An Adaptive Distributed Algorithm for Sequential Circuit Test Generation

James Sienicki and Michael Bushnell
ECE Department
Rutgers University
Piscataway, NJ 08855-0909

Prathima Agrawal and Vishwani Agrawal
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Abstract — We describe the parallelization of sequential circuit test generation on an Ethernet-connected network of SUN workstations. We use the observations of the previous work to execute the program in two phases. All processors simultaneously run the test generation program, Gentest. In the first phase, the fault list is equally divided among processors, each of which derives tests for targets from its list. A time limit is used to abandon the search for a test for hard to detect faults. Any test found is immediately used to simulate all faults, including those assigned to other processors. Due to the sequential nature of the circuit test vectors are not shared, but the fault simulation data are shared among processors. This phase terminates when only hard to detect faults are left. In the second phase, each remaining fault is simultaneously targeted by all processors, which now have different initial states of the circuit. The first processor to find the test interrupts all others, at which point all processors simultaneously target the next remaining fault. The results show that with this dual strategy, the speedup can be made to increase almost linearly, or sometimes superlinearly, with the number of processors.

I. Introduction

Most computer-aided design (CAD) environments of today contain networked computers. These computers (or workstations) are powerful enough for solving many problems. However, there are some CAD problems with exponential complexity. Cooperative use of several computers is desirable for these problems. Test generation for large sequential circuits is one such problem. The basic objective of the present work is to develop a distributed test generation system in which the speedup can be made to increase almost linearly with the number of processors. Theoretically, at least, such speedup may be possible as long as the problem partitioning does not require "grain" splitting. The grain refers to a small piece of calculation that cannot be distributed to multiprocessors.

We develop an adaptive procedure based on the observations reported in the previous work. Significant findings are listed below:

- In a sequential circuit, test vectors must retain the order in which they are generated. Thus, fault partitioning appears to be a reasonable strategy for parallelization [3].

- Unless faults can be partitioned into independent sets (a difficult problem for sequential circuits), multiprocessors will duplicate some computation. Thus, the test generated in one processor may detect faults that are assigned to other processors. Such duplication can be reduced if fault detection data are broadcast as soon as available [11]. This reduces duplication but does not completely eliminate it. In general, the communication on a distributed system can be much slower than the rate of computing possible.

- In certain cases, superlinear speedup has been observed. For hard to detect faults, the search for a test by a single processor takes enormously long time. Suppose the search space is partitioned and several processors simultaneously work in separate partitions. It is observed that often the processor first to find a test does it in much shorter time providing a superlinear speedup [1].

Based upon these observations, we present an adaptive two-phase strategy. The first phase uses fault partitioning in much the same way as described in previous papers [2, 3, 11]. Here, faults are equally divided among processors. They independently generate tests for the assigned faults and only share the fault simulation data. However, the time limit per fault is kept small. As a result, hard to detect faults are abandoned in this phase. This results in considerable savings in the overall time of this phase. In the second phase the abandoned faults are tried one at a time. All processors work on the same fault starting at a different initial state of the circuit. The initialization is provided by the vectors generated in the first phase. The first processor to find a test interrupts all other processors. All
remaining faults are simulated by the processor that has found the test. All processors then start on the next undetected fault.

The “adaptive” part in the two-phase scheme is related to finding the point where the first phase is terminated. Actually, the per-fault time limit, which is responsible for the termination of the first phase, is dependent on the circuit size and structure. The first phase is, therefore, run in passes. These passes are consistent with Gentest’s normal strategy [5]. The program starts with a very small per-fault time limit. After all faults are tried, time limit is increased and the remaining faults are tried again. This process is continued until it is found that the increased time limit has no significant advantage. The remaining faults are then dealt with one at a time by all processors cooperatively. The second phase is somewhat similar to the search-space parallelism. Because of the different initializations, the processors search for a test in different portions of the state-space.

II. Prior Work

There are several techniques for parallelizing ATPG. Fault parallelism partitions the fault list of a circuit among multiple processors. Search space parallelism attempts to reduce individual fault processing times by dividing the search space of a fault among multiple processors. Heuristic parallelism uses different heuristics on multiple processors, relying on the fact that a heuristic performs well on some faults, but poorly on others. The prior work can also be divided between combinational and sequential circuit ATPG. Combinational circuits require only one vector to test a fault, while sequential circuits require multiple vectors to sensitize the fault and propagate its effect to a primary output. These vectors must be applied in the same order in which they were generated, prohibiting the mixing of vectors generated by separate processes.

Agrawal et al. [3] showed the feasibility of performing sequential circuit ATPG with a synchronous version of Gentest using fault parallelism. Their test generation proceeds in synchronized computation and communication phases. The fault list partitions are distributed among the processes which share the detected fault data to prevent redundant computation.

Sienicki et al. [11] have demonstrated an asynchronous version of parallelization that does not require a process to wait for others to finish processing before proceeding to the next computation phase. They have reported superlinear speedups for several benchmark circuits and have provided a mathematical model to describe the results.

Fujiwara and Inoue [7] used a network of workstations to perform combinational ATPG. They use a client-server architecture, where faults are distributed by the server to client processes which generate tests. Faults are distributed in blocks, rather than one at a time, in order to reduce communication overhead. When a client process finishes processing its target faults, it returns the generated vectors and requests another block of faults from the server. They calculate the optimal number of faults for each block, based on communication and computing time.

Patiil and Banerjee [9] analyze fault parallelism for combinational circuits and propose techniques for partitioning the fault list. Their objective is to minimize redundant computation and vector sequence length.

Aguado et al. [4] partition a combinational circuit fault list on a network of workstations using independence and compatibility measures to balance load and reduce redundant computation. Their test generation is performed in three phases: initialization, vector generation, and vector collection. In a parent-child scheme within each computer, the parent process performs ATPG and the child process communicates test vectors to other child processes to be used for simulation and fault dropping. Near-linear speedups were reported for up to five processes running on separate processors. Klenke et al. [8] present a workstation-based ATPG system for combinational (or full-scan sequential) circuits with similar characteristics.

Zaidi and Szygenda [14] have analyzed the various parallelization methods, and concluded that fault parallelism achieves the most consistent speedups.

Chakradhar et al. [6] have recently given a neural network formulation of the ATPG problem with potential for fine grain parallelization. A similar formulation, known as Boolean satisfiability, has been parallelized by Venkatraman et al. [12].

III. A Parallel Processing Model

A. Fault Parallelism

We assume the same model for fault parallelism as presented by Sienicki et al. [11]. We assume that it takes \( k \) units of computation time to find a test for a fault by the ATPG program. Alternatively, we can view these units as the average time to test a fault.

1. Single Process – We assume that after a test has been generated for a fault, that a fraction, \( d \), of the remaining faults will be detected through fault simulation. After fault simulation, any one of the remaining faults is chosen for ATPG. This process is repeated until all faults have been detected or proved redundant.
This model yields a closed form formula for computational effort:

\[
E(N) = k \frac{\ln\left(\frac{1-d}{N-1}\right)}{\ln(1-d)}.
\]

The limiting value of \(E(N)\) as \(d\) tends to zero (no benefit from fault simulation) is \(N\), the number of faults. As \(d\) tends to unity (all remaining faults detected through fault simulation), \(E(N)\) becomes 1.

2. **Multiple Processes** – Consider any one of \(p\) processes, each of which is given \(N/p\) faults. The first fault in each processor takes \(k\) units of time. In this time, however, \(p\) tests are generated. Applying the appropriate changes to the effort formula above yields the following multiprocessor effort formula, \(E(N, p)\):

\[
E(N, p) = k \frac{\ln\left(\frac{1-d^p}{N-p}\right)}{\ln(1-d^p)}.
\]

Since \(E(N)\) is the total computing effort of one processor, the speedup, \(S(N, p)\), is obtained by dividing the single processor effort, \(E(N)\), by the multiprocessor effort, \(E(N, p)\), yielding:

\[
S(N, p) = \frac{\ln\left(\frac{1-d}{N-1}\right)}{\ln\left(\frac{1-d^p}{N-p}\right)}.
\]

This equation is plotted for various values of \(d\) in Figure 1.

![Speedup versus number of processors](image)

**Figure 1: Speedup versus number of processors**

**B. Search Space Parallelism**

The previous models do not consider the effects of the circuit state on the difficulty of ATPG. The computing effort, \(k\), for a fault can vary depending upon the starting state assumed by the ATPG process. This time to detect a fault can be reduced by resorting to *search space parallelism* where the same fault is targeted on multiple processors simultaneously. All processes stop when one process generates a test for a fault or proves that it is redundant. This process can be modeled using *order statistics* [13]. The *minimum order statistic* is the average of the minimum of \(p\) random samples and is denoted by \(\mu_{1,p}\), and is calculated using the following formula

\[
\mu_{1,p}(x) = p \int x [1 - F(x)]^{p-1} \, dF(x),
\]

where \(x\) is a random variable and \(F(x)\) is the *cumulative distribution function* of \(x\). The time for the first process to detect a fault can be approximated using order statistics if the processing time is assumed to be a random variable. This is a reasonable assumption for state dependent faults. What remains is to choose the distribution that best models this state dependency. A convenient distribution is \(x^\delta\), which can model sublinear to superlinear speedups. This distribution has the following minimum order statistic [13]:

\[
k = \mu_{1,p}(x) = \frac{p!}{\prod_{1 \leq j \leq p} (j + \delta)}.
\]

The actual value of \(\delta\) must be determined empirically by gathering statistics for several circuits.

**C. Hybrid Parallelism**

The two models of fault parallelism and search space parallelism can be combined if \(p\) available processors are divided into groups of \(n\) processes. By combining equations 1 and 2 one obtains the following formula

\[
S(N, p, n) = \frac{p \ln\left(\frac{1-d^p}{N-p}\right) \prod_{1 \leq j \leq n} (j + \delta)}{\ln\left(\frac{1-d^p}{N-1}\right) n!(1 + \delta)}.
\]

This quantity can be maximized by choosing the appropriate value of \(n\). A plot of this equation is shown in Figure 2 for 64 processors, two values of \(d\), and \(\delta = 1.5\). It can be seen that for small values of \(d\) the optimal partitioning strategy is pure fault parallelism. For larger values of \(d\), the optimal strategy is a combination of state space and fault parallelism (in this case 8 processors targeting the same fault is optimal). Extrapolating from Weide [13], for values of \(\delta\) between zero and one-half, the optimal strategy is always state-space parallelism.
An implementation that attempts to maximize the speedup must be able to estimate the parameter $d$ from experimental results. One such implementation is described in the following section.

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{speedup_plot.png}
\caption{Allocation of 64 processors with $\delta = 1.5$}
\end{figure}

### IV. Adaptive Algorithm

As described above, the adaptive algorithm is divided into two phases. The first phase performs fault parallelism using the \textit{synchronous algorithm} [3], where each processor performs ATPG on a portion of the undetected faults, but performs fault simulation on all vectors. This is done to drop faults from other processors that are detected by a vector on a different processor. The processors share detected faults to reduce redundant computation. The first phase continues through several iterations of fault parallelism. The first iteration is performed with a time limit of 1 second per fault. If the fault is not detected after 1 second of processing, then it is abandoned. This continues until all faults have been tried. Then the undetected faults are gathered and redistributed among the processors, which then perform another iteration with double the time limit per fault. This continues until some maximum time limit per fault is reached.

The second phase performs state space parallelism, where all processors target the same fault with the same time limit per fault. The first processor to detect a fault or prove it untestable interrupts the execution of the remaining processors. Each fault is attempted only once (one pass through the fault list). Figure 3 gives pseudocode for this algorithm.

### V. Adaptive Algorithm Experimental Results

Table 1 shows the results for 1 to 16 processors using the adaptive method. We used a system of Sparc 10 workstations communicating through Ethernet. The single processor case was run with a 16 seconds per-fault time limit. Of the 474 faults, 424 were detected by 761 vectors. Fifteen faults were found untestable, giving a fault efficiency of 92.6 percent. The CPU time on a Sparc 10 was 2.075 seconds. In the two phase runs for 2 to 16 processors, the per-fault time limit of 4 seconds was used for the first phase. The faults remaining after the first phase were tried by all processors in the second phase. Here the time limit per fault was reduced as the number of processors increased. We notice superlinear speedups for 2 and 4 processors. For eight or more processors to give a similar performance we will need larger problems.

We notice the increase in the number of test vectors, which characteristic of multiprocessor systems. This increase mainly occurs in the first phase where any test generated usually detects many faults. The increase can be reduced by, (a) an improved fault partitioning, and (b) more efficient communication between processors. Such considerations are being studied.

For comparison, Table 2 shows the results of the single-phase strategy. The results here differ from other papers [3] because of different parameters (per-fault time limit, etc.) used by Gentest. Clearly, adaptive method gives better speedup.

Our present experiments are aimed at determining the optimum heuristics for time limits and phase switching. We notice that in Table 1, the speedup beyond eight processors tends to drop below the ideal. Our investigation will also determine whether we are reaching the grain size limit. The circuit $s444$ is fairly small and it is unclear whether the search state space can be divided into 16 independent parts.
Table 1: Adaptive algorithm applied to s444 (474 faults)

<table>
<thead>
<tr>
<th>No. of Proc.</th>
<th>Per-fault time limit (s)</th>
<th>Detected Faults</th>
<th>Untestable Faults</th>
<th>No. of Vectors</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>None</td>
<td>424</td>
<td>15</td>
<td>761</td>
<td>2075</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>424</td>
<td>15</td>
<td>1225</td>
<td>734</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>422</td>
<td>14</td>
<td>1464</td>
<td>382</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td>422</td>
<td>14</td>
<td>1435</td>
<td>272</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>4</td>
<td>422</td>
<td>14</td>
<td>1902</td>
<td>190</td>
</tr>
</tbody>
</table>

Table 2: Single phase (fault parallelism) algorithm applied to s444 (474 faults)

<table>
<thead>
<tr>
<th>No. of Proc.</th>
<th>Per-fault time limit (s)</th>
<th>Detected Faults</th>
<th>Untestable Faults</th>
<th>No. of Vectors</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>424</td>
<td>15</td>
<td>761</td>
<td>2075</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>423</td>
<td>15</td>
<td>1402</td>
<td>864</td>
<td>2.4</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>423</td>
<td>15</td>
<td>2720</td>
<td>560</td>
<td>3.7</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>423</td>
<td>15</td>
<td>2970</td>
<td>314</td>
<td>6.6</td>
</tr>
</tbody>
</table>

VI. Remarks

A. Adaptive Parallelism

Based upon experimentation on most ISCAS 89 benchmark circuit, we stress that the simple adaptive method described above will not work well unless carefully tuned to the problem size [10]. Some hard-to-detect faults are, in fact, untestable and are not state dependent (they are abandoned). No benefit is achieved by targeting them on multiple processors. Some circuits do benefit from state space diversity (s5378), but in many cases the increased fault coverage (and slightly faster detection) did not compensate for the reduced speedup of having many processors working on the same fault simultaneously. The algorithm implemented also lacked a mechanism for determining the optimal processor partitioning strategy. The models presented above provide a theoretical foundation, but these must be coupled with experimental procedures to determine the values for the redundancy and state dependency parameters.

B. Fault Shuffling

Another point that can be made is that by using fault parallelism and shuffling faults we can get the benefits of trying the faults in multiple states, but without wasting time on all processor simultaneously. In this method faults that are not detected on a particular processor will be tried on a different processor in a subsequent iteration. With this method a fault is tried on another processor only if it was not detected by the first processor, reducing the penalty for non-state dependent and abandoned faults. Experimental results have shown that this method produces better fault coverage and faster run times than the traditional fault parallelism [10].

C. Varying the Number of Processors

An alternative adaptive algorithm that varies the number of processors for fault parallelism is also possible. This algorithm requires a model to continuously predict the optimum number of processors to reduce redundant computation and maximize speedup, then a method based on observed fault simulation results would work. This method fits well with the fault parallelism model presented above. We tried a scheme that uses a simple heuristic to switch from 1 processor to N processors after a single pass through the fault list (or through a fraction of the fault list). This heuristic of “filtering” the easy-to-detect faults in a single pass was found to reduce the number of vectors and retain the good speedups [10].

VII. Conclusion

It is generally accepted that the best advantage of multiprocessing is achieved if the scheme of parallelization could be specifically designed to suit the problem and the system architecture. In test generation, we find that the computing requirements can change as we proceed through the solution. An ideal system will, therefore, adapt. The present work is an attempt in this direction. We hope the ideas presented here will lead to further work.

Acknowledgment: This research was supported in part by the Center for Computer Aids for Industrial Productivity (CAIP), an Advanced Technology Center
of the New Jersey Commission on Science and Technology, located at Rutgers University.

REFERENCES


