Appears in the Proceedings of the 31st Design Automation Conference
pp. 717-721, San Diego, CA, June, 1994

ProperHITEC: A Portable, Parallel, Object-Oriented Approach to Sequential Test Generation

Steven Parkes
parkes@crhc.uiuc.edu

Prithviraj Banerjee
banerjee@crhc.uiuc.edu

Janak Patel
patel@crhc.uiuc.edu

Center for Reliable and High-Performance Computing
University of Illinois
1308 W. Main
Urbana, IL 61801

Abstract—Automatic test pattern generation (ATPG) for sequential circuits remains one of the most compute-intensive tasks in the integrated circuit design process. Although numerous attempts have been made to speed the ATPG process via parallelization, these attempts have often proved disappointing when compared against the best available serial algorithms using metrics of resultant quality and performance. In this paper we introduce `ProperHITEC`, a parallel extension of the `HITEC/PROOFS` sequential test generation package. `ProperHITEC` embodies a new approach to parallel ATPG; it is incrementally derived from one of the best-performing serial algorithms and this incremental derivation is implemented via the mechanisms of object-oriented programming. Results of running `ProperHITEC` on three parallel architectures, the Sun 4/600MP, the INTEL iPSC/860, and the Encore Multimax are presented. These results show that `ProperHITEC` achieves results virtually identical in quality to `HITEC` while achieving significant multiprocessor utilization.

1. Introduction

Despite concerted effort, automatic test pattern generation (ATPG) for sequential circuits remains one of the most time consuming tasks in the integrated circuit design process. Yet the use of parallel processing in solving the ATPG problem remains insignificant, even in the face of several proposals for parallel ATPG. This is due in part to the fact that many of the algorithms previously produced are tightly tied to a particular architecture. Moreover, a significant impediment to the development of *useful* parallel applications is the concurrent rapid development of improved serial algorithms. If parallelization is based on a frozen serial algorithm, by the time the parallelization process is complete, the parallel application may lag significantly behind the quality—and sometimes even the performance—of contemporary serial ATPG algorithms.

Thus a strong argument can be made that any practical parallel ATPG algorithm be

1. portable across the widest possible variety of parallel machines
2. based upon the best available serial algorithms

This research was supported in part by the Semiconductor Research Corporation under contract 93–DP–109.

In this paper, we present `ProperHITEC`, a parallel implementation of the `HITEC` sequential test generator. `ProperHITEC` represents two significant contributions to the area of parallel test:

1. `ProperHITEC` uses object-oriented mechanisms to implement a parallel version of the serial algorithm. Not only is the majority of the source code shared between `HITEC` and `ProperHITEC`, the majority of the object code resides in a shared library. Improvements made to `HITEC` are immediately reflected in `ProperHITEC`.
2. `ProperHITEC` is based on the parallelism primitives provided by the ProperCAD II library [1], a C++ object library which provides a fully object-oriented parallel interface based on the Actor model [2] of concurrent object-oriented computation across a wide variety of parallel architectures.

2. Previous work

Several approaches have been proposed for parallelizing test pattern generation:

Fault-parallel *Fault-parallelism* refers to the generation of test patterns for individual faults in parallel. The complete fault set is divided equally among all processors, each processor generating tests for its subset independent of all other processors [3, 4, 5]. While fault-parallelism has the advantage of low communication overhead it is not effective in reducing the time required to generated tests for individual hard-to-detect faults. Moreover, since most test generation systems use fault simulation to capitalize on the serendipitous detection of multiple faults by patterns generated for a single fault, fault-parallelism alone may not lead to high multiprocessor efficiency; any effort expended to detect a fault which will be serendipitously detected is wasted.

Decision-parallel *Decision-parallelism* refers to the evaluation of several decision alternatives in parallel. In this respect, decision-parallelism resembles parallel depth-first search [6]. This technique was proposed in [7] but no heuristics were used to increase the probability of searching in a solution area. A parallel branch and bound algorithm was proposed in [8]. Other implementations have been developed for workstation clusters [9] and shared memory multiprocessors [4], the latter combining fault and decision parallelism. Ramkumar and Banerjee [10] use Charm [11] to develop a parallel version of [4] which runs on a number of architectures. The work presented here adopts many of the techniques of their work.

Heuristic-parallel *Heuristic parallelism* refers to execution of multiple test generation tasks using different heuristics on each processor [3]. Because the number of heuristics available is typically no more than five or six, the scalability of this approach is limited. Moreover, when using different heuristics there is no guarantee that spaces searched by each processor are disjoint, which may lead to excessive redundant work. As in fault parallelism, communication overheads are usually low.

Partition-parallel *Partition-parallelism* refers to the decomposition of the circuit with disjoint partitions assigned to each processor [12]. In this case, the limit on circuit size becomes the sum of the memory on each processor rather than the minimum, as it is in the preceding approaches. To date, it has not been possible to achieve reasonable efficiency using this approach due to excessive communication requirements.

3. HITEC/PROOFS

The HITEC/PROOFS package for sequential test generation and fault simulation [13] exhibits performance and quality among the best known. HITEC is a descendent of and an extension to the PODEM approach. The extensions, roughly in decreasing order of significance, are:

Targeted D-Frontier In order to maximize the number of implications derived from an assignment, many test generators calculate the intersection of the set of single-node dominators of the D-Frontier. Rather than form the intersection, HITEC instead selects the frontier element most likely to propagate to a primary output and assigns a faulty value to that node. A larger number of implications can generated in this manner at the possible expense of a larger number of backtracks.

State Justification HITEC breaks into separate steps intra- and inter-frame justification. HITEC attempts to speed inter-frame justification—also called state justification—by 1) keeping a list of states which have been proven unjustifiable, 2) reducing state assignments whenever possible, 3) taking advantage of the the state produced by running previous test vectors, and 4) determining when a state justification objective has been achieved by previous test vectors.

Fault Simulation HITEC uses fault simulation to drop any faults covered by previously generated vectors. HITEC also orders the fault list to attempt to take maximum advantage of this serendipitous detection.

The overall organization of the HITEC/PROOFS package is shown in Fig. 1. In the original, HITEC and PROOFS run as separate processes, communicating synchronously via a TCP/IP socket. The package is written in C++ using an 'object-based' design style; the object-oriented concept of encapsulation is only weakly represented and the use of dynamic-binding is omitted entirely.

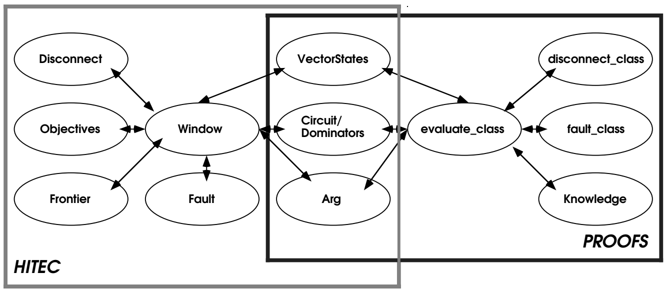

Fig. 1 HITEC/PROOFS Organization

4. ProperHITEC

Our approach to parallelization is an extension of methods developed in earlier work in parallel ATPG [10, 14]. Parallelism is generated in two ways; through fault-parallelism and decision-parallelism (see Fig. 2.)

4.1. *Fault-parallelism*

Because test generation of different faults is largely independent, the impact of fault-parallelism on the HITEC algorithm is minimal. The only heuristics impacted are those related to state knowledge. While in serial HITEC the test generation process can utilize the state in which the circuit was left as a result of running vectors from previous tests, this is not possible in a parallel context where the order of vectors is not fixed until after a test is generated. Because the efficacy of this heuristic is unclear [13] and is not enabled by default in the original HITEC implementation, we did not try to support it nor did we attempt to measure the impact.

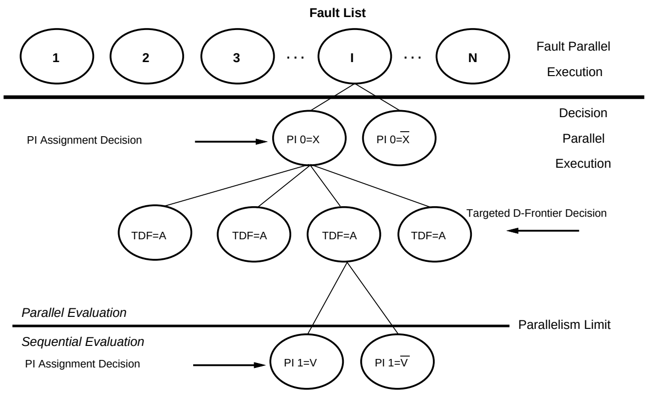

Fig. 2 Parallelism in ProperHITEC

4.2. *Decision-Parallelism*

In decision-parallel execution, when a backtrackable decision is made, rather than perform a depth-first search of the alternatives, new tasks are created to explore simultaneously each alternative. In PODEM, the only decisions are input assignments so the search-space graph for each fault is a binary tree [10, 14]. In `HITEC`, backtrackable decisions are also made on the targeted D-Frontier and thus nodes in the search graph represent either an input assignment with two out-edges or a D-Frontier target operation with N out-edges, one for each node in the frontier.

Only one significant change was required to `HITEC` heuristics to implement decision-parallelism. When a backtrack occurs in a depth-first search, information gained in detecting the backtrack can be passed up the search tree to influence the backtracking process. `HITEC` utilizes such information when a state-justification failure occurs, forcing backtracking to continue until a new state justification goal is generated. When decision-parallelism is used, a breadth-first search is employed and thus there is no way to pass information back during backtracking; another method for communicating the same or similar information is required.

For the state justification problem, it is easy to show that for those cases where the special backtracking mode is used, the current justification goal must be in the *failed states* list, already maintained by `HITEC`. Moreover, the failed states list contains additional information that can reduce the amount of non-solution space searched. Testing of serial `HITEC` using both heuristics showed the cost of useless searching to be far greater than that of checking the failed states list. As a result, the better heuristic was incorporated into both serial `HITEC` and `ProperHITEC`.

Implementation of decision-parallelism involves cloning the current test generation state and the computational weight of this operation can be significant. Since the number of explorations rises exponentially in the number of decisions performed in parallel, beyond a relatively low limit sufficient parallelism exists to keep all processors busy and the benefit of further decision-parallel execution disappears. Therefore, when decision-parallelism is used, a limit on the depth of decisions performed in parallel is set beyond which the serial `HITEC` depth-first algorithm is used.

5. ProperCAD II Overview

The ProperCAD II library [1] provides the primitives used for expressing parallelism in `ProperHITEC` and supports a broad class of shared-memory, message-passing, and hybrid architectures. The ProperCAD II library is built around the Actor paradigm of concurrent object-oriented computing.

5.1. *The Actor Paradigm*

The fundamental object in an Actor paradigm [2] is the actor, an object which communicates with other actors by sending messages. When a message is received, a member function of the receiving actor is invoked to process the contents. In this way, the actor model represents an extension to existing remote-procedure-call (RPC) models.

The basic RPC model is neither object-oriented nor multithreaded. The lack of object-orientation means support for distributed data structures is limited. Moreover, because the basic RPC model is a blocking model, it does not express parallelism; there is still only a single thread of control.

The actor model is implicitly object-oriented and has implicit thread semantics. Each actor is an object in the object-oriented sense and each actor represents a very light-weight thread. Further, because individual actors are single-threaded, a programmer writing in the actor model does not need to deal with low level mutual-exclusion primitives.

Aggregates [15] were developed to extend the synchronous interface of actors. Aggregates are collections of actors which share a single name and support an object-oriented, multiaccess interface. Aggregates find wide use in distributed data structures.

5.2. `Actors` and `Aggregates`

The ProperCAD II library supports actor functionality via the `Actor` and `Aggregate` base classes. An object derived from one of these classes has a name (address) which is valid across the entire parallel application and can be used to express inter-actor communication. For example, in `ProperHITEC` the main computation actor class is `TestGenerator`, derived from the ProperCAD II `Actor` class and the `HITEC` `Window` class:

```
class TestGenerator :   public Actor,
                        public Window { ...  };
```

5.3. `ActorMethods` and `Continuations`

`ActorMethods` are member functions which may be invoked asynchronously. `ActorMethods` are executed via `Continuations`, the object-oriented, parallel extension of member function pointers. A continuation is created by specifying an actor method, indicating which member function to call, and an actor name, representing the actor for which the member is to be called. For example, in `ProperHITEC`, the `test` member of the vector database aggregate class `Vectors` is called by `TestGenerator` actors when a test sequence is found:

```
Vectors::test::Continuation cont ( vectorDB );
cont( vectors );
```

The first line declares a continuation `cont` which when executed will call the member function `test` for the object identified by the aggregate name `vectorDB`. The second line calls the continuation with the vector list `vectors` as an argument.

Execution via continuations differs from that of normal member function execution:

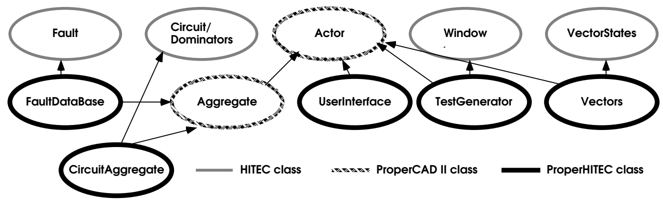

Fig. 3 `ProperHITEC` Organization

- Execution is asynchronous with respect to the caller. The actual execution of the `ActorMethod` occurs at some unspecified time in the future.
- Due to their asynchronous nature, continuation calls do not return a value.

Continuation execution is the sole method of expressing parallelism in an actor program.

6. Implementation details

`ProperHITEC`, based on the original `HITEC` code and the ProperCAD II library, implements the parallelism features described in Section 4. Fig. 3 shows the most significant objects in the `ProperHITEC` implementation along with the `HITEC` and ProperCAD II objects from which they are derived.

`TestGenerator` The `TestGenerator` object is an actor which represents a 'test generator machine.' `TestGenerator` instances can be created for a specified fault to implement fault-parallelism or can be cloned from an existing instance and an alternate assignment to implement decision parallelism. Each `TestGenerator` constructor continuation is assigned a priority which the underlying run-time system uses to order execution. Lexicographic bit string priorities, similar to those of `ProperTEST` [10], ensure that on an individual processor, execution follows that of the serial algorithm; when run on a uniprocessor, execution occurs in exactly the same order as the serial algorithm.

`CircuitAggregate` The circuit is implemented as an aggregate with one representative on each processor. Library primitives implement the data distribution process.

`FaultDataBase` The fault database is implemented as an aggregate with one representative on each processor. Each representative stores the most recent state of the test generation process for each fault. When an update is received from a `TestGenerator` actor or from the vector/states database, the `FaultDataBase` representative invokes serial `VectorsStates` code to record the information locally and, if this results in a change of the local state, broadcasts that information to all other representatives.

`Vectors` The `Vectors` actor is essentially the same as the serial object but uses `ActorMethods` to record new vectors and to notify the `FaultDataBase`. Fault simulation occurs serially on a single processor. Because test generators spend little time in fault simulation, the fault simulator is not a bottleneck.

`UserInterface` The `UserInterface` actor creates the actors and aggregates described above and then creates test generator actors for each undetected fault. If the progressive time limit feature of `HITEC` is used, the process of creating test generator actors is iterated using progressively larger time limits.

7. Results

Parallelization of test generation can be used to either achieve results similar to those of a serial application in less time or to achieve higher quality results in the same time. We present results of both types. All runs were made with fixed time-per-fault limits as indicated and a decision parallelism limit of five decisions.

7.1. *Performance results*

Tables 1, 2, and 3 show the results of `ProperHITEC` for a number of circuits drawn from the ISCAS-89 benchmark set on Sun 4/600MP and Encore Multimax shared-memory multiprocessors and an INTEL iPSC/860 message-passing hypercube. Elapsed wall clock time, T, is reported in seconds and excludes initialization and I/O; some of the parallel machines implement I/O very poorly. Fault efficiency, E, is computed as

$$\text{Efficiency} = \frac{(\text{No. Faults} - \text{No. Aborted})}{\text{No. Faults}}$$

V is the number of test vectors generated. For each circuit, the results for serial `HITEC` and for `ProperHITEC` on a range of configurations are reported.

The results show that `ProperHITEC` achieves consistent speedup with minimal effect on quality across a range of moderately difficult test problems. The only impact on quality is the addition of a small amount of noise in the results. `ProperHITEC` does not achieve significant speedup on 'easy' benchmarks, those for which 100% efficiency is achived by the serial application in a few tens of seconds on contemporary machines. If the first attempt at generating a test virtually always succeeds, the decision parallelism in `ProperHITEC` is not useful and the work done to clone the `TestGenerator` actors is wasted. We are considering techniques for improving the response of `ProperHITEC` for these cases although speeding test generation of such easy circuits is only of marginal interest.

7.2. *Efficiency results*

In addition to providing faster turnaround, parallel processing can be used to achieve higher fault efficiency in a fixed amount of time. Table 4 shows the results of running test generation on `s1494` where the time-per-fault limit was scaled with the number of processors. Consistent improvement is observed for all platforms. Of note is the fact that even though the time limit per fault was raised in tandem with the number of processors,

TABLE 1 RESULTS ON THE SUN 4/670MP

Circuit/ Seconds Per Fault	HITEC			ProperHITEC Processors											
				1			2			3			4		
	T	E	V	T	E	V	T	E	V	T	E	V	T	E	V
s344/20	369.4	95.9	121	374.3	95.9	121	251.9	96.2	110	160.0	96.5	130	156.2	96.2	112
s820/20	435.9	99.3	956	396.8	99.3	956	225.4	99.3	1010	196.3	99.1	1059	140.3	99.1	1013
s953/20	125.8	100	20	134.2	100	20	71.37	100	12	64.17	100	16	47.24	100	12
s1238/2	13.13	100	386	21.64	100	386	15.15	100	390	13.31	100	405	16.18	100	385
s1494/20	722.0	99.1	1058	663.4	98.9	1058	434.1	98.9	1123	350.5	98.9	1153	240.1	99.1	1093

TABLE 2 RESULTS ON THE INTEL IPSC/860

Circuit/ Seconds Per Fault	HITEC			ProperHITEC Processors											
				1			2			4			8		
	T	E	V	T	E	V	T	E	V	T	E	V	T	E	V
s344/20	481.4	94.2	89	485.8	94.2	89	215.8	96.8	105	194.5	96.8	112	142.1	96.5	102
s820/20	438.3	99.3	959	440.8	99.3	959	270.4	99.2	958	158.0	99.3	951	108.0	98.9	1034
s953/20	140.2	100	14	147.7	100	14	89.13	100	14	49.00	100	24	28.66	100	14
s1238/1	14.15	100	374	23.29	100	374	14.69	100	383	12.16	100	369	11.12	100	402
s1494/20	819.8	99.0	1079	821.3	98.7	1079	503.3	99.1	1168	310.1	98.6	1113	192.2	98.8	1151

TABLE 3 RESULTS ON THE ENCORE MULTIMAX

Circuit/ Seconds Per Fault	HITEC			ProperHITEC Processors											
				1			2			4			8		
	T	E	V	T	E	V	T	E	V	T	E	V	T	E	V
s344/20	484.2	93.9	105	493.2	93.9	105	274.3	95.6	95	167.0	95.6	85	131.2	94.7	108
s820/20	1200	97.8	891	1206	98.1	891	761.8	97.6	1008	418.9	97.2	959	255.6	96.9	955
s953/100	572.4	100	20	597.1	100	20	343.3	100	18	252.1	100	14	166.7	100	10
s1238/10	65.41	100	386	97.49	100	386	60.53	100	382	53.83	100	382	55.35	100	406
s1494/10	2615	87.0	492	2920	84.2	402	1654	85.5	460	997.0	83.9	497	540.2	85.5	510

run times still decreased on parallel runs. Once a fault is detected, raising the per-fault time limit does not increase run time; time-per-fault provides only a rough control of test effort.

TABLE 4 INCREASED EFFICIENCY RESULTS

Machine	Seconds Per Fault	Number of Processors	T	E
Sun MP	1	1	707.0	82.2
	2	2	230.7	97.1
	3	3	222.0	97.6
	4	4	185.4	98.1
iPSC/860	2	1	777.4	79.5
	3	2	335.0	94.4
	8	4	212.7	97.8
	20	8	192.2	98.8
Multimax	2	1	2069	42.0
	4	2	1465	67.1
	8	4	1047	77.4
	20	8	540.2	85.5

8. SUMMARY

In this paper we introduced `ProperHITEC`, an approach to parallel test generation based on the `HITEC/PROOFS` package for sequential test generation. We demonstrated the feasibility and usefulness of implementing parallelism via concurrent object-oriented programming techniques. Results of `ProperHITEC` running on three different parallel machines show that `ProperHITEC` achieves results virtually identical in quality to `HITEC` while achieving significant multiprocessor utilization.

REFERENCES

[1] S. Parkes, J. A. Chandy, and P. Banerjee, "ProperCAD II: A run-time library for portable, parallel, object-oriented programming with applications to VLSI CAD," Tech. Rep. CRHC-93-22/UILU-ENG-93-2250, Center for Reliable and High-performance Computing, University of Illinois, Dec. 1993.

[2] G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

[3] S. Chandra and J. H. Patel, "Test generation in a parallel processing environment," in *Digest of Papers, International Conference on Computer Design*, pp. 11–14, Oct. 1988.

[4] S. Patil, P. Banerjee, and J. Patel, "Parallel test generation for sequential circuits on general purpose multiprocessors," in *Proceedings of the Design Automation Conference*, (San Fransisco), June 1991.

[5] P. Agrawal, V. D. Agrawal, and J. Villoldo, "Sequential circuit test generation on a distributed system," *Proceedings of the Design Automation Conference*, June 1993.

[6] V. N. Rao and V. Kumar, "Parallel depth first search, part I: Implementation," *International Journal of Parallel Processing*, vol. 16, no. 6, 1987.

[7] A. Motohara, K. Nishimura, H. Fujiwara, and I. Shirakawa, "A parallel scheme for test-pattern generation," in *Digest of Papers, International Conference on Computer-Aided Design*, pp. 156–159, Nov. 1986.

[8] S. Patil and P. Banerjee, "A parallel branch and bound approach to test generation," *IEEE Transactions on Computer Aided Design*, vol. 9, pp. 313–322, Mar. 1990.

[9] S. Arvindam, V. Kumar, V. N. Rao, and V. Singh, "Automatic test pattern generation on parallel processors," tech. rep., Computer Science Dept, Univ. of Minnesota, May 1990.

[10] B. Ramkumar and P. Banerjee, "ProperTEST: A portable parallel test generator for sequential circuits," *Digest of Papers, International Conference on Computer-Aided Design*, pp. 220–223, Nov. 1992.

[11] W. Fenton, B. Ramkumar, V. A. Saletore, A. B. Sinha, and L. V. Kalé, "Supporting machine independent programming on diverse parallel architecturs," in *Proceedings of the International Conference on Parallel Processing*, Aug. 1991.

[12] R. H. Bell, Jr., R. H. Klenke, J. H. Aylor, and R. D. Williams, "Results of a topologically partitioned parallel automatic test pattern generation system on a distributed-memory multiprocessor," *ASIC '92*, Sept. 1992.

[13] T. Niermann and J. Patel, "HITEC: A test generation package for sequential circuits," *Proceedings of the European Design Automation Conference*, pp. 214–218, Feb. 1991.

[14] S. Patil and P. Banerjee, "Performance trade-offs in a parallel test generation/fault simulation environment," *IEEE Transactions on Computer-Aided Design*, vol. 10, pp. 1542–1558, Dec. 1991.

[15] A. A. Chien, *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs*. The MIT Press, 1993.