Logic Verification Algorithms and their Parallel Implementation

Hi-Keung Tony Ma, Srinivas Devadas and Alberto Sangiovanni-Vincentelli
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

Ruey-sing Wei
AT&T Bell Laboratories, Murray Hill, NJ

ABSTRACT: LOVER incorporates a novel approach to combinational logic verification and obtains good results when compared to existing techniques. In this paper we describe a **new verification algorithm**, LOVER-PODEM, whose enumeration phase is based on PODEM. A variant of LOVER-PODEM, called PLOVER, is presented. We have developed, for the first time, parallel logic verification schemes. Issues in **efficiently parallelizing both general and specific LOVER-based approaches** to logic verification over a large number of processors are addressed. We discuss parallelism inherent in the LOVER framework regardless of what enumeration and simulation algorithms are used. Since the enumeration phase is the efficiency bottleneck in parallelizing LOVER-based approaches, we have developed parallel versions of PODEM-based enumeration algorithms. Experimental results are presented to show that **high processor utilization** can be achieved when these parallelisms are exploited. Speed-up factors of over 7.8 have been achieved with 8 processor configurations.

1. INTRODUCTION

Logic verification tools compare the logic design of integrated circuits at different levels to make sure that, in the synthesis process, no *logic errors* have been introduced. For example, in a silicon compiler environment where a design is translated (synthesized) into a lower level from a higher level description, logic verification is usually performed between functional level (before logic synthesis) and gate level (after logic synthesis), as well as between gate level (before layout generation) and layout level (after layout generation).

Some formal techniques[Rot80,Rot73,Rot77] [Don76,Oda86,Bry85] were proposed in the past but only a few have been applied due to their complexity and computational requirements. In PROTEUS[Wei86] a number of efficient techniques for combinational logic verification has been developed and implemented. The PROTEUS system includes four basic approaches: verification by justification, verification by cube comparison, verification by exhaustive simulation, and verification by cover generation and simulation. The last approach, called LOVER (LOgic VERification) in PROTEUS, is novel and has given some good results compared to existing techniques. Most logic verification algorithms suffer from the problem of multiplicative blow-up. LOVER was developed with the specific goal of eliminating this problem.

In this paper, we present new LOVER-based approaches for verifying the boolean equivalence of two combinational logic circuits. These approaches compare favorably to other LOVER-based approaches.

Large complex logic circuits require significant amounts of cpu time to verify. Parallel logic verification algorithms are therefore extremely attractive. However, to date no logic verification technique has been efficiently parallelized.

Parallel logic verification algorithms based on the LOVER approach are presented. This approach can be extended to other verification techniques. These parallel algorithms can be implemented on a large number of processors while maintaining *high overall efficiency*.

This paper is organized as follows: In Section 2 we review the LOVER approach to logic verification. In Section 3 we describe embellishments to the basic LOVER approach and introduce new verification algorithms. *The parallelism inherent in the general LOVER approach* and a multi-processor implementation exploiting this parallelism is described in Section 4. In the same section a highly parallel version of a specific LOVER-based algorithm, which features a *novel parallel enumeration algorithm* based on PODEM, is described. Experimental results are presented to show that large speed-up can be achieved when either of these parallelisms are exploited.

2. PRELIMINARIES

This section describes the LOVER approach to verifying the equivalence of two logic circuits. LOVER incorporates a two-set/two-phase approach which avoids the multiplicative blow-up problem in traditional logic verification methods and has achieved good results in comparison to existing approaches [Wei86].

Let A and B be the two circuits whose equivalence has to be verified. A cube c from C_A^{ON}[Bra84] (the ON-set cover of circuit A) is *enumerated* and *simulated* on B to check if B produces a 1 at its output. If so, the enumeration/simulation process continues with another cube from C_A^{ON}. If, on the contrary, a 0 appears, the verification is completed with the conclusion that A and B are not Boolean equivalent. If an x (unknown) appears, c is split (cube-split) into smaller cubes and re-simulated until a known value appears at the output of B. Cube-splitting and simulation are implicitly exhaustive. The process continues until all cubes from C_A^{ON} have been simulated. A similar process for C_A^{OFF} (the OFF-set cover of circuit A) is then carried out.

This method is called a two-set/two-phase approach because there are two sets (the ON-set C^{ON} and the OFF-set C^{OFF}) that are to be explicitly verified; and two phases (the enumeration phase and the simulation phase) that are to be performed for each set verification. It is important to note that this framework does not specify which enumeration or simulation algorithm to use. This gives a large degree of freedom in the LOVER approach to verification - many different kinds of simulation and enumeration algorithms can be used. Since simulation is a relatively well understood and developed area, the emphasis is generally placed on developing efficient enumeration algorithms.

Using LOVER, there are only ($n_{ON,A}+n_{OFF,A}$) cube enumeration and simulation passes to be performed where $n_{ON,A}$ and $n_{OFF,A}$ are the number of cubes in C_A^{ON} and C_A^{OFF}, respectively. Though both $n_{ON,A}$ and $n_{OFF,A}$ can be exponentially related to ni (the number of inputs to the verified circuits) in the worst case, the overall complexity is *additive* rather than multiplicative. So the problem of multiplicative blow-up is avoided.

In [Wei86], it is indicated that the performance of the LOVER algorithms vary mostly because of the different enumeration algorithms that are used. Various methods can be used in LOVER for enumeration - justification algorithms as seen in most test pattern generation algorithms are also enumeration algorithms after suitable modifications to the termination condition.

In LOVER, the enumeration phase is followed by the cube simulation phase, hence the interaction between these two processors should be considered to achieve better performance of the overall algorithm. Although it is generally desirable that the number of cubes enumerated should be as small as possible, the possibility of cube-splitting deserves attention. An efficient enumeration algorithm may enumerate C^{ON} or C^{OFF} very efficiently with only a few cubes, but if most of these cubes need to be split several times during simulation, the overall verification time suffers.

For ease of reference, in the remainder of the paper, a LOVER algorithm using a specific justification algorithm X is referred to as LOVER-X. Among all the approaches presented in PROTEUS, it was found that the LOVER-SDIJUST approach was the most efficient[Wei86]. Hence, LOVER-SDIJUST has been used as an example for parallelizing the general LOVER-based approach as described in Section 4.

3. EFFICIENT ENUMERATION ALGORITHMS

3.1. LOVER-PODEM

Enumeration in LOVER can be performed based on the decision tree concept in PODEM[Goe81]. By modifying the termination condition of the implicit enumeration algorithm used in PODEM, both the ON-set and OFF-set can be implicitly, but exhaustively, enumerated. A decision tree for LOVER-PODEM for verifying two functionally equivalent circuits is shown in Fig. 1. In general, two decision trees are required: one for the ON-set verification and the other for the OFF-set.

Each node in the decision tree represents a primary input (PI) assignment. Initially all primary inputs are assigned unknown values. Given an initial objective, i.e. to set a primary output line to a 1 or 0, a path is traced from the objective line backwards to a primary input to obtain a PI assignment. A 1 initial objective corresponds to the enumeration of the ON-set and a 0 initial objective corresponds to the OFF-set enumeration. After each new PI assignment, the circuit is simulated using the current set of PI assignments to see if the value at the target line has been set up. If not, the backtrace process continues. If the desired value has been achieved, a cube in the corresponding set is found and simulated on the other circuit. If the opposite value has been set up, the algorithm backtracks to the last PI assignment, tries the alternative value and flags the node to indicate that both assignment choices have been tried. If the alternative has already been tried, the node is removed and the backtrack process continues until an unflagged node with a possible alternative is reached. The backtrack process is also applied when a desired value has been set up at the target line. This is different from PODEM where the enumeration pro-

cess terminates when the desired value is set up at the target line. When the decision tree is found to be empty in the backtrack process, the total input space for the corresponding set has been implicitly, but exhaustively, enumerated.

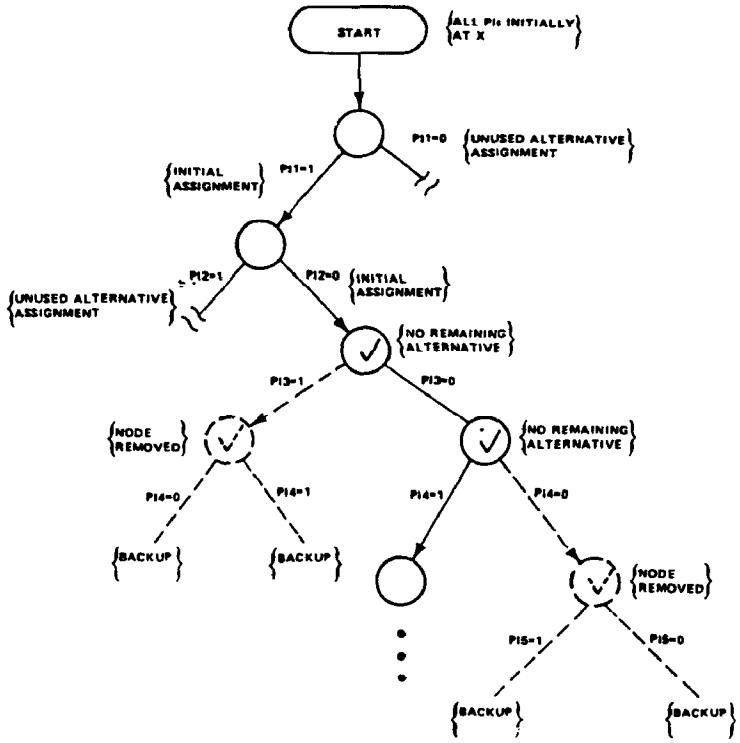

Fig. 1: The Decision Tree of LOVER-PODEM

3.2. PLOVER: A Variation of LOVER-PODEM

In the above enumeration process, the ON-set and OFF-set are enumerated separately with the initial objective being set differently in each case. One can observe, however, that in the enumeration process, backtracking is performed whenever the value of the objective line is set regardless of the value attained. In the enumeration of one of the two sets, cubes in the other set *are actually being generated simultaneously* but discarded. This represents a *wasted effort* in the enumeration process. A variation of the LOVER-PODEM method, called PLOVER, is therefore proposed.

In PLOVER, only one decision tree is used with the initial objective set to either a 1 or a 0 for the primary output. Cubes in both the ON-set and OFF-set are collected whenever the value of the output is set. Simulation of cubes from both sets are performed together rather than separately. The verifications of both sets are interleaved to avoid wasted enumeration effort. The choice of the initial objective, either a 1 or 0, is unimportant in terms of the completeness of the ON-set and OFF-set being generated. It only influences the size of the sets - the set corresponding to the initial objective tends to be more compact. Experience has shown that the enumeration process can be made more efficient if advanced knowledge of the relative sizes of the two sets is available by setting the initial objective corresponding to the bigger set.

4. SCHEMES FOR PARALLEL LOGIC VERIFICATION

The LOVER framework supports various schemes for parallel logic verification. This section describes a static scheduling scheme which can be used regardless of the enumeration algorithms used, and a highly efficient dynamic scheduling scheme using the LOVER-PODEM/PLOVER verification algorithms.

4.1. Static Scheduling

The parallelism inherent in the LOVER framework regardless of what enumeration and simulation algorithms are used can be exploited using a *static scheduling* scheme. In this scheme, each processor works largely independent of the others with very little inter-processor communication. *The enumeration algorithm used has no influence on the speed-up but only on the absolute cpu time expenditure.* The synchronization overhead for this scheme is minimal and the scheme is easily implemented.

For ease of discussion, the following notations will be used. Let A and B be the two circuits that need verification. Let n_i (n_o) denote the number of inputs (outputs) to the circuits. Through segmentation, each single-output circuit, the cone, can be verified individually. The two sets, the ON-set and the OFF-set, of the cone circuit can also be verified independently. Furthermore, for each set verification, the tasks of enumeration and simulation, with proper synchronization, can also be performed in parallel by two distinct processors. This is possible because the two tasks are performed on different data structures: the enumeration is performed on circuit A while simulation is performed on circuit B. Synchronization is necessary because simulation works on a fixed number of cubes that are generated by enumeration. The two tasks are closely interwoven, with simulation following enumeration on every increment of a fixed number of cubes enumerated. The size of an increment is determined by the type of simulation algorithm used. For 32-vector parallel simulation, an increment can contain up to 32 enumerated cubes. For single-vector serial simulation, an increment is an enumerated cube. For 32-vector parallel simulation, an increment of more than 32 enumerated cubes is also possible with the use of buffers which temporarily keep the cubes that are pending for simulation. Buffering is helpful only when the speed of simulation is much faster than enumeration. Even in this case, it should be noted that because synchronization between two tasks must be maintained, the two tasks will terminate at about the same time, the time determined by the slower task. Improving the

speed of the simulation process will not improve the overall throughput. Unless the idle time of the processor dedicated to simulation can be utilized to do other things, there is no reason for using buffering. So immediately we have

$$n_{task} = n_o \times 2 \times 2 = 4n_o \qquad (4.1)$$

tasks that can be performed in parallel. For convenience, each task is denoted as $task(type, i, j)$ where $type$ = "*enu*" or "*sim*" indicating the task is an enumeration or a simulation task; $i = 0$ or 1 indicating it is an OFF-set or an ON-set verification problem; and $j = 1,\ldots,n_o$ indicating which cone circuit is under consideration. As an example, $task(\text{"}enu\text{"}, 0, 4)$ represents the task of enumerating the OFF-set of the 4-th single-output cone circuit. It is assumed that enumeration tasks are always performed on cone circuits of A while simulation tasks are performed on cone circuits of B. The time it takes to complete a specific task $task(type, i, j)$ on a uniprocessor is denoted as $t(type, i, j)$. The time it takes to complete all the tasks on a uniprocessor is denoted as $t_{tot,s}$, and can be calculated by

$$t_{tot,s} = \sum_{j=1}^{n_o} \sum_{i=0}^{1} \left(t(\text{"}enu\text{"}, i, j) + t(\text{"}sim\text{"}, i, j) \right) \qquad (4.2)$$

Let np be the number of processors available for parallel processing in a multi-processor computer. Let $t_{tot,p}$ be the time the multi-processor computer takes to complete all the tasks. Assuming np is large enough so that as many processors as needed are available, then n_{task} processors can be used to carry out all the n_{task} tasks in parallel, each processor working on one task. Then $t_{tot,p}$ is determined by the task which takes the longest time to complete, namely the highest among all the $t(type, i, j)$. Simulation is in general more efficient than enumeration. So $t_{tot,p}$ is in general determined by the most time consuming enumeration task. Let the speedup obtained in this case be denoted as n_{UP}. We have

$$t_{tot,p} = \max_j \left(t(\text{"}enu\text{"}, 0, j), t(\text{"}enu\text{"}, 1, j) \right) \qquad (4.3a)$$

$$n_{UP} = t_{tot,s} / t_{tot,p} \qquad (4.3b)$$

Because n_{task} is directly proportional to n_o, when n_o is big, the number of processors required can be very large. In some cases, the number of available processors may not be enough. When this happens, we can take advantage of the fact that simulation takes much less time than enumeration. For $3n_o \leqslant np < 4n_o$, and for every processor that is needed but is not available, $task(\text{"}sim\text{"}, 0, j)$ and $task(\text{"}sim\text{"}, 1, j)$ for some j are combined and carried out by the same processor. For convenience, the processors which perform simulation (enumeration) tasks are referred to as simulation (enumeration) processors. Thus, for every cone circuit, there are three dedicated processors, one performing the combined simulation task while the other two are performing the enumeration tasks. For the simulation processor, two buffers, one reserved for each enumeration processor, are used to ensure the synchronization between the simulation processor and the two enumeration processors. Since there are two buffers to examine, the simulation processor is kept mostly busy, and thus the utilization of the simulation processor is increased.

Using experimental data over a range of logic circuits, it has been determined that among the three processors dedicated to a cone circuit, typically one of the two enumeration processors represents the bottleneck and determines the time it takes to completely verify that cone circuit. So $t_{tot,p}$ is again determined by the enumeration task which takes the longest time to complete, and n_{UP} remains the same as in the case where $4n_o$ processors were used which implies that overall efficiency has increased.

The above method works until np drops below $3n_o$. When $np < 3n_o$, for every processor that is needed but is not available, instead of combining $task(\text{"}sim\text{"}, 0, j)$ and $task(\text{"}sim\text{"}, 1, j)$ as just described, we can combine $task(\text{"}sim\text{"}, i, j)$ and $task(\text{"}enu\text{"}, i, j)$ for some j and some i and let the combined task be carried out by a processor. Let $TASK(i, j)$ represent the combined task. The time it takes to complete $TASK(i, j)$ on a uniprocessor can be calculated by

$$T(i, j) = t(\text{"}enu\text{"}, i, j) + t(\text{"}sim\text{"}, i, j) \qquad (4.4)$$

In this case, the speedup may be less than n_{UP} if the enumeration task contained in this combined task has a high complexity, making this combined task the bottleneck in the parallel verification process. However, this can be avoided using an intelligent *grouping of enumeration and simulation tasks*, using estimates on the complexity of both phases based on circuit information. If relatively simple enumeration tasks can be combined with their associated simulation tasks and assigned to a processor, the time required for verification will still be determined by the most complex enumeration task and a speed-up of n_{UP} can still be obtained.

When np drops below $2n_o$, it becomes inevitable to allocate more than one cone to a single processor. There are $2n_o$ combined tasks $TASK(i, j)$ and only $np < 2n_o$ processors are available. Let $is\_taken(i, j)$ be a flag for $TASK(i, j)$. $is\_taken(i, j) = 1$ means $TASK(i, j)$ has been assigned to a processor and is either processed or is being processed; $is\_taken(i, j) = 0$ means $TASK(i, j)$ is not yet assigned nor is processed. A processor looks for an $is\_taken(i, j)$ flag that is 0, grasps the task, sets the

is_taken (i, j) flag to 1 and starts processing. When the task is finished, the same process repeats until all *is_taken* (i, j) flags are set to 1.

4.1.1. Grouping of cones in Multi-output circuits

When the number of processors available is small compared to the number of outputs, efficient parallel verification can be achieved by assigning sets of outputs to the different processors, such that the *sum totals* of enumeration and simulation tasks to be performed by each processor are approximately the same. If the sets of tasks are exactly identical in complexity and cpu time requirements, an ideal 100% overall efficiency can be obtained.

To find a good grouping of cones in multi-output circuits, rough estimates of the complexity in enumerating the cones are required. These estimates can be made by examining the structure of the logic network - for example, the number of *levels* in the network, the number of *reconvergent fanouts*, and the relative number of gates at each level.

4.1.2. Static Scheduling Results

A parallel LOVER-SDIJUST algorithm has been developed and implemented on the Sequent Balance 8000 multi-processor [Seq85] using the static scheduling scheme described. The Balance computer available to us is configured with 8 processors.

A pair of cone circuits can be verified using 1-4 processors as summarized by Table 1. Mode 1 is identical to the uni-processor version of the algorithm for each cone. Mode 2 uses two processors one verifying the ON-set and the other the OFF-set. Mode 3 uses three processors, two enumerating the ON and OFF-sets and the third performing simulations. Mode 4 uses four processors, two each for simulation and enumeration. Mode 4 is rarely used since empirical evidence shows that simulation is usually about twice as fast as enumeration making a fourth processor redundant.

Given the number of processors available and the number of cone circuits to be verified against each other the algorithm determines which mode of operation it will adopt. Mode selection takes into account the complexities in verifying each cone circuit in order to minimize idling time of processors (processors which finish early will idle). If the number of outputs is greater than the number of processors, mode 1 is selected, and sets of outputs with approximately equal verification complexities are found. On the other hand, given a two output circuit and 6 processors, mode 3 is automatically adopted.

Table 2 gives the results obtained on two circuits from [Brg85]. Both these circuits are complex and the uni-processor verification time on the Sequent is about 38 and 17 hours respectively using the LOVER-SDIJUST algorithm. Respectable speed-ups have been obtained over 8 processor configurations for

mode no.	no. of processors	comments
1	1	serial algorithm (parallel in cones for entire ckt)
2	2	parallel in sets
3	3	parallel in sets and phases (shared simulation phase)
4	4	parallel in sets and phases

Table 1: Modes in static scheduling

CKT	#inp	#out	speed-up/mode						
			2	3	4	5	6	7	8
C880	60	26	1.90/1	2.60/1	3.30/2	3.82/2	4.70/3	4.70/3	4.70/3
C432	36	7	1.95/1	2.80/1	3.50/2	4.30/2	5.40/2	5.40/2	6.10/3

Table 2: Results using static scheduling

both examples. The modes used for different processor configurations have also been indicated. The first example saturates after 6 processors because verifying one output in the circuit is significantly more time consuming than any of the others, and a maximum of four processors can be used on a single cone given a static scheduling scheme. This output thus becomes the bottleneck in the parallel verification process. Better results are obtained in the second example, although it has fewer outputs, because no single output overwhelms the others in complexity.

In the following section we describe a dynamic scheduling scheme which enables an arbitrary number of processors to be used to verify a single cone circuit and is such that high processor utilization (and overall efficiency) is obtained regardless of the number of processors available and the complexity of individual cone circuits.

4.2. Dynamic Scheduling

In addition to the static scheme, we have also devised a *dynamic scheduling* scheme using the new PODEM decision tree based enumeration method PLOVER, which achieves high processor utilitization on any kind of circuit. In the following section, we will describe how the enumeration method can be efficiently parallelized using dynamic scheduling. Initially, for ease in explanation, a single-output circuit will be assumed while describing the parallel algorithm. Later, we will extend the algorithm to handle multiple-output circuits.

In the LOVER framework as described in Section 2, the two main tasks performed in the verification process are enumeration and simulation. Cubes are continuously enumerated on one circuit and simulated on the other to check any functional discrepancies between the two.

The main goal of dynamic scheduling is to *distribute continually equal amounts of work among processors* to avoid wasteful idling and achieve high processor utilization. Good processor utilization during enumeration can be achieved by repeatedly breaking up the enumeration task(s) into smaller ones and

assigning them to different processors - an enumeration algorithm that is tailored for such a parallel application has been devised.

The parallel enumeration algorithm is based on the PLOVER algorithm described in Section 3. The input space is divided up into disjoint sub-spaces and each processor enumerates all possible input patterns in an assigned sub-space in parallel. Sub-spaces are further broken up, again disjointly, if some processors finish their assigned enumeration in the input sub-space before the others. Thus, even if the initially assigned sub-spaces are very different in enumerative complexity, *processors which complete their tasks early don't remain idle but help other processors* in completing their enumeration task.

Cube simulation on the cone circuit can be performed by any processor whenever the accumulated number of cubes generated by a processor is equal to the number of cubes that can be simulated in parallel by a parallel simulation algorithm. By proceeding in such fashion, an equal amount of verification work is assigned to each available processor and full ultilization of processor time is achieved by continuously keeping all processors at work in parallel.

4.2.1. A Parallel Enumeration Algorithm

The enumeration algorithm used in PLOVER described in Section 3 is well suited to parallel application. Whenever a new PI assignment is made, two disjoint input spaces are implicitly developed by the decision tree. These two input spaces correspond to the 0 and 1 values of the newly assigned input and the old values of all the previously assigned inputs. Some input values may still be unknown. Since these two input spaces are disjoint, they can be enumerated by two different processors in parallel with the guarantee that the resulting two sets of enumerated cubes will also be disjoint. Thus no redundant enumeration work is done using this technique - *each processor enumerates on a different branch of the decision tree.*

Disjoint input spaces are continually generated by all the processors doing the enumeration every time a new PI assignment is made. After a processor performs a PI assignment, it picks one of the disjoint spaces and continues enumeration on that space. As soon as a processor completes enumerating its present input space, it picks up another branch which corresponds to previously generated input spaces by other processors which have not yet been enumerated. This process continues until the entire input space has been enumerated. The selection of a new input space by a processor on the completion of its initially assigned task (this input space would have been generated by some other processor) entails an initialization overhead. It is therefore desirable to select the largest unenumerated input space available which corresponds to the space with the minimum number of assigned primary inputs.

4.2.2. Implementation

The decision tree is an ordered list of nodes and is implemented as a *stack*. Each processor *owns* a separate stack which corresponds to the input space currently being enumerated by it. Whenever a new PI assignment is made, a new unflagged node is pushed onto the top of the stack. And whenever a backtrack step is made, the node on the top of the stack is examined. If the node is unflagged, the alternative value is assigned to the corresponding input and the node is flagged to indicate both choices have been tried. If the node is found to be flagged, it is popped from the stack. Enumeration of a particular input space is completed when the stack becomes empty. The stack is therefore treated as a FILO queue by the owning processor.

The selection of a new input space by a processor is done by popping nodes from the *bottom of the stack* of another processor and pushing them onto the processor's own stack. This popping and pushing process continues until the first unflagged node is reached. This unflagged node is flagged and the corresponding input is assigned the alternative value creating a new disjoint input space on which the processor enumerates. The popping of nodes begins from the bottom of the stack rather than from the top so as to obtain the *largest* unenumerated space to minimize initialization overhead. The implementation of the parallel enumeration algorithm is illustrated in the pseudo-code below.

4.2.3. Global Verification Scheme

Circuits with an arbitrary number of outputs can be efficiently verified using the dynamic scheduling scheme described above by using all the processors to verify each output, and verifying the outputs sequentially. However, greater efficiency is gained by incorporating the dynamic scheduling strategy into a global verification scheme. Initially, each processor tries to pick and verify an output, enumerating and simulating over the entire input space. If a processor runs out of unverified outputs it then helps the processors which have not completed their outputs, via dynamic scheduling. Thus the overhead of selecting new input spaces to enumerate on is minimized. To further minimize the initialization overhead incurred in the selection of a new input space by a processor on the completion of its initial assigned task during dynamic scheduling, a feature called a *preferred stack mechanism* is implemented. This feature restricts a processor to enumerate on unfinished input space of one cone circuit before switching to another one by assigning different priorities to input spaces of different cone circuits during input space selection. This prevents a processor from unnecessary switching between unfinished input spaces of different cone circuits and increases overall efficiency.

```
parallel_enumerate() {
  while (enumeration_not_finished) {
    if (output_is_not_set) {
      find_new_pi_assignment();
      push an unflagged node on top of stack S1;
      simulate the current set of pi assignments;
    }
    else {
      if (output is a 1)
        a cube from ON-set is generated;
      else
        a cube from OFF-set is generated;
      while (S1 is not empty
              AND
        node on top of S1 is flagged) {
        pop a node from the top of S1;
      }
      if (an unflagged node is found) {
        flag node;
        assign alternative value to the primary input;
        simulate the current set of pi assignments;
      }
      else { select:
        select a non empty stack S2 of
          another processor;
        while (node at bottom of S2 is flagged
                    AND
                S2 is not empty ) {
          pop the node and push on top of S1;
          assign the pi value corresponding to that node;
        }
        if (an unflagged node is found) {
          pop the node and push on top of S1;
          flag node;
          assign alternative value to the primary input;
          simulate the current set of pi assignments;
        }
        else
          goto select;
      }
    }
  }
}
```

4.2.4. Dynamic Scheduling Results

Results for five examples using dynamic scheduling are given in Table 3. In the table h, m and s stand for hours, minutes and seconds respectively. The first two examples are benchmark circuits from [Brg85]. The number of outputs for the five examples are 3, 26, 2, 1 and 8 respectively. Regardless of the number of outputs, the number of processors used and the great variations in logic complexities among different cones circuits in the benchmarks, in every case except example 5, the speed-ups are very close to the ideal values. The reason that the speed-up deviates from the ideal value when the number of processors is large in example 5 is because the verification time is so small, circuit read-in time becomes significant; it is about 11s and represents over 40% of the total run-time in the 8-processor case. If we deduct the circuit reading time from the total run-time, the speed-up in the verification phase is again close to the ideal value.

CKT	Number of processors							
	1		2		4		8	
	ABS. time	speed up	ABS. time	speed up	ABS. time	speed up	ABS. time	speed up
C432*	10.9h	1	5.49h	1.99	2.78h	3.92	1.38h	7.92
C880	33.9h	1	17.0h	1.99	8.54h	3.98	4.28h	7.92
ex1	69.7m	1	35.0m	1.99	17.7m	3.94	9.10m	7.68
ex2	95.4m	1	48.4m	1.99	24.2m	3.96	12.2m	7.84
alu4	104s	1	57.5s	1.81	35.3s	2.95	23.2s	4.49

C432* : only the first three outputs

Table 3: Results using dynamic scheduling

A profile of the time each processor spent enumerating and simulating on the various outputs in example C880 is shown in Figure 2 for an 8 processor configuration. In the figure, the time profiles for each output have been normalized and the absolute verification time for each output is indicated to the right of the plot. As can be seen, the outputs which take a long time to verify have been shared out among many processors illustrating the excellent load balancing characteristics of the dynamic scheduling scheme, which is key to obtaining high overall efficiences.

5. CONCLUSION

In this paper, we have presented new algorithms based on the LOVER approach for combinational logic verification. We have developed, for the first time, parallel logic verification schemes and achieved high overall efficiencies over a large number of processors.

Parallelism inherent in the LOVER approach can be exploited using a static scheduling scheme. The advantage with this approach is that it is independent of the enumeration and simulation algorithms used. High speed-ups have been obtained on benchmark circuits.

A dynamic scheduling scheme using a PODEM-based enumeration algorithm has been developed. This scheme produces excellent results on all kinds of circuits, with arbitrary numbers of available processors. Very high processor utilization (more than 95%) which translates to speed-ups greater than 7.8 over 8 processor configurations have been obtained using this scheme.

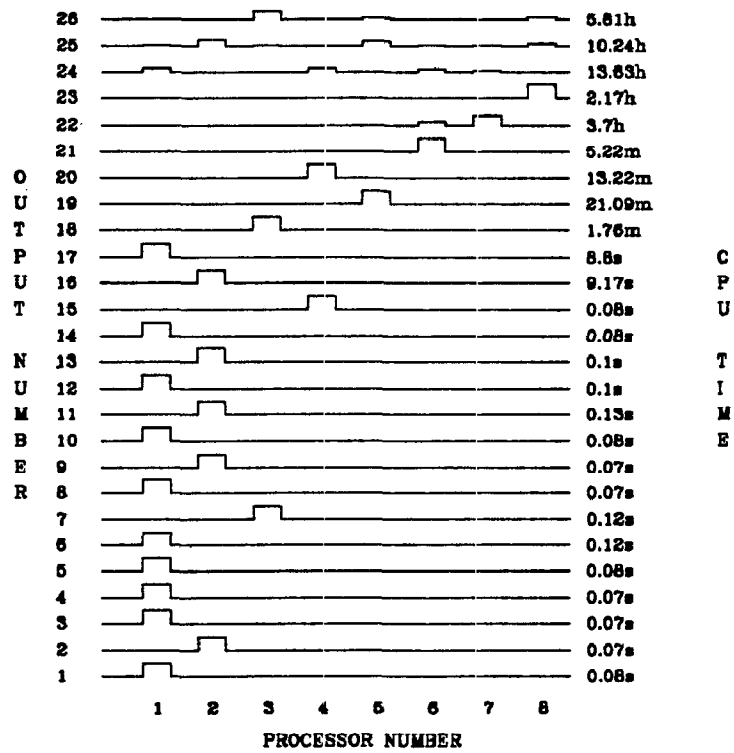

Fig. 2 Processor-Output Time Profiles for C880

6. ACKNOWLEDGEMENTS

This research is supported by the Semiconductor Research Corporation under grant 442427-52055 and by a grant from AT&T Bell Laboratories.

7. REFERENCES

[Bra84]
R. K. Brayton, G. D. Hachtel, C. T. McMullen and A. L. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis," Kluwer Academic Publishers, 1984.

[Brg85]
F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN," Special session on ATPG and fault simulation, Proc. 1985 IEEE Int. Symp. Circuits and Systems, Kyoto, Japan, June 5-7, 1985.

[Bry85]
R. E. Bryant, "Symbolic Manipulation of Boolean Functions", Chapel Hill Conference on VLSI, May 1985.

[Don76]
W. E. Donath and H. Ofek, "Automatic Identification of Equivalence Points for Boolean Logic Verification", IBM Technical Disclosure Bulletin, vol. 18, No 8, Jan. 1976.

[Goe81]
P. Goel, "An Implicit Enumeration Algorithm To Generate Tests for Combinational Logic Circuits" , IEEE Transactions on Computers, Vol C-30, Mar. 1981.

[Oda86]
G. Odawara et. al. "A Logic Verifier based on Boolean Comparison", Proc. 23rd Design Automation Conf. June 1986.

[Rot73]
P. Roth, "VERIFY: An algorithm to verify a computer design," IBM Tech. Disclosure Bull. 15, 2646-2648(1973).

[Rot77]
P. Roth, "Hardware Verification", IEEE Transactions on Computers, Vol C-26, 1977.

[Rot80]
P. Roth, "Computer Hardware Testing and Verification", Computer Science Press, Potomac, Maryland, 1980.

[Seq85]
Sequent Computer Systems, Inc., "Balance 8000 guide to parallel programming", Sequent Computer Systems, Inc., July 31 1985.

[Wei86]
R.S. Wei and A. Sangiovanni-Vincentelli, "PROTEUS: A Logic Verification System for Combinational Logic Circuits", Proc. of International Testing Conference, Sept. 1986.