Processor Verification with *hwBugHunt*

Sangeetha Sudhakrishnan, Liying Su, and Jose Renau
Dept. of Computer Engineering, University of California Santa Cruz
http://masc.soe.ucsc.edu

ABSTRACT

Functional verification of modern processors and complex ASIC designs is a challenging task. Verification is frequently more complex than the design itself. The time required to find the exact source of an error in complex designs represents a significant part of the verification process. Most test suites only report the existence of a bug, but are unable to ultimately discover the line of HDL code where the bug is located. Designing new tools and techniques that reduce these overheads is very important to keep the verification costs under control.

This paper proposes a novel HDL error-discovery tool (*hwBugHunt*) to pinpoint the line of code where a bug is located. *hwBugHunt* works by instrumenting Verilog code and gathering statistics during the execution of various testbenches. The proposed infrastructure is tested on an Alpha-like 21264 Verilog implementation. Our evaluation shows that *hwBugHunt* pinpoints 62% of the bugs introduced on IVM, and it does so with a low overhead and high accuracy.

1 Introduction

Current development costs for top-of-the-line processors are staggering, and are doubling every four years [7]. This cost is increasingly attributable to the growing difficulty to design and verify circuit designs – called the Design and Verification Gap [3]. As a result, according to the ITRS 2002 update [3], ”the increasing level of risk that design cost and design quality present to the continuation of the semiconductor industry” is of serious concern. Verification is widely recognized as the major bottleneck.

The bulk of the time spent on processor front-end design is verification [11] – over 50% in many cases. This is also true for top of the line ASIC designs, where the design verification requirements can be broken down roughly into 42% for testbench development and 58% for debugging [10]. Clearly, techniques that reduce testbench development and debug time can significantly increase the productivity of the design teams. This leads to reduce the time to market and overall design cost.

The work of this paper focusses on front-end verification which strives to correlate the implemented design with the original design specifications.

Designs are validated with testbench suites. However, the testbench-based method suffers from the limitation that while a bug is detected, the offending line of HDL is not. Thus, it may still take a designer substantial effort to localize the specific line of code that caused the problem (58% of the time [10]). This can result in a tedious debug process where a designer may have to search through thousands of Verilog/VHDL lines of code.

To ameliorate the verification process, *hwBugHunt* is proposed – an infrastructure to automatically pinpoint lines of code where HDL functional bugs are located.

To accomplish this, *hwBugHunt* builds on top of the coverage principle. The program instruments the original HDL (Verilog) code to gather coverage metrics, and creates multiple coverage checkpoints as the testbench tests different inputs. By analyzing the differences between coverage checkpoints, *hwBugHunt* is able to pinpoint the line in source code where the bug is located.

hwBugHunt works with any type of testbench, but this paper focuses upon the most challenging case: the integration testbench on a modern out-of-order processor. This testbench verifies that retiring instructions generate the same architectural state. Once a bug is flagged, the proposed infrastructure automatically finds a list of possible bug sources in the HDL code.

There has been much work done in locating the source for errors by looking at the erroneous netlist [2]. The output of the method is a set of locations in the netlist that is the cause for the error. In the work, [6] the authors compare several SAT-based and simulation based methods of locating the source of an error. Most of the work that has been done identify the erroneous gate that causes an error or fault. Our work differs from the above mentioned because, we pinpoint the line of HDL code that causes the testbench to fail and not the erroneous line at the gate level.

In the recent years, there has been work that focusses on the problem of finding bugs in traditional programming languages such as C/C++/Java. The most relevant work includes DIDUCE [9] and Daikon [5]. These two tools find bugs by extracting code invariants during a training phase and enforcing those invariants during the testing phase. Although detecting invariants has a great potential for HDL languages like Verilog, *hwBugHunt* gains its advantage by tracking multiple coverage metrics instead of variable values. Further, *hwBugHunt* has unified the training/testing phase and our work is based on coverage analysis instead of invariant deduction. To our knowledge, *hwBugHunt* is the first tool to use this method as a way to find bugs in HDLs.

IODINE [8] is a tool designed by the same author as DIDUCE to work with processor synthesis. IODINE is capable of extracting design properties using dynamic analysis. This information is used to extend the test suite and detect

possible problems in the design. This is quite different from *hwBugHunt* which searches for specific lines of code that are the source of error.

hwBugHunt is complementary to Assertion Based Verfication (ABV), as it adds value to the debugging phase by localising the source of the bug. ABV can be used for finding and localising the bugs. To employ ABV, there is need to modify the code by typing in the assertions at appropriate locations in the source code. With *hwBugHunt* the focus is on localising the bug as we rely on the testbench to detect it and there is no need to modify the source code.

To test the proposed infrastructure, an HDL design with injected errors is required. For that purpose, several bugs are introduced into the IVM [12] HDL processor description. IVM is a synthesizable Verilog implementation of an Alpha-21264. It is a 4 issue, out-of-order, superscalar processor capable of executing a subset of SPECint. To have a fair evaluation, a representative sample of bugs found in a typical design needs to be inserted in IVM. This requires an assortment of bugs of different types. To accomplish this, the bug classification reported by Al-Asaad [1] is followed. For each non-language specific class of bugs, three different bugs are created.

Bug-finding-tools need to be evaluated by their accuracy to find bugs, the percentage of false positives (correct statements incorrectly flagged as errors) and execution overhead. For example, a tool with many false positives is probably useless even if it is able to find every bug. *hwBugHunt* tries to minimize the number of false positives as it is generally better to ignore a given bug rather than reduce the entire program efficacy by returning many false positives. Further, the simulation overhead must be minimized. Given the time-critical nature of design verification, this coverage analysis must be fast.

For the IVM processor analyzed, *hwBugHunt* correctly pinpoints 62% of the bugs. On average, for the correctly located bugs, there are just 0.6 false positives. *hwBugHunt* keeps the overhead to a minimum even for the non correctly located bugs (38%) where the average number of false positives is just 3.1 (*i.e.*,) *hwBugHunt* reports on an average just 3.1 lines of HDL code as being the source of the bug when in reality they are not. The result is a tool that significantly reduces debug time on modern processors.

The rest of the paper is organized as follows. Section 2 describes the proposed *hwBugHunt* infrastructure. Section 3 describes the evaluation setup. Section 4 evaluates the accuracy and overheads of the models. Section 5 presents conclusions and future work.

2 hwBugHunt

Figure 1 shows the main steps followed by the functional verification teams. Whenever a bug is detected, engineers need to locate it. To do so, engineers, typically, generate an execution trace history or Value Change Dump (VCD) to find a given bug. Once all the outstanding bugs are solved, engineers have to decide whether the test suite is sufficient or whether they still need to add additional tests or expand the inputs set. A test suite is considered "good enough" when a specific set of HDL coverage targets are met, and when no outstanding bugs remain.

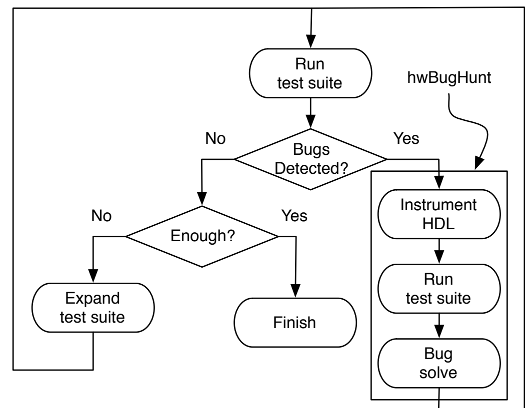

Figure 1: Verification work flow.

hwBugHunt slightly modifies the traditional verification work flow. As Figure 1 shows, three new steps are required once a bug is detected. First, HDL code is instrumented (Section 2.1). Second, the test suite is run to gather information (Section 2.2), and finally, the bug-finding algorithm is run to find the specific location of the bug (Section 2.3).

2.1 HDL Instrumentation

In this work *hwBugHunt* instruments the code adding a preprocessing path. *hwBugHunt* reads a Verilog file and generates instrumented Verilog files. The instrumentation is done to gather metrics for line coverage and toggle coverage. The instrumentation done provides a unique identifier for the corresponding line of code. In the case of toggle coverage, the instrumentation is done so that any change in value is kept track of. In addition to the coverage metrics, the instrumentation of the code also keeps track of whether the testbench reports a pass or fail. We call this instrumented code as *Coverage Mark*. If an error is detected, the simulation is halted and the gathered statistics are dumped to the disk. There is no need for any manual intervention as this is done automatically by the algorithm.

2.2 Gathering Statistics

Traditional coverage simulators keep only one coverage checkpoint, which summarizes the coverage utilization for the whole execution. The key difference, between the method used to instrument HDL code by *hwBugHunt* and those of more traditional coverage gathering methods, is that *hwBugHunt* creates multiple coverage checkpoints as the design under test executes. Instead, *hwBugHunt* creates a new coverage checkpoint each time an instruction retires. Each coverage check point is a collection of *Coverage Mark* that occur since the last coverage check point. The advent of multiple coverage checkpoints allows for improved bug-searching.

Figure 2 shows the DUT running at the same time as the architectural simulator. Each time that a *Coverage Mark* is found (either a new line of code executes, or a wire or register toggles), it is added to the current coverage checkpoint. *hw-*

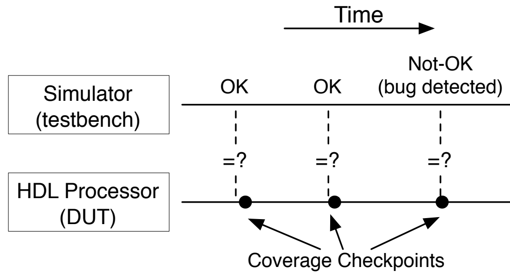

Figure 2: Execution time line as the simulator verifies the results from the DUT.

BugHunt creates a new coverage checkpoint each cycle that an instruction retires. Conceptually, this would require nearly as many checkpoints as execution cycles. This would represent a significant memory overhead. However, it is not necessary to keep all the checkpoints as most bugs can be detected a few hundred cycles after they happened. This value is processsor dependant. A narrower processor may require fewer cycles than a processor with a deeper pipeline. Although configurable, *hwBugHunt* keeps a list of 100 active checkpoints. Once checkpoint 101 is created, the two oldest coverage checkpoints are joined or collapsed.

hwBugHunt uses three types of logic operations to manage checkpoints. Let C_1 and C_2 represent two random checkpoints.

Join	$C1 \cup C2$
And	$C1 \cap C2$
Diff	C_1 - C_2

Table 1: Basic coverage checkpoint operations used by *hwBugHunt*.

Join merges two coverage checkpoints, this is, it does a union of the two coverage checkpoints under consideration. *And* finds the common *Coverage Marks* between two checkpoints. The *Diff* operation corresponds to an inverted implication in boolean logic.

2.3 Bug Find Algorithm

Once the instrumented test suite finishes execution, it provides *hwBugHunt* with a collection of runs. Each run has up to 101 checkpoints [1]. The test suite, obviously, is composed of multiple tests – some of them run without detecting the bug, we label these successful tests as "pass run." Some tests, however, detect the bug and subsequently fail. We label the negative tests as "failed run." Note, that some *pass* runs may contain a bug that went undetected.

The *pass* runs have multiple coverage checkpoints with *Coverage Marks* that did not detect the bug. *hwBugHunt Joins* every *OK* checkpoint into a single coverage checkpoint (*OK* coverage checkpoint), *failed* runs include *OK* and *not-OK* checkpoints. Therefore, it is unclear which coverage checkpoint has the *Coverage Mark* associated with the bug. In the *failed* runs the testbench fails after the bug has been detected. However, the bug could have occurred inside the current checkpoint or anywhere in the previous checkpoints.

[1] A run has less than 101 checkpoint only if the bug is detected before creating the coverage checkpoint 101.

Therefore, for each *Coverage Mark*, we need to calculate a confidence value that this specific *Coverage Mark* is the source of the bug. From the *OK* and *not-OK* gathered coverage checkpoints, four discrete cases emerge as shown in the Equations numbered 1 through 4.

marks not present in pass run	&	marks present in failed run	(1)
marks present in pass run	&	marks present in failed run	(2)
marks not present in the pass run	&	marks not present in the failed run	(3)
marks present in pass run	&	marks not present in failed run	(4)

In equation 1, we find a *Coverage Mark* in the *failed* run that is not present in the *pass* run. This means that the instruction corresponding to this *Coverage Mark* got executed in the *failed* run and was not executed in the *pass* run. Since, this *Coverage Mark* is found in the recent checkpoint, the probability that this instruction caused the testbench to fail is high.

In equation 2, we find a *Coverage Mark* in both the *pass* and the *failed* runs. This means that the instruction associated with the *Coverage Mark* was executed during one or more of the *pass* runs and during the *failed* runs. Such an instruction can be the cause for a bug, but we cannot deduce anything from this scenario.

In equation 3, we find that a *Coverage Mark* is not found in both the *pass* and in the *failed* runs. This means that the corresponding instruction was not executed during the program's run, hence cannot be the source of the bug.

In the case of Equation 4, we see that a *Coverage Mark* that happens during the *pass* run but does not happen during the *failed* run. This means that an instruction that executes in the *pass* runs do not get executed in the last 100 cycles of the *failed* runs. This may have caused the testbench to fail. This *Coverage Mark* is marked with a low confidence.

The compound confidence of each *Coverage Mark* is calculated as shown in the following equation:

```
confidence = count(# of marks not present in pass run &
                   # of marks present in failed run)    *a
           + count(# of marks present in pass run &
                   # of marks not present in failed run)*b
           + count(# of marks not present in pass run &
                   # of marks not present in failed run)*c
           + count(# of marks present in pass run &
                   # of marks present in failed run)    *d
```

A confidence value is generated for each *Coverage Mark*. As a result, to reduce false positives, only *Coverage Marks* with the highest confidence need to be reported as potential bug locations. Following this rule of thumb, we only report the locations for the top 10% elements in the confidence rankings. For example, if there are 50 bugs, each of them with a confidence from 1 to 50, only the top 5 potential bugs are reported.

The pseudo-code for the Bug Find algorithm shown in Figure 3 quickly finds the confidence for each *Coverage Mark*. To simplify the algorithm parameter selection, we use $a = 1$ and $b = c = d = 0$. The algorithm concatenates the *OK* coverage checkpoint with a subsection of each *failed* run (past). The last 100 coverage checkpoints of the *failed* runs are joined to-

gether (recent). Once the two temporal checkpoints are available, a *Diff* operation is performed. The resulting coverage checkpoint has a list of "possible" bug locations. The algorithm reports each *Coverage Mark* or the bug location with a confidence.

```
confidence = 0
for_each_checkpoint in Failed run { |nok|
  for_each_checkpoint in Pass run { |ok|
    confidence++
    # Join all ok ckps with oldest nok ckp
    past   = Join ok[0:100] nok[0]
    # Join recent nok ckps
    recent = Join nok[1:100]
    # What did happen on the last failed run?
    res    = Diff recent past

    for_each_mark in res { |mark|
      counter[mark]++
    }
  }
}
for_all_marks { |mark|
  puts mark, 100*counter[mark]/confidence
}
```

Figure 3: Bug Find algorithm.

Both the *Join* and the *Diff* functions run in $O(n)$, where n is the number of coverage checkpoints per run (100+1). As previously stated, the number of coverage checkpoints is restricted to 100+1 active checkpoints in order to reduce memory overheads. Therefore, assuming that there are m tests, the overall complexity of the Bug Finding algorithm is $O(m * n)$.

3 Evaluation Setup

To evaluate the proposed system we need a representative set of bugs injected into the design under test. To have a fair evaluation, we create random bugs following the bug classification [1]. For each bug category we create 3 bugs with some *pass* and some *failed* runs. The bug number and category relation is shown on Table 2. We do not introduce any Finite State Machine (FSM) bugs because the IVM implementation only has one explicit FSM. Therefore, we can not create multiple FSM bugs.

Design Error Category	Bugs Introduced
Wrong signal source	102, 113, 115
Conceptual error	207, 210, 211
Case statement	301, 302, 304
Wrong constant	601, 602, 603
Logical expression wrong	702, 703, 704
If statement	1101, 1102, 1104
FSM error	none
Wrong operator	1501, 1502, 1505

Table 2: Bugs introduced to IVM.

The created bugs are introduced into the Illinois Verilog Model (IVM) [12]. IVM implements a subset of the Alpha 21264 microarchitecture. To test the bugs introduced, we use the test suite provided with the IVM infrastructure. This test suite is built around simplescalar [4]. To drive the architectural simulator, we run a subset of SPECint (bzip, crafty, twolf, vpr, gcc, mcf, gap) for over 20K instructions for each test. In addition, we also use some kernels that test branches and logic operations.

4 Evaluation

4.1 Overall Results

The results of the Bug find algorithm is summarized in Table 3. The first column is the BugId, bugs are categorised based on the classification reported as in Table 2.

The results from the Bug Find algorithm provides a list of several lines of code that may be the source of the bug. Using *hwBugHunt* confidence estimation, the lines of code flagged by the algorithm are ranked. A 1st position on the ranking indicates that the bug was correctly located with 0 false positives (correct lines of code incorrectly flagged as errors).

In some cases, *hwBugHunt* reported several lines of code with the same ranking, in such cases an average was taken and this was used to report the rank. For example Bug603, 11 lines were marked with the same ranking, so an average was taken to determine the rank (rank: 6 = 1+2+...+11/11). Hence, Bug603 was located with 5 false positives reported. On an average we found that there were only 0.6 false positives for every correctly found bug.

In some cases *hwBugHunt* was not able to locate a source for bug without reporting several lines of code. To minimize the false positives reported, *hwBugHunt* ignores a bug when the confidence is lower than (10%) of the top confidence reported. This makes the false positives equal to the reported bugs when the bug is not found or leser when found. On an average we found that only 3.1 false positives were reported for every bug analyzed. Columnns 2, 3, 4,5 show whether the bug was located or not, the ranking, the confidence and the top confidence for each bug.

The sixth column, 'Reported' shows the number of bugs reported in the top (10%) of the highest confidence. The seventh column in the table shows the number of false positives reported.

A total of 16 different SPECint bencmarks were used. The eighth and ninth columns in the table show the number of Pass-runs (runs where the bug was not located) and the total Failed-runs(where the bug was located). In some cases, these 2 columns do not add upto 16, that reason was because for some of the bugs the tests were failing in the very first few cycles.

From the 21 bugs analyzed, *hwBugHunt* is able to correctly locate 13 bugs in the top 10% confidence (Reported). These reported bugs are localized with high accuracy. On average, correctly located bugs are reported in the 1.6 ranking position. Eight of them are reported as the first ranking position. This has clear advantages to reduce the bug searching time.

Eight (38%) of the introduced bugs are not located. As the detailed results section shows, there are two major reasons: silent bugs and not enough *Coverage Marks*. Silent bugs are triggered bugs not detected by the test suite. The problem with those bugs is that the Bug Find algorithm (Section 2.3) assumes that the *OK* run does not have *Coverage Marks* for the bug. Since the bug is not detected and the *Coverage Marks* associated with the bug are set, the confidence for finding such bugs decreases. This happens in bugs like Bug601 and Bug703 where the bug is notified by in a position with too low confidence.

The lack of related *Coverage Marks* further prevents *hw-*

Bug	Located	Ranking	Confidence	TopConf	Reported	FalsePos	*pass* runs	*failed* runs
102	Yes	1	64%	64%	1	0	12	4
113	No	NA	NA	36%	6	6	13	3
115	Yes	1	55%	55%	9	0	11	5
207	Yes	1	56%	56%	1	0	6	10
210	No	31	32%	63%	3	3	5	10
211	No	312	3%	25%	2	2	6	9
301	Yes	1	45%	45%	5	0	4	12
302	Yes	1	28%	28%	2	0	5	11
304	Yes	2	60%	60%	4	1	7	5
601	No	238	10%	44%	3	3	7	9
602	Yes	1	50%	50%	4	0	12	4
603	Yes	6	20%	20%	11	5	11	5
702	Yes	1	20%	20%	1	0	10	5
703	No	53	11%	30%	4	4	5	9
704	No	6	30%	38%	3	3	4	10
1101	Yes	2	68%	68%	3	1	10	6
1102	No	NA	NA	40%	1	1	9	7
1104	Yes	1	36%	36%	3	0	5	11
1501	Yes	1.5	14%	14%	7	0.5	9	7
1502	Yes*	1	37%	37%	2	0	4	11
1505	No	NA	NA	16%	3	3	1	14
Found Avg.	13 Yes	1.6	43%	43%	4.1	0.6	8.9	7.4
Not Found Avg.	8 No	–	–	36%	3.1	3.1	6.25	8.9

Table 3: *hwBugHunt* overall results for all the bugs analyzed.

BugHunt from localizing certain bugs. For example, Bug1102 and Bug1505 are not localized at all. This is because there are no *Coverage Marks* associated with these bugs. The solution to this approach is to introduce additional *Coverage Mark* types on the instrumented HDL code. This is left for further work.

Although several bugs are not located, *hwBugHunt* keeps the overhead to a minimum. Minimized overhead is one of the design principles of *hwBugHunt*. This is achieved on all the bugs where the number of false positives is just 3.1. This means that even for the not located bugs, just analyzing 3.1 potential bugs is enough to decide to fall back to more traditional bug detection methods. Together with the low execution overhead and success rate of (62%), we believe that *hwBugHunt* will contribute effectively to reducing the time and effort required in verification.

4.2 Detailed Results

We now proceed to evaluate some of the successes and failures of *hwBugHunt*. This involves a more detailed analysis such that the benefits and shortcomings of *hwBugHunt* are well understood.

To perform this analysis, we specifically analyze the results in Table 3 and each failing bug is individually evaluated.

Bug113 swaps the names of the stores wires on the ROB. IVM can retire up to 2 stores per cycle, this bug performs the retirement out of order for two consecutive stores. There are two reasons why this bug is not detected: lack of failing test suites and *Coverage Marks*. Bug113 only has one failing test. In addition, it is not detected because many stores can still retire correctly (both are set or unset), and there is no *Coverage Mark* trying to track when one store is retired and not the other.

Bug210 loses registers when the processor pipeline stalls. The problem with this bug is that its detection requires several hundreds cycles. The testbench does not detect a bug until the processor pipeline deadlocks due to lack of registers. A more detailed testbench that tracks the physical register used by the retiring instruction should suffice to detect this bug.

Bug211 changes the recycle policy on the ROB. This bug is not detected due to a similar reason as Bug113. Nevertheless, several of the top 10 bug rankings point to the ROB file.

Bug601 changes the opcode for which a shift operation is performed. The bug is not detected because the alternative opcode generates similar results for several tests. As a result the ranking decreases to 238.

Bug703 and Bug704 are not detected, the reason being the short length of the *pass* runs. As a result, the bug find algorithm produces low confidence values even for the most confidence bug (Confidence). Increasing the test suite with tests that do not fail would improve the reporting accuracy.

Bug1102 is not reported on the top 10% confidence sample (Reported). This bug modifies the rename logic by selecting an incorrect forwarding on the group of instructions renamed. The *Coverage Marks* inserted are not enough to localize the bug because all the related *Coverage Marks* are set by failing and not-failing testbenches. Unless additional *Coverage Marks* are tracked the bug could not be detected.

Bug1502 is detected, but the variable reported is not the reason for the bug but a cause of the bug. Bug1502 incorrectly searches the oldest load in the load queue to do forwarding. *hwBugHunt* is not able to detect the incorrectly updated pointer, but it detects that the load id generated for some instructions as potential bugs.

Bug1505 is not detected due to the lack of *pass* runs. This bug incorrectly enables store instructions. As a result, only two small tests that do not perform memory operations are able to pass the test. Although *hwBugHunt* ranks the memory module 2 out of the 3 false positive reported, *hwBugHunt* is unable to locate the source. Adding additional *pass* tests could improve the accuracy.

4.3 Sensitivity Analysis

To better understand *hwBugHunt*, this section performs two sensitivity studies. First, we partially deactivate different coverage metrics. Second, we introduce two bugs at once to see

the effectiveness of *hwBugHunt* with multiple bugs.

Coverage Sensitivity: *hwBugHunt* instruments the HDL code to gather multiple coverage metrics. Each of them has a different instrumentation overhead. While line coverage has a smallest performance overhead, toggle coverage has the maximum performance overhead. To verify the importance of the different coverage metrics gathered, we deactivate some of them and analyze the accuracy of the bug finding algorithm when less coverage information is provided.

With line coverage only, most of the bugs are missing. More interesting is the case when all but the toggle coverage are activated. The resulting simplified framework locates 7 bugs instead of 13 bugs. Removing the toggle coverage only improves the accuracy of Bug704. While the default *hwBugHunt* ranks this bug in 6th position, without toggle coverage the same bug is ranked on 2nd position.

For the bugs analyzed, we conclude that the four types of coverage utilized are required.

Multiple Bugs Sensitivity: *hwBugHunt* is built around the assumption that one bug happens at a time. Obviously, several bugs can happen simultaneously, and different testbenches can fail due to different bugs. To evaluate the effectiveness of *hwBugHunt* under such conditions, we activate several bugs.

When Bug302 and Bug1104 are active at the same time, *hwBugHunt* fails to locate them. Both bugs happen on the same file (data cache). When both are active, *hwBugHunt* pinpoints them but not on the top 10% confidence.

In another experiment, Bug207 and Bug301 are active at the same time. In this case, *hwBugHunt* correctly finds both bugs in the top 10% confidence.

Mixed results are achieved with Bug301 and Bug601. In that case, only the Bug301 is located on the top 10% confidence. Nevertheless, once the bug is solved Bug601 can be studied in isolation and therefore located correctly. A similar result is achieved when Bug302 and Bug702 are activated simultaneously. In this case, only Bug302 is located by *hwBugHunt*.

5 Conclusions

hwBugHunt targets the 58% part of the verification pie that accounts for debugging [10] This work puts forth an attractive means for reducing a substantial percentage of the time spent during debugging. With verification being recognised as a major bottleneck in the design cycle, tools like *hwBugHunt* that target specifically to cut down the costs of verification can have a huge impact on lowering the overall development costs.

Although there are many tools on the market to reduce verification costs, to our knowledge they focus on many aspects of verification like bug detection, formal verification, detect the faulty gate in netlist, and/or speedup the simulations, but none of them pinpoint the line of HDL code where the bug is located. The closest thing are lint tools, but those tools always perform static checking and never use dynamic information provided by testbenches. As a result, we see this work as highly novel with a great potential impact to the verification industry.

Acknowledgments

We like to thank the reviewers for their feedback on the paper. This work was supported in part by the National Science Foundation under grants 0546819 and 720913; Special Research Grant from the University of California, Santa Cruz; Sun OpenSPARC Center of Excellence at UCSC; gifts from SUN, Altera, Xilinx, and ChipEDA. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] H. Al-Asaad, D.V. Campenhout, J.P. Hayes, T. Mudge, and R. Brown. High-Level Design Verification of Microprocessors via Error Modeling. In *IEEE International High-Level Design Validation and Test Workshop*, pages 194–201, 1997.

[2] M. Fahim Ali, A. Veneris, A. Smith, S. Safarpour, R. Drechsler, and M. Abadir. Debugging sequential circuits using boolean satisfiability. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 204–209, Washington, DC, USA, 2004. IEEE Computer Society.

[3] Semiconductor Industry Association. International Technology Roadmap for Semiconductors (ITRS), 2002.

[4] D. Burger, T.M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, 1996.

[5] M.D. Ernst, A. Czeisler, W.G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 449–458, New York, NY, USA, 2000. ACM Press.

[6] Görschwin Fey, Sean Safarpour, Andreas Veneris, and Rolf Drechsler. On the relation between simulation-based and sat-based diagnosis. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1139–1144, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[7] R. Goodall, D. Fandel, A. Allan, P. Landler, and H. R. Huff. Long Term Productivity Mechanisms of the Semiconductor Industry. www.sematech.org, 2002.

[8] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty. Iodine: a tool to automatically infer dynamic invariants for hardware designs. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 775–778, New York, NY, USA, 2005. ACM Press.

[9] S. Hangal and M.S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM Press.

[10] Collett International. 2003 IC/ASIC Design Closure Study, 2003.

[11] R. Liang. Personal communication. Sun Microsystems, 2006.

[12] N.J. Wang, J. Quek, T.M. Rafacz, and S.J. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *International Conference on Dependable Systems and Networks*. IEEE Computer Society, Jun 2004.