TEST GENERATION IN A PARALLEL PROCESSING ENVIRONMENT

Susheel J. Chandra and Janak H. Patel

Computer Systems Group
Coordinated Science Laboratory
University of Illinois
1101 W. Springfield Ave.
Urbana, IL 61801

Abstract

The availability of low-cost, high-performance, general-purpose parallel machines has made parallel processing viable for the development of CAD applications. In this paper we will identify the key issues which surface when an attempt is made to parallelize the test generation process. We will illustrate how different test generation strategies can be mapped onto different classes of parallel machines including loosely coupled distributed systems, distributed-memory systems with message passing architectures and tightly coupled multiprocessor systems with shared global memory. We will predict the performance of these mapping strategies by using uniprocessor turnaround times and an estimate of the communication delays.

1. INTRODUCTION

The availability of low-cost, high-performance general-purpose, parallel machines has made parallel processing viable for the development of CAD applications. However, to effectively use these systems, new algorithms and techniques for mapping CAD applications onto parallel machines have to be developed. These algorithms must distribute the load evenly among processors and minimize the overhead from inter-processor communication and synchronizing activities [1].

In this paper we will identify the key issues which surface when an attempt is made to parallelize the test generation process. We will illustrate how different test generation strategies can be mapped onto parallel machines, and predict their performance by using uniprocessor turnaround times and an estimate of the communication delays. Three different classes of machines will be considered:

(1) A loosely coupled distributed system such as a collection of workstations connected through a local area network.
(2) Distributed-memory systems with message passing architectures e.g. the Intel iPSC.
(3) Tightly coupled multiprocessor systems with shared global memory, e.g. the Alliant FX series.

The performance issues and tradeoffs on distributed-memory multiprocessor systems are very similar to those on the loosely coupled distributed system. Tightly coupled systems differ from these two in that the communication delays are significantly less. For loosely coupled distributed systems we will use a uniform partitioning model to estimate the speedup achieved by using multiple processors.

The remainder of the paper is organized as follows. Section 2 identifies some issues that must be addressed in order to speedup the test generation process. In Section 3 strategies to parallelize test generation using a single heuristic are suggested and evaluated. Section 4 deals with multiple heuristic schemes to parallelize test generation. Section 5 discusses some proposed methods to exploit fine grain parallelism in the test generation process.

2. PARALLELIZING TEST GENERATION

There are two main issues that must be addressed in order to speedup the test generation process [2]:

(1) How to deal efficiently with a large number of faults.
(2) How to generate test-vectors for faults which cause a great number of backtracks.

Most test generation algorithms focus on the second problem. The two state of the art test generation algorithms PODEM [3] and FAN [4] both rely heavily on the use of heuristics to speed up the test generation process by reducing the number of backtracks. However, the effectiveness of the heuristics employed depends to a large extent on the quality of the testability measure.

Parallel processing seems to be a viable technique to tackle both these problems simultaneously. In the following sections we will present and evaluate strategies to map test generation algorithms onto parallel machines. Experimental results obtained by using a local implementation [5,6], of the PODEM algorithm will be presented. We will present both, simple strategies using a single heuristic as well as more sophisticated ones that use multiple heuristics.

3. PARALLEL TEST GENERATION USING A SINGLE HEURISTIC

The most obvious, simple and possibly the most effective scheme for exploiting parallelism in the test generation process is to distribute faults evenly among processors and let each processor independently generate test vectors for the faults assigned to it. The turnaround time can be further improved by partitioning and distributing the faults cleverly so that the load on each processor is approximately the same.

3.1. Systems with no Shared-Memory

These include both loosely coupled distributed systems connected through a local area network and tightly coupled multiprocessor systems with a distributed-memory that use message passing for communication. Figure 1 depicts the architecture of such a system. The individual processors do not share any memory. We assume that initially, both the circuit netlist and the test generation software reside in the local memory of the primary processor. We will consider two minor variations of the simple scheme. In the first variation the netlist of the circuit and the test generation program are broadcast by the primary processor to all the participating secondary processors. If the system does not have broadcast capability, then
the information will have to be transmitted sequentially or in other ways depending on the interconnection network of the system. All the work including parsing the circuit, calculating Controllability/Observability (C/O) values and all computations are replicated on each processor.

In the second variation of the simple scheme the primary processor performs certain tasks that are common and must be performed for each run of the test generation software. These include parsing the circuit and calculating the C/O values. This information is then broadcast to all the participating secondary processors along with the test generation program.

It is obvious that the turnaround time for both these variations will be approximately the same. However, the work performed by the individual processors will be slightly less for the second variation. Let us evaluate the speedup achieved by this scheme using a uniform partitioning model. The model assumes that the secondary processors receive the same amount of data and send back the same amount of results. It also assumes that the execution costs are identical and independent of the processor and the processed data. We will assume a network of \( n \) workstations connected via a packet oriented bus such as an ethernet. Let \( C(m) \) be the cost for sending \( m \) bytes from any processor to any other processor assuming no contention for the bus. Let \( B \) be the size of the broadcast data and \( R \) the size of the data received from each secondary processor. The speedup \( \sigma \) is then given by the following equation:

\[
\sigma = \left[ \frac{t}{t + C(B) + (n-1)C(R)} \right]^{(n)} \quad \text{(2.1)}
\]

where \( t \) is the execution time on each of the secondary processors and \( t \) is the turnaround time on a uniprocessor.

To get a feel for the numbers involved, consider a network of 5 SUN 3/50 workstations running Unix 4.2BSD, connected via a 10 Mbits/second ethernet. Although the ethernet is capable of delivering 1.25 Mbytes/second, the actual communication cost \( C \) involves cost due to software for several layers of network protocol. This cost \( C(m) \), was determined empirically [7], as a function of the number of bytes transferred \( m \), and is given by:

\[
C(m) = 7.93 \times 10^{-3}m + 9.62 \quad \text{(2.2)}
\]

where \( C(m) \) is expressed in milliseconds. Note that for a tightly coupled multiprocessor system using a message passing paradigm the communication cost \( C(m) \) would be considerably less. The speedup \( \sigma \) can thus be calculated using equations 2.1 and 2.2 and uniprocessor turnaround times. Table 1 gives the speedup numbers for \( n = 5 \) processors for the ISCAS benchmark circuits [8]. A local implementation of the PODEM algorithm [5,6] was used to obtain the uniprocessor turnaround times on a SUN 3/50. A CPU time limit of 2 secs/fault was imposed, and faults exceeding this limit were dropped.

The numbers in Table 1 indicate that using five processors will give near linear speedups for all the circuits. Figure 2 shows the speedup achieved as a function of the number of processors for three different circuits. For the smaller circuit c880, where the average time per fault is small and each processor is working on a small number of faults, the speedup achieved levels off after about 20 processors and eventually begins to start falling off around 64 processors. For the larger circuit c3540 the speedup is monotonically increasing even up to 128 processors. In this case, the average time per fault is higher and each processor is assigned about 26 faults. In Figure 3 the individual contributions of computation and communication to the total time are shown for the three circuits. It is evident from the graphs of Figure 3 that, as the number of processors increases, the communication time begins to play a major role in the total time taken and thus causes the speedup to decrease. Also, the number of processors at which the maximum speedup is achieved, increases with circuit size.

![Architecture of a distributed system](image)

**Figure 1: Architecture of a distributed system**

![Speedup versus number of processors](image)

**Figure 2: Speedup versus number of processors.**

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Number of Faults</th>
<th>Uniprocessor Time (ms/fault)</th>
<th>Speedup (( \sigma ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>c432</td>
<td>524</td>
<td>98.49</td>
<td>4.65</td>
</tr>
<tr>
<td>c499</td>
<td>758</td>
<td>298.05</td>
<td>4.90</td>
</tr>
<tr>
<td>c880</td>
<td>942</td>
<td>62.89</td>
<td>4.57</td>
</tr>
<tr>
<td>c1355</td>
<td>1574</td>
<td>270.32</td>
<td>4.92</td>
</tr>
<tr>
<td>c1908</td>
<td>1879</td>
<td>336.32</td>
<td>4.95</td>
</tr>
<tr>
<td>c2670</td>
<td>2595</td>
<td>293.72</td>
<td>4.98</td>
</tr>
<tr>
<td>c3540</td>
<td>3428</td>
<td>784.87</td>
<td>4.86</td>
</tr>
<tr>
<td>c5351</td>
<td>5350</td>
<td>239.67</td>
<td>4.86</td>
</tr>
<tr>
<td>c6288</td>
<td>7744</td>
<td>1046.50</td>
<td>4.99</td>
</tr>
<tr>
<td>c7552</td>
<td>7552</td>
<td>515.34</td>
<td>4.92</td>
</tr>
</tbody>
</table>

**Table 1: Speedup for \( n = 5 \) processors.**
3.2. Shared Memory Systems

For shared memory multiprocessors the broadcast time $C(B)$ will be eliminated. This is because the test generation program and the circuit netlist will reside in the common global memory. However, all the participating processors may not be able to access the shared memory at the same time. This will introduce additional delays and increase the turnaround time. The exact nature of memory interference will depend on the memory organization, the interconnection network, and the size of the local memory if any.

4. PARALLEL TEST GENERATION USING MULTIPLE HEURISTICS

There is empirical evidence [6] to suggest that certain heuristics are complementary in that faults dropped by one heuristic are often detected by the other. A composite strategy that uses multiple heuristics was evaluated in [9] and found to be very effective. This strategy uses a small time limit per fault but switches to a different heuristic for faults dropped in the first pass. In all five different testability measures were evaluated in [9] with a time limit of 2 secs/fault per measure. Faults dropped by one measure were passed on to the next measure, until all five measures were tried. Thus in the composite strategy, a fault which is dropped by all five measures will contribute 10 seconds to the total time on a uniprocessor.

In this section we will propose two different schemes to parallelize this composite strategy and discuss the related communication and synchronization issues. The first scheme is the uniform partitioning scheme in which faults are evenly distributed among processors and each individual processor computes tests for the faults assigned to it using the composite strategy. An alternate scheme is to have each individual processor use a different heuristic and work on the same fault at the same time. The first one to generate a test vector signals the others to abort and go on to the next fault. We will refer to this as the concurrent heuristic scheme.

4.1. Loosely Coupled Distributed Systems

The first scheme is very easily adapted to loosely coupled distributed systems and the issues involved and the speedup calculations are very similar to those in Section 3.1. The second scheme however is not very well suited to this type of architecture primarily due to lack of fast synchronization.

4.2. Tightly Coupled Multiprocessors Systems

The second scheme is more suitable for tightly coupled multiprocessor systems with synchronization support in hardware. To evaluate the speedup achieved by using this scheme, consider $N$ clusters of 5 processors each. Each processor in a cluster runs one of the 5 different heuristics evaluated in [9]. Faults are evenly distributed among the clusters. Consider the case where a single cluster of 5 processors is used. Each of the five processors in the cluster runs a different heuristic with a time limit of 2 secs/fault. Thus a fault that is dropped by all five measures will contribute only 2 seconds to the total time. Once again we will use uniprocessor turnaround times to demonstrate which scheme performs better. To get an estimate of the performance of the second scheme, tests were generated for each of the benchmark circuits using the five different heuristics, and the minimum of the five different times was used for each fault. Note that, this does not take into account penalties due to synchronization and memory contention. These results are summarized in Table 2.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Uniprocessor</th>
<th>Uniform Partitioning</th>
<th>Concurrent Heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>c432</td>
<td>189.31</td>
<td>4.81</td>
<td>2.13</td>
</tr>
<tr>
<td>c499</td>
<td>803.75</td>
<td>4.96</td>
<td>6.07</td>
</tr>
<tr>
<td>c880</td>
<td>61.74</td>
<td>4.56</td>
<td>1.11</td>
</tr>
<tr>
<td>c1355</td>
<td>304.36</td>
<td>4.93</td>
<td>1.20</td>
</tr>
<tr>
<td>c1908</td>
<td>867.14</td>
<td>4.98</td>
<td>2.85</td>
</tr>
<tr>
<td>c2670</td>
<td>675.12</td>
<td>4.95</td>
<td>3.44</td>
</tr>
<tr>
<td>c3540</td>
<td>1686.61</td>
<td>4.99</td>
<td>3.27</td>
</tr>
<tr>
<td>c5315</td>
<td>306.24</td>
<td>4.88</td>
<td>1.54</td>
</tr>
<tr>
<td>c6288</td>
<td>1185.72</td>
<td>4.99</td>
<td>2.02</td>
</tr>
<tr>
<td>c7552</td>
<td>656.63</td>
<td>4.94</td>
<td>1.83</td>
</tr>
</tbody>
</table>
It is interesting to note that in case of one particular circuit, c499, a speedup of over 6 was achieved using only five processors. This can be explained as follows. Consider the case where 5 different heuristics $H_1, \ldots, H_5$, are being used each with a time limit of 2 secs/fault. Suppose a fault is dropped by heuristics $H_1$ and $H_2$ but is detected by $H_5$ in 0.2 secs. In the uniprocessor case or in the uniform partitioning case, if the heuristics are run in the order $H_1$ through $H_5$, this fault will contribute 4.2 secs to the total time. However, in the case of the concurrent heuristic scheme this fault will contribute only 0.2 secs to the total time. Hence it is possible to get better than linear speedups in case of the concurrent heuristic scheme. In most cases however, the uniform partitioning scheme performs better than the concurrent heuristic strategy.

5. FINE GRAIN PARALLELISM

All of the strategies discussed and analyzed above, exploit coarse grain parallelism. However, more sophisticated approaches that attempt to exploit fine grain parallelism are being developed [2, 10]. One such approach exploits AND/OR parallelism during the decision making process [10]. In a conventional test generation algorithm running on a uniprocessor, when there are several ways to satisfy an objective, a choice is made. If this choice later proves to be incorrect, the algorithm backtracks and makes an alternate choice. In a parallel processing environment, we can evaluate these choices in parallel and thus avoid the need for backtracking. This is termed OR parallelism. Similarly, if in satisfying an objective, there are some necessary subgoals which have to be satisfied, these goals can be solved in parallel resulting in AND parallelism.

6. CONCLUSIONS

The most obvious and simple approach of dividing the faultlist uniformly over the available processors appears to be the most promising way to parallelize test generation. However, more sophisticated approaches which handle multiple nodes in the decision tree concurrently, or attempt to exploit AND/OR-parallelism in the decision making process [10] are being developed. Their performance on large practical circuits remains to be evaluated. It should be noted that fault simulation was not considered in any of the above analysis. If fault simulation is used as part of the test generation process, other approaches may have to be considered in order to parallelize fault simulation in conjunction with test generation.

7. REFERENCES