1995

Parallel Test Generation With Low Communication Overhead

Sivaramakrishnan Venkatraman
LSI Logic Corporation

Sharad C. Seth
University of Nebraska-Lincoln, seth@cse.unl.edu

Prathima Agrawal
AT&T Bell Laboratories, Murray Hill. NJ

Proceedings of the 8th International Conference on VLSI Design, 1995. doi: 10.1109/ICVD.1995.512088

Parallel Test Generation With Low Communication Overhead

Sivaramakrishnan Venkatraman
LSI Logic Corporation
Milpitas, CA 95035

Sharad Seth
University of Nebraska-Lincoln
Lincoln, NE 68588-0115

Prathima Agrawal
AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

In this paper we present a method of parallelizing test generation for combinational logic using boolean satisfiability. We propose a dynamic search-space allocation strategy to split work between the available processors. This strategy is easy to implement with a greedy heuristic and is economical in its demand for inter-processor communication. We derive an analytical model to predict the performance of the parallel versus sequential implementations. The effectiveness of our method and analysis is demonstrated by an implementation on a Sequent (shared memory) multiprocessor. The experimental data shows significant performance improvement in parallel implementation, validates our analytical model, and allows predictions of performance for a range of time-out limits and degrees of parallelism.

1 Introduction

In this paper we present a parallel implementation of the boolean satisfiability algorithm in which a test for a given fault (if it exists) is found cooperatively by the available processors. The results of test generation for the hard-to-detect faults in the ISCAS-85 benchmark circuits prove that the parallel test generation scheme results in significantly improved fault coverage and CPU time. We also propose a probabilistic model which accurately predicts the performance of parallel versus sequential implementation.

Our contribution differs from the earlier work in several important ways:

(a) *Processor Scheduling and Scalability*: In our scheme each processor can schedule another idle processor and distribute work. There is no centralized scheduler that can become a bottleneck with large number of processors. Once scheduled a processor can proceed independently. We also try to minimize the communication overheads for scheduling a new processor.

(b) *Search Space Partitioning*: We use a greedy heuristic for search space partitioning which tends to break the search space evenly amongst available processors and is easy to implement. Our heuristic is much simpler than that used by Patil and Banerjee [1], yet the results show it to be very effective.

(c) *Multiple Heuristics*: Both our sequential and parallel algorithms incorporate four different heuristics in order; one is tried only if all the preceding ones have time out. The effectiveness of orthogonal heuristics has been shown for sequential algorithms [2, 3] and suggested for parallel ones [4] but not evaluated on actual implementations. Our results show that in the context of good search-space partitioning, multiple heuristics are not as effective for the parallel case as they are for the sequential case.

(d) *Performance Analysis*: We believe our use of a *long-tail* distribution [5] to model the detection times of HTD faults accurately captures the intractability of test generation for a small fraction of faults using a given algorithm. The single parameter of the distribution is a measure of algorithm-specific testability of a circuit for HTD faults. The model is characterized and validated by experimental data. It can be used for predictions of performance for a range of time-out and fault-coverage limits.

Due to space limitation, we assume the reader is familiar with the boolean satisfiability formulation of the test generation problem and the concepts related to its solution, as described by Larrabee [3]. In particular, familiarity will be assumed with the following concepts: the *implication graph* constructed from the binary clauses in the boolean formula, the *base*

Support of this research by AT&T Bell Laboratories and a Collaborative Research grant from NATO is gratefully acknowledged.

level system for finding a satisfiable solution, and the heuristics added to improve the base level system. Amongst the various suggested heuristics (*nonlocal implications, active clauses, unique sensitization* etc.) we retain only the active clauses in our sequential implementation and add the following four to define a static ordering of the variables:

(F1) Forward Cone with one: Associate with each variable the number of nodes in the implication graph that are forced to be true when this variable is assigned the true value. Then sort the variables in the decreasing order of this number. The 2SAT solutions to the boolean formula are iterated in the *descending* order of variable assignments, that is, first the true value is tried for a variable and then, if backtracking makes it necessary, the false value is tried.

(B0) Backward Cone with zero: This is similar to the above except that the sorting key for variables is the number of nodes in the implication graph that are forced to be false when the false value is assigned to the variable. Further, the 2SAT solutions are iterated in the *ascending* order of variable assignments.

(F0) Forward Cone with zero: The variables are sorted as in *F1* but assigned in the *ascending* order.

(B1) Backward Cone with one: The variables are sorted as in *B0* but assigned in the *descending* order.

2 Parallel Test Generation

Three broad alternatives are available for parallelizing a test generation algorithm: *algorithmic partitioning, fault partitioning*, and *search-space partitioning*. We evaluated each alternative for parallelizing the boolean satisfiability algorithm and chose the last alternative. A detailed justification for this choice can be found elsewhere [6].

In search space partitioning multiple processors divide the space of input vectors amongst themselves in order to find a test for a given fault [7, 1]. An important design issue in search space partitioning is the choice of the target fault set. Because of the overhead involved in allocating disjoint parts of the search space to available processors and coordinating their results, search-space partitioning is not cost-effective for the easy-to-detect (ETD) faults. This is particularly true for the boolean satisfiability algorithm which was proposed for only hard-to-detect (HTD) faults even on a uniprocessor [3]. For this reason, we report the results of our parallel implementation only for the HDT faults.

Another important design issue is whether to use a *static* or a *dynamic* partitioning of the search space. We implemented a static partitioning algorithm on Sequent and found that (a) the processor utilization was quite low, and (b) the parallel implementation showed only marginal performance improvement over the sequential implementation [6]. These results led us to consider a dynamic partitioning of the search space.

2.1 Dynamic Partitioning

In the parallel test generation phase, each processor is given a complete copy of the implication graph and the set of 3CNF clauses so that they can proceed independently to find a solution. When one processor finds a solution it stops all the other processors. When all the processors exhaust their search space without a solution, the fault is declared to be redundant.

Amongst the many possible alternatives for dynamic allocation of search subspaces to processors, we chose a greedy distributed algorithm that tries to minimize communication during scheduling of a new process to an idle processor. All processors work completely asynchronously and no processor works as a centralized scheduler. Whenever a processor becomes free, it gets work dynamically from one of the busy processors. The background information in the next paragraph is essential to understanding the details of how this is accomplished.

In the test generation algorithm, a binary direction variable (*dir*) can be either "Forward" or "Backward". In the Forward direction the algorithm guarantees that the current variable assignments are consistent with the 3CNF clauses and the next yet-to-be assigned variable is chosen for assignment. The Backward direction, on the other hand, results from falsification of the formula by the current assignments; the next step of the algorithm is to backtrack and undo the most recent variable binding. A *choice point* is a variable that has been bound to a value in the forward direction and the alternate value is yet to be tried.

In our parallel implementation, when a processor is in the forward direction and finds another processor free, it splits work at the first (most recently) available choice point. It passes the partial assignment list (upto the choice point) to the new processor and readjusts its search space to reflect this reduced work (see Figure 2.1). This greedy work splitting is chosen in order to keep the message transferred between the processors minimal. The free processor, on receiving the work, initializes its implication graph with the partial assignment list and assigns the opposite (alternate) value to the choice point. Then the free processor carries out the implications of these assignments to recreate the starting state of the implication graph. Since

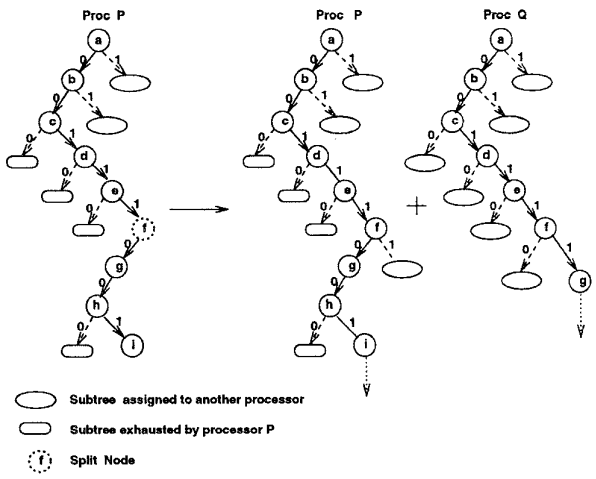

Figure 2: The dynamic sharing of work between processors.

there is no backtracking involved in this initialization, the time required is minimal: in the worst case it is proportional to the product of the number of assignments and the size of the implication graph. After initialization, a processor starts exploring its search space looking for a test.

In comparing the above scheme from that proposed by Patil and Banerjee [8] there are three major differences:

(a) Their scheme relies on a centralized scheduling of work compared with the distributed scheduling used here. When a large number of processors are involved, the scheduler can become a bottleneck in centralized scheduling.

(b) The search space is split in their scheme between two processors by assignment of alternate choice points in the search tree to each processor. Thus a much more complex data structure needs to be maintained for the search tree by each processor.

(c) In their scheme individual processors do not have any control on when the search space is split between processors. The scheduler does this whenever there is an idle processor available. We use a distributed control hence it is easy to incorporate strategies to minimize overhead of starting a new process. The current implementation includes the following two optimizations in addition to the basic search-space splitting scheme outlined above:

- A processor is not allowed to split its work until it has expended a certain threshold value of processing time between work splits.
- A processor is not allowed to split its work if it has less than certain threshold number of nodes to be assigned.

All processors work independently, giving and taking work from each other until a processor comes up with a test or all of them exhaust their search space.

3 PERFORMANCE ANALYSIS

It is assumed that test generation for a fault is aborted if it takes longer than a predefined *timeout* limit. We consider *speedup* and *fault coverage* as the measures of performance. Our analysis uses a probabilistic model to predict the performance of a parallel test generation algorithm on the HTD faults. It is based on the assumption that each HTD fault is independently targeted for test generation (which agrees with our implementation). However, it can be adapted to the case when faults are dropped by fault simulation.

3.1 Uniprocessor Test Generation Time

We assume a *long-tail* distribution for the detection times of HTD faults in a circuit for a given test generation algorithm. It is characterized by *values so removed from the mean, the median, and other 'typical indicators of location' that they do* not *seem to be generated by the same mechanism as the values near the median* [5]. Those with experience in running test generation programs will recognize this to be commonly the case with detection times of HTD faults.

Let $F_1(t)$ represent the probability of a randomly chosen HTD fault having a detection time less than t. In other words, $F_1(t)$ can be considered as the fraction of the HTD faults having a detection time less than t. According to the long-tail distribution:

$$F_1(t) = 1 - \frac{1}{(1+t)^{\alpha_S}} \quad t \geq 0, \quad \alpha_S \geq 0 \qquad (1)$$

Here, the parameter α_S may be regarded as an algorithm-specific testability measure of the circuit. The density function $f_1(t)$ is given by the derivative of the the distribution function $F_1(t)$

$$f_1(t) = F_1^{'}(t) = \frac{\alpha_S}{(1+t)^{\alpha_S+1}} \qquad (2)$$

Let X be the timeout value. Now, the average test generation time per fault comprises of two components:

1. the weighted sum of the time spent on the faults detected, $\int_0^X t.f_1(t)dt$, and
2. the weighted sum of the time spent on the faults aborted after the timeout value X, $X\left(\int_X^\infty f_1(t)dt\right) = X\left(1 - \int_0^X f_1(t)dt\right)$

Hence, the average test generation time $T_1(X)$, after simplification, is given by

$$T_1(X) = \frac{1-(1+X)^{\alpha_S-1}}{(1-\alpha_S)(1+X)^{\alpha_S-1}} \quad (3)$$

This equation gives the average test generation time per fault on a uniprocessor for a timeout value X.

3.2 N-Processor Test Generation Time

If we have N independent processors working on disjoint subspaces then we could expect the probability of a random HTD fault having a detection time greater than t to be $\frac{1}{(1+t)^{N\alpha_S}}$. But since the processors are not completely independent and there are overheads involved in parallelizing, the actual measure of parallelism is not N but a fraction of N. The fraction of HTD faults having a detection time less than t, with N processors is given by

$$F_N(t) = 1 - \frac{1}{(1+t)^{\alpha_P}}, \quad t \geq 0 \quad \alpha_P \geq 0 \quad (4)$$

where α_P is the algorithm dependent testability measure similar to α_S. Following a derivation similar to that of Equation (3), we get

$$T_N(X) = \frac{1-(1+X)^{\alpha_P-1}}{(1-\alpha_P)(1+X)^{\alpha_P-1}} \quad (5)$$

where $T_N(X)$ is the average test generation time per fault on the N-processor with a timeout value of X.

3.3 Performance measures

For a given value of timeout X, the speedup can be computed immediately as the ratio of $T_1(X)$ (Equation 3) and $T_N(X)$ (Equation 5).

The fault coverages in the two cases are obtained from Equations (1) and (4).

$$F_1(X) = 1 - \frac{1}{(1+X)^{\alpha_S}} \quad (6)$$

$$F_N(X) = 1 - \frac{1}{(1+X)^{\alpha_P}} \quad (7)$$

For a given fault coverage FC, the timeout value required in the sequential case, X_1 can be derived from Equations (6) and (7). Then substituting these values of timeouts in Equations (3) and (5) we can get an equation for speedup for a given fault coverage.

4 RESULTS

We implemented the parallel algorithm and evaluated its performance on the ISCAS-85 benchmark circuits. In this section we present uniprocessor and 6-Processor results of test generation for HTD faults based on our implementation. The HTD faults were obtained by running Podem on a uniprocessor with a small timeout limit. For the circuit C880, all the faults were detected in preprocessing only hence this circuit was not included in our evaluation.

All results were obtained on the Sequent Symmetry system with the Intel 80386 processor running at 16 MHz. First, a sequential version of the boolean satisfiability algorithm was implemented to obtain data for the uniprocessor performance. To calibrate the performance of this implementation we compared it against the Nemesis data [3] and found that, after accounting for the machine differences, our timings were quite comparable to the Nemesis system.

We further improved the performance of our sequential version before comparing it against the parallel implementation. This was achieved by trying the four heuristics mentioned in Section 1 serially, each with a timeout value of 1 sec. The heuristics were tried in the following order: F1, B1, B0, and F0. As expected, this implementation was able to achieve a higher fault coverage at the expense of longer CPU times.

The results of the sequential and parallel implementations with the composite heuristics, are shown in Table 1. We report only the times taken for satisfying the CNF formula; the times for parsing the circuit and extracting the boolean formula are not reported since they are incurred only once for each circuit.

For the parallel implementation, each processor ran composite heuristics with the same timeout value as the sequential implementation. Perfect fault coverage was achieved in all cases for the parallel implementation and the speedup values ranged from 1.29 to 12.34. As is evident from the table, the sequential implementation needed to rely on the multiple heuristics lot more often and could not achieve perfect fault coverage in some cases. This fact is essentially responsible for the superlinear speedup achieved in some cases.

Table 1: Sequential/Parallel Performance on Sequent with Composite Heuristics

Circuit	#HTD Faults	Detected or Proved Redundant by				Faults Dropped	% HTD FC	Time (sec)	Speedup
		F1	B1	B0	F0				
C432	12	9/12	1/-	0/-	1/-	1/0	91.7/100	99.8/8.0	12.34
C499	20	17/20	3/-	-/-	-/-	0/0	100/100	32.5/3.8	8.52
C880	0	-/-	-/-	-/-	-/-	-/-	-/-	-	-
C1908	37	36/37	0/-	0/-	0/-	1/0	97.3/100	49.5/10.5	4.70
C2670	94	71/94	23/-	-/-	-/-	0/0	100/100	275.3/123.5	2.22
C3540	166	166/166	-/-	-/-	-/-	0/0	100/100	79.7/61.9	1.29
C5315	82	82/82	-/-	-/-	-/-	0/0	100/100	90.6/64.4	1.41
C6288	10	2/9	1/0	4/1	0/-	3/0	70/100	226.8/43.9	5.16
C7552	650	621/637	7/9	0/4	0/-	22/0	96.6/100	2100.5/887.9	2.36

Additional experiments were carried out to validate our performance analysis model in Section 3. Because of space limitation we omit the details here and only describe below the results briefly (interested reader may want to refer to the report [6]).

The unknown parameters of our model, α_S and α_P, relate to the testability of the circuit for the sequential and parallel implementations respectively. These were estimated by nonlinear regression analysis on data obtained by running test generation on a sample of HTD faults. The fault coverage and the speedup projections (for a fixed timeout) from the model thus characterized were found to be close to the experimental value. The model was also used to estimate speedup for a fixed fault coverage.

5 CONCLUSION

Our processor scheduling algorithm involves no central processor and requires minimal amount of interprocessor communication. For this reason, we believe, the results reported here should apply equally well to loosely coupled scalable MIMD architectures that are increasingly becoming available. Since our parallel scheme is concerned only with search space partitioning and processor scheduling, it requires only an incremental effort over that required for implementing the boolean satisfiability algorithm on a standard workstation.

References

[1] S. Patil and P. Banerjee, "A parallel branch and bound algorithm for test generation," *IEEE Transaction on Computer-Aided Design*, vol. 9, pp. 313–322, March 1990.

[2] H. B. Min and W. A. Rogers, "Search strategy switching: An alternative to increased backtracking," in *Proc. International Test Conference*, pp. 803–811, August 1989.

[3] T. Larrabee, "Test pattern generation using boolean satisfiability," *IEEE Transaction on Computer-Aided Design*, vol. 11, pp. 4–15, January 1992.

[4] S. J. Chandra and J. H. Patel, "Test generation in a parallel processing enviornment," in *Proc. International Conference on Computer Design: VLSI in Computers and Processors*, pp. 11–14, IEEE CS Press, 1988.

[5] B. Mandelbrot, *Mathematical Explorations in Behavioral Sciences*, ch. The class of long-tailed probability distributions and the empirical distributions of city sizes, pp. 322–332. Homewood, Ill: Richard D. Irwin Inc. and the Dorsey Press, 1965.

[6] S. Venkatraman, S. Seth, and P. Agrawal, "Parallel test pattern generation using boolean satisfiability," Technical Report UNL-CSE-92-024, CSE Department, Univ. of Nebraska-Lincoln, 1992.

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

[8] S. Patil and P. Banerjee, "A parallel branch and bound approach to test generation," in *Proc. 26th Design Automation Conference*, pp. 339–345, June 1989.