A Runtime Metric of Design Confidence

EECS 578 Final Project, Fall 2006

Kenneth Zick
University of Michigan
kzick@umich.edu

Joseph Lee Greathouse
University of Michigan
jlgreath@umich.edu

ABSTRACT

We offer a series of statistical approaches that quantify confidence in a digital system's design at runtime. Using runtime checkers to detect errors, we present a system to collate information about these errors and assign a confidence number to both a design and modules in the design. We then show methods for implicating unchecked modules for failures deeper in the system. Using these methods, we give a series of examples of a small system with numerous errors among its modules. We show that even without checking all modules, we can often find the modules in which errors originate and replace these modules, resulting in the betterment of the system as a whole.

Keywords

Runtime verification, reliability, design confidence, fault tolerant computing.

1. INTRODUCTION

Verification engineers fight what seems like a losing battle with the continuous rise of the complexity in digital systems. Academia has responded by attempting to make new types of verification schemes viable. These include a number of formal methods for verifying the correctness of a circuit mathematically [3]. These formal methods are often too computationally intensive to use against whole commercial designs, but they may be put to other uses. For instance, there are formally verified runtime checking mechanisms that can detect failures in a system while the system runs its normal programs [1].

The idea behind runtime verification schemes is watching the system during its operation and finding/correcting errors as they happen. These methods, if they can correct errors, often do so silently, fixing the system during its operation so that the user never sees the problem. While this seems like a panacea to the problem of design complexity, the cost of fixing runtime errors is reduced system performance during error-correction. Due to this performance loss, system designers may require a design that has very few failures in an effort to ensure high performance.

Similarly, real-time systems will have difficulties with a design that suddenly changes its performance characteristics due to errors. This does not, however, mean that a runtime checker would not be useful for such designs. Tools such as formal checkers [4] and onboard assertions [9] can be used to help debug a design by finding errors and reporting them during operation.

1.1 Contribution Overview

We present a method for quantifying the confidence in a system's design based on runtime checking information. We assign a confidence number to individual modules in a design based on the error information obtained from onboard error-detection units. This information can be returned to verification engineers for bug-hunting purposes, or it can be used internally to help a system detect and work around its own bugs.

We assign these confidence numbers using a metric based upon the number of errors detected across the circuit. It is impossible, however, to watch all outputs of all modules in a complex design. As such, we need a method for assigning errors to modules that we cannot monitor directly. We present a technique called module-level probabilistic diagnosis to assign partial-errors to unwatched modules. Additionally we analyze three different methods of assigning failures.

1.2 Background Information

The idea of a runtime checking is central to this paper. Much of the literature focuses on using runtime checkers to greatly reduce the number of errors in a system, thereby allowing designs with bugs to operate accurately in the field. We leverage the idea of a module-level checker that watches the outputs of certain parts of the design and checks them for accuracy. An example of this is the formally verified checker processor presented in [13]. We utilize design ideas like this as the core error-finding mechanism for our metric.

We utilize statistical learning approaches in this paper to justify our confidence metrics. Causal networks and probabilistic methods are important for assigning errors across the system at runtime. Techniques such as this are found in the literature [7], but our method is based on both module- and system-level runtime checking, rather than design-time bug categorization. Causal reasoning has been used to perform automated diagnosis, by identifying the next set of diagnostic tests [8]. Our approach does not involve a sequence of tests or complete diagnosis, but rather acts as a statistical estimate.

Some runtime techniques for fixing errors rely on the verification engineer to find an error and describe it to the checker in order to fix it [12]. We feel that the presented technique is compatible with systems such as this because our work is focused on finding which bugs are particularly crucial to repair and from where these bugs originate.

While there is an entire field devoted to the reliability of systems, the models presented in this paper are not meant to estimate the reliability of a system over its lifetime. Instead, they are designed to estimate the correctness of a system's design.

1.3 Organization

We will begin this paper by discussing our runtime metric of design confidence. We will describe what exactly we mean by runtime confidence and detail the types of statistics needed to obtain it. We will follow this with methods for obtaining these statistics. This is contained in section 2.

Section 3 will explore our current methods for module-level probabilistic diagnosis. We will explain a weighted-graph method for assigning the blame for a detected failure. We will also discuss the implication matrix method of holding these weights, as well as two methods for determining these weights.

Our experimental data is contained in section 4. We show our confidence metric in action on a buggy design of small-to-moderate size in simulation. The design is tested blindly and we use our confidence estimations to decide which parts are most likely causing failures at the system level.

In section 5 we discuss future work that could improve the ideas presented here, as well as ideas for work that can utilize the idea of dynamic design confidence.

2. DESIGN CONFIDENCE

2.1 Runtime Confidence

We present the idea of runtime design confidence in this section. This is a scheme to quantify confidence in the correctness of a design using runtime information. We define confidence in a design as an estimated probability that a design will run correctly (and continue to run correctly) when it is in a certain system environment. As such, a confidence score is concerned with the probability of failure, not the number bugs. While a design may have many bugs, if they never appear, we can be confident that the design will work correctly the majority of the time.

In contrast to reliability estimates, which predict the average time a class of parts will take to fail, confidence estimates predict whether a single product will continue to operate correctly for the foreseeable future. This is based on the assumption that past failures forecast future failures of the same type. Confidence estimates cannot tell us how long it will be until a catastrophic failure of the part. Rather, they are useful to answer "is this a good design?"

We aim to make this confidence number useful for identifying problematic regions in a design, indicating erroneous parts by giving them lower module-level confidence scores. The scores themselves should be comparable across systems of similar design and modules of different complexity. Confidence scores should be useful for comparing the correctness of similar designs in similar environments. Finally, because these confidence numbers are based on runtime information, they need to be constantly updated during system operation.

2.2 Runtime Prediction of Failure Rates

It can be difficult to accurately model the future failure rate of a complex design before it is deployed into the field. Traditional methods such as 'bug curves' give an indication of how many bugs are likely to be found before deployment, but provide little guidance about whether subtle bugs will be exposed in the field, nor do they estimate how often those bugs will cause a failure.

We propose a straightforward method of predicting failure rates using the runtime statistics. First, we treat runtime failure statistics as an estimate of a design's inherent failure rate for a given system environment. This is standard statistical parameter learning, where the parameter in question is the inherent failure rate [10]. Second, we use the observed failure rate as a prediction of the future failure rate, according to the maximum likelihood hypothesis. If a design has failed once every million instructions in the past, maximum likelihood suggests that the actual failure rate is one in a million. This hypothesis is only an approximation, and it requires that we collect enough data. But it has proven to be useful in many situations, and it acts as a valuable starting point that is easy to implement in hardware.

To these ends, we give a rough formula for design confidence as

$$DesignConfidence_x = 1 - FailureRate_x \quad (1)$$

This yields a confidence that lies between zero and one. Zero implies a completely broken design where every instruction fails. One only implies that the design may not have any problems. Note that even though we have full confidence in the design we do not rule out the possibility of future failures. We can only say that we have not seen any failures, so we believe that the part will continue to function properly.

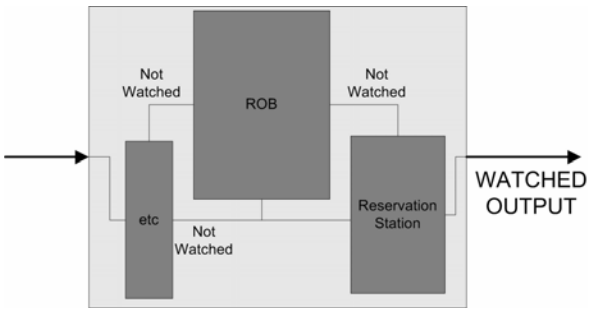

Figure 1. Modules may go unwatched by the checker

2.3 Failure Statistics

We must keep track of statistics about failures in the system in order to calculate the failure rate of a design. This is done with the help of runtime checkers that watch for failures. The source module is marked with an error when a module failure is caught by the onboard checker. We can keep track of these errors, whether the checker fixes them or not, to give a failure rate for use in design confidence calculation. In this manner we can calculate design confidence not just for the system as a whole, but also for individual modules in the design.

Because we base our failure statistics (and thus our design confidence) on the number of errors seen, our confidence metric is a function of the environment the system is in at runtime. This is one of the benefits of runtime checker: the system may reach many states that it would not see during normal lab tests. Conversely, we cannot give a firm confidence in the design because the design will never run through all types of programs.

An additional challenge of calculating failure statistics comes from the inability to watch every output in a design. How should we assign failure statistics to a circuit we cannot check directly? See Figure 1 for an example of this situation. In this example, the ROB module is completely contained in a watched module, but it may still be erroneous. We present plans for assigning blame to unwatched modules in the following section.

3. MODULE-LEVEL PROBABILISTIC DIAGNOSIS

3.1 Overview

There are significant barriers to diagnosing design defects in an automated fashion. In high-availability systems we can typically not afford to take the system offline to perform a complete diagnostic procedure. The same can be said for real-time systems. Even for systems without these requirements, it can be difficult to automatically pinpoint the root cause of a design defect and institute a workaround.

We propose a method of estimating the cause of design failures when a complete diagnosis is not feasible. In our method, a partial high-level diagnosis is performed while the system remains online; modules are identified as faulty with certain probabilities. These probabilistic failures are tracked by the failure statistics, just as with actual failures. Thus the statistics provide an indirect measure of a design's level of defectiveness, which in turn determines our level of confidence. It is important to measure confidence at the module level in order to feed information back to module designers and verification team, and just as important, to allow reconfiguration of systems where modules act as the field replaceable units.

3.2 Weighted-Graph Representation

To enable module-level probabilistic diagnosis we build a directed weighted graph of the system, similar to a causal network [8]. The nodes of the graph represent modules, and links represent signals flowing from one module to another. In our contribution, the links are weighted to indicate the importance of the interconnection. In other words, they indicate how much responsibility a source module is assumed to have for failures detected at the destination module. Some methods for determining the weights are discussed in the following section.

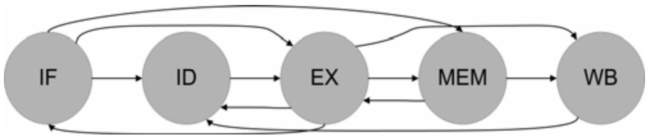

Figure 2. Directed weighted graph model of an example system. Nodes have an implied link to themselves.

3.3 Methods for Determining Weights

We propose a systematic calculation of weights based upon an analysis of the design structure. In particular, we recommend computing the contributions of each module to the logic cone that feeds a checker. As a heuristic, we treat each module as a black box and tally the number of signals connecting each pair of modules. The weighting function for a link between module *i* (source) and module *j* (destination) is as follows.

$$w_{ij} = \begin{cases} 1 & \text{if module j has a checker and i=j} \\ X & \text{(don't-care) if module j has no checker} \\ \sum 1 / (FO_s \times (\#\text{ inputs to j})) & \text{for all inputs } s_{i\rightarrow j} \end{cases} \qquad (2)$$

If a module has a checker, it is implicated with an integer weight of 1 whenever a fail is detected by that checker. If a module has no checker, the weight of an incoming link is a don't-care because fails can never be detected at that destination.

The most interesting case is for links from a module without a checker to another module that has one. In such a case, we compute the proportion of responsibility associated with the source module, by computing the proportion of signals feeding into the destination module. If 30% of the inputs to module *j* come from module *i*, then module *i* will be implicated at a level of 0.30 for every failure caught at module *j*. There is one more modification we must make, and that is to divide by the fan-out of each input. If a line fans out to 3 modules with checkers, then its individual contribution to the weight is divided by 3, so the module will not get implicated multiple times. Finally, we perform a summation over all of the inputs to determine the weight for the link.

3.4 Implication Matrices

The full set of weights can be stored in an implication matrix. Table 1 illustrates an example matrix for a system with 3 modules (A,B,C). Module C does not have a checker, so the implication column for C is a don't-care. The columns for A and B indicate how much the source modules are implicated for failures detected in A and B. As an example, when a failure is detected at module A, module C is implicated at a level of 0.8.

Table 1. Implication matrix for a system with 3 modules. Only modules A and B have associated checkers.

Src \ Dest	A	B	C
A	1	0.9	X
B	0.2	1	X
C	0.8	0.1	X

In the general case of a system with more than 1 checker per module, the matrix would contain *m* rows for the modules and *c* columns for the checkers.

As long as a system's interconnections are static, then the implication matrix can be accurately determined at design time. The matrix values would then be encoded into the confidence logic for use at runtime.

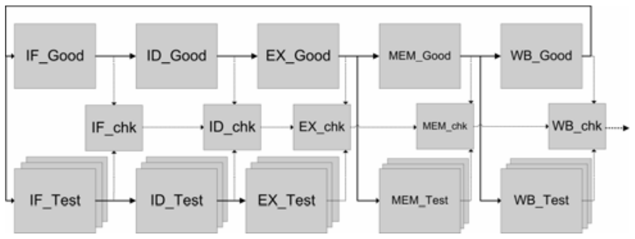

Figure 3. Experimental Setup. The known good designs at the top are compared to the DUT at bottom

4. EXPERIMENTAL DATA

4.1 Experimental Setup of a Small Design

We tested our confidence estimate ideas against a small-to-moderate sized Verilog design run in simulation. The design itself was a five-stage pipeline split into a module for each stage and an encompassing module for the pipeline and its registers. To simulate a runtime checker, we kept a known-good version of each stage in the pipeline and compared its outputs to the DUT version of that respective stage. If there were differences in the output signals we would flag that stage and propagate the error along with the instruction until commit. Figure 3 shows this setup.

Our test designs were modeled on the known-good design with a range of errors manually inserted by the authors. Some errors were modeled after bugs found in the original design of the pipeline, while others were more general. Two versions of each of the five pipeline stages were made, leading to 32 possible configurations of the whole system.

The design was run through a 50,000 instruction suite of small tests, and the 32 designs ranged from 33 to 177 erroneous instructions. The same 50,000 instruction test was run for every configuration during these tests.

To expedite the large number of test runs required of these experiments, we created a Java program which implemented our refinement heuristic. The program utilized the failure data that was generated by our Verilog simulations. The program would start with a given system configuration and track the failure statistics (and design confidence values) for each of the 10 versions (2 versions of each module). It would then execute our reconfiguration decision procedure (heuristic) and move on to the next step with a new system configuration. The decision procedure would continue for 15 steps. Finally, this same procedure was run for all possible starting configurations, and the results were averaged over all runs.

4.2 Experiment 1

4.2.1 Hypothesis

Starting from any system configuration, our design confidence scheme and refinement procedure will lead to an improved system design in a small number of refinement steps. The number of steps will be much less than the number of unique system configurations; it will linear in the total number of module versions.

Table 2. Example refinement sequence. By improving module-level confidence, the system-level pass rate also improves, after temporary decreases at steps 2 & 3.

Step	System config.	System pass/fail rate	Refinement decision
0	00000	0.9968	WB: v0->v1
1	00001	0.9980	ID: v0->v1
2	01001	0.9979	MEM: v0->v1
3	01011	0.9978	EX: v0->v1
4	01111	0.9986	IF: v0->v1
5	11111	0.9991	no change

4.2.2 Description

This experiment shows that if a module's outputs are monitored, it is possible to accurately find the most erroneous individual modules in most cases. Beginning at any of the possible 32 starting configurations we would switch out the single module that would increase the overall system confidence the most. Table 2 shows an example refinement sequence starting from configuration 0000 while Figure 4 shows the confidence in a pair of modules from the starting sequence 01010.

We test the idea that even though bugs in one module may cause failures in subsequent modules, we can correctly identify the worst modules eventually and find a good system configuration.

This experiment shows the best-case situation for our method of switching modules. It will be useful in future tests to compare their correctness over time versus this case.

4.2.3 Results

Figure 5 details the results of experiment 1 averaged over all starting states. The average correctness of the designs rises over the course of the refinement, and by steady-state, 2/3^{rds} of designs are in the optimal configuration. Of those that are left, their correctness has still improved from their initial configurations. The average system performance converges after about 10 steps, which matches the total number of design versions.

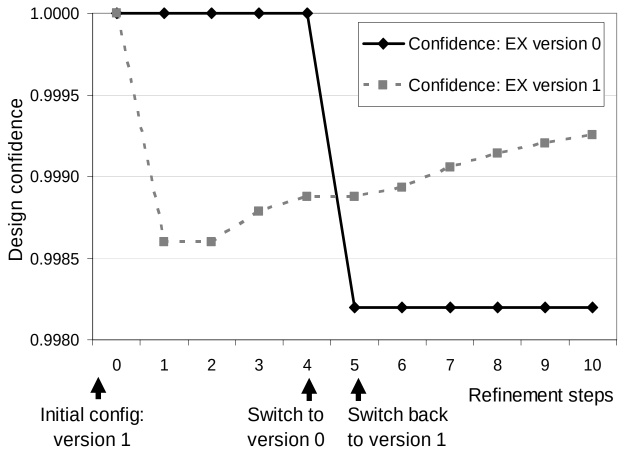

Figure 4. Confidence in two module versions over time

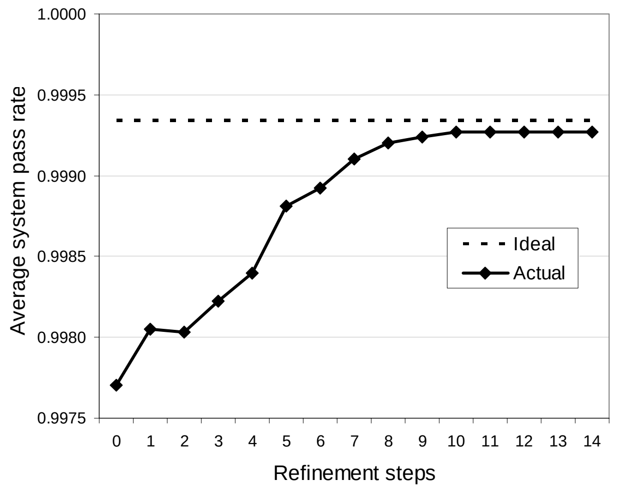

Figure 5. Experiment 1 Results. Pass-rate of system on average from all 32 start-states increases over refinements

4.3 Experiment 2

4.3.1 Hypothesis

In a system with only partial checking, probabilistic diagnosis (with implication weights=1) will provide better system reliability than an alternative method without any back-propagation of errors.

4.3.2 Description

In this experiment we test the efficacy of probabilistic diagnosis in a system with only partial checking. Specifically, checkers are enabled for only three of the five stages (EX, MEM, and WB). Two of stages (IF and ID) have no checker, so the confidence of the respective design versions must be inferred indirectly.

Similar to Experiment 1, we simulated the system for all possible starting configurations, using the given refinement heuristic. We did this with probabilistic diagnosis turned off (e.g. zero implication, weights of unchecked modules are 0), and again with probabilistic diagnosis turned on (implication with weights of 1).

4.3.3 Results

The results indicate that system reliability is in fact improved with probabilistic diagnosis. Please refer to Figure 6. Without probabilistic diagnosis, the overall system fail rate converges to 62.78 fails per test, while with probabilistic diagnosis, the system fail rate reaches 47.97, a 24% improvement.

Note that the reliability with zero implication initially outperforms unit implication. We feel that this is because a locally optimum configuration can be found more quickly, due to the smaller search space. Specifically, without implication, the IF and ID modules never encounter any failures, and retain full design confidence (1.0). Thus the system never reconfigures those modules, and only needs to reconfigure the remaining three modules. As a result, with no implication, the system can achieve a local optimum in fewer than 8 refinement steps. With unit implication, the system is able to estimate the design confidence of the IF and ID versions, and has a larger space of system configurations. Unit implication performs better than no implication after about 8 refinement steps, which matches our intuition.

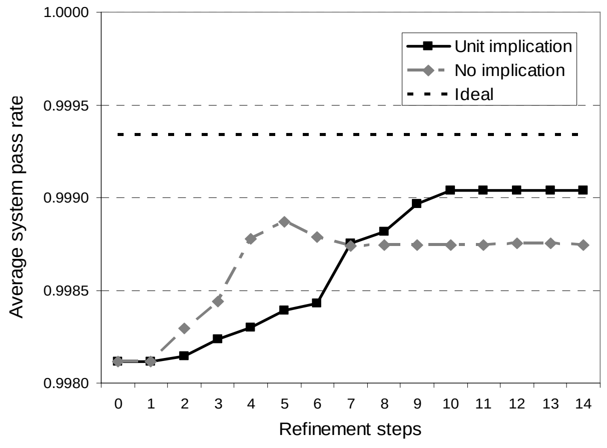

Figure 6. Experiment 2 Results. Unit implication outperforms zero implication after just a few steps.

4.4 Experiment 3

4.4.1 Hypothesis

Proportional weighting will provide better system reliability than unit or zero weighting.

4.4.2 Description

We simulated two proportional weighting schemes and compared the results to unit and zero weighting (experiment 2). As before, for each scheme we simulated all possible starting configurations, and used the given heuristic for refinement decisions.

The first weighting scheme counts the numbers of input *lines* (individual bits) feeding a module with a checker. The second scheme counts the number of input *signals*, where data busses are treated as a single signal. The idea is that not all lines are created equal; it is counterproductive to give disproportionate weighting to data busses when small control lines are often more important to the correct operation of the design.

Table 3. Implication matrix for proportional weighting by individual bits

Src\Dst	IF	ID	EX	MEM	WB
IF	X	X	0.321	0	0.474
ID	X	X	0.465	0.015	0.044
EX	X	X	1	0.985	0.007
MEM	X	X	0.214	1	0.474
WB	X	X	0	0	1

Table 4. Implication matrix for proportional weighting by signals (data busses collapsed to 1 signal)

Src\Dst	IF	ID	EX	MEM	WB
IF	X	X	0.702	0	0.111
ID	X	X	0.277	0.4	0.667
EX	X	X	1	0.6	0.111
MEM	X	X	0.021	1	0.111
WB	X	X	0	0	1

4.4.3 Results

The results indicate that both proportional weighting schemes outperform unit and zero weighting. Furthermore, the scheme with proportional weighting by signals (40.41 fails per test) performs better than weighting by individual lines (45.56 fails per test) by 11.3%.

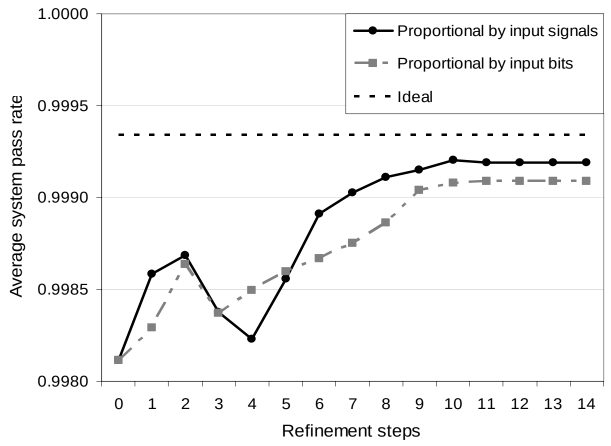

Figure 6. Experiment 3 Results. Collapsing data busses into a single signal results in a better final design.

5. FUTURE WORK

We feel there are opportunities for additional research into the types of design defects that tend to escape verification and are uncovered at runtime. Specifically, we would like to have a better taxonomy of these defects, including local and interoperability defects, and a better understanding of frequencies of occurrence on leading-edge systems. While we have anecdotal evidence and first-hand experience with these types of defects, low-level systematic data is lacking [2].

We would like to see our approach applied to non-processor systems in the future. ASICs with a modular design and a method of runtime verification would be prime candidates. The particular method of checking could be a checker processor, assertions, watchdog timers, etc. Deriving implication values for the checkers would be more challenging than in the system analyzed here.

Further study would be required to understand the possibilities for proportional weighting heuristics. For instance, our scheme depends on signal fan-out, so systematic experiments could be performed on signals with various fan-outs. Additionally, we believe our proportional weighting heuristic could be significantly extended, for instance by analyzing a logic cone at finer levels of granularity (i.e. levels lower than a module).

We feel that the techniques presented in this paper would be well suited for research into hardware design diversity. The idea of design diversity is to employ redundancy at the design level (via multiple unique variants of a design) to protect against design defects. Prior research has demonstrated that certain types of systems can benefit from design diversity, including systems with strict requirements for high-availability, safety, or autonomy. One incarnation of diversity called N-version programming [5] has the drawbacks of high cost and a possibility of correlated errors in multiple variants. For modern distributed systems, a more promising technique is N-self-checking, which allows N variants to operate independently, each performing checking locally. It remains to be seen whether the cost of developing N variants can be mitigated by the greater availability of IP libraries, open-source designs, and the like. While typically used for software, design diversity holds promise for reconfigurable hardware designs as well [11][6]. A method is required for evaluating the utility of the various designs; the methods presented here may be along the lines of what is needed.

Finally, while focused on design defects, out techniques may be just as applicable to measuring a design's inherent predisposition to soft errors. Future work could determine the feasibility of our high-level runtime confidence and refinement, as an alternative to existing techniques that deal with soft errors at lower levels.

6. CONCLUSIONS

We have presented a method for estimating the correctness of a processor-based design and its constituent modules based on runtime error information. Our experiments have shown that by watching the outputs of less than all modules we are able to estimate which modules in a design are erroneous and give a quantified confidence in them.

Based on our experimental data we feel that our metric for design confidence will be useful in a number of endeavors from design verification to autonomous system reconfiguration.

Our module-level implication weighting is a method for assigning partial blame to modules which are not under direct test. This will allow our confidence metric to be used in systems without full checker coverage. We hope in future work to find better weighting schemes to more accurately model the complexities of designs with many interactions.

7. ACKNOWLEDGMENTS

We would like to thank Professor Valeria Bertacco and the students of the EECS 578 course for the opportunity to pursue this project. We thank Prof. Mark Brehob and the EECS 470 students for the Verilog code which acted as our starting point. Ken would like to acknowledge the support of the NASA GSRP Fellowship.

8. REFERENCES

[1] Austin, T., "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design", *Micro-32* (Nov 1999)

[2] Avizienis, A., and Yutao He, "Microprocessor entomology: a taxonomy of design faults in COTS microprocessors", *Dependable Computing for Critical Applications 7*, 3 – 23 (1999)

[3] Bertacco, V. *Scalable Hardware Verification with Symbolic Simulation,* 1st ed., Springer, New York (2005)

[4] Chatterjee, S., Weaver, C., and Austin, T., "Efficient Checker Processor Design", *Micro-33* (Nov 2000)

[5] Chen, L., Avizienis, A., "N-version Programming: A Fault Tolerance Approach to Reliability of Software Operation", *Fault-Tolerant Computing, Twenty-Fifth International Symposium on*, 113-119 (Jun 1995)

[6] Greco, J., Cieslewski, G., Jacobs, A., Troxel, I.A., and George, A.D., "Hardware/software Interface for High-performance Space Computing with FPGA Coprocessors", *Aerospace Conference*, 2006 IEEE (2006)

[7] Malka, Y., A. Ziz, "Design Reliability-Estimation through Statistical Analysis of Bug Discovery Data", *Proc. Design Automation Conference* (June, 1998)

[8] McDermott, R. M., and Stern, D., "Switch Directed Dynamic Causal Networks – A Paradigm for Electronic System Diagnosis", *24th ACM/IEEE Design Automation Conference* (1987)

[9] Nacif, J.A.M., *et al*., "The Chip is Ready. Am I done? On-chip Verification using Assertion Processors", *Proc. VLSI-SOC'03* (2003)

[10] Russell, S., and Norvig, P., *"Artificial Intelligence: A Modern Approach"*, 2nd ed., Prentice Hall (2003)

[11] Shriver, P. M., Gokhale, M.B., Briles, S.D., Kang, D.-I., M. Cai, McCabe, K., Crago, S.P., and J. Suh, "A Power-Aware, Satellite-Based Parallel Signal Processing Scheme," *Power Aware Computing*, Series in Computer Science, Kluwer Academic/Plenum Publishers, New York, NY (2002

[12] Wagner, I., Bertacco, V., and Austin, T., "Shielding against design flaws with field repairable control logic", *Design Automation Conference, 2006 43rd ACM/IEEE.* Page(s):344 – 347 (July 2006)

[13] Weaver, C., and T. Austin, "A Fault Tolerant Approach to Microprocessor Design", *Proc. Dependable Systems and Networks*. 411-420 (2001)