A Parallel Algorithm for State Assignment of Finite State Machines

Gagan Hasteer **Prithviraj Banerjee**

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois
1308 West Main St.
Urbana, IL 61801

Abstract

Optimization of huge sequential circuits has become unmanageable in CAD of VLSI due to enormous time and memory requirements. In this paper, we report a parallel algorithm for the state assignment problem for finite state machines. Our algorithm has three significant contributions: It is an asynchronous parallel algorithm portable across different MIMD machines. Time and memory requirements reduce by a factor of P (the number of processors), enabling it to handle large problem sizes which the sequential algorithm fails to handle. The quality of the results for multiprocessor runs remains comparable to the sequential algorithm on which it is based.

Index Terms : Encoding Hypercube, State Assignment, Memory Scalabilty, Conflict Resolution.

1 Introduction

With the rapid improvement in VLSI technology, circuit design is becoming extremely complex and is placing increasing demands on CAD tools. Parallel processing is becoming an attractive solution to reduce the inordinate amount of time spent in VLSI circuit design [1]. This has been recognized by several researchers in VLSI CAD with application in cell placement [2], circuit extraction [3], and logic synthesis [4].

In this paper, we present a parallel algorithm, named ProperSTATE, for the state assignment problem of finite state machines (FSM). The parallel algorithm is based on one of the recent sequential algorithms, MUSTANG. FSMs form an important class of VLSI circuits and are especially useful for designing the control unit of various microprocessors. In general, a huge sequential circuit can be considered to be a big FSM. In such a scenario, even a circuit with modest number of latches (say 20) has more than a million states(2^{20}). Researchers have tried to use techniques like FSM decomposition and isolation of smaller interacting FSMs to make the time and memory requirements manageable [5]. These smaller sub-circuits are locally optimized by conventional techniques like state minimization and optimal state assignment.

The penalty for lack of global optimization techniques severely affects the quality of the results. In other words, the area requirement is significantly more for a piecewise locally optimized sequential machine than for a globally optimized one. To this effect our attempt to parallelize MUSTANG assumes significance. Besides being compute intensive in nature, MUSTANG uses four different heuristics and is run four times to generate state assignments, the best of which is chosen. Hence the time requirements increase by a factor of four. Table 2 shows the large time requirements for MUSTANG.

Another important motivation factor for our parallel implementation is the enormous memory requirement of huge FSMs. In the past, a popular approach has been to partition the problem across processors, invariably suffering a loss of quality. Our algorithm has three significant contributions: (1) It is an asynchronous parallel algorithm portable across different MIMD machines. (2) The uniprocessor version of ProperSTATE is much more efficient than MUSTANG and results in an order of magnitude reduction in time requirements. (3) Time and memory requirements reduce by a factor of P (the number of processors),

Acknowledgement: This research was supported in part by the Advanced Research Project Agency under contract DAA-H04-94-G-0273 and in part by Semiconductor Research Corporation under Contract SRC 94-DP-109. We are also grateful to the National Center for Supercomputing Applications (NCSA) for providing us access to the Thinking Machines CM-5 multicomputer.

maintaining the quality comparable to MUSTANG.

The rest of this paper is organized as follows: In §2, we define the state assignment problem and review previous work done in state assignment for FSMs. §3 describes the ProperSTATE algorithm in detail. Quality improvement is discussed in §4. §5 discusses the performance of the ProperSTATE algorithm on several parallel machines. We conclude the paper with a discussion on future work in §6.

2 Preliminaries

In this section we give the motivation for the state assignment problem. This is followed by a brief description of the work done on state assignment.

2.1 Definitions

A *variable* is a symbol representing a single coordinate in the Boolean space (e.g. a). A *literal* is a variable or its negation (e.g. a or $\bar{a}$). A *cube* is a set of literals and represents the Boolean function corresponding to their product (e.g. $\{a, \bar{b}\}$ is a cube representing the product $a\bar{b}$). An expression is a set of cubes and represents the Boolean function corresponding to their disjunction (e.g. $\{\{a, \bar{b}\}, \{b, \bar{a}\}\}$ is an expression representing the function $a\bar{b} + b\bar{a}$).

The most commonly used representation of an FSM is a *State Transition Graph* (STG). An STG G(V,E,I(E),O(E)) is a set of vertices V connected by directed edges belonging to E. Each edge is labelled by two fields : an input field I(E) and an output field O(E). The set of vertices are in 1-1 correspondence with the states of the FSM. A directed edge (v_i, v_j) joins v_i to v_j if there is a transition from the state corresponding to v_i to the state corresponding to v_j in the FSM. I(v_i, v_j) is the set of primary inputs that cause the transition from v_i to v_j and O(v_i, v_j) is the set of primary outputs produced on that transition.

In order to make a fair comparison between MUSTANG and different processor runs of ProperSTATE we give equal weight to all the benchmarks. We assess the quality in terms of scaled count rather than actual literal count. *Scaled literal count* for an alogrithm with respect to MUSTANG is defined as

$$m_{xy} = \frac{100}{n} * \sum_{i=1}^{n} \frac{LC\ of\ BM_i\ for\ the\ algorithm}{LC\ of\ BM_i\ for\ MUSTANG}$$

where LC refers to the literal count and BM stands for benchmark.

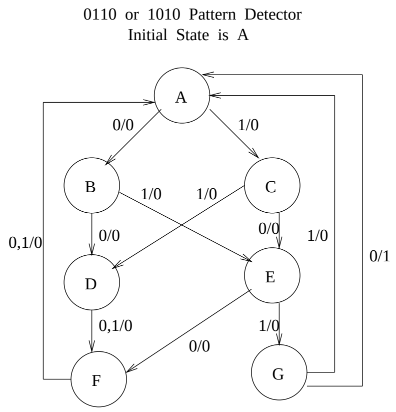

Figure 1: STG for an FSM (pattern detector)

2.2 The State Assignment Problem

Figure 1 shows the state transition graph (STG) for a Finite State Machine which outputs a 1 on detecting either of the two patterns 0110 or 1010. 3 bits are needed to encode the states of this FSM. There is a 1 bit output. This translates to 4 Boolean functions. Let x represent the input, y_1, y_2 and y_3 the 3 bits which encode the states of the FSM, and z the output. Consider the following state assignment: A(000), B(001), C(010), D(011), E(100), F(101), G(111). Boolean equations for the next state functions are

$$z = \bar{x}y_1y_2y_3 \tag{1}$$

$$y_1 = x\bar{y_1}y_3 + \bar{x}\bar{y_1}y_2 + y_1\bar{y_2}\bar{y_3} \tag{2}$$

$$y_2 = \bar{x}\bar{y_1}\bar{y_2}y_3 + x\bar{y_2}\bar{y_3} + x\bar{y_1}\bar{y_3} \tag{3}$$

$$y_3 = \bar{x}\bar{y_2}\bar{y_3} + x\bar{y_1}\bar{y_2} + \bar{x}\bar{y_1}y_3 \tag{4}$$

The combinational circuit required to realize these functions requires 13 gates assuming complemented inputs are available.

On the other hand the assignment A(000), B(010), C(011), D(110), E(111), F(100), G(101) yields the following next state functions

$$z = \bar{x}y_1\bar{y_2}y_3 \tag{5}$$

$$y_1 = \bar{x}y_1\bar{y_2} + x\bar{y_1}\bar{y_2} + xy_1y_2y_3 \tag{6}$$

$$y_2 = \bar{y_3} \tag{7}$$

$$y_3 = y_2 \tag{8}$$

The combinational circuit required to realize these functions requires only 5 gates.

Thus the state assignment problem involves assigning binary coded values to the states of an FSM in such a way that the area of the combinational circuit, required to realize the FSM, is minimized.

2.3 Related Work

A lot of work has been done for the state assignment problem. MUSTANG [6], NOVA [7], and JEDI [8] are some of the recent algorithms. The algorithms targeting two-level PLA based implementations such as NOVA optimize for the number of product terms called cubes. However, the *number of literals in a factored form* for the logic is the standard estimate used by algorithms designed for multi-level implementations such as MUSTANG and JEDI.

Multilevel logic optimization systems such as SIS [9] primarily use algebraic techniques to factor and decompose the Boolean equations by identifying common sub-expressions. Hence the heuristics in both MUSTANG and JEDI try to maximize the number and size of common sub-expressions. MUSTANG finds pairs or clusters of states which, if kept minimally distant in the Boolean space representing the encoding, result in a large number of common sub-expressions. This minimizes the number of literals in the resulting combinational logic network after multi-level logic optimization.

In a related work [10] we have reported a parallel implementation of the simulated annealing based JEDI algorithm. While the quality of parallel JEDI is 10% superior than that of ProperSTATE the run times are about 200 times more. Hence parallelizing both the algorithms provides the designer a tradeoff between time and quality.

3 ProperSTATE

We have built ProperSTATE around an existing uniprocessor implementation of a state assignment algorithm, MUSTANG. The input to MUSTANG is an FSM in the form of a state transition graph. MUSTANG solves the state assignment problem in two phases. In Phase 1 it evaluates a gain estimate of encoding two states at a minimal distance for all pairs of states. In Phase 2 it tries to maximize the total gain while mapping the states on the encoding hypercube. In ProperSTATE, we exploit parallelism by evaluating the gains as well as encoding the state in parallel. In addition, ProperSTATE has a third phase in which the processors exchange and enhance their encodings. In the following subsections we discuss the parallel implementation of each of the phases in detail.

3.1 Gain Function Computation

For computing an estimate of the gain obtained by encoding two states closely, MUSTANG provides an option between four heuristics- a fanout oriented algorithm and a fanin oriented algorithm, both targetted for two-level and multi-level implementations. All these algorithms generate a complete undirected weighted graph $G_M(V, E_M, W(E_M))$ where V is the set of states of the FSM, and $W(e_{ij})$ is the estimate of the gain that can be obtained by encoding states $v_i, v_j \in V$ close to each other.

The fanout oriented algorithm works on the output and the fanout of each state. Pairs of present states which assert similar outputs and produce similar sets of next states are given high edge weights. This has an effect of maximizing the size of common cubes in the Boolean functions corresponding to the output and the next state lines. In contrast, the fanin oriented algorithm, works on the input and the fanin of each state. Pairs of next states which are produced by similar inputs and similar sets of present states are given high edge weights. This has an effect of maximizing the number of common cubes in the boolean functions for the next state lines. Without loss of generality we consider the fanin oriented algorithm.

3.1.1 Sequential algorithm

Let the number of inputs be N_i, the number of outputs N_o and the number of states N_s. The fanin algorithm proceeds as follows:

1. For each node (state) in the state transition graph, look at all the edges fanning in. For every node, construct $2N_i$ input factors, two for each input (corresponding to the input being a 1 or a 0). Each input factor is a value which indicates the number of times that input asserts the state being considered. If two states are asserted by similar inputs then their input factors for those inputs will be high.

2. For each node (state) in the state transition graph, look at all the edges fanning in. For every node, construct N_s present state factors, one for each state. Each present state factor is a value which indicates the number of times that present state leads to the state being considered.

3. For each pair of states in the FSM, generate a gain estimate of encoding them closely. This is done by multiplying their corresponding input and present state factors and summing them up. If two next states are asserted by similar inputs and produced by similar present states then they could potentially share a large number of common cubes. Their factors when multiplied give a large gain value.

3.1.2 Parallel Implementation

The state transition graph is replicated on all processors. ProperSTATE divides the set of states of the FSM among its processors for computation of the gain function. The work done in Steps 1 and 2 is proportional to the number of edges, whereas that in Step 3 is proportional to the number of nodes. Since Step 3 is computationally more intensive, a load balance is achieved by cyclically partitioning the nodes. All edges fanning out of a node are given to the owner processor of that node. This enables each processor to compute the $2N_i$ output factors and N_s next state factors for all its owned nodes independent of other processors (Steps 1 and 2). In practice, the cyclic partition of nodes leads to a more even distribution of edges than a blocked distribution. For computing Step 3, each processor needs the output and next state factors of all the nodes. This requires a broadcast of the computed factors to every processor. Thus each processor computes N_s edge weights (gain estimates) for each of its local nodes. Each weight in G_M is computed by the two processors owning the corresponding states of the STG. G_M which requires $O(N_s^2)$ space for its edge weights is thus divided among the processors and requires N_s^2/P space per processor.

The asynchronous nature of the algorithm enables overlap of computation and communication. Also to avoid storage of all output and next state factors received from all the processors, which require $O(N_s^2)$ storage, they are discarded immediately after use. At no time is the information from two different processors stored simultaneously. This is essential for achieving memory scalability.[1]

3.2 Encoding Phase

In Phase 1 each processor generates N_s edge weights for every node it owns. These weights are then sorted together in decreasing order to generate the local weight list. At this stage the processor is ready to go ahead with the encoding of the states. This is essentially equivalent to mapping the set of states on a Boolean hypercube. The dimension of the hypercube is equal to N_b, the number of bits for encoding the states of the FSM. First we will discuss briefly the encoding algorithm used by MUSTANG and then describe its parallel implementation in detail.

[1] All $O(N_s^2)$ data structures required by the intermediate computations in the sequential algorithm have been handled using a partitioned approach. Entire data structures are not required simultaneously for any computation. Hence, each processor works on the part of the data structure it requires and then discards it.

3.2.1 Sequential algorithm

MUSTANG tries to heuristically minimize the function

$$\sum_{i=1}^{N_s} \sum_{j=i+1}^{N_s} we(s_i, s_j) * dist(enc(s_i), enc(s_j))$$

The function $we()$ returns the edge weight between vertices s_i and s_j in G_M. $dist()$ returns the Hamming distance between the two binary codes assigned to s_i and s_j. If a pair of states has a high edge weight then the algorithm attempts to encode them as close as possible to minimize this sum. It looks for clusters or pairs of states to encode together. Clusters of nodes with cardinality no greater than $N_b + 1$ and consisting of edges of maximum total weight are identified. One of them is picked as a *seed* and embedded in the encoding hypercube. Rest of the states in the cluster are embedded at a Hamming distance of 1 from the seed.

Figure 2 shows the encoding phase of MUSTANG on a 9 state FSM. (s1,s2) has the highest weight in the sorted weight list. s1 is picked as the seed and the states with maximum weights with s1 are chosen to form a cluster of size $5 (N_b = 4)$. The seed is embedded first and remaining nodes are embedded on distance-1 vertices. After embedding this cluster the weight list is scanned to find the next potential seed. (s3,s4) has the next highest edge weight and both are still unencoded. Hence, s3 is picked as the seed and a cluster is formed in a similar manner. This cluster is of size 4 since all other states have been encoded. The encoding space is searched for the first vertex which has 3 distance-1 neighbors unassigned. 0111 satisfies the constraint and is chosen to embed the seed s3. Again the remaining vertices are embedded on the distance-1 neighbors. This is the final encoding as no other state remains unencoded.

3.2.2 Parallel Implementation

The encoding hypercube is partitioned among the different processors. Each processor depending on its identity, works on a different partition. The sorted list of gain values is scanned to identify clusters. The seed of these clusters is embedded in the processor's own partition. The remaining states in the cluster are embedded as explained. If there are 8 processors and a 7-dimension hypercube (7 bit encoding for states) then each processor works on a 4-dimension hypercube $(2^7/8 = 2^4)$. But if each processor is restricted to a smaller encoding space then it cannot encode large

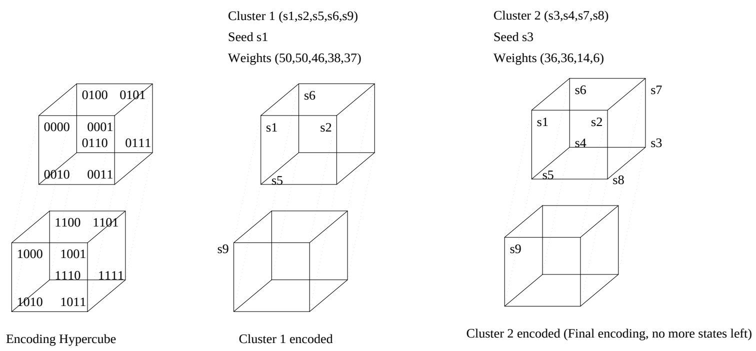

Figure 2: An example showing the Encoding Phase of Mustang

clusters. In this case, each processor would be encoding clusters of maximum size 5 whereas the sequential algorithm encodes clusters of size 8. This would lead to a significant degradation in quality. Hence, each processor is allowed to claim vertices from another processor's partition. These claims are resolved in the merge phase along with other contentions as explained in §3.3. Due to partitioning of the encoding space, each node of the hypercube loses $log_2 P$ distance 1 neighbors. After embedding states at a distance of 1 from the seed on the local partition if some states of the cluster still remain unencoded then the processor claims non-local distance 1 neighbors.

A state may be encoded by two or more processors. This happens because all processors work on their local gain lists which contain the gain estimates of its local nodes with all the nodes of the state transition graph. A node may end up being included in different clusters on different processors. This is another correction to be performed in the merge phase.

3.3 Merge Phase

After each processor embeds its clusters in its local partition of the hypercube they merge their information pairwise using a tree based merge approach. The following tasks are done at each merge step:

- Exchange the best encoding information
- Resolve claims on the same encoding space
- Correct codes for state with multiple assignments

The two processors send their encoding information to each other. In general there would be clusters of states, which a processor could not code in the encoding phase due to lack of information about their gain estimates. So each processor selectively enhances its encoding using the incoming information. The resolution of claims is complicated by the fact that one of the contenders could be a seed of a cluster. The gain estimate realized in an encoding is preserved and communicated along with the coding information. In case of a conflict the choice with better gain is picked. If a seed loses a claim then its cluster is also removed from the encoding hypercube. A similar technique is applied for picking the better choice for states with multiple assignments. Due to the dynamically changing encodings and states being removed, the associated gain values realized have to be consistently updated. A detailed description of the utility and update of the gain values is give in §4.

The heuristic used for correction of codes actually results in the two processors resolving the conflicts differently. As an example if two nodes encoded with the same gain claim the same encoding, then the processor resolves the conflict in favor of the one which is local to the processor at the time of the merge. Hence, all the processors are allowed to generate complete encodings for the FSM through the $log_2 P$ merge phases. Some states may remain unencoded after the merge phases. They are coded by picking up the maximum possible gain in the local gain list of that processor for that state and trying to realize that gain. The final encodings obtained by each processor are in general, different from each other.

Figure 3 shows the encoding and merge phases on the FSM of Figure 2 for a 4 processor run. For simplicity the first cluster to be encoded by each processor is made to include all local states. Since each cluster requires a local unencoded state as a seed, no more clusters are generated. The weights refer to the gain estimates of the corresponding nodes with the seed. Each processor embeds the seed first (e.g. P0 encodes s1 as 0000, P1 encodes s2 as 0111). The remaining nodes are embedded on local distance-1 vertices. When these vertices are exhausted, non-local distance-1 neighbors are claimed (e.g. P0 encodes s2 as 0001, s5 as 0010 on local vertices but s6 as 0100, s9

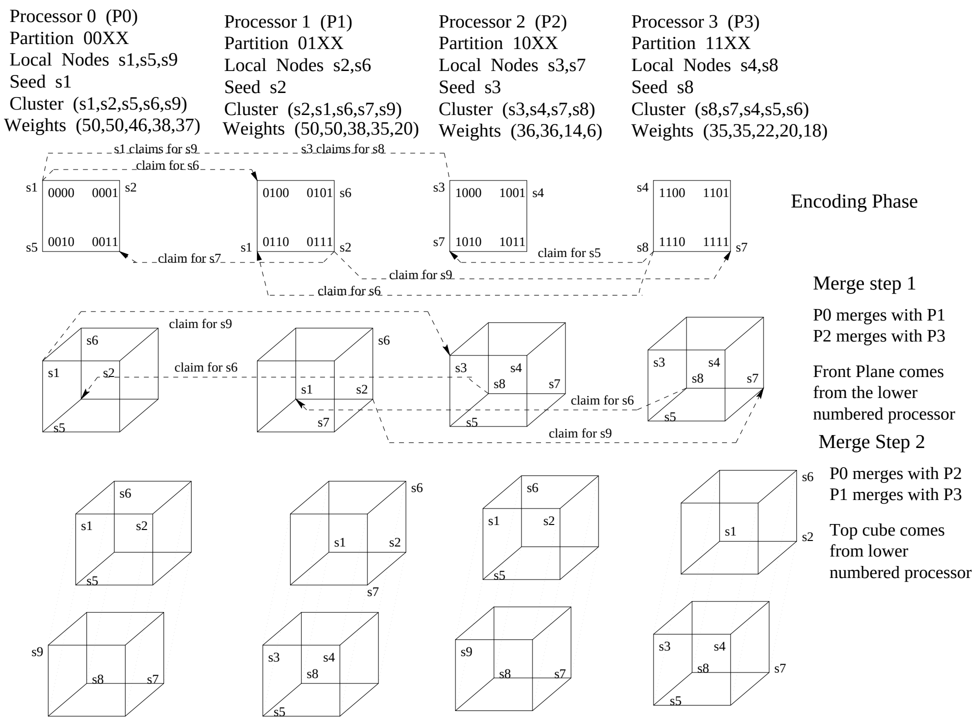

Figure 3: An example of Encoding and Merge Phases of ProperSTATE on 4 processors

as 1000 on non-local vertices).

There are $log_2 4 = 2$ merge phases. In the first merge P0 exchanges information with P1 (and P2 with P3). On P0 the encoding of P1's seed s2 is rejected since it is already encoded with the same gain estimate. Hence, none of the encodings proposed by P1 is accepted by P0. P0's claim for s6 is settled without any conflict. The merge step on P1 proceeds similarly. The local cluster of P2 is weak as seen from the gain estimates. P3 proposes an alternate encoding for s7 with a higher gain. This is accepted by P2. Hence, vertex 1010 is released and later on taken by s5. P3, however, loses its claim for s4. Same conflicts are resolved on P3 similarly; however, there is a difference. The proposal for encoding s7 by P2 faces two conflicts on P3, one with s5 which claims the same vertex and the other with already encoded local s7. The unsettled claim for s6 made by P3 is recorded by P2 while accepting s8. P2 also drops its claim for s8 on P0 after encoding s8.

In the second merge step, P0 exchanges information with P2 and wins its claim for s9. Since the losing node is the seed, its cluster (which has been reduced to s3,s4) is also undone. Both processors end up in an identical encoding after this merge. An initial suboptimal decision for encoding a weak cluster by P2 has been rectified in this process. The merge of P1 and P3 has two conflicts. P1 claims vertex 1111 for s9 and loses the claim because of lower gain on both the processors. s7 is encoded by both the processors with the same gain estimate. Thus both processors stick to their own encoding for s7. After the merge each processor encodes any unencoded state and outputs a complete solution with the estimated total gain.

4 Quality Control

The solution to the state assignment problem is basically an exploration of a search space (Boolean hypercube) to allocate encodings to the states of an FSM. Effectively we have utilized the parallelism to search more during the merge phase. Instead of keeping other processors idle after they communicate their information, they are programmed to explore other possible encodings. The sequential algorithm greedily picks up clusters. After encoding them it does not have a backtrack mechanism to undo any encoding. Hence, a suboptimal decision made during this phase cannot be changed. On the other hand, the $log_2 P$ merge phases of the parallel algorithm involve a correction step. This step implicitly provides a backtrack in favor of a better solution by undoing an encoding with lesser gain for a more prolific encoding. As a side effect of the correction step the clusters tend to

Table 1: Quality Comparison between various runs of ProperSTATE and MUSTANG(Dashed entries correspond to large FSMs which were not optimized by SIS

Quality in Literal counts

FSM BENCHMARK	Number of States	MUSTANG	ProperSTATE					
			1 PE	2 PE	4 PE	8 PE	16 PE	32 PE
bbara_bbtas	128	1110	1064	1039	1049	1110	1164	1304
scf	121	1046	918	1002	1016	1144	1221	1180
s298	218	8219	8031	8016	8739	7642	8324	8576
fetchXlog	442	9240	9139	9043	9265	9450	9960	9514
dvramXmarkl	423	10248	10147	9980	9973	9814	9877	10721
fetchXkeyb	494	-	-	-	-	-	-	-
planetXfetch	1248	-	-	-	-	-	-	-
Scaled count		100	95.9	96.4	99.1	100.1	100.3	100.3

reduce in size with each merge step. This has an effect of degrading the quality.

In order to pick up the best output of the P outputs produced, a gain estimate is kept with each state encoding. This estimate is consistently updated during the merge phase. Each processor calculates the total gain associated with its output by summing up the gains for each individual encoding. The output with the maximum gain is picked as the final solution. This solution may not be the best because:

1. A fanin oriented algorithm maximizes the number of common cubes in the Boolean functions representing the combinational logic. The gain values estimate this number. However, implicitly the size of these common cubes could be maximized in a solution. This solution may turn out to be better than the one with highest gain. The same logic applies to the fanout oriented algorithm.

2. MUSTANG optimizes for common cubes but ignores other optimizations such as common kernel extraction and Boolean factoring. Some solutions may yield better results after these optimizations, available in the optimization tool.

3. It is very complicated and computationally intensive to calculate the exact gains realized as the encoding builds up. The interaction between two or more clusters and states which are encoded independent of each other are not modeled in our gain estimates.

With increasing number of processors we get a wider range of solutions to choose from, but the probability of picking a sub-optimal solution also increases. With an increasing number of processors, the total number of encoded clusters increases. In the extreme case, when $P > log_2|V| + 1$ the number of encoded clusters becomes $|V|$ in the worst case ($\frac{|V|}{P}$ clusters per processor). Since each cluster is of size $O(log_2|V| + 1)$, the total number of encodings to be undone is $|V|log_2|V| - |V| = (|V| - 1)log_2|V|$. This breaks up lots of clusters and reduces their size a lot, offsetting the advantage of the correction. Therefore quality degrades for 16 and 32 processor runs.

5 Experimental Results

ProperSTATE has been implemented as part of the ProperCAD project [11]. In [12] we have shown that the maximum number of encoding conflicts that can occur in ProperSTATE is bounded by $\frac{2|V|log_2 P}{log_2|V|}$. In the worst case these conflicts reduce the size of each cluster by $log_2 P$. Thus, the quality degradation is proportional to $log_2 P$. We have also proven that the time overhead for correction of conflicts is bounded by $\frac{4|V|}{log_2|V|}$. This is a strong result as it shows independence of the overhead from the number of processors in the system. We have derived the time and space bounds for ProperSTATE and shown that both time and memory requirements reduce by a factor of P for a P-processor run asymptotically as the number of states $|V| \to \infty$. The analysis is not presented here due to lack of space.

In this section, we will describe the results obtained by applying ProperSTATE on various circuits in the MCNC benchmark suite on a SparcServer1000 shared memory multiprocessor, and a CM-5 distributed memory multicomputer. The output ob-

[2]These tables show the sum of the individual times for four different heuristics of ProperSTATE and MUSTANG.

Table 2: ProperSTATE on a CM-5 (distributed memory multicomputer)[2]
Time in seconds(Speedup)

FSM BNCHMARK	MUS-TANG	ProperSTATE 1 PE	4 PE	8 PE	16 PE	32 PE
bbara_bbtas	324.0	12.48(1.0)	3.76(3.32)	2.16(5.8)	1.52(8.21)	1.24(10.06)
scf	149.6	28.4(1.0)	7.76(3.66)	3.76(7.55)	2.2(12.9)	1.68(16.9)
s298	3416.0	79.28(1.0)	20.2(3.92)	11.24(7.05)	6.72(11.8)	4.56(17.38)
fetchXlog	54032.8	588.6(1.0)	140.64(4.18)	71.56(8.22)	38.76(15.18)	23.7(24.86)
dvramXmark1	51886.0	1983.7(1.0)	481.28(4.12)	169.4(11.7)	64.64(30.68)	32.48(61.1)
fetchXkeyb	timeout	5747.8(1.0)	1381.8(4.16)	588.8(9.76)	291.84(19.7)	163.4(35.2)
planetXfetch	memout	memout	8148.15	2398.16	1162.32	554.48

tained from ProperSTATE is passed through multi-level logic optimization scripts of SIS, a logic synthesis tool from Berkeley. The quality of the solution is measured as the number of literals in a factored form of the logic after optimization. Less literals imply less area for implementing the combinational logic and hence better quality.

Literal count after optimization has been reported for 7 benchmarks. For each benchmark, all the four weight generation heuristics were run and the best quality was picked. ProperSTATE does about 4% better in quality than MUSTANG on a single processor. This quality improvement is attributed to the fact that ProperSTATE attempts to encode clusters of size $N_b + 1$, whereas MUSTANG tries to encode clusters of size N_b. The key observation is that the seed of a cluster can be connected to N_b neighbors, making the maximum possible cluster size $N_b + 1$. Another heuristic used by ProperSTATE is not to attempt a cluster encoding if the size of the cluster is below a certain threshold. This heuristic not only improves quality but also reduces the run time. As seen in Table 1, the quality gradually degrades with an increase in number of processors, but still remains comparable to MUSTANG even for a 32 processor run. The reason for this degradation has been explained in §4.

The memory requirement of ProperSTATE data structures reduces with an increase in the number of processors. Thus *planetXfetch* which was too large to fit on 2 processors, was successfully run on 4 processors on the CM-5. Due to a lack of tools for parsing sequential circuits and generating STGs out of them, we tried to construct representative FSMs of very large sizes. Results are reported for FSMs containing up to 1250 states which would correspond to a moderate size sequential circuit with 11 latches. These were synthesized through the cross products of the FSMs in the MCNC benchmark. A cross product generator [3] was written which takes as input two FSMs M_1 and M_2 and outputs another FSM $M_3 = M_1 X M_2$. *fetchXlog, dvramXmark1, fetchXkeyb, planetXfetch* were generated using this tool. SIS was unable to optimize very large FSM benchmarks like *planetXfetch* which were encoded by ProperSTATE. A partitioned parallel implementation of SIS is in progress currently in the ProperCAD group. Using this memory scalable implementation we hope to optimize these benchmarks.

As seen from Tables 2 and 3 the uniprocessor implementation of ProperSTATE is much more efficient than MUSTANG. After selecting a potential vertex for encoding the seed, MUSTANG searches the entire encoding space of size 2^{N_b} to find a vertex set which is suitable for encoding the corresponding cluster. In contrast, ProperSTATE looks at only N_b distance-1 neighbors for each vertex to check whether a vertex set can accomodate the cluster. Further, ProperSTATE does not attempt a cluster encoding if the size of the cluster is below a certain threshold, thus saving on run time. We attribute the superlinear speedups as seen in Table 2 to the cache effects. Since the memory requirements decrease with increasing number of processors, the cache hit ratio increases. The sorting time also speeds up by a factor greater than 1 (reduces from $|V|^2 log_2 |V|^2$ to $\frac{|V|^2}{P} log_2 \frac{|V|^2}{P}$). Further the searching of the boolean hypercube for cluster formation may result in superlinear effects.

[3]Given two FSMs M_l and M_2 their cross product $M_3(M_1 X M_2)$ is defined as a machine which has the ability to execute M_l and M_2 in parallel. If $M_1 = (S_1, X_1, Z_1)$ and $M_2 = (S_2, X_2, Z_2)$ then $M_3 = (S_3, X_3, Z_3)$ where
$S_3 = ((s_1, s_2) | s_1 \in S_1$ and $s_2 \in S_2)$,
$X_3 = ((x_1, x_2) | x_1 \in X_1$ and $s_2 \in X_2)$,
$Z_3 = ((Z_1, Z_2) | Z_1 \in Z_1$ and $Z_2 \in Z_2)$.

Table 3: ProperSTATE on a SparcServer1000 (8 processor shared memory multiprocessor)
Time in seconds(Speedups)

FSM BENCHMARK	MUSTANG	ProperSTATE			
		1 PE	2 PE	4 PE	8 PE
bbara_bbtas	83.6	2.0(1.0)	1.6(1.25)	0.8(2.5)	0.4(5.0)
scf	67.6	8.8(1.0)	6.0(1.5)	2.8(3.14)	1.2(7.33)
s298	842.4	14.0(1.0)	11.6(1.2)	5.6(2.5)	3.6(3.9)
fetchXlog	11861.6	99.2(1.0)	70.4(1.4)	39.2(2.53)	21.6(4.6)
dvramXmark1	11434.4	574.0(1.0)	335.6(1.71)	155.6(3.69)	55.6(10.32)
fetchXkeyb	20911.2	340.8(1.0)	202.8(1.68)	108.4(3.14)	44.8(7.6)
planetXfetch	Timeout	8639.2(1.0)	5954.4(1.74)	2596.0(3.33)	1010.4(8.55)

6 Conclusions and Future Work

We have demonstrated that it is possible to design and implement a portable parallel state assignment tool that is cleanly and efficiently interfaced with MUSTANG using the ProperCAD II environment. We have shown that ProperSTATE gives very good speedups on shared memory as well as message passing machines. Uniprocessor implementation of ProperSTATE does much better than MUSTANG both in time and quality. Further, the merge phase, partially compensates the quality deterioration due to increasing number of processors by implicitly correcting some sub-optimal decisions made earlier. The state assignment of large benchmarks which do not fit on single processors can be done using ProperSTATE because of its memory scalability.

References

[1] P. Banerjee, *Parallel Algorithms for VLSI Computer-aided Design Applications.* Englewoods-Cliffs, NJ: Prentice Hall, 1994.

[2] J. S. Rose, W. M. Snelgrove, and Z. G. Vranesic, "Parallel cell placement algorithms with quality equivalent to simulated annealing," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 387–396, Mar. 1988.

[3] K. P. Belkhale and P. Banerjee, "Parallel algorithms for VLSI circuit extraction," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 604–618, May 1991.

[4] K. De, J. A. Chandy, S. Roy, S. Parkes, and P. Banerjee, "Portable parallel algorithms for logic synthesis using the mis approach," in *Proceedings of the International Parallel Processing Symposium*, (Santa Barbara, CA), pp. 579–585, Apr. 1995.

[5] P. Ashar, A. Ghosh, S. Devadas, and A. Newton, "Implicit state transition graphs: Applications to sequential logic synthesis and test," *International Conference on Computer Aided Design*, pp. 84–87, Nov. 1990.

[6] S. Devdas, H. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli, "MUSTANG: state assignment of finite state machines targetting multilevel logic implementations," *IEEE Transactions on Computer Aided Design*, pp. 1290–1300, Dec. 1988.

[7] T. Villa and A. Sangiovanni-Vincentelli, "NOVA: state assignment of finite state machines for optimal two-level logic implementation," *IEEE Transactions on Computer Aided Design*, pp. 905–924, Sept. 1990.

[8] B. Lin and A. Newton, "Synthesis of multiple level logic from symbolic high-level description languages," *Proceedings of the International Conference on VLSI*, pp. 187–196, Aug. 1989.

[9] E. M. Sentovich *et al.*, "SIS: a system for sequential circuit synthesis," tech. rep., Electronics Research Laboratory, Department of Electrical Engineering and Computer Science, University of California, Berkeley,, 1992.

[10] G. Hasteer and P. Banerjee, "Simulated annealing based parallel algorithm for state assignment of finite state machines." submitted to EuroPar, Aug. 1996.

[11] S. Parkes, J. A. Chandy, and P. Banerjee, "A library-based approach to portable, parallel, object-oriented programming: Interface, implementation, and application," in *Supercomputing '94*, (Washington, DC), pp. 69–78, Nov. 1994.

[12] G. Hasteer and P. Banerjee, "A parallel algorithm for state assignment of finite state machines." submitted to the IEEE Transactions on Computers, 1996.