Automated Debugging with High Level Abstraction and Refinement

Sean Safarpour [1] Andreas Veneris[2,3]

***Abstract*— Design debugging is a manual and time consuming task which takes as much as 60% of the verification effort. To alleviate the debugging pain automated debuggers must tackle industrial problems by increasing their capacity and improving their performance. This work introduces an abstraction and refinement methodology for debugging that leverages the high level information inherent to RTL designs. Function abstraction uses the modular nature of designs to simplify the debugging problem. If required, refinement re-introduces the necessary circuitry back into the design in order to find all error locations. The abstraction and refinement process is applied throughout the design's hierarchy allowing for a divide and conquer methodology. The proposed technique is shown to reduce the memory requirement by as much as 27× and reduce the run-time by two orders of magnitude over a conventional debugger.**

I. Introduction

The continuous increase in design size and complexity has led to a significant escalation in the verification cost and effort. This trend is confirmed in the industry as the number of verification engineers has quadrupled with respect design engineers over the last decade [1]. As a result, functional verification has received much attention from the academic and industrial communities to alleviate this pain [2], [3], [4]. In contrast relatively little attention is paid to the task of debugging, or locating the error source, once verification fails.

Today, debugging is predominantly performed manually by verification engineers with no more than waveform viewers and navigation tools at their disposal. The debug process is comprised of manually collecting information from the failed simulation trace (or counter-examples) and back-tracing with "what-if" analysis until the error source is identified. As typical design block sizes today exceed the half million synthesized gates mark and traces range from a few hundred to a few thousand of clock cycles, debugging has grown to take as much as 60% of the verification effort [5]. As a result, scalable automated debugging techniques have become an urgent necessity to alleviate this pain [1].

Broadly speaking, there are two factors that influence the effectiveness of automated debugging. The first factor is the design size that impacts the solution space. As designs are implemented at higher levels of abstraction and the gate count increases, the number of suspect error locations that needs to be examined also increases. In turn, a debugger's performance can dramatically slow down [6], [7]. Secondly, the length of the error trace, that is, the number of clock cycles from the beginning of simulation until the design fails, increases the solution space one must examine. Modern debugging solutions must cope with the complexity introduced by these factors to be adopted by the industry.

This work aims to bridge this gap between current debugging capabilities and contemporary industrial needs. It does so by introducing the concepts of abstraction and refinement for automated debuggers using high level information. More specifically, the modular and hierarchical nature of a design at the RTL is leveraged to develop an efficient abstraction and refinement methodology. In the past, similar techniques have led to dramatic improvements in the scalability and applicability of model checking methodologies [8], [9], [10]. More recently, results from abstraction/refinement at the gate level demonstrated significant performance gains for debugging as well [11].

The proposed functional abstraction-based debugging methodology operates iteratively on the design hierarchy. At the topmost level, it begins by abstracting or "simplifying" high level components. Next, a conventional debugger [12], [7], [6] is applied to the abstracted model to identify all error locations. Since the debugging problem is much smaller than the original, it can be solved much more efficiently. If the debugger fails to identify all error locations, refinement is performed to systematically re-introduce the abstracted design components back into the design.

This pairing sequence of debugging and refinement is repeated until all solutions are found at the given hierarchy level. Next, the process is repeated at the next lower level to improve the granularity of the solutions. The methodology terminates once all the error locations are found at the lowest hierarchy level (*i.e.* gate level). It is important to notice that the proposed theory is not tied to any particular debugging technique and applies to SAT, simulation, and BDD based methods [12], [6]. Extensive experiments on large industrial problems demonstrate a drastic memory reduction of over 27× and run time reductions of over two orders of magnitude. This work demonstrates that abstraction and refinement has a significant impact on the performance of automated debugging.

This paper is organized as follows. Section II provides background material while the main contribution is introduced in Section III. Section IV and V present the experimental results and the conclusion, respectively.

II. Preliminaries

A circuit C (combinational or sequential) at the RTL can be hierarchically composed of modules or functions. In this work, a function is said to generate a Boolean value for a variable y based on m input variables $x_1, x_2, ..., x_m$ and zero or more state variables. In this work, we are primarily concerned with the structural connectivity between the input variables and the

[01] Vennsa Technologies Inc., Toronto, ON M5V 3B1 (sean@vennsa.com)
[02] University of Toronto, ECE Department, Toronto, ON M5S 3G4 (veneris@eecg.toronto.edu)
[03] Athens University of Economics and Business, CS Department, Athens, 10434

output variable y of a function. As a result, we label the function of y as $f(x_1, x_2, ..., x_m)$ and omit its dependence on any state variables. The terms modules, components and functions are used interchangeably to refer to entities implementing functions as defined above. For example, a Verilog function or a collection of logic gates and flip-flops can define a module. Each module implements a multi-output function $F = \{f_1(X), f_2(X), ..., f_p(X)\}$ where each single-output function f_i is defined on input variables $X = \{x_1, x_2, ..., x_q\}$. In the remaining paper, single output functions and multi-output functions are not distinguished unless explicitly stated otherwise.

Modules can also contain sub-modules thus resulting in a hierarchy tree H for the design [13]. A hierarchy tree H contains nodes representing modules and edges representing parent and child (sub-module) relationships. The hierarchy tree H can contain many levels, each function is tagged with a superscript that indicates its level and a subscript to uniquely label the function. For example a function F_j^i is at level i of the tree and it can have sub-functions F_k^{i+1} and F_l^{i+1} at the next level $i+1$. The output of the entire design C is represented by F_1^0 at *root* level 0.

A. Debugging Background

Error traces (or counter-examples) returned by simulation or formal verification tools along with a corresponding set of correct output vector sequences comprise the set of diagnosis vectors V. Given an erroneous design C with the corresponding set of diagnosis vectors V, design debugging identifies components (gates, modules, etc.) responsible for the erroneous behavior. We assume that the reference (golden) model can only be simulated to provide the correct output values for V. For instance, the golden model can be in some high level language (C/C++, Matlab, etc) and provide no structural similarity to the RTL. It has been shown in [14] that the lack of similarity increases the debugging problem complexity exponentially with respect to the *error cardinality* N.

In this paper, a user-defined number $maxN$ denotes the maximum number of errors the debugger is limited to find. If $maxN$ is smaller than the actual number of errors, then not all error locations are found. A debugger begins with $N = 1$ and increases its value when it does not return with a solution until $maxN$ is reached. In this process, the tool returns all *equivalent* error locations responsible for the failure under a vector set V [6]. These locations are called *error suspects*. For a debugging methodology to remain complete and return the actual error site, all equivalent error locations must be returned by the debugger for a given $maxN$ [6].

B. Abstraction/Refinement in Model Checking and Debugging

Abstraction and refinement techniques are used readily in model checking to mitigate the exponential nature of the state space [8], [9], [10]. Roughly speaking, an *abstract model* is derived by removing some state elements from the *concrete* design using some abstraction function $\hbar$. If model checking determines that a universal property holds in an existentially abstracted model, then it must also hold in the concrete design [8]. However, if a property does not hold in the abstract model, then the corresponding counter-example must be validated in the concrete design. If the counter-example does not expose a failure of the property in the concrete design it is said to be *spurious* [9]. In this case, the abstract model is refined by reverting some of the abstracted components and continuing the model checking process.

Recently, the concept of abstraction/refinement was utilized to improve performance in conventional debuggers [11]. Under this new framework, an abstract model of the gate level design is first created to undergo debugging by removing state a set of state elements. Since this representation contains less logic than the original one, the size of the problem may be reduced considerably, in favor of debugging. Debugging an abstract model can sometimes return abstracted state elements as error sources. In these cases, a refinement procedure replaces some of the abstracted variables with the original state elements. Note that this technique does not leverage any high level information when performing abstraction and thus cannot iterate over the different hierarchy levels.

III. Debugging with Function Abstraction

The abstraction/refinement method introduced in [11] operates on the circuit's state elements. Although powerful, deciding which states to abstraction is not a trivial task. In contrast, function abstraction, presented in this paper, utilizes the high level information contained in the circuit RTL and operates on functions and modules providing a natural way to structure the debugging problem. Furthermore, the hierarchical composition of designs can be leveraged to apply abstraction in a systematic and iterative manner.

A. Formulating the Problem

At any hierarchy level, the original design C can undergo *function abstraction* based on the functions available at that level. The abstraction function is given by

$$C' = \hbar_f(C, i),$$

where C is the original circuit and i is a given hierarchy level. The mapping of $\hbar_f$ is a design C' which contains a set of functions $Abs_i = \{F_j, F_k, ...\}$ abstracted at hierarchy level i. In other words, the circuitry corresponding to Abs_i is removed from C and hierarchy H, resulting in C' and H'. In our methodology, determining which functions to abstract is found through heuristics outlined in Section IV. After removing Abs_i from C, the functions $\{F_j, F_k, ...\}$ are replaced with new primary input $\{X_{F_j}, X_{F_k}, ...\}$ in C'.

Example 1 *Figure 1 (a) and (b) show a design with its corresponding hierarchy tree. The functions of the design are $F_1^1, F_2^1, F_3^1, F_4^1$ at level 1 and F_5^2, F_6^2 at level 2. By abstracting functions F_1^1 and F_3^1 the resulting design C' with the new primary input $X_{F_1^1}, X_{F_3^1}$ is shown in Figure 1 (c). The corresponding hierarchy tree H' for the new abstracted design is shown in Figure 1 (d). Notice that the sub-module of F_1^1, F_5^2 is also abstracted in C'.*

As a consequence of the abstraction operation, some of the transitive fanin of the abstracted functions may be dangling, that is, at level i, fanin circuitry of Abs_i may not be connected

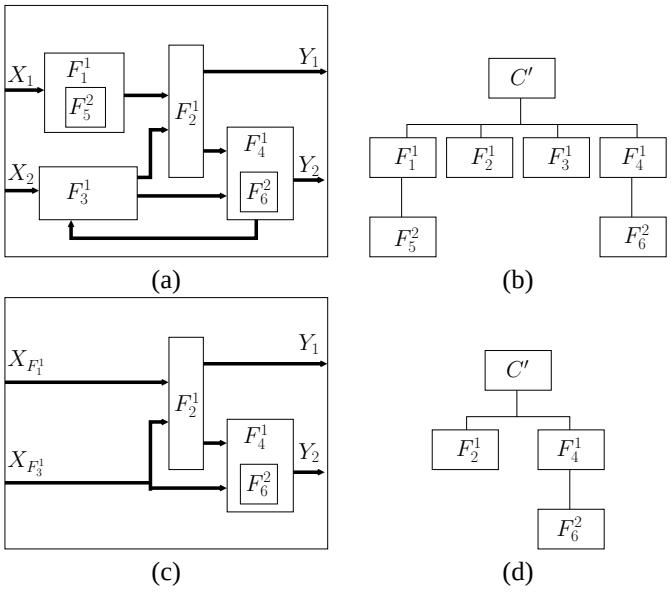

Fig. 1. Circuit and its hierarchy before and after abstracting F_1^1 and F_3^1

to other circuit components once Abs_i are removed. This dangling circuitry can be deleted with a logic removal algorithm to further simplify C'.

Since the abstracted design is less complex than the concrete one, it can be used to formulate an easier debugging problem. However, as demonstrated in [11] with theorems and examples, the abstract model C' contains the set of newly added primary input, which are unconstrained in the vectors V. As a result, a debugging engine may assign arbitrary logic values to these variables while operating on the problem. Such logic value assignments traditionally made by debuggers may be *unjustifiable* in the concrete design C where these variables are constrained by their original fanin logic. Consequently the solutions returned by the debugger in this formulation cannot be trusted and may be incorrect.

To resolve this situation, the logic values of the abstracted functions $Abs_i = \{F_1, F_2, ...\}$ must be captured and used to constrain the primary input $\{X_{F_1}, X_{F_2}, ...\}$. The constraints can be captured by storing the values of the output $\{F_1, F_2, ..\}$ of the functions to be abstracted during the simulation of C with the stimulus input sequence in V. These values must be amended to V to create a new stimulus V' to constrain the primary input $\{X_{F_1}, X_{F_2}, ...\}$ of C'.

The amended stimulus V' and the abstracted design C' can be provided to an automated debugger to find the error suspects or the error sources. The debugger can determine the location of errors in C' but it can miss errors residing within the abstracted circuitry. To find all equivalent locations, a mechanism is required to identify when suspects may be missed. This is accomplished by adding correction models on the added primary input X_F for each abstracted function F. In the context of SAT-based debugging a *correction model*, as defined [11], [6], is a multiplexer that allows the debugger to identify any gate as suspects (including primary inputs). Similar models exist for other debugging techniques [13]. Figure 2 shows the correction models as black dots on the example of Figure 1 at level 1. Notice that suspects are added for each of the functions at level 1 in addition to the

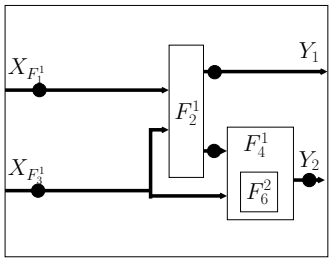

Fig. 2. Abstract circuit with black circles representing correction models

primary inputs $X_{F_1}^1$ and $X_{F_3}^1$. If any of the new primary inputs are found as suspects, then the error in the concrete design may be encapsulated inside the abstracted functions. These solutions are called *spurious* because they do not correspond to components in the concrete design.

B. Spurious Solutions

Abstract modules corresponding to spurious solutions must be refined to determine where the error location is with respect to the concrete design. The refinement process re-introduces the abstracted modules corresponding to spurious solutions into the design C'. That is, if one of the suspects found is a primary input X_{F_j}, then the function F_j is added to C' with its output connected to the fanin of the removed X_{F_j}. Once the original functions are re-introduced, all its transitive fanin must also be added to the abstract design C'.

After performing refinement, the debugger must be called again to determine the error location corresponding to the previously found spurious solution. The steps of debugging and refinement are repeated until no spurious solutions are found. Algorithm 1 presents the described abstraction and refinement flow at a given hierarchical level. Lines 2, 4, and 5 perform module abstraction, constrain the input and perform modular debugging. Refinement is performed on line 8 if spurious solutions are found. As explained in [11], $maxN$ can be increased to guarantee correctness and completeness.

Theorem 1 states that the design C' resulting from Algorithm 1 contains the modules/functions necessary to find all the equivalent error locations. Lemma 1 below is useful to prove Theorem 1 and other results in this section.

Algorithm 1 Module Abstraction and Refinement Debugging

```
1:  Solutions = ∅, N = 1
2:  C' = ħ_f(C, level)
3:  while (1) do
4:     V'= extract_constraint(C, C', V)
5:     New_sols = debug(C', V', N, level)
6:     for all Sol ∈ New_sols do
7:        if (spurious_solutions(Sol, C')) then
8:           C' =refine(Sol, C', level)
9:           N = 0
10:       else
11:          Solutions = Solutions ∪ Sol
12:       end if
13:    end for
14:    N = N + 1
15:    if (N > maxN) then
16:       return {Solutions, C'}
17:    end if
18: end while
```

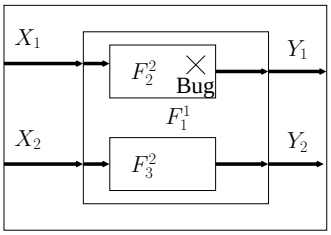

Fig. 3. Function F_1^1 is composed of functions F_2^2 and F_3^2

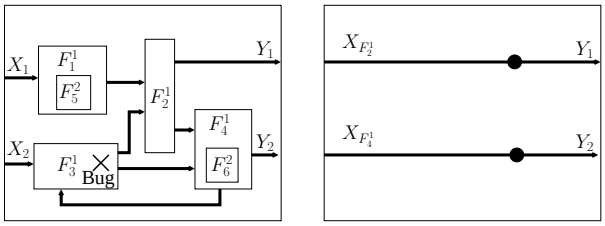

Fig. 4. Before and after abstracting functions F_2^1 and F_4^1 with bug in F_3^1

Lemma 1 *Assuming that an erroneous behavior is only observed at the primary output of a design, a module m_c demonstrates an erroneous behavior only if its parent module m_p demonstrates an erroneous behavior.*

Proof: Take any parent module m_p of module m_c from a hierarchical design. All paths from any set of output of module m_c to the primary output of the circuit will contain the output of modules m_p. Thus, the outputs of m_p dominate any set of output of m_c. As a result, the parent module demonstrates an erroneous behavior if an error behavior from m_c is observed at the primary output. ■

Theorem 1 *The debugging technique presented in Algorithm 1 finds all equivalent error modules at a given hierarchy level.*

Proof: Based on Lemma 1, error modules demonstrate an erroneous behavior only if their parent modules demonstrate an erroneous behavior. This means that if a child module at level i is erroneous, then the debugger will find the parent modules erroneous at levels $j < i$. At any hierarchy level, if the erroneous module or its parents are abstracted, the corresponding spurious solutions will be found. Refinement ensures that the content of the abstracted modules are re-introduced into the design thus allowing the erroneous module to be identified. Since all equivalent error locations cannot be distinguished for a given set of vectors, all equivalent error modules will be found. ■

C. Hierarchical abstraction

This function abstraction scheme introduced in the previous section is most effective when it is used in a hierarchical manner, where module-based debugging can be applied at each hierarchy level [13], [15]. At each level i of the hierarchy, the design can be represented by a set of functions $\{F_1^i, F_2^i, ...\}$. As discussed in the previous section, the iterative sequence of abstraction, debugging and refinement is applied until no spurious solutions are found. Thus at each level the abstracted functions in C' and their children in the hierarchy tree H are not required for debugging. In other words, it is not necessary to consider the internal logic of abstracted functions in C' at level i when debugging at the level $i+1$ as presented in Theorem 2.

Theorem 2 *After applying Algorithm 1, only the modules at level i present in design C' are necessary for debugging at level $j \geq i+1$.*

Proof: By the contrapositive of Lemma 1, no errors can be encapsulated in child modules if the parent modules are not refined at level i. Thus, to debug errors at level $j \geq i+1$, it is not necessary to consider abstracted modules at level i. ■

Apart from the functions abstracted at level i, more functions can be abstracted at level $i+1$. Intuitively, at each level of the hierarchy, functions are comprised of one or more sub-functions. This allows for sub-functions to be abstracted at lower levels of hierarchy even though their collection cannot be abstracted at a higher level. For example, consider Figure 3 where an error resides in F_2^2. At level 1, function F_1^1 cannot be abstracted since it contains the error, however, at level 2 sub-function F_3^2 may be abstracted since it is independent from F_2^2 and its output.

D. Multiple Errors

The module-based abstraction and refinement methodology described above is capable of debugging multiple error locations. The process for finding multiple error locations is similar to that in [11]. In that process, the cardinality used to find the errors starts at one and increases as the error location is not found. However, when spurious solutions are found and the corresponding module is refined, the error cardinality must be reset to 1. For function abstraction, the same process must be followed since resetting the cardinality is crucial for maintaining correctness.

Consider the example in Figure 4 where the modules $Abs_1 = \{F_2^1, F_4^1\}$ are abstracted at level 1. The abstraction results in the removal of modules F_1^1 and F_3^1 as well because they fan-in to Abs_1. The pre- and post- abstraction circuits are shown in Figure 4 (a) and (b), respectively. Assuming that the error is in module F_3^1, the error effect can propagate to the output of Y_1 and Y_2. Under those conditions, the debugger will not identify a single error module, but will find the error pair of $X_{F_2^1}$ and $X_{F_4^1}$. Through refinement, these modules and their fanin circuitry will be re-introduced in the circuit. At this point, the number of error modules the debugger seeks must be reset to 1, otherwise, the single error source inside F_3^1 will be missed.

E. Overall algorithm

Hierarchical debugging and function abstraction can be combined to give further gains when debugging a design. Hierarchy-based debugging can identify erroneous modules in a divide and conquer approach while abstraction simplifies each step of the problem. The overall proposed technique for function abstraction is shown in Algorithm 2. Here the

Algorithm 2 Hierarchical Debugging

```
1: Solutions = ∅, level = 0, N = 1, C' = C
2: while (1) do
3:    level = level + 1
4:    {New_sols, C'} = Module_debug(C', level, N)
5:    if New_sols = ∅ then
6:       return Solutions
7:    else
8:       Solutions = Solutions ∪ New_sols
9:    end if
10: end while
```

debugging problem is solved iteratively by descending the hierarchy while the details of abstraction and refinement are performed as require at each level by *Module_debug* according to Algorithm 1.

IV. Experiments

This section presents experimental results for the proposed high level abstraction and refinement methodology. All the circuits used are Verilog designs from the OpenCores.org website [16] except for an industrial communication design (*comm*). Each circuit contains a functional level error such as an incorrect statement, incorrect module instantiation, bad wiring between modules, etc. These RTL errors typically represent tens or hundreds of gate-level errors. The debugger used in all experiments is the module-aware SAT-based automated debugger of [13]. This set of experiments is conducted on a 64 bit Intel Core 2 Quad processor with 2.66 GHz and 8GB of memory.

Table I presents a summary of the circuits and the statistics using SAT-based debugging [13]. Columns one, two, and three show the name of the debugging problem based on the design, and its size in terms of gates and state elements (DFFs), respectively. Column four presents the length of the erroneous trace in terms of clock cycles required to observe the erroneous behavior from an initial state. When the trace is too long for the debugger, the trace is reduced to only contain the last 25 or 40 transitions in order to make automated debugging feasible. The number of clock cycle traces used to formulate the debugging problem are presented in the parentheses in column four. The column *# literals* presents the total number of literals generated in the CNF of the debugging problem [13]. Finally, columns *time (s)* and *mem (M)* show the total run-time, in seconds, required to solve the problem and the required memory, in MB, respectively. Notice that problem `comm2` requires more than 8GB to formulate the problem and thus runs out memory.

Table II presents the result of the proposed technique where initially all functions are abstracted. In other words, we rely on refinement to re-introduce all the circuitry required to debug the design. Column one shows the names of the problems, while column two shows the maximum error cardinality (*maxN*) required to solve the debugging problem. As discussed in Section III-D the cardinality required to locate the bug using an abstracted design can be larger than required to solve the original problem. Even though the problems shown here have a single functional-level (RTL) error, for the problem `comm1` and `comm3` a higher cardinality of 2 is used by the algorithm to find the error site.

TABLE I
Summary of problems for function abstraction

design	Problem statistics			Debugger engine [13]		
	size	# DFF	# cyc (used)	# literal	time (s)	mem (M)
wb_con1	81K	818	19 (19)	519K	58.74	619
wb_con3	81K	818	1387 (40)	1273K	205.16	1250
fdct1	264K	5461	189 (40)	1705K	555.37	4400
mem_ctrl1	39K	1145	1318 (40)	3888K	55.13	850
vga1	147K	17102	16100 (40)	8680K	1635.78	4700
vga2	147K	17102	141 (40)	213K	236.16	1350
comm1	450K	30339	19 (25)	1912K	1575.67	5080
comm2	454K	26852	88 (25)	N/A	N/A	Mem out
comm3	454K	26852	1387 (25)	278K	809.31	4831

In Table II, the column labeled *# itr* states the number of refinement and debugging iterations required to find all equivalent locations (number of times line 5 of Algorithm 1 is run). The column *mod refined / total* presents the number of modules refined out of the total number of modules in the concrete design. These modules are the only ones required to diagnose the error. The smaller this number is, the more effective is the abstraction technique. The next three columns, *# literals*, *time (s)*, and *peak mem (M)* present the benefit of the proposed technique in terms of the number of literals required in the problem formulation, the total run time in seconds and peak memory requirement by the entire algorithm.

The improvement provided by the proposed technique when compared to a state-of-the-art method such as this of [13] is shown in the last columns of Table II. Its effectiveness is attributed to reducing the problem size which is directly related to the number of literals. For example, in `vga1` where 5 / 14 modules are used, it leads to 630.48× reduction in literals which results in a 260.89× improvement in run time and 27.17× reduction in overall memory requirement. For all problems, the number of refinement and debugging iterations performed is larger than one. Therefore, it is clear that each iteration is much easier and faster when abstraction is used, thus it is more advantageous to run more iterations on easier problem than fewer iterations on harder problems.

In Table II there are two problems that experience a slow-down. For problem `fdct1`, six iterations are required to solve the problem, at which stage all 5 modules are used. Thus in this case, the extra iterations simply add overhead as the entire circuit is needed in order to solve the problem. The problem `vga2`, also experiences a slow down, but in this case, a 2.26× reduction in memory is observed. In this case, unlike the overall trend, the simpler and faster debugging problems cannot compensate for the extra iterations performed.

Figure 5 (a), (b) and (c) provide detail into the numbers of Table II for `vga2`, `fdct1` and `comm1`, respectively. These figures illustrate the relationship between the run time shown in solid line and the number of literals shown in dashed line against the refinement and debugging iterations. Notice the general trend where both run time and number of literals appear to increase exponentially with the increase in the number of iterations. For the majority of cases where the proposed technique is effective, abstraction allows the problem to be solved with a fraction of its size thus leading to smaller memory requirements and run times. Considering problem `vga2`, notice that for iterations 3,4,5 the solve time is quite high thus not providing any run time benefit.

TABLE II

RESULTS OF PROPOSED FUNCTION ABSTRACTION AND REFINEMENT TECHNIQUE

design name	abstracted problem stats						comparison to original		
	maxN	# itr	mod refined/total	# literals	time (s)	peak mem (M)	lit reduced (×)	speed up (×)	mem reduced (×)
wb_con1	1	3	3 / 8	116K	25.55	253	4.49	2.30	2.45
wb_con2	1	4	4 / 8	141K	149.12	469	9.05	1.38	2.67
fdct1	1	6	5 / 5	1705K	638.78	4400	1.00	0.87	1.00
mem_ctrl1	1	4	12 / 14	113K	12.02	200	34.53	4.59	4.25
vga1	1	2	5 / 14	14K	6.27	173	630.48	260.89	27.17
vga2	1	5	6 / 14	94K	436.38	1052	2.26	0.54	1.28
comm1	2	8	10 / 129	38K	108.32	772	50.37	13.11	6.58
comm2	1	9	10 / 129	25K	1403.47	640	—	—	> 12.50
comm3	2	8	8 / 129	80K	63.94	317	3.47	12.66	15.24

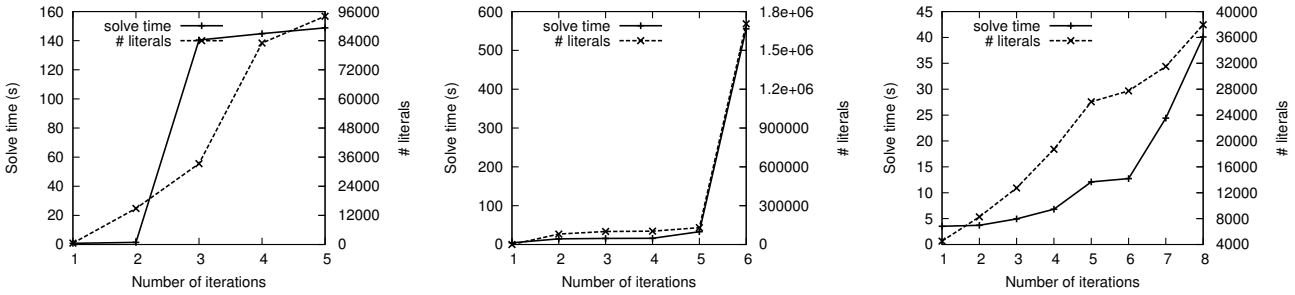

Fig. 5. Solve time and # literals in problem vs. the # of refinement and debugging iterations for `vga2` , `fdct1` and `comm1`

In general, it is found that more aggressive abstraction leads to more debugging and refinement steps. However, due to the simplicity of the design when abstracted aggressively, the initial debugging and refinement iterations are relatively much easy problems and thus quicker to solve. This behavior is observed in Figure 5 where the initial iterations have a faster run time than later ones. It may be possible to find an abstraction heuristic that can balance the number of iterations and the functions abstracted, but this is not a trivial task. However, in these experiments, it is found that simply abstracting all functions and modules is quite effective as it results with the smallest problems possible.

V. CONCLUSION

This work presents abstraction and refinement techniques for automated debugging that leverage the high level information contained in RTL design. More specifically, functions of the designs are first abstracted resulting in smaller debugging problems. To ensure that all the equivalent error locations are found in the original design, a refinement process is performed. The performance of the methodology is enhanced as it is applied hierarchically to the RTL design. The experiments demonstrate two orders of magnitude of performance improvement over a conventional debugging.

REFERENCES

[1] International Techonology Roadmap for Semiconductors, "http://www.itrs.net/links/2006update/2006updatefinal.htm," 2008.

[2] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[3] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*. Kluwer Academic Publishers, 2003.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 1579. Springer Verlag, 1999, pp. 193–207.

[5] H. Foster, "Assertion-based verification: Industry myths to realities (invited tutorial)," in *CAV*, 2008, pp. 5–10.

[6] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.

[7] K.-H. Chang, I. Markov, and V. Bertacco, "Automating post-silicon debugging and repair," *IEEE Trans. on Comp.*, p. to appear, 2008.

[8] E. Clarke, O. Grumberg, and D. Long, "Model checking and abstraction," in *Symposium on Principles of Programming Languages*, 1992, pp. 342–354.

[9] E. Clarke, A. Gupta, and O. Strichman, "SAT-based counterexample-guided abstraction refinement," *IEEE Trans. on CAD*, vol. 22, no. 7, pp. 1113–1123, 2004.

[10] P. Bjesse and J. Kukula, "Using counter example guided abstraction refinement to find complex bugs," in *Design, Automation and Test in Europe*, 2004, pp. 156–161.

[11] S. Safarpour and A. Veneris, "Abstraction and refinement techniques in automated design debugging," in *Design, Automation and Test in Europe*, 2007, pp. 1182–1187.

[12] S. Huang and K. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publisher, 1998.

[13] M. F. Ali, S. Safarpour, A. Veneris, M. Abadir, and R. Drechsler, "Post-verification debugging of hierarchical designs," in *Int'l Conf. on CAD*, 2005, pp. 871–876.

[14] A. Veneris and I. N. Hajj, "Design error diagnosis and correction via test vector simulation," *IEEE Trans. on CAD*, vol. 18, no. 12, pp. 1803–1816, 1999.

[15] G. Fey, S. Safarpour, A. Veneris, and R. Drechsler, "On the relation between simulation-based and SAT-based diagnosis," in *Design, Automation and Test in Europe*, 2006, pp. 1139–1144.

[16] OpenCores.org, "http://www.opencores.org," 2008.