Circuit Profiling Mechanisms for High-Level ATPG

Jorge Campos and Hussain Al-Asaad
Department of Electrical and Computer Engineering
University of California, Davis, CA
E-mail: {jcampos, halasaad} @ece.ucdavis.*edu*

***Abstract*—Our Mutation-based Validation Paradigm (MVP) is a validation environment for high-level microprocessor implementations. To be able to efficiently generate test sequences, we need to enable MVP's ATPG to learn important details of the circuit under validation as a means to explore critical new circuit scenarios. In this paper, we present new profiling mechanisms that can exist either as a pre-processor that gathers circuit information prior to the circuit validation process, or as run-time entities that allow MVP to learn from its progressive experience.**

I. Introduction

Our Mutation-based Validation Paradigm (MVP) [1][2][3][4] technology contains the fundamental techniques for analyzing high-level circuit implementations, and is unique in the way it exploits these techniques to validate circuit implementations. MVP's methods help deliver certainty into a circuit validation project in two ways: (*i*) It provides real-time observability into the validation effort through a concurrent mutant simulator that quantifies the circuit coverage (certainty level) at every simulation time-frame, and (*ii*) it employs deterministic circuit analysis techniques that, together with the observability provided by its concurrent mutant simulator, allow MVP's ATPG effort to consistently explore new corners in a circuit's architectural landscape. These contributions enable MVP to mitigate the risk of validation false-positives due to unexposed bugs, which is commonly encountered when random or pseudorandom ATPG fail to travel towards unexplored portions of the circuit under validation.

MVP is a circuit validation tool for high-level hardware descriptions, and its purpose is to provide expert deterministic validation methods to the average design engineer. MVP provides a complete and automated strategy for analyzing high-level hardware descriptions that only leaves the circuit design engineer to decide what portions of the circuit to validate, and not how to validate it. These circuit analysis abilities allow MVP to perform automated white-box circuit validation on high-level RTL descriptions while providing the simplicity of black-box validation to its users.

MVP does not require a priori information on the circuit under validation for it to be effective, but instead gathers this information real-time. Use of MVP's fundamental circuit analysis abilities alone cause it to be burdened by the analysis of irrelevant HDL code segments, and by the traversal of already-explored architectural states. We can significantly improve MVP's run-time performance by implanting mechanisms that enable it to *learn* important details of the circuit under validation as a means to explore critical new circuit scenarios. These mechanisms can exist as a pre-processor that gathers circuit information prior to the circuit validation process, as well as run-time entities that allow MVP to learn from its experience.

MVP can handle complete implementations because it only uses high-level information, and only uses the hardware description language (HDL) information relevant to the set of constraints when identifying all relevant architectural states. In this paper, we define a circuit architectural state that satisfies the set of constraints under consideration as a *prospect state* (*pState*).

Generating input stimuli that satisfy a set of constraints requires the solver to identify all prospect states for each time frame, and eliminate the prospect states that can not be used to satisfy a test sequence. This is a problem for modern superscalar microprocessor implementations because of their inherently large state space. Therefore to be able to efficiently identify and analyze the architectural states (*prospect states*) that can possibly satisfy the set of constraints, we need to reduce the search space (via profiling) in the analysis process as early as possible.

The rest of this paper is organized as follows. Section II presents several pre-processor profiling techniques used by MVP and Section III presents the details of MVP's run-time profiling techniques. Section IV presents some preliminary experimental results and Section V concludes the paper.

II. Pre-processor Circuit Profiling

The pre-processor to MVP's circuit validation process should be a light-weight task that provides MVP with valuable insight capable of directing its test pattern generation process towards a solution. It is because of this low-overhead demand that the pre-processor should not attempt to solve actual ATPG constraints, but rather solve early the sub-problems that provide MVP with the most valuable information. Instead of analyzing the implications that the circuit has onto each statement in the hardware description as is done in the real-time circuit analysis process (implications of the range of values for β and γ on the assignment to α as shown in Fig. 1a), we can implement the light-weight circuit profiler for the pre-processor by having it analyze the implications each statement has onto the overall circuit (assignment to α in Fig. 1b).

A. Assignment Statement Profiling

Our previous work discusses how an ATPG constraint is solved by exploring all relevant assignment statements that can satisfy its unresolved data implications [3]. Therefore whenever attempting to satisfy a constraint (especially when it is dependent on an enumeration data type), this solver process will likely be repeated for a great deal of assignment statements that cannot help satisfy the constraint. Much of this

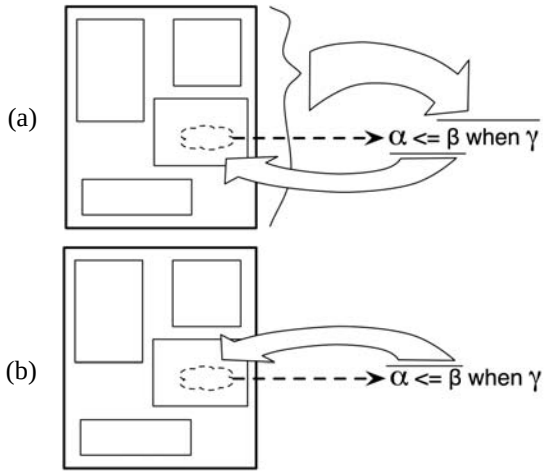

Fig. 1. (a) Run-time and (b) pre-processor circuit profiling.

dead-end work can be prevented by using MVP's *statementList* data structure [3], which holds a circuit's HDL information, to *index* each assignment statement with the identifier value implications that it has onto the hardware description. This *statementList* data structure is utilized by MVP to hold the HDL information of the circuit under validation.

The indexing process is easily implemented by using MVP's available resources. Satisfying a constraint requires MVP to first convert the assignment statement being considered into a data dependency graph (DDG) and to solve its implications. Solving this DDG provides every identifier within every conjunction with the explicit range in values that satisfies this assignment statement. Therefore to profile this specific assignment statement, the data implications it imposes onto the circuit are extracted directly from the identifiers within the solved DDG.

This indexing pre-processor is effective because many assignment statements in a hardware description simply transfer a constant value onto an identifier. This is particularly true for enumeration data types, as they are commonly used to explicitly control a finite state machine (FSM). This allows MVP to peek into each statement to expose most data contradictions before committing itself to solving that possible solution path.

B. Implicit Memory Element Profiling

For complete circuit analysis, MVP must explore all signals in the hardware description in search for implicit memory elements. It does this by negating the explicit guards to all assignment statements onto the signal being analyzed, and inserting them into a single conjunction (unified by Boolean AND operators). This process exploits MVP's efficient DDG solver, and a DDG that does not evaluate to false signifies an implicit memory element. This process therefore takes all implicit memory elements, and defines them explicitly by creating an entry for a corresponding memory-preserving assignment statement within the *statementList* data structure such that the guards to this entry denotes the memory-preserving condition.

C. Basic-Block Guard Profiling

In most cases where a data contradiction is encountered when solving a constraint, the contradiction arises from the union of the guards in the multiple prospect code paths. That is, the guards gathered from satisfying the constraint of a current unresolved identifier will more than likely conflict with the guard of a previously resolved identifier in the constraint. Experiencing an identifier value contradiction within the guard of a basic block is significantly more costly than experiencing a contradiction within the statement itself because the aggregated guards leading up to a basic block is larger in most cases than any of the assignment statements in that basic block, and this guard is repeatedly utilized by all statements within the basic block. Therefore the performance of this guard profiling pre-processor is slower than that of the assignment statement profiling pre-processor, but the runtime performance advantage it provides is equally as significant.

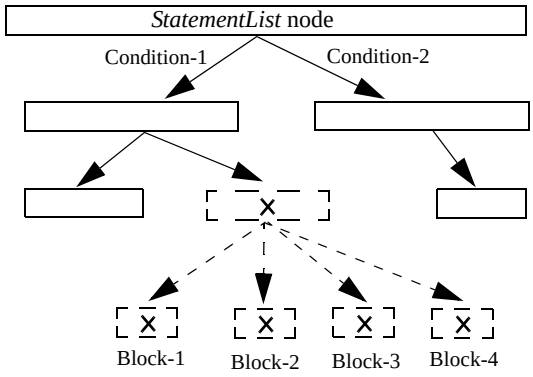

Fig. 2. Basic-block guard profiling.

It is possible to take advantage of the *statementList* data structure once again to hold pre-solved identifier values from the guards to all basic blocks. These solved identifier values can be used to expose data contradictions between a prospect code path's guards and the identifiers within a constraint being solved, and would prevent the costly dead-end task of converting a set of guard statements into a corresponding solved DDG that would evaluate to 'false'. For those assignment statements that are reachable, this profiling effort can retain the solved DDG to optimize all later uses by any assignment statement within the basic block.

This process of indexing all leaf *statementList* nodes with the solved identifier values to its guards can be performed as a pre-processor or at run-time. Given that MVP already analyzes all identifiers to expose implicit memory elements, which requires it to evaluate the guards to all statements, it is natural to implement this basic block guard profiler as a part of the pre-processor. For any given *statementList* node, the set of guards are obtained by appending its guard to those of all its ancestor nodes [3]. We can take advantage of the fact that the guards are distributed throughout the *statementList* tree (Fig. 2) by gathering the list of solved identifier values at each node, and recursively providing a copy of this list to all its children so it may append onto it.

This recursive process to obtaining guard information provides us with two key advantages. The data-sharing nature of this recursive algorithm allows it, as a pre-processor that starts at the root *statementList* node, to reduce the amount of redundant work that would be performed if it were to be executed at run-time starting at a leaf node. The second advantage is that it can identify all unreachable basic blocks within the hardware

description without having to analyze all basic blocks. As shown in Fig. 2, an identifier value contradiction exposed within the guard of an internal *statementList* node will automatically denote all its children statements as unreachable as well.

Unreachable code blocks commonly exist within CASE statements. The "when others" clause of a CASE statement is commonly utilized as a safe-guard that ensures an internal signal to the circuit is not created as an implicit memory element as a result of unhandled guard cases. However, when all possible cases have been handled explicitly, then the "when others" clause will create an unreachable block of code that will never be executed. Unfortunately for MVP, it attempts to satisfy a constraint by starting at all relevant assignment statements. MVP must therefore be aware of which HDL code blocks are irrelevant as a means of reducing the ATPG search space.

III. Runtime Circuit Profiling

MVP's run-time circuit validation process should be a complete task focused on exploring uncharted territory within the processor. In an ideal problem, it would be possible to travel throughout a hardware description's architectural state space without retracing one's steps. Unfortunately, 100% FSM coverage commonly requires a significant amount of redundant state exploration. Therefore as MVP gets further into its validation process, it will be forced to retrace more of the previously-explored state space in order to reach the target architectural state that defines the ATPG goal. Also, there are many architectural states that have a high occurrence frequency as they are a precursor to a wide range of other architectural states, thus retaining some of their pre-solved information can optimize MVP's performance in the long run. This section focuses on the run-time circuit profiling efforts that can allow MVP to breeze through the already-explored state space when attempting to satisfy a unique ATPG goal.

A. Finite State Machine Profiling

An FSM is usually described by $y = \delta(s, x)$, where a target state y can be reached from state s when the FSM's inputs are x [5]. When the target state y can be reached by multiple states $s_1 \ldots s_n$, we can use a weight scheme such that the state s with the lowest weight provides MVP with two advantages:

- When the pStates have never been explored (thus they are un-indexed), it will allow MVP to choose the state s with the least number of constraints that will need to be satisfied at the subsequent ATPG iteration. If the reset state is among the set, it will be characterized by the lack of constraints that need a subsequent ATPG iteration, therefore resulting in a weight of zero.
- When any of the pStates has been previously explored, its weight will be lower than all unexplored pStates, and will provide MVP with guidance towards the reset state as all subsequent pStates will continue to have lower weights.

To implement this weight-assigning process, we simply need to implement our ATPG solver as shown in Fig. 3. Line 5 selects the optimal candidate for the next ATPG iteration by selecting the pState with the lowest weight. If the selected pState has a weight of zero, the previous recursive call to the *multiFSM_solve* algorithm has its length value *l* updated to zero on line 7, and it is returned SUCCESS signifying that the reset state has been reached on line 8. The previous recursive call to the *multiFSM_solve* function will then be in charge of updating the weight values on line 11, incrementing the weight for its previous recursive call on line 12, and then commencing as usual.

```
multiFSM_solve(testSequence TS, pStateSet Y, int l)
1.  If (l = 0) return FAIL
2.  If (Y = reset state)
3.    return SUCCESS
4.  P ← solve(Y)              //Generates set of solutions
5.  For each t ∈ P, s.t. t has the lowest weight in P
6.    If (weight(t) = 0)
7.      l ← 0
8.      return SUCCESS
9.    S ← get_previous_timeFrame(t)
10.   If (multiFSM_solve(TS, S, l – 1) = SUCCESS)
11.     assign_weight(S, l)
12.     l ← l + 1
13.     TS ← TS + getPrimaryInputs(S)
14.     return SUCCESS
15. return FAIL
```

Fig. 3. MVP's optimized ATPG algorithm.

The goal of exploring an FSM is to generate a test sequence that maps the hardware description's architectural state from its reset state onto any architectural state that satisfies the given set of constraints. This process begins at the target architectural state, and continues to traverse the circuit backwards in time until the reset state is reached. To optimize this ATPG effort, we need to enable MVP to intelligently navigate through a circuit's FSM.

A hardware description is characterized by the inter-dependent FSMs from all of its internal registers, thus developing a macroscopic understanding on the overall FSM will require us to understand all possible state combinations (the cross product) from all these smaller inter-dependent FSMs. We can therefore simplify the FSM profiling task by placing our focus at the individual FSMs for each register as they make up the building blocks for the overall FSM.

Our objective in performing FSM profiling on the overall circuit is to achieve the profiling tasks on the individual FSMs, and employ a mechanism that translates this low-level FSM profiling information into a circuit-wide FSM profiler. The concept is simple enough, but the implementation is tricky because MVP does not manage these FSMs explicitly. It would be possible to provide MVP with the mechanisms that allow it to build and analyze these interacting FSMs explicitly, but that would only require it to perform another level of computations that should not be necessary. MVP's strength is in its ability to analyze the circuit under validation by focusing on the source code, and it is possible to exploit MVP's source code database of the circuit under validation to achieve similar profiling results.

Let us take a moment to translate these FSM profiling goals into MVP's language. The low-level FSM profiling is meant to account for the many inter-dependent FSMs, and so it must therefore analyze the FSM associated with each identifier that represents an internal register. MVP currently uses a construct entitled as an *identifierSet*, whose purpose is to keep track of every HDL location that each identifier is assigned a value

onto. Initially, the objective of this construct was to optimize the algorithm that generates all possible pStates from a given identifier constraint by having the sources to all possible solutions be readily available in one data structure. Therefore, we can also use all entries corresponding to a constraint's identifier to provide us with the FSM profiling information we need. We can exploit the fact that MVP accesses this *identifierSet* data structure each time it attempts to use a code path as a solution by also having MVP leave behind real-time low-level circuit profiling information whenever it successfully utilizes this data source to satisfy a constraint.

The aforementioned global FSM profiling effort is meant to interpret the low-level FSM profiling information and identify the shortest FSM path that can reach the circuit's reset state. We know the low-level profiling effort should be performed when MVP attempts to use a line of HDL source code for satisfying a circuit constraint, therefore we should take a step back and identify which MVP construct is analyzing these lines of code and could stand to benefit from the low-level profiling efforts. Looking at Fig. 3, we can see that the resulting test sequence is generated by instantiating pStates as the mechanisms that carry the potential solutions as they are being generated, and thus the pState construct should be used to manage the global FSM profiling effort.

The low-level FSM profiling effort is focused on depositing information onto each statement in the hardware description to record its scope and the success it provides. Conversely, the global FSM profiling effort is focused on unifying the information gathered from all statement sources that represent a given solution as a means to avoid costly or irrelevant scenarios. The remainder of this section revolves around these concepts.

FSM Weight Indexing. MVP's ATPG algorithm is able to independently find the reset state through exploration of an FSM, but this alone requires much backtracking. We can therefore exploit its ability to find and detect the reset state by appending the explored states in each FSM (the explored assignment statements for the identifier behind the FSM's register) with a weight value equal to its distance from the reset state. If MVP is instructed to generate a test sequence with a length of at most l, then we can assign each state an initial weight $>> l$.

Unfortunately, the task of assigning weight values to a processor's architectural states is not so straightforward. This is because each pState is influenced by multiple implicit FSMs, and is pieced together by several concurrent assignment statements that successfully satisfy all simultaneous constraints. MVP, therefore, is not assigning weight values to explicit architectural states, but rather is assigning weights to the assignment statements that were used to piece them together.

MVP can perform its run-time weight-assigning process following every ATPG iteration to update each assignment statement's resulting distance to its FSM's reset state. Any given assignment statement may impact several distinct architectural states, and thus its weight value may have multiple sources. For the sake of allowing MVP to move towards an optimal solution while keeping the ATPG implementation simple, we will allow each assignment statement to store the lowest weight value it is assigned. Using a given assignment statement's lowest assigned weight value, say w, is reasonable because that statement has the potential of providing an instruction sequence of size w again in the future. Therefore giving preference to this assignment statement over other alternate assignment statements of higher weight when solving a constraint allows MVP to choose the ATPG path with the highest probability of producing the shortest path to the reset state.

Prospect State Weight Estimation. MVP's ATPG algorithm analyzes multiple pStates at every time frame, from which it must choose one to attempt and reach the reset state. Therefore providing MVP with a weighing scheme for its pStates can help it easily identify the most effective solution path. The motivation for extracting a weight value from a pState is twofold, as mentioned at the start of this section. In choosing the ideal pState, MVP must first favor those solutions to which a path to the reset state has already been identified; otherwise it must favor the pStates with the least number of constraints to justify. These two objectives must be handled inherently by a single weighing scheme.

However, finding a balance between these two objectives is not trivial because the first requires that the pState have been solved in order to extract an accurate weight from the utilized HDL statement sources, and the second requires the pState to *not* have been solved. Using our FSM weight indexing scheme where we index each RTL assignment statement with its known distance to the reset state, we can attain a weight value to a solved pState because it will then have assignment statements associated to it that were used to satisfy its constraints. Thus for the first case, if a pState has not been solved, then it will not have these HDL statement sources that are necessary to estimate its distance to the reset state. Conversely for the second case, the number of constraints to resolve in a pState obviously can only be evaluated before these constraints are resolved.

In identifying a pState's weight, MVP must use a unifying scheme that satisfies both of the preceding objectives. MVP will first have to solve the pState, and then adapt its weight-assigning scheme to handle the second case which favors the pState with the least number of ATPG constraints. It can do this adaptation by counting the number of constraints that will propagate onto the following ATPG iteration. And to estimate the weight that gives preference to those previously-solved pStates closest to the reset state, we can multiply this number of constraints that need to be resolved in the next ATPG iteration by the average weight of the assignment statements associated to the solved constraints. A pState whose constraints were solved in a previous ATPG problem will have assignment statements associated to it whose weight is lower than the maximum weight, and thus its average weight will naturally be lower than the maximum weight.

Modified ATPG Algorithm. MVP's pState-weighing scheme requires us to modify MVP's ATPG algorithm as depicted in Fig. 4. The *get_previous_timeFrame()* algorithm extracts, from a pState y, all the pStates s that can transition into it. It requires y to have been solved (have all its constraints satisfied), and it returns a set of pStates s that have not been solved. Thus, the objective of this modification is to ensure that MVP's ATPG algorithm calls the weight estimation procedure on solved pStates only, as well as perform weight indexing using these solved pStates.

The most significant change that allows us to satisfy these objectives is that the algorithm now expects the alternate

```
Precondition: pStates in Y have been pre-solved

multiFSM_solve(testSequence X, pStateSet VS, pStateSet Y, int l)
1.  If (l = 0) return FAIL
2.  For each p ∈ Y, s.t. p has the lowest weight in Y
3.    P ← get_previous_timeFrame(p)
4.    For each p' ∈ P
5.      If (p' is masked by some state in VS)
6.        delete p'
7.      else
8.        VS ← VS + p'      // Store copy of p' into visited set VS
9.        If (weight(p') = 0)      // Reset state has been found
10.         l ← 0
11.         Y ← {p}          // Remove unexplored pStates in Y
12.         X ← getPrimaryInputs(p')
13.         return SUCCESS
14.       S ← solve(p')      // Attain incoming pStates S from
                             // target pState Y
15.       If (multiFSM_solve(TS, VS, S, l – 1) = SUCCESS)
16.         weight(p) ← l
17.         Y ← {p}          // Remove unexplored pStates in Y
18.         l ← l+1
19.         X ← X + getPrimaryInputs(S)
20.         return SUCCESS
21. return FAIL              // No ATPG goals (Y is empty) or
                             // no solution exists
```

Fig. 4. MVP's modified ATPG algorithm.

ATPG objectives *Y* to be a previously solved set of pStates. Having *Y* be a solved set of pStates allows MVP to immediately use its weight estimation methods for identifying the ATPG goal in *Y* that is estimated to be closest to the circuit's reset state. Afterwards, this modification converts the chosen path in *Y* into the alternate sets of constraints *P* that define the preceding architectural states. If the pState set in *P* contains the reset state, then the ATPG iteration is complete. Otherwise the set in *P* is solved to define the set of previous time frames *S* that can transition into *Y*, and to define the inputs that allow this transition to take place. The preceding pStates in *S* are themselves justified towards the reset state by invoking a recursive call to the ATPG algorithm.

B. Explored State-Space Tracking

The ATPG algorithm in Fig. 4 will commonly receive, from line 3, pStates that have been traversed by a previous recursive call within the same ATPG iteration. When this happens, those pStates should be ignored because re-analyzing them will not help the ATPG algorithm get any closer towards a solution. Ignoring the visited pStates is both an up-stream and down-stream process. Preventing the ATPG algorithm from revisiting a pState that is visited earlier in the same test sequence will prevent the ATPG algorithm from analyzing FSM loops. Furthermore, preventing the ATPG algorithm from revisiting a pState that was visited by a previous test sequence branch that failed to generate a result will prevent the ATPG algorithm from analyzing unsuccessful paths more than once.

This explored state-space tracking effort is implemented on lines 5, 6, and 8 of Fig. 4. Line 5 checks if the current pState *t* to be analyzed has been previously visited by that same ATPG iteration. If it has been previously visited, then line 6 deletes it and allows the subsequent iteration of the FOR loop on line 4 to analyze the next pState in the solution set *P*. If it has not been previously visited, then line 8 allows the ATPG algorithm to store *p'* into the visited set *VS* and proceed as usual.

We can identify if a pState *p'* has been previously visited by identifying if *p'* is masked by the set of visited pStates in *VS*. A pState is defined by a set of internal and input identifiers, and their corresponding range in values. For the purpose of obtaining a clear perspective on when one pState masks another, let us realize that an identifier missing from a pState signifies that the corresponding identifier has a complete range in values. In terms of identifiers, an identifier with a range in values *v* is masked by a corresponding identifier instantiation with a range in values *v'* if and only if (IFF) the range in values for *v* are encapsulated by the range in values for *v'* ($v \in v'$). We can therefore identify if a pState *t'* is masked by a pState *t* IFF the set of identifiers referenced by pState *t* is a subset of the identifiers referenced by *t'*, and IFF the range in values of the identifiers in *t* encapsulate the range in values of the corresponding identifiers in *t'*.

IV. Experimental Results

Results on MVP's effectiveness have been generated by following the key steps in a validation paradigm: coverage metric definition, error modeling, circuit simulation, and ATPG. All experiments have been performed on a Dual 2.5GHz G5 workstation under OS X Tiger using gcc 4.0. MVP has been implemented as a library using GNU's autotools (*autoconf*, *automake*, *libtool*) in 20K physical lines of C++ code.

The Motorola 68K implementation [6] analyzed for this paper is microarchitectural by nature. The collection of mutation control errors (MCEs) [1][2] are generated to bind all possible combinations between the explicit *state* signal to all other control signals. The set of control signals also includes the *next_state* signal, which allows us to stimulate the data paths as well as the FSM transitions. Having a combination of values between a control signal (imagine an input to a multiplexer) and the explicit state as ATPG constraints allows us to stimulate every combination of the control signal's set of resulting data paths at every explicit microarchitectural state. Furthermore, having a combination of values between the *state* signal and the *next_state* signal allows us to stimulate every transition in the microarchitectural FSM.

The Motorola 68K explicit FSM allows us to easily understand MVP's effectiveness in deciphering a circuit's state machine by observing the shortcuts it exploits when satisfying a set of constraints. Each of MVP's ATPG iterations generates a test sequence that maps any target architectural state to the reset state. However if the reset state has not been identified, the search process is blind. After even one successful ATPG iteration, MVP's effectiveness in reaching the reset state is highly optimized by its run-time circuit profiling methods. We can quantify MVP's effectiveness in learning a circuit's FSM by counting the amount of backtracking experienced by each ATPG iteration. Fig. 5 quantifies how suddenly the backtracking is reduced by MVP's run-time circuit profiling methods after every successful ATPG iteration.

MVP's true effectiveness is due to its ability to continuously traverse unexplored portions of a circuit's architectural state-space. The circuit design industry currently employs random and pseudo-random ATPG when exploring large circuit imple-

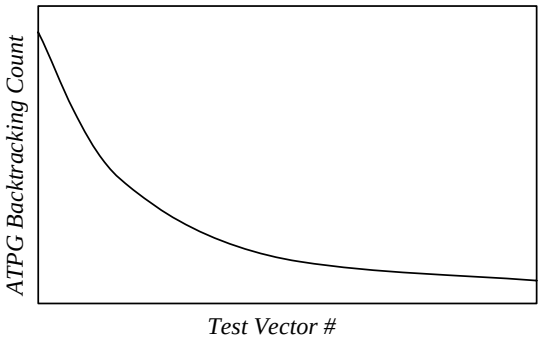

Fig. 5. MVP's FSM-learning effectiveness.

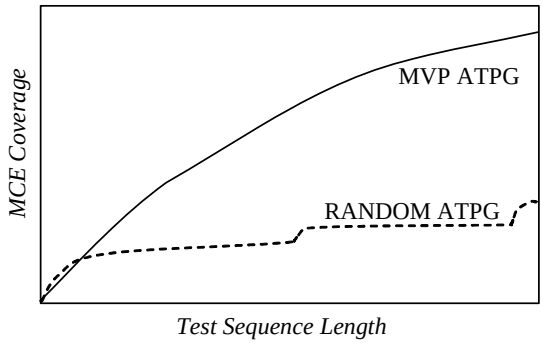

Fig. 6. MVP's ATPG effectiveness.

mentations. This provides them with a high rate of simulation iterations, but they gamble on their random approach to eventually reach a high coverage of their circuit's state-space. By applying a given set of mutants onto MVP's concurrent mutant simulator, we can directly compare the effectiveness in MVP's approach to the random methods commonly used to expose circuit design errors. Fig. 6 presents the effectiveness provided by MVP in comparison to random ATPG.

If we look closely at the start of the simulation in Fig. 6, we can see that random ATPG is initially more effective than MVP's deterministic ATPG. The reason for this is because every test sequence generated by MVP begins at the circuit's reset state. As a result, the initial test vectors in each of MVP's ATPG traverse already-explored architectural states. This is acceptable because using the reset state as the common test sequence starting point provides us with two advantages: (*i*) The ATPG unit is able to perform real-time profiling such that each HDL line of code can hold its weight with respect to its known shortest distance to the reset state, and (*ii*) any given test sequence that exposes an actual circuit design error is self-sustained and can be utilized independently, as it begins with the circuit's reset state and ends when the circuit design error is exposed.

By looking closely towards the end of simulation in Fig. 6, we can see that MVP's deterministic ATPG is significantly more consistent and effective than a random or pseudo-random ATPG approach. The simulation results from the random ATPG soon levels off, and has eventual bursts of effectiveness whenever the random test vectors happen to reach an unexplored portion of the architectural state-space. These sudden burst of productivity cannot be predicted, and are commonly a source of false-positives in circuit validation. MVP's deterministic ATPG, however, exploits MVP's simulation statistics to determine the ATPG goals that can allow it to continuously reach unexplored portions of the state-space. Therefore MVP, unlike random and pseudo-random ATPG, has *consistent* bursts of productivity due to MVP's closed-loop strategy between circuit simulation and automated test pattern generation.

V. Conclusions

We have presented circuit profiling mechanisms that allow our mutation-based validation system to learn as it generates test sequences. These mechanisms are either a pre-processor that gathers circuit information prior to the validation process or a run-time entity that progressively gathers circuit information during the validation process. Our preliminary experiments show that MVP's effectiveness in reaching the reset state is highly optimized by its circuit profiling methods. Moreover, the experiments show that the backtracking in MVP's ATPG is reduced by using run-time circuit profiling methods after every successful ATPG iteration.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 0092867.

References

[1] J. Campos and H. Al-Asaad, "Concurrent design error simulation for high-level microprocessor implementations", *Proc. AUTOTESTCON*, 2004, pp. 382-388.

[2] J. Campos and H. Al-Asaad, "Mutation-based validation of high-level microprocessor implementations", *Proc. International High-Level Design Validation and Test Workshop*, 2004, pp. 81-86.

[3] J. Campos and H. Al-Asaad, "MVP: A mutation-based validation paradigm", *Proc. International High-Level Design Validation and Test Workshop*, 2005, pp. 27-34.

[4] J. Campos and H. Al-Asaad, "Search-space optimizations for high-level ATPG", *Proc. International Microprocessor Test and Verification Workshop*, 2005, pp. 84-89.

[5] F. Corno et al., "SymFony: A hybrid topological-symbolic ATPG exploiting RT-level information", *IEEE Transactions on Computer-Aided Design*, Vol. 18, pp.191-202, February 1999.

[6] http://www.opencores.org/projects.cgi/web/system68/overview.

[7] I. Ghosh and M. Fujita, "Automatic test pattern generation for functional register-transfer level circuits using assignment decision diagrams", *IEEE Transactions on Computer-Aided Design*, Vol. 20, pp. 402-415, March 2001.

[8] J. Shen and J.A. Abraham, "An RTL abstraction technique for processor microarchitecture validation and test generation", *Journal of Electronic Testing: Theory and Applications*, Vol. 16, pp. 67-81, February-April 2000.

[9] L.-C. Wang and M.S. Abadir, "On efficiently producing quality tests for custom circuits in PowerPC^{TM} microprocessors", *Journal of Electronic Testing: Theory and Applications*, Vol. 16, pp. 121-130, February-April, 2000.

[10] F. Corno et al., "Automatic test program generation from RT-level microprocessor descriptions", *Proc. International Symposium on Quality Electronic Design*, 2002, pp. 120-125.

[11] M. N. Velev, "Collection of high-level microprocessor bugs from formal verification of pipelined and superscalar designs", *Proc. International Test Conference*, 2003, pp. 138-147.

[12] C.-C. Yen, J.-Y. Jou, and K.-C. Chen, "A divide-and-conquer-based algorithm for automatic simulation vector generation", *IEEE Design and Test of Computers*, Vol. 21, pp. 111-120, March-April 2004.

[13] D. Moundanos, J. A. Abraham, and Y. V. Hoskote, "Abstraction techniques for validation coverage analysis and test generation", *IEEE Transactions on Computers*, Vol. 47, pp.2-14, January 1998.