 Aggregated Holes

Meaningful holes can be automatically discovered between uncovered tasks that are quantitatively similar. A simple metric for similarity is the Hamming distance between two holes. This corresponds to the number of attributes on which the two differ. The distance will be one for holes that differ in only one attribute, two for holes that differ in two attributes, and so on, up to the number of attributes in the coverage space. We aggregate together any two holes whose Hamming distance is one. Thus, the two uncovered tasks <0,2> and <0,3> from Figure 3 can be aggregated into the single hole <0,{2,3}>.

Hamming distances can be also computed on aggregated holes.

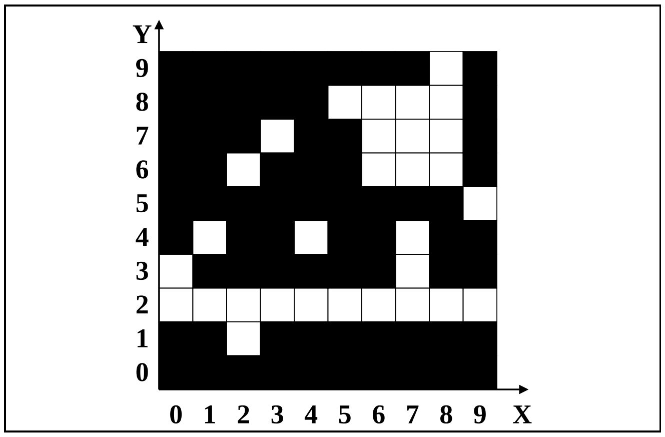


Figure 4: Holes visualization example

$$\begin{array}{l} <0,2> \\ <0,3> \\ <7,2> \\ <7,3> \\ <7,4> \end{array} \Longrightarrow \begin{array}{l} <\{0,7\},2> \\ <\{0,7\},3> \\ <0,\{2,3\}> \\ <7,\{2,3,4\}> \end{array} \Longrightarrow \begin{array}{l} <\{0,7\},\{2,3\}> \\ \\ <7,\{2,3,4\}> \end{array}$$

Figure 5: Aggregated holes calculation

Again, the distance between any two holes is equal to the number of differing attributes, but now the comparison is done for aggregated sets as well as atomic values. The process can be applied iteratively until no more new aggregated holes are discovered. Figure 5 shows how the five tasks <0,2>, <0,3>, <7,2>, <7,3>, and <7,4> can be aggregated together until only the holes <{0,7},{2,3}> and <7,{2,3,4}> remain. This technique is similar to Karnaugh binary mapping [8] and is useful for much the same reasons.

3.2 Partitioned Holes

A second technique for performing hole analysis is to group together holes that have been defined to be conceptually similar. Grouping is done according to semantic partitioning of the attribute values provided by the user when defining the coverage model. Attribute values that are partitioned together are frequently tested together, and holes in one value may occur for the whole group.

Grouping is used to describe larger holes and to provide more focus on the meaning of the holes. For example, the second hole in Table 4 is a grouping of many smaller holes. This grouping allows us to detect the common denominator (and perhaps the common cause) of all the sub-holes, that none of the instructions in the SquareRoots group produced denormalized results. After some analysis, it was determined that square root instructions in fact should not produce such results, and the hole was converted into a restriction in the coverage model.

Both aggregated and partitioned holes look to combine individual, uncovered tasks into larger holes that in turn can suggest reasons for the observed gaps in coverage. In both cases, similarity between tasks is used to determine which tasks should be combined. The difference between the two is that the similarity for partitioned holes is semantic and based upon groupings provided by the user, whereas for aggregated holes it is quantitative and discovered automatically by the tool.

3.3 Projected Holes

Consider the first hole described in Table 4, which has the specific value `fadd` for the instruction attribute and wild cards in all the other attributes. This hole denotes that all the tasks with `fadd` are not covered and it describes 152 tasks. We call this a projected hole. The dimension of a projected hole is the number of attributes in it that do not have specific values.

Any coverage model of n attributes can be viewed as a set of tasks or points in an n-dimensional space. Viewed as such, a projected hole of dimension 1 contains tasks that lie on a line in hyperspace, a projected hole of dimension 2 describes tasks that lie on a plane, and so on. A projected hole of a higher dimension subsumes all holes of a lower dimension that it contains. For example, the hole $p = < *, *, x_3 \ldots, x_n >$ of dimension 2 contains all the tasks described by the subspace $q = < x_1, *, x_3 \ldots, x_n >$. In such a case, we say that p is the *ancestor* of q, and that q is the *descendant* of p. In general, the higher the dimension, the larger and, therefore, more meaningful the hole it describes. Since a hole of higher dimension is both more informative and more concise than the holes it subsumes, it becomes unnecessary, and in fact redundant, to report on the latter.

One goal of hole analysis is to find projected holes of the highest dimensionality. The algorithm shown in Figure 6 first iterates over all covered events and marks them and all their ancestors. Next, it iterates over all potential holes, starting with those of the highest dimensionality ($< *, *, \ldots, * >$). Any subspace that is unmarked is reported as a significant hole and its descendants are recursively marked.

```
Data   : set $T$ of covered tasks
Result : list of projected holes

// mark phase
foreach $t \in T$ do
    if $t$ is not marked then
        mark $t$
        recursively mark all ancestors of $t$
    end
end

// report phase
for $i = k$ downto 0 do
    foreach subspace $v$ with dimension $i$ do
        if $v$ is not marked then
            report $v$ as hole
            recursively mark all descendants of $v$
        end
    end
end
```

Figure 6: Projected holes algorithm

A projected hole is considered *lightly covered* if the percentage of tasks subsumed that have been covered is less than some given threshold. It is fairly easy to adapt the projected holes algorithm to report on lightly covered holes.

Another effective approach is to use a classification algorithm, such as ID3 [11], to find projected holes. Most classification algorithms use some kind of entropy measurement on the data to guide the construction of the decision tree used in the classification. In our case, the target function is the coverage data and ID3 is used to find a large cluster of uncovered tasks.

Table 4 shows some of the holes found in the floating point model when the model was implemented at one of the design laboratories as part of their floating point test plan. The first hole indicates that none of the tasks related to the `fadd` instruction were covered. The hole was caused by a bug in the specification to the test

Instr	Result	Round Mode	Round Occur	Hole Size
fadd	$*$	$*$	$*$	152
Square Roots	+DeNorm +MaxDeNorm +MinDeNorm	$*$	$*$	72
Estimates	$*$	$*$	True	144
$*$	$\pm$MaxDeNorm	$*$	True	263

Table 4: Coverage holes report

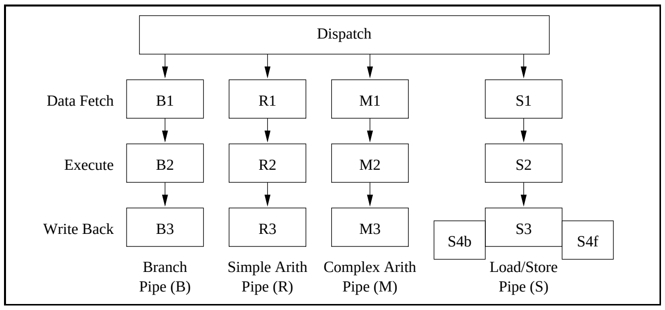


Figure 7: NorthStar pipeline structure

generator that omitted the `fadd` instruction from the list of instructions that needed to be generated. The next two holes show two groups of tasks that cannot be achieved. The first points to square root instructions (fsqrt, fsqrts, frsqrte) with small results, while the second points to estimation instructions (fres, frsqrte) with inexact results. Both holes are covered by restrictions that eluded the developer of the model. After their detection, the two holes were converted to restrictions. The last hole in the list corresponds to a group of tasks with specific results and rounding that was hard to generate, and was therefore only lightly covered (less than 10 percent). This hole was filled by enhancing the capabilities of the test generator.

A significant feature of our coverage tools is that the language they use to report holes is equivalent to the one used by the user for defining restrictions. The hole <Arith,SNaN,*,*>, for example, is expressed by the restriction $Instr \in Arith \Rightarrow Result \neq SNaN$. This means that holes can be easily translated into restrictions if necessary. In many cases, the translation can be automated.

4. INTERDEPENDENCY MODEL

In this section, we describe the coverage model and hole analysis that was done for checking register interdependency in the pipelines of the NorthStar processor. NorthStar is a member of the multi-threaded, super-scalar PowerPC family of processors developed by IBM. As depicted in Figure 7, the processor consists of four execution pipelines: branch (BPipe), simple arithmetic (RPipe), complex arithmetic (MPipe), and load/store (SPipe). Single cycle, fixed point arithmetic instructions can be handled either by RPipe or MPipe. All other arithmetic instructions, including floating point instructions, are handled by MPipe. Each pipeline consists of a data fetch, execution, and write back stage. The load/store pipe has two additional stages that are used for thread switching. The NorthStar processor does not allow out-of-order execution, so that all instructions progress through their respective pipelines in the same order in which they appear in the test.

A coverage model was defined to check for register interdependency between two instructions that appear together in the pipeline.

Attr	Descr	Values
I_1	Instr 1 type	FP-ld-simple, FP-st-simple, fix-ld-simple, fix-st-simple, fix-ld-complex, fix-st-complex, FP-M, fix-M, fix-R, branch, cache, condition, sync-call-return
P_1	Instr 1 pipe	BPipe, RPipe, MPipe, SPipe
S_1	Instr 1 stage	1-5
I_2	Instr 2 type	same as I_1
P_2	Instr 2 pipe	same as P_1
S_2	Instr 2 stage	same as S_1
Dep	Dependency	RR, RW, WR, WW, None

Table 5: Attributes of the interdependency model

$I_1(I_2)$ is a simple fixed point instruction
    $\Rightarrow P_1(P_2)$ is RPipe or MPipe
$I_1(I_2)$ is a complex fixed point or an FP instruction
    $\Rightarrow P_1(P_2) =$ MPipe
$I_1(I_2)$ is a load/store instruction $\Rightarrow P_1(P_2) =$ SPipe
$I_1(I_2)$ is a branch instruction $\Rightarrow P_1(P_2) =$ BPipe
$P_1 = P_2 \Rightarrow S_1 \neq S_2$
$P_1(P_2) \neq SPipe \Rightarrow S_1(S_2) <= 3$
$I_1(I_2)$ is a branch instruction $\Rightarrow Dep =$ None
$I_1(I_2)$ is a cache or complex store instruction
    $\Rightarrow Dep \neq$ WR (RW)
$S_1 = 2$ and I_1 is not a complex load instruction
    $\Rightarrow Dep \neq$ WR
$S_2 = 2$ and I_2 is not a store or mtspr instruction
    $\Rightarrow Dep \neq$ WR
$S_1 >= S_2$

Table 6: Restrictions for the interdependency model

The attributes of the model are shown in Table 5. The model consists of seven attributes: the type, pipeline, and stage of one instruction; the type, pipeline, and stage of a second instruction; and the register interdependency that may exist between the two. We say that two instructions in a pipeline have a register interdependency if they both access the same register during execution. The model distinguishes between read/read (RR), read/write (RW), write/read (WR), and write/write (WW) dependencies depending if the first/second instruction was reading or writing to the register. The model story looks for coverage of all possible interdependencies (including none) between all combinations of instructions in any possible pipeline stage.

An example of a coverage task in the model is the task <fix-R, RPipe, 2, fix-ld-simple, SPipe, 1, RR>, which corresponds to a simple fixed point instruction in stage 2 of the RPipe which has a read/read dependency with a fix-ld-simple instruction in stage 1 of the SPipe.

Restrictions in the model arise for various reasons: pipelines are dedicated to handle only certain types of instruction, limitations that exist on the pipeline architecture, certain instructions never read or write to registers, instructions are restricted in what dependencies they can have at certain stages of execution, and out-of-order execution is not supported by the processor. A partial list of restrictions for the interdependency model is shown in Table 6.

After performing hole analysis on the interdependency model,

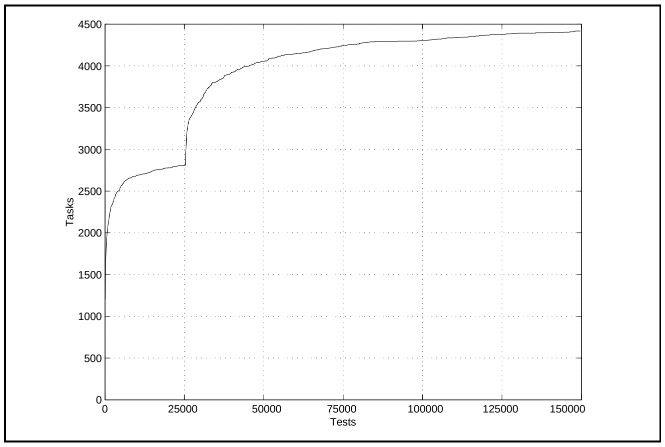


Figure 8: Interdependency coverage progress

two holes of significance were discovered. The first was the result of a bug in the test generator that was used to produce the test inputs. The test generator did not consider the fact that a store and update instruction also writes to a register, and therefore, such events were generated with low probability. The bug was fixed with the help of the hole that was discovered.

The second hole was due to the fact that some tasks need a rare combination of events to occur before they can be observed. When not enough thread switches are created, for example, the logic that checks dependency between the foreground and the background threads, about a third of the total pipeline logic, is not sufficiently tested. The solution was to reduce the timeout between threads and increase the mix of instructions which cause thread switches. While the test generator was capable of generating such events, the likelihood of doing so unintentionally was low. To improve coverage, the test generator was instructed by the verification engineer to concentrate on these corner cases. Figure 8 shows that a significant improvement in coverage was observed after about 25,000 tests were measured when these changes were made.

5. CONCLUSIONS

In this paper, we described hole analysis, a method that provides concise information about uncovered and lightly covered areas in cross-product functional coverage models. We presented a number of classification techniques used in holes analysis. Using two case studies, we showed how hole analysis can assist users of coverage tools to improve the quality of their verification.

We illustrated that effective coverage models are built hierarchically and through iterative refinement. We also showed that holes and restrictions can be similarly expressed, leading to two main advantages. First, the learning curve for writing functional coverage models and for using hole analysis is reduced. Also, the translation of holes into missing restrictions can be automated in many cases.

We are currently investigating several other techniques for increasing the analysis capabilities of our coverage tools. Items on which we are working include using machine learning and datamining [4] techniques to detect coverage holes that are not rectangular or not parallel to the attribute axis (e.g., hole such as $x+y<7$). We are also looking at automatic and semi-automatic techniques for discovering relationships between the input to the simulation (e.g., test cases, test cases specification) and the coverage data. This type of analysis can identify which inputs contribute more towards coverage and should therefore be utilized more often.

6. REFERENCES

[1] A. Aharon, D. Goodman, M. Levinger, Y Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. Test program generation for functional verification of PowerPC processors in IBM. In *Proceedings of the 32nd Design Automation Conference*, pages 279–285, June 1995.

[2] A.M. Ahi, G.D. Burroughs, A.B. Gore, S.W. LaMar, C.R. Linand, and A.L. Wieman. Design verification of the HP9000 series 700 PA-RISC workstations. *Hewlett-Packard Journal*, 14(8), August 1992.

[3] B. Beizer. The Pentium bug, an industry watershed. *Testing Techniques Newsletter, On-Line Edition*, September 1995.

[4] P. Cabena, P. Hadjinian, R. Stadler, J. Verhees, and A. Zanasi. *Discovering Data Mining, from Concept to Implementation*. Prentice Hall, 1997.

[5] L. Fournier, Y. Arbetman, and M. Levinger. Functional verification methodology for microprocessors using the Genesys test-program generator. In *Proceedings of the 1999 Design, Automation and Test in Europe Conference (DATE)*, pages 434–441, March 1999.

[6] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. User defined coverage - a tool supported methodology for design verification. In *Proceedings of the 35th Design Automation Conference*, pages 158–165, June 1998.

[7] A. Hajjar, T. Chen, I. Munn, A. Andrews, and M. Bjorkman. High quality behavioral verification using statistical stopping criteria. In *Proceedings of the 2001 Design, Automation and Test in Europe Conference (DATE)*, pages 411–418, March 2001.

[8] M. Karnaugh. The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers*, 72(9):593–599, November 1953.

[9] B. Marick. *The Craft of Software Testing, Subsystem testing Including Object-Based and Object-Oriented Testing*. Prentice-Hall, 1985.

[10] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture*. Morgan Kaufmann, 1994.

[11] J. R. Quinlan. Inductions of decision trees. *Machine Learning*, 1:81–106, 1986.

[12] S. Ur and A. Ziv. Off-the-shelf vs. custom made coverage models, which is the one for you? In *proceedings of STAR98: the 7th international conference on software testing analysis and review*, May 1998.

[13] Focus. http://www.alphaworks.ibm.com.