A Shared Memory Parallel Algorithm for Logic Synthesis *

Chieng-Fai Lim† **Prithviraj Banerjee**† **Kaushik De**† **Saburo Muroga**‡

† Coordinated Science Lab
1308 W. Main Street
Urbana, IL 61801, USA

‡ Digital Computer Lab.
1304 W. Springfield Av.
Urbana, IL 61801, USA

Abstract

The Transduction Method [9] has been shown to be a powerful tool in the optimization of multi-level networks. However, this algorithm takes large execution times, making these tools difficult to use for large networks. In this paper, we present a parallel algorithm for logic synthesis based on the Transduction Method that is suitable for execution on a shared-memory multiprocessor. This implementation, called Parallel TRANSduction (PTRANS), also uses partitioning to break up large networks and performs inter- and intra-partition dynamic load balancing. We report experimental results on an 8-processor Encore multiprocessor for various benchmark networks.

1 Introduction

Combinational logic synthesis deals with the optimization of combinational logic to realize a specific function in either two level or multilevel form, and typically optimizes the area or delay of the resultant circuit. Combinational logic can be synthesized using two-level logic such as PLAs. Many efficient algorithms for two-level logic minimization such as ESPRESSO [1] have been developed. However, many combinational logic functions can be more efficiently realized with multi-level networks. Several tools such as SOCRATES [2], MIS [3], SYLON-XTRANS [4], and BOLD [5] have also been developed for the synthesis of such multi-level networks. Logic synthesis for medium sized circuits have large computing times and memory requirements. For larger circuits of the future, sequential algorithms will not be able to address their complexity. In this paper we will present a parallel algorithm for combinational logic synthesis.

Recently, researchers have started to investigate parallel algorithms for problems in logic synthesis and verification. Galivanche and Reddy [6] have proposed a parallel implementation of two-level network synthesis based on ESPRESSO. The authors presented parallel versions of the procedures in two-level synthesis, such as reduction, expansion and complementation of cubes, and reported speedups of about 7 on an 8 processor shared memory multiprocessor. Hachtel et al. [7] have proposed parallel tautology checking with application to logic verification. Ma et al. [8] have proposed parallel logic verification algorithms based on implicit enumeration. Both the above work on parallel logic verification reported good speedups. Until now, therefore there has been no successful results published on parallel logic synthesis for multilevel circuits.

The successful implementations of SYLON-XTRANS [4] have proven that the Transduction Method [9] is a powerful tool in the synthesis of multi-level networks. However, the Transduction Method has the disadvantage of large execution times for large networks. In this paper, we present a parallel algorithm for logic synthesis, called Parallel TRANSduction (PTRANS), that employs both the power of the Transduction Method, as well as the advantage offered by parallel processing, to synthesize multi-level logic networks of high quality quickly.

2 Review of the Transduction Method

2.1 Terminology

In this paper, we will consider loop-free multi-level networks consisting of only simple gates such as AND, OR, NAND, NOR and NOT gates. However, it is conceptually trivial to extend the methods discussed in our paper to handle more complex gates like XOR and XNOR.

A multilevel network consists of gates and connections. A network can be viewed as a directed acyclic graph, consisting of the gates as the nodes in the graph and the connections as the edges in graph. The Transduction method is based on the concept of *sets of permissible functions* [9]. The output function of a gate can be any member of its set of permissible functions without changing the functionalities of the primary outputs of the network. There are two types of permissible functions, the *maximum set of permissible functions (MSPF)* and the *compatible set of permis-*

*This research was supported in part by National Aeronautics and Space Administration under Grant NAG 1-613, and in part by the Semiconductor Research Corporation under Contract SRC 91-DP-109.

sible functions (CSPF). The MSPF is more powerful, but it requires very large computation time. On the other hand, the CSPF of a gate or a connection is a subset of its MSPF, calculated based on some ordering of the connections in the network. However, the computation time needed for CSPF is much shorter than that required for MSPF, though it is less effective.

2.2 Types of Transformation

A suite of transformations and optimizations can be applied to the network with the help of the CSPFs of the gates and the connections. The Transduction method consists of four main transformations, namely, *gate substitution*, *gate merging*, *generalized gate substitution* and *gate input reduction*. It also *prunes* redundant connections in the network.

In *gate substitution*, a gate v_i can be substituted by another gate v_j if the output function of v_j is a member of the CSPF of v_i. Then v_i can be deleted from the network. In *gate merging*, two gates v_i and v_j whose CSPFs intersect are tried to be merged by synthesizing a new gate whose output function is a member of the intersection of CSPFs of two gates. The connection to the new gate is from a minimal set of existing gates in the network. Then both v_i and v_j can be removed from the network. A more general version of gate substitution is *generalized gate substitution*, where each output connection of a gate v_i is substituted by an existing gate in the network. Then v_i can be removed from the network. Finally, in *gate input reduction*, one tries to reduce the number of input connections to a gate v_i by trying to synthesize a new gate whose output function is a member of the CSPF of v_i such that the number of inputs to the new gate is less than that of v_i. Then the new gate can replace v_i, and v_i can be removed.

3 Parallel Algorithm for Transduction Method

3.1 Partitioning Networks for Parallel Execution

An obvious way of extracting significant speedup out of the logic synthesis application is to generate a large number of partitions and to synthesize each partition independently. The results of the individual partitions can then be merged back. Unfortunately, such an approach has the problem that with increasing number of partitions, the quality of the overall network degrades in terms of area and testability. Hence, one is forced to resort to intra-partition parallelism which is much harder to exploit.

In PTRANS, the following procedure was used. Networks that are too large to fit into the memory as a whole during minimization are partitioned into as many partitions as required using the partitioning algorithm [10]. However, the size of each partition is the maximum size that satisfies the memory constraints. Subsequently, a combination of inter- and intra-partition parallelism is applied to the partitions during minimization.

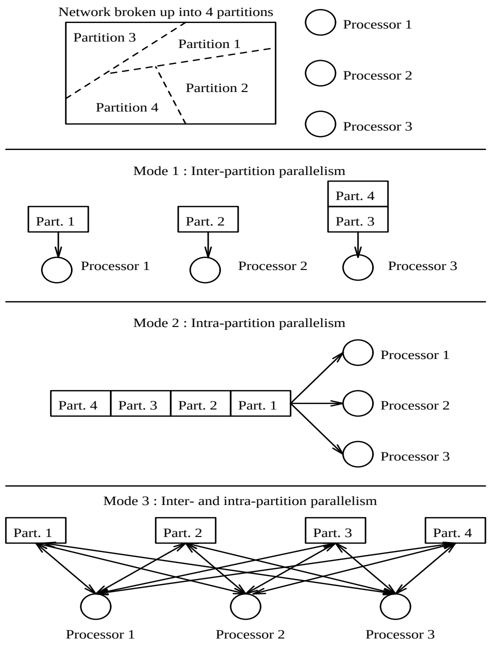

Figure 1: The three modes of execution of PTRANS

3.2 Parallel Execution Model

We use a master-slave model for PTRANS. In such a model, every processor can be either a master or a slave at any instance of time. A master is responsible for distributing jobs to the slaves and synchronizing them. It also retains some jobs for itself. When all of the slaves have completed their jobs, they pass their results back to the master. On finishing its jobs, the master will then combine all of the results together. In PTRANS, we allow multiple masters to be present. As for slaves, they reside in a **slave pool** whenever they are idle. A master can then look into this pool for idle slaves when it has jobs that can be performed in parallel.

Let M be the number of partitions to be minimized. M can vary from 1 to the maximum number of partitions that can be minimized simultaneously in the computer's memory. Let P be the number of processors which PTRANS is executing on, and N be the minimum of M and P. Initially, there are N masters and P-N slaves. Every master gets M/P partitions and maintains them in a *partition-list*.

For every partition in its partition-list, a master will remove it from the list and apply the Transduction procedures to minimize it. In each procedure, there are entry points where the master will enter the slave pool to look for idle slaves and exist where slaves can

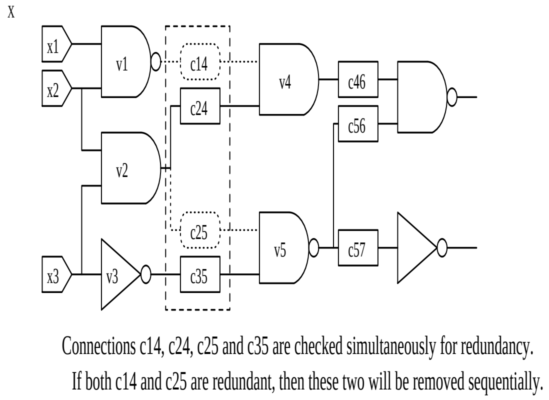

Figure 2: An example of pruning with CSPF

pass their results back to their masters and return to the pool. The use of these entry points and exits forms the basis of dynamic load balancing between processors in PTRANS.

When a master has minimized all of the partitions in its partition-list, it looks for the longest partition-list among the other masters, transfers half of the list's contents to its own list, and continues processing the transferred contents. On the other hand, if this master cannot find any non-empty partition-list, all of the M partitions have been (or are being) minimized. In this case, it changes its status to a slave and resides in the slave pool as an idle processor. A special situation arises when this master is last among the N original masters to change into a slave. When this happens, all of the M partitions have already been minimized. Hence, PTRANS will terminate. With this model, PTRANS is able to perform all of the three modes of execution as shown in Figure 1.

3.3 Parallel Computation of Output Functions and Sets of Permissible Functions

In this section, we will describe the approach to compute the output functions and CSPFs of gates and connections in parallel. The gates and connections in the same level with respect to the primary inputs can have their output functions computed in parallel. Similarly, the CSPFs of gates and connections in the same level with respect to the primary outputs can be computed in parallel too.

Suppose we want to compute the output functions of the gates and connections of a network in parallel. We start with the primary inputs and distribute them evenly among the processors for parallel computation. After this, we distribute the connections which are connected to these inputs, and process them in parallel. Next, we pick those gates in the network whose input connections have all been processed, distribute them, and compute their output functions in parallel too. This is repeated until every gate and connection has been processed. An exit is located after this so that the slave processors involved in the computation can return to the slave pool.

Obviously, the processors need to be synchronized every time before the gates and connections are distributed to them. Between two consecutive synchronizations, the processors are actually processing a level of gates or connections with respect to the primary inputs. During the parallel processing of such a level, a processor may sometimes complete its job before others. When this happens, the processor will identify which of the other processors has the most unprocessed gates and connections left, remove half of that processor's unprocessed items, and process them. This load balancing strategy minimizes the amount of idleness at each processor.

Using a similar approach, we can also compute the CSPFs of the gates and connections by starting with the primary outputs.

3.4 Parallel Pruning

Pruning can be performed with either CSPF or MSPF. We will first describe how pruning with CSPF is performed in parallel.

The procedure for parallel pruning with CSPF is a modification of the procedure for parallel computation of CSPFs, which was described in the previous section. In this procedure, we check for the redundancy of a connection immediately after its CSPF has been computed. Hence, after the CSPFs of every gate and connection in a partition have been computed in parallel, we know which of the connections are redundant. The master in charge of minimizing this partition will then remove these connections. We have decided to use a single processor to remove the redundant connections because we need to adjust only a few pointers in the program data structures during each removal, which could be performed very quickly. Hence, it is not worth parallelizing. After removing the connections, the master initiates the recomputation of the CSPFs for the next iteration of the procedure. This is repeated until no further redundant connections can be found. Figure 2 shows an example of an iteration of this pruning procedure.

On the other hand, in the procedure for parallel pruning with MSPF, we do not wait until the MSPFs of all the gates and connections have been computed before performing any removal. This is because after a connection has been removed, we need to recompute the MSPFs. Hence, a lot of work will be wasted if we were to compute all of the MSPFs first. In this procedure, we let the processors compute the MSPFs of gates and connections in parallel, similar to the procedure used for the computation of CSPFs. However, after a processor has computed the MSPF of a connection and

Table 1: Comparison of MIS 2.1, XTRANS 1.1, and PTRANS.

CKT	Init. (g/c)	Non-part. MIS 2.1 (g/c)	Partitioned # part.	Partitioned XTRANS (g/c)	Partitioned PTRANS (g/c)
f51m	131/270	110/225	1	70/128	81/157
5xp1	129/279	92/185	1	62/112	79/153
9sym	205/470	175/408	1	162/346	173/360
bw	205/481	123/250	1	144/262	145/289
sao2	129/310	100/211	1	99/195	99/213
vg2	158/391	66/141	1	82/158	73/163
rd73	135/325	67/134	1	79/156	115/243
duke2	485/1224	282/627	1	352/736	348/730
misex1	69/154	47/88	1	46/84	51/102
misex2	87/233	78/159	1	94/181	82/178
misex3c	493/1231	311/730	1	326/680	363/776
C432	198/411	-	2	-	166/345
C499	526/942	521/925	2	-	493/892
C880	342/688	-	4	-	313/632
C1355	492/1018	519/523	4	-	507/1022
C1908	599/1220	-	4	-	448/914

has determined that the connection is redundant, it informs the other processors working on the same partition to halt. The master in charge of minimizing this partition will then remove the redundant connection, while the slaves return to the slave pool. After this, the master initiates the recomputation of the MSPFs for the next iteration of the procedure.

As with CSPF, there may be insufficient memory to store the MSPFs of every gate and connection. However, MSPFs are not stored in the disk since they are not used in any of the other transformations in PTRANS. The MSPF of a gate or connection is deleted once it has been used for the computation of the MSPFs of all of the relevant immediate predecessor gates or connections.

3.5 Parallel Gate Substitution with CSPF

The main idea of gate substitution is to search a given network for pairs of gates such that in each pair, one of the gates can substitute for the other. However, as the network needs to be loop-free, the substitute gate must not be a successor of the gate to be substituted. Hence, to minimize the chance of this from happening, the substitute gate and the substituted one should be as near as possible to the primary inputs and primary outputs, respectively.

To explain how parallel gate substitution works, we will describe an iteration of the procedure. In each iteration, we try to generate, in parallel, a set of pairs of gates, $\{(v_i, v_j)\}$, such that in each pair, the gate v_i can substitute for the corresponding gate, v_j. Initially, the master, which is in charge of minimizing a partition, will enter the slave pool and look for idle slaves. After this, every processor (master or slave) will each take note of a *unique* gate, v_{j_x}, in the partition. (We use 'unique' in the sense that each processor chooses a different gate.) This v_{j_x} is obtained by scanning the partition, starting with the primary outputs, for the first gate which has not yet been taken note of by any processor. After this, the processor will then search the partition for another gate, v_{i_x}, which can substitute for v_{j_x}. This search begins with the primary inputs. If the search is successful, the processor will halt the other processors working on the same partition. The master then performs the substitution while the slaves return to the slave pool. After this, the master gathers idle slaves from the slave pool and together, they examine the remaining gates in the partition for substitution.

In order to obtain better processor utilization, the master may also enter the slave pool to look for more slaves after it has taken note of its unique v_{j_x}. This will happen when the master has not yet owned P-1 slaves. The new slaves (if any) will then join the other existing slaves in the search for substitutable pairs of gates in the network by first obtaining their unique gates.

After each iteration of the procedure, the master recomputes of the output functions and CSPFs of the gates and connections in the network before proceeding to the next iteration.

Similar algorithms are developed for the other operations like gate merging, generalized gate substitution, gate input reduction etc. The details of those algorithms are reported in [11].

Table 2: Variation of network quality versus number of processors.

CKT	# part	Initial (g/c)	1 proc (g/c)	2 proc (g/c)	4 proc (g/c)	8 proc (g/c)
f51m	1	131/270	81/157	81/157	81/157	81/157
5xp1	1	129/279	79/153	79/153	78/151	75/147
9sym	1	205/470	173/360	173/362	180/379	187/394
bw	1	205/481	145/289	146/291	150/298	149/300
sao2	1	129/310	99/213	99/213	109/231	111/235
vg2	1	158/391	73/163	73/162	73/162	73/162
rd73	1	135/325	115/243	115/243	115/241	110/235
duke2	1	485/1224	352/736	344/715	343/718	352/737
alupla	1	114/223	103/205	103/205	103/205	103/205
misex1	1	69/154	51/102	51/102	50/100	51/101
misex2	1	87/233	82/178	82/178	87/178	82/178
misex3c	1	493/1231	363/776	360/770	355/755	356/766
C432	2	198/411	166/345	166/345	166/345	166/345
C499	2	526/942	493/892	493/892	493/892	493/892
C880	4	342/688	313/632	313/632	313/632	313/632
C1355	4	492/1018	507/1022	507/1022	507/1022	507/1022
C1908	4	599/1220	448/914	449/914	448/914	453/940

4 Experimental Results

The PTRANS algorithm has been implemented on an 8-processor Encore 510 Multimax shared-memory (64 MBytes) multiprocessor where each processor is an NS32532 processor rated at 5 MIPS.

4.1 Comparison of Quality

We will first compare the sequential implementation of our transduction method of logic synthesis to other existing logic synthesis tools. In Table 1, we compare the qualities of networks synthesized by MIS 2.1 [3], SYLON-XTRANS 1.1 [4], and PTRANS on a single processor of the Encore Multimax computer. The qualities of the final networks were measured in terms of the number of simple gates and connections (g/c). MIS 2.1 was executed on the whole network using the Boolean script, using don't-cares and disjoint support filtering. XTRANS 1.1 was run without the use of XTRANS-INI (which generates its own initial networks which has been reported elsewere to help produce high quality networks), but from the same initial network as described above. In the following comparison table, a '-' sign means either the corresponding program ran out of memory, could not finish within 30 hours, or was unable to handle the large number of inputs in the network.

The qualities of the circuits synthesized by the transduction method (XTRANS and PTRANS) were comparable to MIS 2.1 for the smaller circuits, and better than MIS 2.1 for the larger circuits. Between XTRANS and PTRANS in Table 1, PTRANS generally produced networks larger than those produced by XTRANS. This is because XTRANS 1.1 recognizes EXCLUSIVE-OR and EXCLUSIVE-NOR gates,

Table 3: Speedups obtained by using intra-partition parallelism only.

CKT	1 proc Time(s)	2 proc	4 proc	8 proc
f51m	217	1.8	3.2	3.4
5xp1	182	1.9	2.9	4.4
9sym	5408	2.8	4.1	8.3
bw	793	2.2	2.7	3.7
sao2	1556	1.9	3.6	3.5
vg2	1603	1.6	2.8	4.6
rd73	967	2.1	3.0	5.5
duke2	19650	2.5	4.2	7.3
alupla	6560	1.9	3.6	6.5
misex1	115	1.8	3.2	4.8
misex2	264	1.7	3.1	3.7
misex3c	72834	2.5	3.7	4.9

which are presently not accepted by PTRANS.

Table 2 shows the variation in quality of the circuit produced with increasing number of processors. For networks consisting of multiple partitions, the partitions were minimized using intra-partition parallelism only. As shown in Table 2, the network qualities did not vary appreciably with increasing number of processors.

4.2 Runtime and Speedup Results

The results obtained by using intra-partition parallelism only on various *MCNC* benchmark examples are presented in Table 3.

Table 4 shows the results obtained with inter-partition parallelism. It has been observed that with increase

Table 4: Speedups using Inter-Partition Parallelism.

CKT	part 1 (sec)	Part 2 (sec)	Part 3 (sec)	Part 4 (sec)	Spdup (4 proc)
C432	1975	3024	-	-	1.6
C499	1660	3597	-	-	1.5
C880	52	415	385	185	2.5
C1355	4876	3727	138	24721	1.3
C1908	3651	2168	191	221	1.7

Table 5: Speedups obtained for using both intra- and inter-partition parallelism.

CKT	# of part	1 Proc Time(s)	2 P	4 P	8 P
C432	2	4911	2.0	3.4	5.9
C499	2	5231	1.9	3.7	7.5
C880	4	1535	2.0	3.8	7.0
C1355	4	33517	2.0	3.9	7.4
C1908	4	6219	2.0	3.8	5.3

in the number of partitions, the runtimes were seen to go down dramatically, but so is the quality of the resultant circuits. Hence one is forced to limit the number of partitions of each circuit. The number of partitions is governed by the memory limitations of the computer. Even though there is total parallelism across the partitions, the speedups obtained are not that high, because partitions take a wide variety in execution times, and there is a tremendous load imbalance across the processors. The final execution time of all partitions is governed by the slowest partition's execution time.

For networks consisting of multiple partitions, the partitions were minimized simultaneously using both inter- and intra-partition parallelism and the speedups are shown in Table 5.

We note that the speedups obtained for such networks were generally higher than those shown in Table 3. This was because when multiple partitions were simultaneously minimized, the average number of processors associated with each partition was smaller. Thus, the processors were more efficiently utilized and this explains the higher speedups obtained.

5 Conclusions

In this paper, a parallel algorithm (PTRANS) implementing the Transduction Method has been proposed and implemented using BDDs. The algorithm is targeted to synthesize very large networks. The large networks are partitioned into several smaller partitions using a novel partitioning algorithm [10] that tries to ensure that not too many redundancies will be created after the synthesized partitions are reconnected together. PTRANS uses a combination of inter-partition and intra-partition parallelism on these partitions.

To support intra-partition parallelism, we have developed parallel algorithms for computing output functions and sets of permissible functions, as well as algorithms for parallel network transformation and reduction. PTRANS achieves good speedups and high processor utilization with simply intra-partition parallelism. When combined with inter-partition parallelism, the speedups achieved are even higher.

References

[1] R. K. Brayton and et al., "ESPRESSO-II: A New Logic Minimizer for Programmable Logic Arrays," *CICC*, pp. 370–376, June 1984.

[2] A. J. de Geus and W. Cohen, "A Rule-based System for Optimizing Combinational Logic," *IEEE Design and Test*, pp. 22–32, August 1985.

[3] R. Brayton, R. Ruddel, A. Sangiovanni-Vincentelli, and A. Wang, "MIS: A Multiple-level Logic Optimization System," *IEEE Transactions on Computer-Aided Design*, pp. 1062–1081, November 1987.

[4] X. Q. Xiang and S. Muroga, "SYLON-XTRANS: A Multilevel Logic Network Synthesizer," *International Workshop on Logicx Synthesis*, 1989.

[5] K. A. Barlett, D. Bostick, G. Hachtel, R. Jacoby, and M. Ligh-ner, "BOLD: A Muliple-level Logic Optimization System," *International Conference on Computer Aided Design*, 1987.

[6] R. Galivanche and S. M. Reddy, "A Parallel PLA Minimization Program," *Design Automation Conference*, pp. 600–607, 1987.

[7] G. D. Hachtel and P. H. Moceyunas, "Parallel Algorithms for Boolean Tautology Checking," *ICCAD*, pp. 422–425, 1987.

[8] H. T. Ma, S. Devadas, and A. S. Vincentelli, "Logic Verification Algorithms and their Parallel Implementations," *24th DAC*, 1987.

[9] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney, "The Transduction Method - Design of Logic Networks Based on Permissible Functions," *IEEE Transactions on Computers*, pp. 1404–1424, Oct. 1989.

[10] K. De and P. Banerjee, "Logic Partitioning and Resynthesis for Testability," *International Test Conference*, pp. 906–915, 1991.

[11] C.-F. Lim, "Parallel Algorithm for Multi-Level Logic Synthesis Using the Transduction Method," Master's thesis, Univ. of Illinois, 1991.