International Conference on Computer Aided Design (ICCAD), November 1992

Portable Parallel Test Generation for Sequential Circuits*

Balkrishna Ramkumar
Dept. of Electrical & Computer Engg.,
University of Iowa,
Iowa City, Iowa 52242
ramkumar@eng.uiowa.edu

Prithviraj Banerjee
Center for Reliable & High-Perf. Computing,
University of Illinois,
Urbana, IL 61801
banerjee@crhc.uiuc.edu

Abstract

We report a new parallel test generation algorithm, ProperTEST, for sequential circuits that is **portable** *across a range of MIMD parallel architectures. It uses* **prioritized** *execution to ensure consistent speedups as the number of processors is increased. This consistency is achieved without loss in fault coverage with increase in number of processors. This also enables the use of parallel processing to improve the fault coverage when the execution time is bounded. Results on ISCAS 89 benchmark programs are provided on a shared memory machine, a message passing machine and a network of Sun workstations. ProperTEST was run* **unchanged** *on these different architectures.*

1 Introduction

Much of the work on parallel algorithms for CAD applications reported to date suffers from a major limitation. To date, parallel algorithms have been invariably designed with a specific underlying architecture in mind. As a result, these programs perform poorly on architectures other than the one for which they were designed. As a result,a parallel algorithm needs to be developed afresh for every target MIMD architecture. To make matters worse, the software development cycle for parallel programs is considerably longer than for sequential programs.

The goal of the ProperCAD project [5] at the University of Illinois is to address these problems. We have developed an environment for writing portable parallel CAD algorithms across MIMD machines.[1] All programs written using this environment run *unchanged* on a range of parallel machines including the Sequent Symmetry, the Intel i860 hypercube and a network of Sun Sparc workstations.

In this paper, we describe a new parallel algorithm for test generation, ProperTEST, for sequential circuits. ProperTEST is unique in that it is *portable* across MIMD machines, delivers *consistent* speedups with increase in the number of processors, and, at the same time, retains the quality of the fault coverage and efficiency of the test generator as the number of processors is increased. ProperTEST exploits parallelism at two levels: (a) the evaluation of different faults, and (b) during search for a sequence of test vectors for a given fault. It is based on a coarse-grained data-flow model of execution that makes it possible to obtain high processor utilization with low overhead. ProperTEST is also novel in that it is designed around an existing sequential algorithm. This makes it possible to automatically benefit from future improvements to the sequential algorithm.

2 The Parallel Algorithm

The parallel algorithm presented in this paper is based on the PODEM algorithm [2] and restricts itself to gate-level test generation for sequential circuits and a single stuck-at fault model.

In Figures 1 and 2 , we provide the outline of the ProperTEST algorithm. Initially one process is created per fault. Note that the destination processor for the processes is not provided. The underlying environment is responsible for load balancing these processes.

When a process for fault F is picked up for execution, the next primary or pseudo-input pi to be assigned is identified (together with its assigned value v and the time frame in which the assignment was made), based on heuristics defined in the uniprocessor code. Typically, this is one iteration of the PODEM

*This research was supported in part by the National Aeronautics and Space Administration under grant NAG 1-613, in part by the Semiconductor Research Corporation under grant SRC 91-DP-109, in part by the National Science Foundation Presidential Young Investigator Award NSF MIP 86-57563 PYI.

[1] This environment is built on top of a platform for portable parallel programming called the Chare kernel [1].

```
(1) Initially create one process per fault (see Figure 2)
      Distribution of processes not concern of programmer
      All processors maintain a bit vector (one bit per fault) and
            a count per fault for managing time/backtrack limits
      Every process carries a depth bound =
            # of PI/PSI's to be assigned in parallel

(2) When a process P for fault F is picked up:
      If bit for F in bit vector is set, discard P
      If count for F indicates time/backtrack limit exceeded,
            discard P
      If number of PI/PSI's assigned for fault F < depth bound
            Get next PI/PSI according to chosen heuristic
            Create one process for each of the two possible
            assignments to next PI/PSI
            Terminate process P
      Else
            Perform test generation sequentially using best
                  available uniprocessor algorithm
            If test vectors found for F or F aborted
                  Set bit for fault F on local processor
                  If test vectors found
                        Run fault simulator
                        Set bits in bit vector for all faults covered
            Broadcast bit vector to all other processors
            Update count for Fault F
            Broadcast count to all other processors

(3) If message picked up is a bit vector (due to broadcast)
      Bitwise OR the bit vector with local processor bit vector

(4) If message picked up is a count for Fault F
      Update local copy of count for Fault F
```

Figure 1: An outline of the ProperTEST algorithm for parallel test generation.

loop in which no backtracks are permitted. Two process are then created: one carries the assignment v to pi and the second process carries the assignment $pi =$ complement of v. The two processes thus represent two disjoint regions of the search space defined by the choice point at pi on the stack. The creation of processes in this manner is carried on recursively. At any node N in the search tree, process P representing N carries the sequence of assignments to primary or pseudo-inputs carried by its parent process together with the assignment made to "next" primary or pseudo-input at N. Assignments made in the parallel search space are not 'backtrackable'. Should the current set of assignments lead to a conflict, the process simply terminates. The recursion terminates at a predefined depth in the search space.

When the recursion is terminated, each leaf node L in the parallel search space represents a sequence of assignments to primary or pseudo inputs made along the path from the root of the search tree to L. A uniprocessor test generator is then invoked with these assignments to primary or pseudo-inputs marked as fixed non-backtrackable assignments. Processes representing these nodes in the search space are the *leaf* processes in the parallel search tree. The remainder of the search is then performed sequentially until a

Figure 2: A typical search space created by ProperTEST.

user-specified time limit or backtrack limit has been reached.

3 Prioritized Execution

In test generation for sequential circuits, fault ordering has been shown to have a significant effect on the total execution time [4]. To achieve this, a preprocessing pass is used to determine a good fault ordering. In our algorithm, we create one process per fault in the fault set and assign them priorities that reflect the ordering from such a preprocessing pass.

At run time, the user provides an overall time (or backtrack) limit T per fault which bounds the total amount of computation time that may be devoted to one fault. Note that this limit is cumulative across the available processors.

To provide consistent speedups, we also assign priorities to nodes in the search space to reflect the order in which they are visited by the underlying uniprocessor test generator. Bit-vectors are used to represent priorities; two bit-vectors are compared using lexicographic ordering. In lexicographic ordering, bits are compared from left to right and bit 0 has higher priority than bit 1. For example, the bit-vector 00100 has higher priority than 01. The assignment of priorities to nodes in the search space for a fault is illustrated in Figure 3. The use of priorities in this manner for combinatorial search can also be found in [6].

This assignment of priorities (see Figure 3) ensures that the available processors focus their effort in the leftmost bottom-most region of the search space.

Figure 3: The use of priorities to guide the parallel search so as to minimize wasted effort during speculative search.

Table 1: Execution times (in seconds) of ISCAS89 sequential benchmark circuits on a network of Sun Sparc workstations. The figure in brackets denotes the time limit imposed per fault in seconds.

Network of Sun workstations

Circuit	PE	Tgen.	Fsim.	Cov.	Eff.	Vec.
s386 (15)	1	311.1	3.0	81.8	100	333
	8	39.2	1.2	81.8	100	369
s713 (2)	1	42.2	3.9	81.6	97.6	182
	8	7.1	1.2	81.9	98.3	246
s1196 (2)	1	8.8	13.2	99.8	100	365
	8	1.7	4.1	99.8	100	370
s1238 (2)	1	18.6	15.8	94.7	100	383
	8	3.7	5.2	94.8	100	406
s5378 (1)	1	1384.2	200.2	70.6	72.1	849
	8	298.7	49.6	68.6	70.2	1095

This maximizes the likelihood that the search is performed in a region that would have been explored during uniprocessor search. This, in turn, tries to maximize the overlap between the search spaces explored during uniprocessor and parallel search. Since the use of priorities also cause the same search order during parallel search, we get consistent speedups and quality with a very high degree of probability.

4 Performance

In this section, we present the performance of the ProperTEST test generator on a variety of parallel machines, including a Sequent Symmetry, an Intel i860 hypercube and a network of Sun Sparc workstations. We would like to emphasize that ProperTEST ran *unchanged* on all these machines. The SCOAP [3] heuristic was used to obtain all the data presented in this paper. A limit of 255 time frames per fault was imposed. All data was obtained using time limits on each fault.

Table 2: Execution times (in seconds) of ISCAS89 sequential benchmark circuits on the Sequent Symmetry. The figure in brackets denotes the time limit imposed per fault in seconds.

Sequent Symmetry (Shared Memory)

Circuit	PE	Tgen.	Fsim.	Cov.	Eff.	Vec.
s386(15)	1	164.4	15.1	78.4	96.6	255
	16	143.5	5.5	78.9	97.1	403
s713(1)	1	41.6	25.9	81.9	95.9	206
	16	4.6	5.8	81.9	95.7	300
s1196(1)	1	46.2	72.0	99.6	99.8	369
	16	3.9	11.7	99.8	100	412
s1238(1)	1	85.3	87.3	94.5	99.0	376
	16	7.7	14.8	94.6	99.0	442
s5378(1)	1	1648.4	1078.7	66.1	67.2	769
	12	181.2	198.7	65.4	66.5	1141

Table 3: Execution times (in seconds) of ISCAS89 sequential benchmark circuits on the Intel i860 hypercube. The figure in brackets denotes the time limit imposed per fault in seconds.

Intel i860 hypercube (Message Passing)

Circuit	PE	Tgen.	Fsim.	Cov.	Eff.	Vec.
s386(15)	1	184.4	2.7	81.8	100	330
	8	28.8	1.3	81.8	100	418
s713(2)	1	27.0	3.7	81.9	98.8	217
	8	6.6	1.0	81.9	98.8	221
s1196(1)	1	5.5	10.1	99.8	100	365
	8	0.9	3.2	99.8	100	394
s1238(1)	1	11.7	12.3	94.7	100	383
	8	2.2	3.7	94.8	100	406
s5378(5)	1	6016.5	184.8	73.4	75.3	985
	8	901.7	38.6	70.8	72.7	1228

No changes were made to the underlying uniprocessor test generator. On one processor the behavior of ProperTEST is *identical* to the uniprocessor test generator except for the overhead of process creation incurred by the former.

In Tables I-IV, the execution times for a subset of the ISCAS 89 benchmarks are reported.[2] The execution times reported do not include the time to read in the circuit or to print the results.

In Tables I-III, for each circuit, column 2 reports the number of processors used, columns 3 and 4 provide the time in seconds spent for test generation and fault simulation. Columns 5 and 6 give the fault coverage (i.e. percentage of faults which were covered by the test generator and fault simulator during execution), and the test generator efficiency (i.e. the percentage of faults for which tests were generated,

[2] Data on the other ISCAS 89 sequential benchmark circuits are available from the authors on request.

Table 4: The use of parallel processing to improve the fault coverage and test generator efficiency. All times reported are in seconds. The figure in brackets denotes the time limit imposed per fault in seconds.

Intel i860 hypercube

Circuit	# PEs	Exe. Time	Coverage	Efficiency
s344 (2)	1	63.5	90.6	91.8
s344 (20)	8	97.5	93.6	95.0
s349 (2)	1	60.6	90.6	92.3
s349 (20)	8	109.9	93.4	95.4
s820 (2)	1	1004.0	52.1	55.2
s820 (20	8	1081.5	56.2	60.4
s832 (2)	1	1048.0	48.5	53.3
s832 (20)	8	1053.8	55.3	61.1
s1494 (2)	1	1826.6	44.0	46.8
s1494 (10)	8	756.6	70.2	73.4

or were identified to be redundant.) for each circuit. In column 7, we provide the number of test vectors produced during the entire execution.

In Tables I-III, for each circuit, the test generation and fault simulation times reported represent the time spent in the *uniprocessor* test generator code. This data does *not* include the time to create processes, load balance them, kernel overheads, communication overheads etc. These constitute the total overhead of parallelization. On one processor, the overhead observed for the ISCAS 89 circuits was less than 5% of the execution time on all machines. It was highest, not surprisingly, on the network of workstations. On the other two machines, the overhead on one processor averaged less than 2% of the execution time.

4.1 Improving Execution Time

In Tables I-III, we present the performance of ProperTEST on a network of 8 Sun Sparc workstations, a Sequent Symmetry, an Intel i860 hypercube. For each circuit, the data reported includes the time limit per fault, the test generation time and the fault simulation time. Good speedups were consistently obtained for all the ISCAS 89 circuits. Also evident is the consistent quality of the results obtained. The quality of the fault coverage and efficiency is retained as is evident in Tables I-IV. Some small degradation is seen for the number of test vectors produced which increases as the number of processors is increased. This can be reduced even further by better heuristics for fault ordering.

4.2 Improving Fault Coverage

In Table IV, we show how parallel processing can be used to improve the fault coverage and efficiency of the test generator. ProperTEST was run on five ISCAS 89 benchmark circuits on the Intel i860 hypercube. The time limit per fault was increased as the number of processors were increased as shown in Table IV so as to give a fixed total execution time for test generation. A consistent improvement in fault coverage and efficiency was clear with increase in the number of processors. Once again, the use of priorities was instrumental in keeping wasted search to a minimum. This is evident from the modest increase (on average) in the number of processes created for each benchmark circuit as the time limit is increased.

5 Conclusions and Future Work

We have designed and implemented a portable parallel test generator, ProperTEST, that is cleanly and efficiently interfaced with a PODEM-based uniprocessor test generator using the ProperCAD environment [5]. We have shown that ProperTEST runs *unchanged* on a range of MIMD machines including shared memory machines, message passing machines and a network of workstations. A novel algorithm based on the use of bit-vector priorities was used to deliver good consistent speedups without degradation in the fault coverage and test generator efficiency. We believe that this is the first parallel test generation algorithm for sequential circuits to effectively exploit nonshared memory machines and a network of workstations in this manner.

Another important feature of this algorithm is that improvements to the uniprocessor test generator will result in improvements to the parallel performance of the ProperCAD algorithm without any effort. It is only necessary to keep the interface between the uniprocessor and parallel test pattern generator unchanged. We expect to report improved performance from the parallel test generator as and when new optimizations for the uniprocessor test generator are developed.

Our sincere thanks to Srinivas Patil for providing us with his uniprocessor code for test generation.

References

[1] Fenton, W., Ramkumar, B., Saletore, V.A., Sinha A.B., Kalé, L.V. Supporting Machine Independent

Programming on Diverse Parallel Architectures. In *ICPP*, Aug. 1991.

[2] Goel, P. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Trans. Comp.*, C-30:215–222, Mar. 1981.

[3] Goldstein, L.H., Thigpen, E.L. SCOAP: Sandia Controllability/Observability Analysis Program. In *Proceedings of the 17th DAC*, Jun. 1980.

[4] Niermann, T. *Techniques for Sequential Circuit Automatic Test Generation*. PhD thesis, Dept. of Elec. & Comp. Engg., Univ. of Illinois at Urbana-Champaign, Sep. 1990.

[5] Ramkumar, B., Banerjee P. ProperCAD: A Portable Object-oriented Parallel Environment for VLSI CAD. In *ICCD*, Oct. 1992.

[6] Saletore, V.A. *Machine Independent Parallel Execution of Speculative Computations*. PhD thesis, Dept. of Elec. & Comp. Engg., Univ. of Illinois at Urbana-Champaign, 1991.