BugMD: Automatic Mismatch Diagnosis for Bug Triaging

Biruk Mammo[1], Milind Furia[1], Valeria Bertacco[1], Scott Mahlke[1], Daya S Khudia[2]
[1]University of Michigan, Ann Arbor, MI, USA
[2]Intel Corporation, Santa Clara, CA, USA
{birukw, furia, valeria, mahlke}@umich.edu
daya.s.khudia@intel.com

ABSTRACT

System-level validation is the most challenging phase of design verification. A common methodology in this context entails simulating the design under validation in lockstep with a high-level golden model, while comparing the architectural state of the two models at regular intervals. However, if a bug is detected, the diagnosis of the problem with this framework is extremely time and resource consuming. To address this challenge, we propose a novel bug triaging solution that collects multiple architectural-level mismatches and employs a classifier to pinpoint buggy design units. We also design and implement an automated synthetic bug injection framework that enables us to generate large datasets for training our classifier models. Experimental results show that our solution is able to correctly identify the source of a bug over 70% of the time in an out-of-order processor model. Furthermore, our solution can identify the top 3 most likely units with over 90% accuracy.

1. INTRODUCTION

For most of today's industrial designs, a majority of design development time is spent on verification [8]. Software simulation tools frequently cannot tackle the complexity of system-level validation, forcing design teams to rely exclusively on acceleration and emulation platforms. Unfortunately, these platforms are plagued by limitations on the design's signals that can be monitored during validation [16], often consisting only of the architectural-level state updates [4]. Once a bug is detected in such an environment, information to accurately pinpoint its origin is scarce. In addition, such bugs tend to hide in extremely complex corner cases, often detected long after their occurrence. Engineers engage in a time-consuming, mostly ad-hoc, triaging process to identify the design units responsible for a bug and then carve a unit-level analysis to root-cause it. This challenging process of locating bugs in complex designs is one of the factors for the rising rate of escaped functional bugs, as evidenced by bugs reported in processor errata [10, 3].

In this work, we propose a **high-accuracy, automatic bug triaging solution for low-observability system-level validation environments** that slashes the amount of time spent on triaging bugs. Our solution, called BugMD (**Bug M**ismatch **D**iagnosis), is designed to operate in acceleration and co-simulation validation methodologies, where bugs are detected by comparing the state of the design-under-verification (DUV) against a high-level instruction set simulator (ISS). ISSs are capable of executing an assembly program at very high performance (only 2-3 times slowdown from the manufactured silicon processor) and are considered the golden reference model to determine correct program output. We leverage this same framework to provide diagnostic/localization information, in addition to the detection capabilities that are already part of this approach. BugMD, illustrated in Figure 1, is capable of identifying the most likely unit(s) responsible for a bug, based on the manifestations observed at the architectural state of the design, *i.e.*, all those states that can be observed from the firmware/software layer: memory content, memory mapped registers for accelerators and IP components, and architected state for processors. Ater detecting a bug manifestation by comparing the architected state of the DUV with that from the ISS, we **update the ISS state with the erroneous DUV state**. This way, both models are pursuing the same incorrect execution, and we can detect

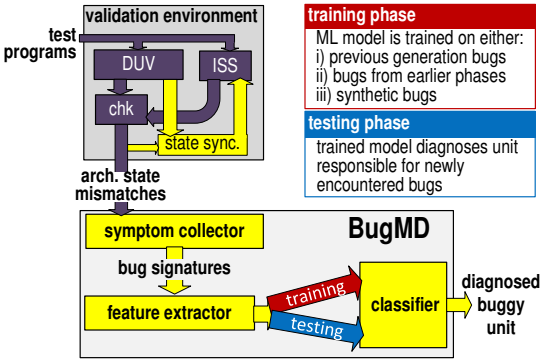


Figure 1: **BugMD overview**. In co-simulation and acceleration methodologies, the design-under-verification (DUV) executes in lockstep with a golden instruction set simulator (ISS) (purple). BugMD (yellow) augments this methodology with a mechanism to update the ISS state upon a bug manifestation, so that the two models are re-synchronized and subsequent manifestations of the bug can be observed. The complete set of bug manifestations logged by the symptom collector are assembled into a "bug signature", which is then transformed into a "feature vector" by the feature extractor, and transferred to the classifier for diagnosis.

ICCAD '16, November 07-10, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2966986.2967010

subsequent impacts of that same bug. Note that in a traditional validation setting, localization is attempted based on the first, single mismatch between the two models. For each bug manifestation, we log a new bug symptom, which we aggregate into a **bug signature**. The collected symptoms are compressed into feature vectors that are transferred to a classifier model [1] for analysis and localization. Our classifier is trained on known bugs, *e.g.*: bugs fixed in previous design generations and bugs fixed in earlier phases of verification. In addition, we created a **systematic and efficient infrastructure to train the BugMD classifier with synthetic bugs** when those from prior design generations or design phases are not available. Our solution can be deployed alongside current methodologies with minimal changes: we simply need to be able to update the ISS state with that of the DUV after a bug manifests, so that we can gather additional symptoms for that same bug.

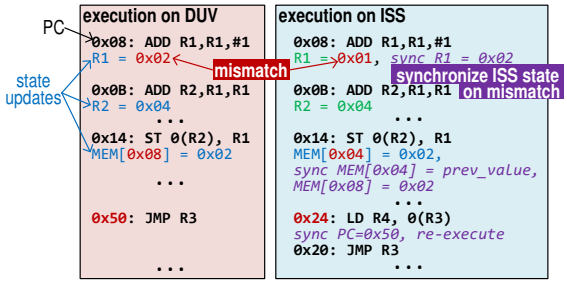
Contributions: In this work, we propose BugMD – a novel approach for localizing bugs using multiple architectural level bug manifestations, which we refer to as bug signatures. We introduce a state-synchronized ISS co-simulation strategy to construct bug signatures by **collecting multiple independent bug symptoms beyond the first manifestation of a bug**. We identify features and classifier algorithms best suited for pinpointing bug sites from bug signatures. Furthermore, we introduce **an automated synthetic bug injection framework to train our classifier model.**

2. SYSTEM DESCRIPTION

2.1 Architectural bug signatures

Once a bug manifests at the architectural level, it corrupts program state. This state corruption often cascades down to subsequent instructions in the program, leading to a permanent deviation of the DUV execution from that of the ISS. As a result, conventional debugging techniques rely exclusively on the first bug manifestation to diagnose a bug. However, if cascading corruptions could be prevented and multiple manifestations of the same bug could be observed as several distinct symptoms, interesting patterns could emerge. Some patterns may be identified by investigating multiple manifestations of a bug over several distinct test runs: for instance, a bug may have similar symptoms on different test cases. More interestingly, patterns may exist among the multiple symptoms obtained from a single test execution.

Consider the example in Figure 2. A bug in the dispatch logic of a 4-wide, out-of-order core grabs the wrong register value for one of the four instructions it dispatches in a cycle. The first manifestation of the bug is a mismatch in the value being written to a register. For the engineer that observes this mismatch, there could be several reasons for it: the execution logic may have performed the wrong computation, the writeback logic may have corrupted the result, the instruction decoding logic may have provided the wrong operands, *etc.* In the worst case, the engineer would have to analyze traces of internal design signals for as far back as thousands of cycles (*e.g.*, if the affected instruction was a load operation that missed in the caches) in order to discover what went wrong. However, if we synchronize the ISS state with the state of the DUV and resume execution (and comparisons), patterns emerge that would help narrow down the likely culprits. Here, the 4-instruction periodicity



bug signature snippet

cycle	instr. count	mismatch type	instr. type	design value	ISS value
100	1	REG VALUE	ALU	0x02	0x01
103	5	MEM ADDR	MEM	0x08	0x04
112	9	PC	CONTROL	0x50	0x24
		...			

Figure 2: **Bug signature example**. The state updates from execution on the DUV are compared with those from the ISS. We record mismatches as symptoms and aggregate them into bug signatures. We synchronize the ISS state to avoid cascading failures.

of mismatches, the diversity of affected instruction types, and the differences between the correct and incorrect values give hints that the bug may be located in the dispatch logic.

BugMD identifies and collects bug symptoms from each mismatch. Mismatches are of the form: i) a write to the wrong register or memory location, ii) a wrong value written to a register or memory location, or iii) a mismatch in the expected program counter. In addition, we consider high-level failures, such as divide-by-zero errors, program hangs, invalid traps, page faults, *etc.* Previous works have utilized these high-level failures to detect the presence of transient and permanent faults in hardware [11, 12, 17]. A symptom includes the type of mismatch, the current simulation cycle, the instruction count, the state update values from the design and the ISS, and the instruction type. For each test execution, we aggregate the symptoms from multiple mismatches and failures into a bug signature. This aggregation continues until we complete the execution of the test program, the program terminates due to abnormal conditions (*e.g.*: segmentation faults, division by zero, *etc.*), or until a fixed number of instructions are executed after the first bug manifestation.

The ISS modifications that BugMD requires are simple enhancements to the ISS data structures used to maintain architected state. Firstly, all ISS architectural state updates should be made visible to BugMD's symptom collector. Ability to read ISS state updates is a feature that is already available as it is required for comparing DUV and ISS states. Secondly, the ISS should be enhanced to allow modification of its architected state from BugMD's synchronization mechanism. This enhancement should be fairly straightforward to implement for anybody with even moderate familiarity with the inner workings of the ISS. Finally, note that our ISS synchronization approach may trigger unexpected behaviors in the ISS. We expect the ISS designers to ensure that their ISS can execute correctly from any legal architected state. However, a synchronization event may lead the ISS to an illegal state, which may result in side-effects from executions that are incorrect or not defined by the architecture. Examples include the execution of unimplemented instructions, invalid system calls, ISS crashes, *etc.* BugMD treats these side-effects as bug symptoms.

We expect the effort required to implement ISS modifications to be minimal. Moreover, a single ISS is typically used over multiple design variants and generations; thus, the one-time modification effort is amortized over the lifetime of the ISS. It took one graduate student less than a week's worth of effort to enhance the ISS used in our experimental framework with the modifications described above. It took even less effort to modify another ISS for a subset of the Alpha instruction set architecture (ISA). Note that a new ISS needs to be modified for use with BugMD. The architectural bug signatures generated by the modified ISS, however, are architecture independent. Hence, BugMD's feature extractor and classifier do not require any modifications for use with new ISSs or designs.

2.2 Extracting features from signatures

Each manifestation of a bug can be characterized by the nature of the mismatches it generates. For instance, an instruction may write the wrong value to the wrong register. This generates two distinct mismatches, a register index mismatch and a register value mismatch, each with its own characteristics. The difference between the wrong and correct value, the hamming distance between them, and the locations of the wrong bits are some examples of characteristics we may record as symptoms for the manifestation.

We represent the multiform, variable-length bug signatures with fixed-length, real-valued vectors of features. The quality of the classification depends heavily on the quality of the characteristics chosen to represent the bug signatures and the vector length (also known as input dimensionality). After several experiments to identify good feature vectors with low input dimensionality, we chose the feature extraction methodology, illustrated in Figure 3, which captures the differences between the DUV and ISS values as well as the distribution of mismatch types and instruction types among symptoms in a given bug signature. Each feature is a summary of symptoms observed within a user-specified window of instructions (window of 10,000 instructions in our experiments), starting from the first manifestation. For each symptom, we first compute the differences and hamming distances between the DUV and ISS states. We then compute features that are arithmetic means and standard deviations of the differences and hamming distances for each pairwise occurrence of mismatch and instruction type. In addition, we include the total number of symptoms in the signature and we compute the fraction of pairwise occurrences of mismatch and instruction types. Using this feature extraction approach in our experiments, we extracted 470 features that enabled good localization accuracy, while keeping feature vector lengths reasonable.

2.3 Classifier design

The BugMD classifier plays the most important role in identifying likely bug sites. We investigated several machine-learning algorithms to identify one that best suits our needs. Note that the selection of a classifier and its accuracy are highly dependent on the set of features selected to represent bug signatures. Thus, we investigated several different approaches for generating features, along with classifier algorithms, before selecting the feature set described in Section 2.2, which demonstrated the best overall performance for all applicable classifier models. Below is a brief discussion of the classifiers we investigated.

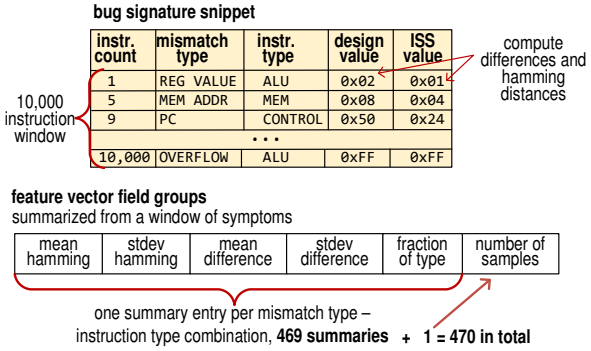


Figure 3: **Feature vectors**. Each feature vector is a summary of symptoms within a user-specified window. For each combination of mismatch and instruction type, the average and standard deviations of the hamming distances and differences between the DUV and ISS states are computed. The distribution of each combination and the total number of symptoms observed in the window are also computed.

Decision Tree learns simple rules from training data. This classifier performed well on our feature sets, despite its simplicity. It is the fundamental building block of Random Decision Forests, which we ultimately selected.

Random Decision Forest constructs multiple Decision Trees, each from different random training samples, and takes the majority vote as the final output. We chose this model as it performed better than all other techniques we investigated. We hypothesize that this method captures the non-linear boundaries between bugs from different units better. Moreover, by aggregating results from multiple decision trees, it can better tolerate errors due to overfitting.

Linear Support Vector Machine (SVM) is one of the most used machine-learning algorithms for classification. For our application, however, it performed poorly and was slow. We hypothesize that this occurred mainly because our data points are not linearly separable.

Simple Feed-forward Neural Network is a network of artificial neurons that can approximate any function. We investigated several options for network parameters and found cases where our network learned better than SVM and was comparable to Decision Trees. However, it was always inferior to Random Decision Forest.

Custom Multi-layer Neural Network is a network we designed specifically for our application, loosely based on convolutional neural networks, which are commonly employed in image classification. Unlike the feature sets used for the other techniques, the inputs for this network are much closer to the raw bug signatures; the network attempts to learn the correlations between symptoms by itself. This classifier led to better localization accuracy than SVM, but slightly worse than our simple feed-forward network.

In addition, we also investigated a K-Neighbors classifier, a Naïve Bayes classifier, an Ada Boost classifier, a voting classifier that aggregates multiple types of classifiers, and a custom hierarchical classifier. The Random Decision Forest classifier outperformed every other classifier we investigated and is our classifier of choice. In setting up the Random Decision Forest, we configured the number of trees to be anywhere between 32 and 1024, without noticing any significant difference in the quality of results. Our experience indicates that the actual process of training the classifier takes only minutes.

2.4 Synthetic bug injection framework

Our classifier model is trained with a database of known bugs and their corresponding feature vectors, derived from multiple buggy executions. This training database can be developed from bugs that were fixed in previous design generations or during earlier phases of the verification effort. Engineers label each training input with the design unit found to be responsible for it. Note that only the engineer needs to have the design-specific knowledge to label each input – BugMD's classifier is unaware of the meaning behind each label. Once trained, a classifier be deployed for use with previously never seen before signatures.

Typically, a larger training database results in better training. In situations where there are not enough known bugs for training, a set of synthetic bugs can be used in their place. We developed a synthetic bug injection framework, inspired by software mutation analysis techniques [18], that randomly injects mutation bugs into gate-level netlists of the design units. Our tools first synthesize each design unit into a technology-independent, gate-level netlist. They then parse the netlist to select random gates and insert a mutation for each gate that takes the same inputs as the original gate but has a different functionality. For example, a 3-input OR gate may be the chosen mutant for a randomly selected 3-input AND gate in the design. For each original-mutation pair, a multiplexer is inserted to select between the two outputs. Each multiplexer represents a synthetic bug in the design; to activate a bug, its associated multiplexer is set to choose the output from the mutant gate.

Our mechanism for injecting synthetic bugs and collecting training data is extremely low-cost and low-effort. The process of injecting synthetic bugs and collecting their symptoms can be completely automated, allowing it to be performed without taking resources away from other verification efforts. Moreover, the process can start before the availability of the whole design. For instance, it is common practice to develop partial system models to validate specific features: we can thus use these models to gather bug signatures for bugs injected in the units included in these partial models. In addition, we can leverage simulation independence to gather signatures for multiple bugs concurrently. Note that, much like in training with real bugs from prior designs, synthetic bug signatures can also be shared across multiple generations and variations of a design. This further amortizes the cost of generating synthetic bug signatures.

The type of functional bugs we are interested in are usually introduced at the behavioral-level. Note that our gate-level synthetic bug model is an approximation of bugs at the behavioral-level. In addition, synthetic bugs may interact with real, hidden bugs in the system causing unpredictable outcomes. Investigating other synthetic bug models and the interaction with other real bugs is left for future work.

3. EXPERIMENTAL RESULTS

We developed our solution on 3 different designs, implementing two ISAs. For lack of space, we report detailed results for only one of our experimental frameworks. This framework comprises a 4-wide, out-of-order processor design described in Verilog and a C++ ISS for the system-level validation flow. BugMD includes a symptom collection and synchronization harness, a synthetic bug injection toolkit, feature extraction scripts, and a machine learning script implemented using Python's scikit-learn [14] library. We only report results obtained using the Random Decision Forest classifier configured to comprise 64 trees. Our processor design, which implements the SimpleScalar PISA instruction set, was generated using the FabScalar toolset [5]. We modified the SimpleScalar-based ISS that is bundled with FabScalar for use with BugMD. For our test programs, we used several distinct sections of the bzip benchmark that ships with FabScalar. We opted not to include other benchmarks since we found the diversity in the different sections of the bzip benchmark to be sufficient for our experiments. We grouped the design's Verilog modules into the 12 distinct units described in Table 1.

unit	# cells	description
FETCH1	598,671	Instruction fetch logic, stage 1
FETCH2	8,852	Instruction fetch logic, stage 2
DECODE	4,654	Instruction decoding logic
INSTBUF	24,483	Buffer for decoded instructions
RENAME	10,232	Register renaming logic
DISPATCH	998	Instruction dispatch logic
ISSUEQ	39,994	Issue logic and queue
REGREAD	27,205	Physical register file
EXECUTE	24,827	Integer execution logic
LSU	26,129	Load-store-unit
RETIRE	104,069	Reorder buffer and writeback logic
MAPTABLE	3,036	Map table for register renaming

Table 1: **Design units.** The FabScalar design modules were grouped into 12 hardware units. We estimated the size of each unit by counting the number of cells in its synthesized netlist. Units with large storage structures have the largest sizes: *e.g.*, FETCH1 contains the branch target buffer. Engineers label bug signatures used for training with their corresponding buggy unit.

We grouped the subset of the PISA instructions that FabScalar implements into 6 types, namely Control, ALU, Multiply/Divide, Load, Store, and Other. Table 2 lists the groups of mismatch types that we log in our bug signatures. For certain mismatch types, we observed that some features generated by our feature extraction methodology were not useful (*e.g.*, differences for x-prop group of mismatch types and instruction types for termination mismatch types), and thus were excluded from our feature set. We collected symptom signatures until our tests completed/terminated or until they executed 10,000 instructions after the first mismatch. Our resulting feature vectors were 470 features long.

Using our synthetic bug injection framework, we injected 590 bugs in each of the units in Table 1 adding up to 7080 bugs in total. Since we did not have a database of previously fixed bugs for our DUV, we relied exclusively on these synthetic bugs, both to train BugMD and to evaluate its diagnosis capability. To collect bug signatures, we activated one bug in the design at a time and executed test programs, both on the DUV and the ISS. We ran 6 distinct test program executions while activating a single bug per execution and collected over 40,000 bug signatures. We then divided the feature vectors generated from these signatures into disjoint training and testing datasets, ensuring that each unique bug for each unit was exclusively either in the training dataset or the testnig dataset. We trained BugMD using the training dataset and evaluated its localization capabilities using the testing dataset.

group	# of types	mismatched property
value	2	the value of a register or a memory update
address	2	the address of a memory update or the index of a register update
valid	2	the validity of an update
size	1	the size of a memory update
control	2	the program counter and syscalls
bounds	3	out-of-bounds data and instruction accesses
x-prop	10	presence of X bits in updates
abnormal	8	abnormal conditions such as division by zero and ISS anomalies
final	4	simulation termination conditions: normal finish, hang, livelock, and crash
TOTAL	34	

Table 2: **Mismatch types.** Bug symptoms consisted of mismatch types within the groups shown here. In addition to state mismatches, we included erroneous conditions such as division-by-zero and ISS crashes. We had a total of 34 mismatch types.

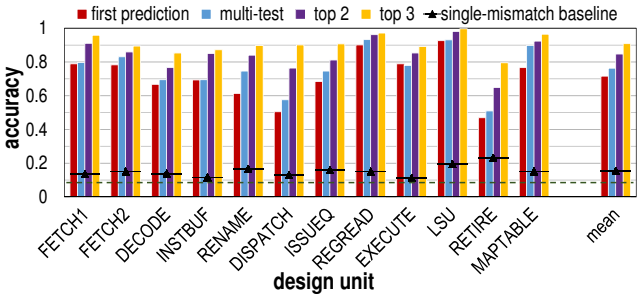


Figure 4: **Breakdown of localization accuracy for each unit**. Our classifier pointed to a single unit with an average accuracy of 70%. This first prediction was further enhanced to 77% if the same bug was exposed by multiple distinct tests and the most common prediction was chosen. BugMD narrowed the search space from 12 units to 3 likely units with a 90% accuracy. The dashed green line indicates the accuracy for a random guess.

3.1 Localization accuracy

We investigated different approaches to boost the accuracy of classification. Figure 4 reports the results of our investigations using a training dataset of 35,352 feature vectors and a testing set of 3,756 feature vectors. We implemented a simple, automated, single-mismatch baseline heuristic that emulates the initial triaging efforts of an engineer who does not have BugMD. This baseline assumes a validation environment where a set of symptoms for only one bug manifestation can be collected and analyzed during testing. We designed our baseline to closely match the initial bug triaging steps that would be taken in the absence of multiple, independent symptoms. Note, however, that in a real-world scenario, a verification engineer would run more tests while observing multiple internal design signals and draw from accumulated experience to refine the estimate from the first attempt. Our baseline heuristic, shown with the black horizontal segments, correctly identified a buggy unit on the first try about 15% of the time on average.

Our classifier correctly identified a buggy unit on the first try about 70% of the times, as demonstrated by the red "first-prediction" bars; 90% of the times, the buggy unit was within the top 3 predictions, as shown by the gold "top 3" bars in the figure. When we ran multiple tests for a bug and took the most common prediction as the final prediction, we achieved an overall accuracy of 77%, shown by the blue "multi-test" bars. Note that a purely random guess would only have an 8.3% chance of correctly localizing a bug. We observed that some units, such as the LSU, had bugs that were often correctly localized whereas others, such as the DISPATCH unit, exhibited bugs that were difficult to localize. The LSU bugs mostly affected just memory operations and were easy to discern. The small size of the DISPATCH unit limited the number of options our bug injection framework had when injecting bugs, resulting in low-quality training. The relationship between unit size and localization accuracy is to be studied in future work. For other units, the variation is due to factors including the diversity of bug signatures from difficult-to-localize units, which made their feature vectors appear to be closer to those from other units than to each other. For instance, feature vectors from bugs in the FETCH1 unit demonstrated a lot of PC and instruction overflow mismatches. Thus, the classifier opted to localize bugs with a large fraction of PC and instruction overflow mismatches to the FETCH1 unit.

BugMD performed significantly better than our single-mismatch heuristic mainly due to the richer information available to it from the multiple symptoms for each bug – our feature sets and the classifier were able to utilize this extra information to discern bug signatures from different units.

3.2 Distribution of localizations

We analyzed the diagnosis outputs from BugMD to understand the distribution of correct and incorrect predictions. We report this distribution in Figure 5. We observed that for all units, correct outcomes far exceeded incorrect localization to any single other unit. For some units, the other unit that they were most frequently misclassified to is fairly reasonable. For example, a FETCH1 bug was often misclassified as a FETCH2 bug, a RENAME bug was often misclassified as a MAPTABLE bug, and vice versa for both. These misclassifications are close enough to the actual bug site that they can be considered useful. However, bugs in DISPATCH and RETIRE were often undesirably misclassified as FETCH1 bugs. This behavior indicates that a non-trivial portion of DISPATCH and RETIRE feature vectors fell within the FETCH1 classification boundary learned by our classifier. We observed that a large proportion of FETCH1 bugs resulted in a PC mismatch, which then led to early termination either due to invalid instructions fetched from the wrong address or out-of-bounds instruction addresses. A fraction of the bugs in the DISPATCH and RETIRE units affected the targets of branch instructions, which also eventually manifested as PC mismatches and early termination. Some bug signatures with early termination had as few as only two symptoms, which made it difficult to identify discerning patterns, leading BugMD to incorrect diagnoses. However, our classifier still localized to the correct unit at least 3 times more frequently than to the incorrect unit. Note that there is no symmetric relationship in localization accuracies: *e.g.*, the percentage of FETCH1 bugs that appear to be DISPATCH bugs is not the same as the percentage of DISPATCH bugs that appear to be FETCH1 bugs. Thus, Figure 5 is not symmetric across the diagonal.

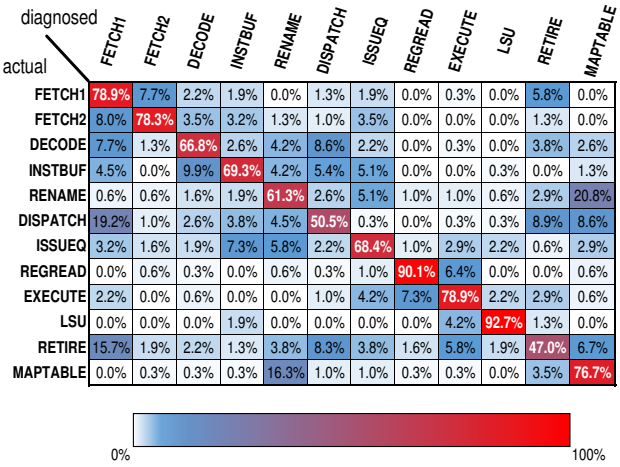


Figure 5: **Distribution of prediction accuracies**. For the bugs in each of our 12 design units, we kept track of the correct outcomes and where bugs were wrongly localized to. Correct outcomes, shown on the diagonal, far exceeded incorrect ones. For some unit pairs, for example FETCH1 and FETCH2, incorrect outcomes could still be considered useful.

3.3 Size of training dataset

Finally, we investigated the impact of training data size on the accuracy of classification. To this end, we partitioned our dataset into several disjoint training and test sets. We trained our classifier using feature vectors generated from the training set and asked it to classify feature vectors generated from the test set. Figure 6 summarizes the results from this study. The mean classification accuracy across all 12 design units increased with increasing training data size. Bugs in certain design units were easier to localize than others. We show the individual accuracy trend for the unit with the most correctly localized bugs – the LSU – and one with the least – DISPATCH. We observed that larger training set sizes generally led to better localization accuracies but the benefits started diminishing after a training size of about 27,000. Highly localizable units, such as the LSU, were not very sensitive to the size of the training set; their bug signatures were quite distinct. The quality of training for smaller units, such as DISPATCH, was limited by the number of gates to inject bugs into; our injection framework had very little room to select and inject high-quality bugs that would result in bug signatures representative of the unit.

4. DISCUSSION

4.1 Execution reproducibility

BugMD relies on validation methodologies that detect bug manifestations by comparing execution on a DUV with execution on an ISS. For such a methodology to work, the bug-free architectural updates from an execution on the DUV should be identical to those from an execution on the ISS. Similarly, test executions should be deterministic: one execution on the DUV should be identical to another. For a buggy execution, we expect the bug to manifest consistently for all executions of the same test program. This is a reasonable expectation in typical validation environments. Adapting BugMD for non-deterministic validation environments is left for future work.

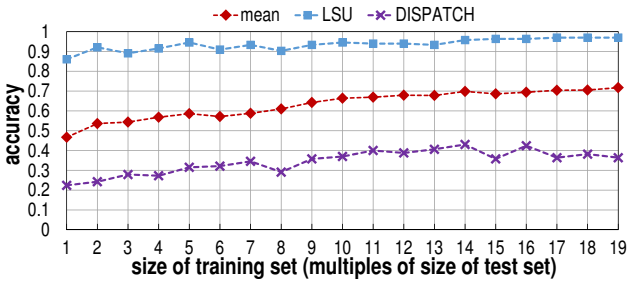


Figure 6: **Sensitivity to training dataset size**. We divided our dataset into disjoint training and test sets. In all cases, the test set contained 1980 feature vectors. The training sets contained feature vectors close to multiples of the test set (1980, 3960, 5940, *etc.*) as shown. Bugs in LSU were localized correctly even with small training sets, while bugs in DISPATCH were difficult to discern.

4.2 Multiple active bugs

Our discussions so far assume that a bug signature contains symptoms from the manifestations of a single bug in the system. This assumption may not hold in a real-world validation environment: there is never a guarantee that only one bug will manifest during an execution. We believe that BugMD can be easily extended to operate in such scenarios. A bug signature that contains symptoms from multiple bugs can be considered to be the signature for a "composite bug" that is an aggregate of the multiple manifesting bugs. If the manifesting bugs reside in different units, this composite bug can be considered to reside in a "composite unit" that is the union of the multiple units. We can then enhance BugMD with the ability to localize to composite units in addition to regular units by treating the composite units just like any other unit. Engineers can then run other tests to discern among the bugs in a given composite bug. An experimental investigation of this approach is left for future work.

4.3 BugMD for post-silicon validation

In post-silicon validation environments, checking after each instruction completion is impractical. While our discussion so far presumes the ability to access and compare architectural level state after every committed instruction, this is not strictly required. With a small change in the checking mechanism, BugMD could require much less frequent checking, allowing it to be deployed in post-silicon. This change entails the use of a binary search-and-compare method for identifying mismatches, as detailed below, by running a test program multiple times, either from the beginning or from a last known clean checkpoint.

We first run a test on the DUV, either to completion or abnormal termination (in the presence of fatal bugs), and compare the final architectural state with the final state from the ISS. If there is a mismatch, we re-run the test for half the number of cycles to completion, and repeat the check. Since the ISS is often not cycle-accurate, we can not simply execute the test program on the ISS for half the number of cycles to get the equivalent execution for comparison. We first need to extract the number of instructions completed on the DUV from on-chip performance counters and then execute the test program on the ISS until the same number of instructions are committed. We repeat this process until we find a cycle where there is no mismatch between the

DUV and the ISS. We take a checkpoint of the DUV and ISS states and continue the binary search forward. Upon detecting a mismatch, we synchronize the ISS state with that from the DUV and continue the search for more mismatches until the desired number of instructions after the first mismatch has been executed. Even though this approach requires a test to be run multiple times, it reduces the number of comparisons by several orders of magnitude. A similar search based approach has been used to successfully isolate faults in industrial designs [2].

5. RELATED WORK

In recent years, a few solutions have been proposed to support bug localization during post-silicon validation. BPS [6] logs measurements of signal activity from multiple executions using a hardware structure added to the design. A clustering algorithm is later used to process these logs to discern among failing and passing tests and identify a candidate set of signals responsible for an intermittent bug. Symbolic QED [13] achieves coarse-grain localization of bugs to processor cores, cache banks, and the crossbar in a multicore SoC by utilizing a combination of bounded model checking, partial instantiations of the design, and test transformations enabled by an extra hardware module added to the fetch stage. Unlike BPS and Symbolic QED, BugMD relies neither on hardware modifications added to the design nor transformations applied to test programs. In addition, it is not limited to post-silicon and can operate on any validation environment that supports DUV-to-ISS state comparisons. Unlike Symbolic QED, BugMD localizes bugs at a much finer granularity, among a larger number of units in out-of-order processor cores.

Several other works have also been proposed that triage bugs by leveraging a combination microarchitectural knowledge [7], microarchitectural or low-level signal traces [7, 15], and SAT-based debugging techniques [15]. Some of these solutions [7, 15] rely on machine-learning algorithms for clustering and/or classification to bin failures and identify their root-causes. BugMD only relies on architectural updates, does not need access to the RTL source code, is entirely simulation/execution-based and does not require any microarchitectural knowledge for its operation.

Finally, Friedler, *et al.* [9] have proposed using information derived from executing a test-case on an ISS to localize the set of instructions responsible for a functional data flow bug in the DUV. Unlike BugMD, this solution focuses on identifying instructions and does not provide any indication of the hardware units responsible for the bugs.

6. CONCLUSIONS

In this work, we presented BugMD – an automatic bug triaging solution. BugMD compares a design's architected state with a golden state from an instruction set simulator to collect multiple symptoms for a single bug in a single test run. These multiple manifestations of bugs form bug signatures that are then passed through a machine-learning backend to obtain a prediction of likely bug sites. To train the machine-learning classifier, we developed a synthetic bug injection framework for generating large training datasets when real, previously diagnosed bug signatures are either unavailable or insufficient. Despite leveraging only architectural-level mismatches without any microarchitectural knowledge, our experiments show that BugMD can identify the correct location over 70% of the time on first try. When considering multiple top candidates, the buggy design unit is among BugMD's top three likely candidates in over 90% of our cases. Future work will investigate a cooperative selection of feature extraction approaches and classifier algorithms to further improve the accuracy of localization.

Acknowledgments. This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA, and NSF grant #1217764.

7. REFERENCES

[1] E. Alpaydin. *Introduction to Machine Learning*. MIT Press, 2014.

[2] M. Amyeen, S. Venkataraman, and M. Mak. Microprocessor system failures debug and fault isolation methodology. In *Proc. ITC*, 2009.

[3] ARM. *Cortex®-A72 MPCore software developers errata notice, product revision r0*, Jan. 2016. v5.0.

[4] D. Chatterjee, A. Koyfman, R. Morad, A. Ziv, and V. Bertacco. Checking architectural outputs instruction-by-instruction on acceleration platforms. In *Proc. DAC*, 2012.

[5] N. Choudhary, S. Wadhavkar, T. Shah, H. Mayukh, J. Gandhi, B. Dwiel, S. Navada, H. Najaf-abadi, and E. Rotenberg. Fabscalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template. In *Proc. ISCA*, 2011.

[6] A. DeOrio, Q. Li, M. Burgess, and V. Bertacco. Machine learning-based anomaly detection for post-silicon bug diagnosis. In *Proc. DATE*, 2013.

[7] M. Farkash, B. Hickerson, and B. Samynathan. Data mining diagnostics and bug MRIs for HW bug localization. In *Proc. DATE*, 2015.

[8] H. Foster. Trends in functional verification: A 2014 industry study. In *Proc. DAC*, 2015.

[9] O. Friedler, W. Kadry, A. Morgenshtein, A. Nahir, and V. Sokhin. Effective post-silicon failure localization using dynamic program slicing. In *Proc. DATE*, 2014.

[10] Intel Corporation. *6th Generation Intel® Processor Family Specification Update*, Sept. 2015. Rev. 001.

[11] D. S. Khudia and S. Mahlke. Low cost control flow protection using abstract control signatures. In *Proc. LCTES*, 2013.

[12] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. Trace-based microarchitecture-level diagnosis of permanent hardware faults. In *Proc. DSN*, 2008.

[13] D. Lin, E. Singh, C. Barrett, and S. Mitra. A structured approach to post-silicon validation and debug using symbolic quick error detection. In *Proc. ITC*, 2015.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12, 2011.

[15] Z. Poulos and A. Veneris. Exemplar-based failure triage for regression design debugging. In *Proc. LATS*, 2015.

[16] A. Veneris, B. Keng, and S. Safarpour. From RTL to silicon: The case for automated debug. In *Proc. ASP-DAC*, 2011.

[17] N. J. Wang and S. J. Patel. Restore: Symptom-based soft error detection in microprocessors. *IEEE Trans. on Dependeable and Secure Computing*, 3(3), 2006.

[18] M. Woodward. Mutation testing – its origin and evolution. *Information and Software Technology*, 35(3), 1993.