Parallel Placement On Reduced Array Architecture

C. P. Ravi Kumar and Sarma Sastry*

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-0781

1 Abstract

This paper presents a hardware accelerator for a *module placement algorithm* based on the Divide and Conquer paradigm. We consider a partitioning algorithm for the approximate solution of a large placement problem. This algorithm divides the set of logic modules into small clusters and generates an optimal placement for each cluster. Finally, in a pasting step, the algorithm combines the optimal solutions for the smaller problems into a near-optimal solution for the original placement problem. Our algorithm lends itself very naturally to a parallel realization, and maps nicely onto an SIMD organization. Considerations such as cost effectiveness and suitability to VLSI implementation led us to the selection of the *Reduced Array Architecture* as the target architecture for the placement accelerator.

2 Introduction

In this paper, we are concerned with the module placement problem which arises in the automatic layout of VLSI circuits[5,7]. Placement involves the positioning of circuit elements on the layout surface, such that some objective function is optimized. For example, it is usual to choose the *total wire length* as a cost function. Module placement has been identified as an NP-complete problem [5]. Heuristic algorithms cut down the large search space of the placement problem and obtain near-optimal placements in polynomial time. However, when VLSI circuits are involved, the problem size N is large and polynomial time algorithms are also highly compute-intensive. There is thus a need to accelerate placement algorithms through techniques such as parallel processing [2,12]. In this paper, we discuss the hardware acceleration of a placement algorithm based on the *Divide and Conquer* paradigm.

For an introduction to VLSI Layout, we refer the reader to [5,7,12]. In Section 3, we provide a brief survey of related work on hardware accelerators for VLSI Layout. Section 4 presents a partitioning approach to placement and motivates the reader towards parallelization of the placement algorithm. The Reduced Array architecture (RAA) [1,14] has recently acquired considerable importance in several applications such as signal processing and image processing. Section 5 begins with a brief overview of RAA and throws light on the use of the architecture in Design Automation applications. We present the hardware organization of the PPS in section 5, and describe the parallel implementation of our placement algorithm on RAA . In Section 6, we present the results of a simulation study of the proposed parallel placement system.

*This work was supported in full by a grant from the Powell Foundation.

3 Previous Work

Recognizing that CAD tasks are highly compute bound, parallel schemes have been proposed for the execution of various steps in VLSI Design Automation [2,12]. Referring to VLSI layout, general purpose parallel computers have been used to efficiently execute placement algorithms [11,3]. Several special purpose placement engines have been presented in [9,4,15,13]. It is interesting to observe that in all the above placement systems, the basic approach is to obtain an approximate solution to the combinatorial optimization problem using the *local search* paradigm. These systems exploit two sources of concurrency inherent in local search algorithms

- Application of local transformations to a large placement
- Evaluation of the resulting change in cost function

For example, the Module Interchange Placement Machine of [9] uses a special purpose microprogrammed hardware optimized for computing wire lengths. This speeds up the most compute bound step of the iterative improvement algorithm, namely, the cost evaluation step. A 4-stage arithmetic pipeline is used to calculate the incremental change in cost function after every local transformation.

The authors of [15] use an SIMD architecture with a two-dimensional processor array structure. Perturbing a placement using adjacent pairwise interchange, and deciding whether or not to accept the new placement are both done in parallel. A similar strategy is employed in [4], where a *force directed pairwise interchange* algorithm is implemented on a two- dimensional array of processors.

More recently, attempts have been made to apply the *Simulated Annealing* [10] technique to the placement problem. An annealing algorithm also involves as its core step, the perturbation of a system configuration and evaluation of the resulting change in cost function. Multiprocessor implementations of placement based on Simulated Annealing are reported in [11] and [3]. A special purpose SIMD architecture for the above task is proposed in [13]. Once again, the essential idea used in the design of these architectures is to speed up the computation of the core step of the annealing algorithm using parallel processing.

The parallel algorithm presented in this paper differs signif-

icantly from the approaches described above. We use a partitioning algorithm for logic module placement which has a very natural parallelization. The placement scheme is described in the following section.

4 Partitioning and Parallel Placement : An Overview

In this Section, we describe two methods to break up a large placement problem so as to achieve the following important goals :

- Implement the placement algorithm in a parallel fashion
- Obtain solutions close to the optimal solution.

Noting that the principal input to a placement algorithm is the *connectivity matrix* C, we require a method to break up the matrix C into smaller pieces. There are two ways to proceed to decompose the C-matrix. The first method is to perform an intelligent partitioning of the circuit into smaller clusters, so that strongly connected modules fall into a given cluster. The C-matrix is used to arrive at this *constructive partition*. The clusters represent placement problems of smaller size. Each cluster can therefore be placed optimally using purely enumerative techniques. Semi-enumerative techniques such as *Branch and Bound*, which are known to produce near-optimal solutions, may also be used. We refer to the placement of a cluster as *Local Placement* hereafter. The reader's attention is drawn to the fact that this is a Divide and Conquer strategy with much time expended on the *Divide* step.

The second alternative is to arrive at an arbitrary decomposition of the C-matrix. This corresponds to an arbitrary partition of the circuit into smaller pieces, or *clusters*. Note that such a scheme spends little time in the *divide* phase. Each cluster is subjected to a local placement as in the first method. Finally, we *shrink* each cluster into a module, hence obtaining a placement problem whose size is the same as the number of clusters. Once again, the size of the resulting placement problem is small, and, as a result, it can be solved optimally in a reasonable amount of time. This approach differs from the first one in that the connectivity information is not used at all in the *divide* step. We began with a random partition of the circuit, and, clearly, there are many such random partitions to investigate. If there are N modules in the original placement problem, and we break it up into smaller problems of size m, there will be (N/m) resulting subproblems. We assume that N is a multiple of m , and use the notation $P$for the quantity (N/m) which represents the number of clusters in the *reduced placement problem* . There are exactly

$$Q(N,m,P) = \frac{N!}{(m!)^P \times P!} \tag{1}$$

ways of dividing a set of N modules into subsets of m modules each. If we are indeed seeking the optimum solution to the original placement problem (of size N), we must investigate the entire search space consisting of $Q(N,m,P)$ partitions. We make several observations at this stage. If we formulated the placement problem as one of optimally assigning N modules to N slots on a chip surface, the search space of the optimization problem consists of N! assignments. However, when we formulate the problem in terms of partitions, and assume that the local placement is $free$, we see that the search space shrinks to $Q(N,m,P)$. This becomes more clear if we imagine that the clusters are being placed concurrently. Given a random partition of N modules into Ppartitions, let us assume that a parallel computing system assigns one processor to each partition. Each processor operates on a partition (cluster of size m) and solves the assignment problem of size m. Next, the (N/m) clusters are placed on the chip; this step corresponds to an assignment problem of size (N/m). The parallel system must exhaust $Q(N,m,P)$ partitions in order to arrive at an optimal placement of N modules. This is consistent with

$$N! = \left\{\frac{N!}{(m!)^{N/m}(N/m)!}\right\}\left\{(m!)^{N/m}(N/m)!\right\} \tag{2}$$

When the values for m and $P$are chosen somewhere midway between 1 and N, the function $Q(N,m,P)$ exhibits exponential increase with N. As a result the computational requirements of the placement scheme are very large. We therefore propose a local search mechanism to arrive at a suboptimal solution to the placement problem. In this method, only a fraction of $Q(N,m,P)$ partitions are investigated before arriving at a suboptimal solution. We give the algorithm **PA** below.

```
procedure PA (N, m, P);
constant ξ;
begin
    repeat
        Randomly Partition N logic modules into
        Pclusters of size m;
        for every cluster i ∈ {1 .. P} do
            LocalPlace (m, i);
        PlaceClusters (P);
    ξ times;
end
```

The procedure *PA* is largely self explanatory. The constant ξ used in the procedure dictates the fraction of the search space considered by the placement algorithm in arriving at the suboptimal solution. Clearly, $1 \leq \xi \ll Q(N,m,P)$. The subroutine *LocalPlace* used in the procedure produces an optimal placement of m modules. The subroutine *PlaceClusters* optimally places the clusters.

As remarked earlier, the procedure PA lends itself to parallel implementation. We can execute the Pcalls to subroutine *LocalPlace* in parallel by assigning one processor to each invocation of the subroutine. Furthermore, the execution of each invocation of the subroutine may itself be parallelized. We have remarked that *LocalPlace* and *PlaceClusters* use enumerative techniques for placement. However, well known heuristics like local search can be used in the design of these steps also. As a result, the parallel placement techniques discussed in Section 3 can be effectively used to implement *LocalPlace* and *PlaceClusters*. Such a scheme leads to a highly parallel placement system. In the following section we describe the organization of the PPS.

5 Organization of a Parallel Placement System

5.1 The Reduced Array Architecture

The reduced array architecture (RAA) [1,14] essentially consists of a one dimensional array of processors and a two dimensional array of memory elements. The intent here is to organize a

small number of processing elements (PE's) to access a large two dimensional array of data. Hence the term *reduced array*.

The RAA consists of n PEs which access a two dimensional memory with row and column access capability. The organization has n^2 memory modules with processor PE_i accessing the memory in the *ith* row and *ith* column. The organization of such an architecture is shown in Figure 1 and details of the PE are shown in Figure 2.

The processors operate in Single Instruction Multiple Data (SIMD) mode. i.e. the n PEs are supervised by a central Control Processor (CP), each PE receiving the same instruction from the CP but operating on different data sets. The basic capabilities of the CP include decoding the vector instructions of the user programs, determining which PEs are to be activated, loading the index and data registers of the PEs via a system data bus or by broadcasting data on the data bus using a control bus. Since not all the PEs need be active during every instruction cycle, a simple masking scheme can be used by the CP to activate or deactivate one or more PEs. The CP has a global n-bit masking register which indicates the status of each of the n PEs. Details of such an organization may be found in [8].

Figure 2 shows a minimal hardware configuration that a Processing Element is assumed to have. Each PE_i contains an ALU and a fixed number of r registers $R_1, R_2, \ldots, R_r$ which hold operands and intermediate results, a status flag A, which indicates whether PE_i is active or inactive during an instruction cycle and a register ID which holds an m-bit (assuming $n = 2^m$) address of PE_i. For instructions that include memory reference operations, each PE_i accesses the memory location in row i or column i as specified by the Memory Address Register (MAR). The row (column) index registers, IR (IC) are used as offsets into the row (column) memory module being accessed by the PE. Finally, a Memory Data Register, MDR is used to hold data read from or to be written to memory module in row i or column i. Each PE is connected to exactly one row and one column of memory modules. For example, PE_0 is connected to memory modules $M_{0j}, 0 \leq j \leq n-1$ and $M_{i0}, 0 \leq i \leq n-1$. This implies that memory module $M_{ij}, 0 \leq i, j \leq n-1$ can only be accessed by PE_i and PE_j.

The memory array can be accessed as follows: During a memory access, a PE can read or write from a single memory location in its row or column. A single bit is used to indicate whether the access is *row access* or *column access*. Each PE specifies a $\log_2 n$ bit address in its MAR to select the memory module to be accessed. Thus in row access, each PE_i accesses a memory location in row_i i.e. PE_i accesses a location M_{ij}, where j is specified in the MAR of PE_i. Similarly in column access, PE_i is allowed to access column i of memory array. The location M_{ki} accessed by PE_i is specified by the MAR in PE_i. Notice that there can not be any memory access conflict, since during a memory access, either all PEs access their row or all PEs access their column (but not both). Note also that the memory module M_{ij} acts as a shared space between PE_i and PE_j.

As stated above, apart from the memory access feature, the organization is similar to a standard SIMD machine. The SIMD organization suits well due to the regular data flow inherent in the PPS. Also, the memory address registers in each PE provide individual access to row and column of memory. This feature will be exploited by our parallel algorithms to execute the same program but on different data with different memory access patterns. Throughout the paper we make the following standard assumptions used in the analysis of algorithms.

1. The input data is initially loaded into n^2 memory modules. This can be done in $O(n)$ time by loading the data using the n PEs in the array.

2. The total execution time is the sum total of time for computation and time for data movement. A basic computation step (multiply, add, divide) as well as basic data movement (row or column access) is assumed to take one unit of time.

The RAA has several advantages of over other parallel processing architectures which have been proposed or built for for specific applications. First of all, the RAA is a general purpose machine. It has been used for various applications such as image processing [1] and signal processing [14]. Second, the RAA has a regular structure which makes it suitable for a VLSI implementation.

5.2 Placement on the Reduced Array Architecture

In what follows, we propose the architecture of a hardware accelerator for the placement algorithm discussed in the previous section. We begin by observing that an array architecture suits well to parallelize the algorithm PA, since the processing elements of the array architecture can be used to concurrently execute all the invocations of *LocalPlace*. Accordingly, the Parallel Placement System proposed in this section has an SIMD architecture with a Control Processor (CP) and PProcessing Elements (PEs). There are P^2 blocks of memory elements, each block containing m^2 memory cells. There are therefore N^2 memory elements in the PPS. The processors and memory blocks are organized in the form of an RAA as described in Section 5.1 Thus, processing element PE_i has access to the the row i of memory blocks,and column i of memory blocks. Therefore, the memory block (i, i) may be considered as the *local memory* of the processor i. The memory block $(i, j), i \neq j$, may be considered as the *mailbox memory* shared by processing elements i and j. The processors PE_i and PE_j can exchange data through the memory block (i, j); the same is true about the memory block (j, i). The CP has access to all the memory blocks. With the assumption of the above configuration, a parallel placement scheme based on the Divide and Conquer strategy can be implemented as described below.

5.2.1 Algorithm PPA

Step 1 (Initialization)

The CP generates an initial partition of the modules . The N modules to be placed are divided into Pgroups of m elements each. Note that the number of PEs is P= N/m. Let the set of modules received by processor PE_i be denoted by M_i. We sometimes refer to this as a *cluster* of m modules. A vector M is used to store the initial partition. Note that the length of the vector M is m, and there is one such vector corresponding to every PE.

The initial partition can be :

1. A constructive partition (e.g., a suboptimal partition obtained using a *connected components* algorithm [6]) *or*
2. A random partition (e.g., $M_i = \{(i-1)m+1, ..., im\}$

for $i = 1, 2, \ldots, P$.

Step 2 (Local Placement in Parallel)

The CP broadcasts an *LPLACE* instruction to all the PEs. The vector instruction LPLACE initiates a local placement of modules in each of the PEs. Each PE works on a placement problem of size m. Since m is a small number, the PEs use techniques such as enumeration, partial enumeration, or Branch and Bound to arrive at an optimal local placement. Since m is a constant, the amount of time spent by PE_i on the placement of the set of modules in M_i is the same for all $i = 1, 2, \ldots, P$. At the end of the local placement step, each PE has the cost of the local placement. We denote the cost of the local placement by PE_i as LC_i.

Step 3 (Computation of collapsed C-matrix)

The PPS computes the collapsed C-matrix, which we shall refer to as C'. If this step is carried out sequentially, it would require $O(N^2)$ steps. On the PPS, however, the step can be parallelized. It is easy to see that the Collapsed C-matrix has P^2 elements. We can assign one PE to compute a set of P elements in the matrix C'. A speedup factor of $P$is therefore achieved in the computation of the matrix C'.

Step 4 (Placement of Clusters)

Using the matrix C', the CP places the clusters. The CP also uses an enumerative technique to optimally place the clusters, since the number of clusters is small. In this process of cluster placement, the CP calculates the cost of placing the clusters. We refer to this as the Interconnection Cost (IC). It may be noted that steps 2 and 3 above can be executed in parallel if we permit concurrent reads [8]. This implies that the CP executes the cluster placement algorithm at the same time when each individual PE executes the local placement algorithm to place the modules within a cluster.

Step 5 (Evaluation of Total Cost)

The CP broadcasts an *ECOST* instruction. The total cost of the placement is now evaluated. Referring to the global cost as GC, we have:

$$GC = \sum_{i=1}^{P} (LC_i) + IC \qquad (3)$$

For example, if the cost of a placement is characterized by the wire length, GC corresponds to the total wire length, LC corresponds to the cluster wire length, and IC to the wiring required to connect the clusters.

Step 6 (Accept or Reject new placement)

After the execution of steps 1 through 4, the CP has access to the global cost of the current configuration and the global cost of the *previous* configuration. (For the first configuration, we set COST(previous configuration) $= +\infty$).

The CP compares the two global costs and decides to retain the old configuration of the new configuration depending on some heuristic such as iterative improvement [5] or the Metropolis procedure [10].

Step 7 (Generate new partition)

The CP broadcasts a *SHUFFLE* instruction. In the shuffle step, the partition of the original placement problem is altered in a random fashion. Thus, λ modules are randomly picked from λ different clusters ($\lambda \leq P$). These λ modules are permuted among the clusters to which they belong as described later in Section 5.3. Therefore, we obtain a different partition of N modules into P clusters.

Step 8 (Repeat steps 2 through 7)

The CP decides to reiterate steps 2 through 7 depending on some predefined criterion such as maximum time required to be spent on the placement.

5.3 Implementation of PPA on the Reduced Array

5.3.1 Random partition generation

In what follows, we describe a scheme to generate random partitions and evaluate the cost of local placement using the PEs on the RAA. In the implementation discussed, we assume that the control processor has an N-bit register R. The CP broadcasts the contents of R to all the PEs following the broadcast of the *SHUFFLE* instruction. The register R must be viewed as a bit-vector of length N, where N is the number of modules in the original placement problem. For all i in the range 1 to N, we have :

- If $R_i = 1$, it means that the module i will be displaced from its current partition (cluster).
- If $R_i = 0$, the module i will continue to stay in its current partition.

We denote by M_i the partition corresponding to a module i. It is easy to see that $M_i = \lceil i/m \rceil$. The shuffle operation will move module j from partition M_j to partition M_i if the following conditions are met:

1. If two bits R_i and R_j are both set to 1, and $j > i$,
2. If $R_k = 0, \quad \forall \subset < k < j$.

A $\lambda - shuffle$ consists of a round robin movement of λ modules among λ different clusters. Consider the following example with $N = 12$, $m = 3$, $P = 4$.

i	1	2	3	4	5	6	7	8	9	10	11	12
R	0	1	0	0	1	0	1	0	1	0	0	1
M	1	1	1	2	2	2	3	3	3	4	4	4

In the above example, $R_2 = R_5 = R_7 = R_9 = R_{12} = 1$ and $M_2 = 1, M_5 = 2, M_7 = M_9 = 3, M_{12} = 4$. The modules will be permuted as follows: $12 \leftarrow 2 \leftarrow 5 \leftarrow 7 \leftarrow 9 \leftarrow 12$.

Several observations can be made on the basis of above discussion. To bring about a λ-shuffle, the register R must contain exactly λ 1's. If we divide the bit vector R into Pparts, it is clear that at most one bit in each part need be set to 1 in order to create a displacement in the corresponding partition. If more than one bit in a part is 1, only the leftmost 1 is retained. In the above example, bits 7 and 9 in partition 3 are set to 1. The effect of $R_9 = 1$ is to move module 9 from partition 3 to partition 3, which is a *useless* move. So, we can achieve the same effect by setting $R_9 = 0$ and retaining $R_7 = 1$. It is clear that λ must vary from 1 to $P - 1$. This is in agreement with the fact that the number of 1's in R varies from 2 to P. The register R is

available to all the PEs. Each PE uses bit manipulation instructions to retain the leftmost 1 in its bit-partition and sets all the remaining 1's in its bit-partition to 0. By looking at the register R, each PE can find the pattern of required data movement.

5.4 Data Movement after a shuffle

The connectivity matrix is stored in the memory of the PPS in the following manner. If the partition corresponding to a processor i is $\{q_{i_1}, q_{i_2}, \ldots, q_{i_m}\}$ the m^2 elements of the connectivity matrix C that correspond to the modules in the partition are stored in the memory module (i, i). This enables each PE_i to use the elements in module (i, i) to perform a local placement *without causing memory conflicts.* The following Lemma is used as the key fact in the solution.

Lemma 1 *A λ-shuffle of a partition can be realized by using at most $(\lambda - 1)$ 2-shuffles.*

Proof : The Lemma states a fundamental result in the theory of permutations. A partition of N modules, when written down without the separators between partitions, is, infact, a permutation of $1 \ldots N$. The result therefore holds for partitions. □

We have discussed how a bit-vector of length N can be used to introduce λ-shuffles. In light of the above lemma, a λ-shuffle can be realized by considering only two 1's of the bit-vector R at a time. An example will clarify things.

Example: Let $N = 12$, $m = 3$, $\lambda = 4$. The separator | is used between partitions. Let X denote the initial (before shuffle) partition and X_f denote the final partition (after shuffle).

i :	1	2	3	4	5	6	7	8	9	10	11	12
R :	0	0	1	0	1	0	1	0	0	0	1	0
X :	4	5	7	3	2	1	9	8	6	10	11	12
X_f :	4	5	2	3	9	1	11	8	6	10	7	2

The partition X_f can be realized starting from partition X by the application of 3 2-shuffles R_1, R_2, and R_3 as shown below.

R_1 :	0	0	1	0	1	0	0	0	0	0	0	0
X :	4	5	7	3	2	1	9	8	6	10	11	12
X_1 :	4	5	2	3	7	1	9	8	6	10	11	2
R_2 :	0	0	0	0	1	0	1	0	0	0	0	0
X_1 :	4	5	2	3	7	1	9	8	6	10	11	2
X_2 :	4	5	2	3	9	1	7	8	6	10	11	2
R_2 :	0	0	0	0	0	0	1	0	0	0	1	0
X_2 :	4	5	2	3	9	1	7	8	6	10	11	2
X_3 :	4	5	2	3	9	1	11	8	6	10	7	2

It may be observed that $X_3 = X_f$, in accordance with Lemma 1.

5.5 Data Movement for a 2-shuffle

Consider the situation where the N logic modules are randomly distributed into (N/m) partitions initially. Without loss of generality we can consider the following partition:
Modules $(i-1)m+1$ through im are assigned to the partition i (*i.e.,* modules $1, \ldots, m$ are assigned to partition M_1, modules $m+1, \ldots, 2m$ to partition M_2 , etc). For the configuration described above, the connectivity matrix is stored in the memory in the *shuffle row-major order*[8].

Now consider the operation of shuffling 2 modules q_1 and q_2. For all $x, y \in \{1, 2, \ldots, N\}$ the effect this operation on the storage of the connectivity matrix is one of

1. renumbering all the entries of the form (q_1, x) to (q_2, x)
2. renumbering all entries of the form (q_2, y) to (q_1, y)
3. renumbering all the entries of the form (x, q_1) to (x, q_2)
4. renumbering all entries of the form (y, q_2) to (y, q_1)

In the language of hardware, this means *exchange the contents of rows and columns corresponding to q_1 and q_2.* Since the RAA allows simultaneous access to different rows and columns of the memory blocks, the above mentioned data transfer can be made concurrently. Figure 3 below illustrates data movement with an example where $N = 9$, and $P = m = 3$. Initially, modules 1,2, and 3 are assigning to P_1, modules 4,5,6 to P_2, and modules 7,8, 9 to P_3. We assume that the shuffle operation causes a round robin movement of modules 1, 5, and 7. Snapshots of the memory before and after this shuffle are shown in Figures 3 through 6. Note that this data movement can be done in 2 transpositions, namely the transposition of modules 1 and 5, and transpositions of module 5 and 7.

The data movement after the transposition of modules 1 and 5 is explained with reference to Figures 3 - 6. There are two steps in which such a data movement is done:

1. Interchange the *rows* 1 and 5 in the memory
2. Interchange the *columns* 1 and 5 in the memory.

Figures 4 and 5 show the contents of the memory after the execution of Steps 1 and 2 above. A similar series of data movements can be used to affect a transposition of modules 5 and 7. The final configuration of memory is shown in figure 6.

6 A Study of PPS through Simulation and Analysis

Referring to placement algorithm PA presented in Section 4, we note the following. If we let $\mu(N) = \log \frac{N}{P}$ we see that the algorithm PA proceeds by subdividing the original placement problem into $2^{\mu(N)}$ small placement problems. Since we assumed that the smaller placement problems are solved by enumeration, there exist constants α and β such that the execution time of each local placement is $\leq \alpha\beta^m$ [5]. The execution time of algorithm PA is therefore bounded by

$$\xi[2^{\mu(N)}\alpha\beta^m + \alpha\beta^P + O(N^2)] \tag{4}$$

The term O (N^2) comes from the fact that, before the placement of clusters begins, the collapsed C-matrix C' must be computed, which requires O (N^2) arithmetic operations. Every SHUFFLE instruction is followed by a data movement as explained earlier; this data movement takes O (N) time. Using PPS, we obtain a speedup which is slightly less than Pdue to the following reason. All the local placements are made concurrently; this contributes a speedup of P. The computation of C' is speeded up by a factor of P as we explained in Section 5. However, the placement of clusters is performed by the CP in a separate step after the local placements are done. The expected speedup of PPS is, therefore,

$$P\frac{2^{\mu(N)}\alpha\beta^m + \alpha\beta^p + O(N^2)}{2^{\mu(N)}\alpha\beta^m + P\alpha\beta^P + O(N^2)} \tag{5}$$

6.1 Selection of m and P

In Section 5, we have mentioned that the parameters m and Pmust be selected somewhere midway between the values 1 and N. Such a selection is a trade off between the number of processors and the amount of parallelism, and results in a cost effective parallel solution. We can state our goal as the minimization of the function (m+P), subject to the constraint that the product m.P is fixed at N. Such a minimum occurs when $m = P = \sqrt{N}$. The corresponding value of $Q(N, m, P)$ is :

$$\frac{N!}{\sqrt{N}^{\sqrt{N}+1}} \tag{6}$$

Our simulation results indicate that such a selection of m and P is not the best so far as the quality of the final layout is under consideration. Choosing $m = N^{2/3}$ and $P = N^{1/3}$ yielded the best solutions.

6.2 Simulation of PPS

We have conducted a simulation study of PPS on a Sun Workstation. The simulator was programmed in C. We compared the performance of the parallel placement scheme with a sequential placement algorithm based on *iterative improvement*. We selected the *total wire length* as our cost function. The local placement was performed using the iterative improvement heuristic rather than an enumerative technique.

The following inferences can be drawn from the Figures 7 through 9. The selection of cluster size m is crucial in deciding the performance of the PPS. As indicated in Section 6.1, choosing $m = N^{2/3}$ and $P = N^{1/3}$ yield the best solutions. The striking difference which we observed between the partitioning algorithm and the iterative improvement algorithm is in the number of *improvement cycles* γ required by each of these schemes to arrive at a configuration whose cost is a certain fraction of the final suboptimal solution. It is clear that in either of the two cases, γ is a measure of the *rate of convergence* of the optimization algorithm. Our results show that in a given amount of time τ, PPS yields far better quality solutions than a sequential iterative improvement. The rate at which the cost function converges towards the final value is extremely large compared to that of the conventional iterative improvement technique. Simulation results are plotted in Figures 7, 8, 9.

7 Conclusions

In this paper, we have presented a parallel scheme to place a large number of circuit modules. Our approach is based on a Divide and Conquer paradigm. The placement algorithm partitions the circuit into clusters and uses parallel processing to optimally place each of the clusters. The clusters are then placed optimally on the carrier. In the latter phase, information regarding connectivity between clusters is derived from the original connectivity matrix by a collapsing procedure. We have described an implementation of the parallel placement algorithm on the Reduced Array Architecture. A simulation of the PPS is described and results of simulation are presented.

References

[1] H.M. Alnuweiri and V.K. Prasanna Kumar. *Efficient Image Computations on VLSI Architectures with Reduced Hardware*, 1987 Workshop for Pattern Analysis and Machine Intelligence.

[2] T. Blank. *A Survey of Hardware Accelerators used in Computer-Aided Design*, IEEE Design and Test, August 1984, pp. 21 - 39.

[3] A. Casotto et al.. *A Parallel Simulated Annealing Algorithm for the Placement of Macro-cells*, IEEE-Trans. on Computer-Aided Design, Sep 1987, Vol. CAD-6, Number 5, pp. 838-847.

[4] D.J. Chyan and M.A. Breuer. *A Placement algorithm for Array Processors*, Proc. of the 20th Design Automation Conf., 1983, pp. 182 - 188.

[5] M. Hanan and J.M. Kurtzberg. *Placement Techniques*, in Design Automation of Digital Systems Vol. 1 : Theory and Techniques, Ed., M.A. Breuer, Eaglewood Cliffs, New Jersey, Prentice Hall, 1972, pp. 213 - 282.

[6] E. Horowitz and S. Sahni. *Fundamentals of Data Structures*, Computer Science Press, Maryland, USA, 1983, pp. 352.

[7] T.C. Hu and E.S. Kuh. *Theory and Concepts of Circuit Layout*, in VLSI Circuit Layout : Theory and Design, Ed. T.C. Hu and E.S. Kuh, IEEE Press, 1985, pp. 144 - 152.

[8] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing*, McGraw-Hill Book Company, 1985.

[9] A. Iosupovici et al.. *A Module Interchange Placement Machine*, Proc. of the 20th Design Automation Conf., June 1983, pp 171-174.

[10] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. *Optimization by Simulated Annealing*, Science, vol. 220, no. 4598 pp. 671-680, May 13, 1983.

[11] S. A. Kravitz. *Placement by Simulated Annealing on a Multiprocessor*, IEEE-Trans. on Computer-Aided Design, July 1987, Vol. CAD-6, Number 4, pp. 534-550.

[12] C. P. RaviKumar. *Hardware Accelerators for VLSI Layout*, M.E. Project Thesis, Dept. of Comp. Sci. and Automation, Indian Institute of Science, 1987.

[13] C. P. Ravi Kumar and L.M. Patnaik. *Parallel Placement based on Simulated Annealing*, Proc. of the Int. Conf. on Comp. Des., Oct 1987.

[14] Sarma Sastry and V. K. Prasanna Kumar. *Efficient Signal Processing on a reduced hardware VLSI Array*, Proc. of the 1987 Int. Conf. on Parallel Processing.

[15] K. Ueda et al. *A Parallel Processing Approach for Logic Module Placement*, IEEE-Trans. on Computer-Aided Design, Jan 1983, Vol. CAD-2, Number 1, pp. 39-47.

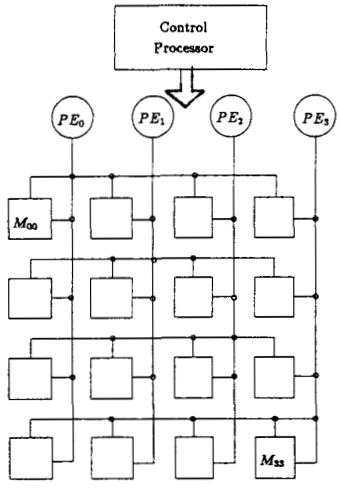

Figure 1: Array Architecture for $n = 4$

$$\begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} & c_{16} & c_{17} & c_{18} & c_{19} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & c_{26} & c_{27} & c_{28} & c_{29} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} & c_{36} & c_{37} & c_{38} & c_{39} \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} & c_{46} & c_{47} & c_{48} & c_{49} \\ c_{51} & c_{52} & c_{53} & c_{54} & c_{55} & c_{56} & c_{57} & c_{58} & c_{59} \\ c_{61} & c_{62} & c_{63} & c_{64} & c_{65} & c_{66} & c_{67} & c_{68} & c_{69} \\ c_{71} & c_{72} & c_{73} & c_{74} & c_{75} & c_{76} & c_{77} & c_{78} & c_{79} \\ c_{81} & c_{82} & c_{83} & c_{84} & c_{85} & c_{86} & c_{87} & c_{88} & c_{89} \\ c_{91} & c_{92} & c_{93} & c_{94} & c_{95} & c_{96} & c_{97} & c_{98} & c_{99} \end{pmatrix}$$

Figure 3: Initial C-Matrix Storage

$$\begin{pmatrix} c_{51} & c_{52} & c_{53} & c_{54} & c_{55} & c_{56} & c_{57} & c_{58} & c_{59} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & c_{26} & c_{27} & c_{28} & c_{29} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} & c_{36} & c_{37} & c_{38} & c_{39} \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} & c_{46} & c_{47} & c_{48} & c_{49} \\ c_{11} & c_{12} & c_{13} & c_{14} & c_{15} & c_{16} & c_{17} & c_{18} & c_{19} \\ c_{61} & c_{62} & c_{63} & c_{64} & c_{65} & c_{66} & c_{67} & c_{68} & c_{69} \\ c_{71} & c_{72} & c_{73} & c_{74} & c_{75} & c_{76} & c_{77} & c_{78} & c_{79} \\ c_{81} & c_{82} & c_{83} & c_{84} & c_{85} & c_{86} & c_{87} & c_{88} & c_{89} \\ c_{91} & c_{92} & c_{93} & c_{94} & c_{95} & c_{96} & c_{97} & c_{98} & c_{99} \end{pmatrix}$$

Figure 4: C-Matrix after Step 1

$$\begin{pmatrix} c_{77} & c_{72} & c_{73} & c_{74} & c_{71} & c_{76} & c_{75} & c_{78} & c_{79} \\ c_{27} & c_{22} & c_{23} & c_{24} & c_{21} & c_{26} & c_{25} & c_{28} & c_{29} \\ c_{37} & c_{32} & c_{33} & c_{34} & c_{31} & c_{36} & c_{35} & c_{38} & c_{39} \\ c_{47} & c_{42} & c_{43} & c_{44} & c_{41} & c_{46} & c_{45} & c_{48} & c_{49} \\ c_{17} & c_{12} & c_{13} & c_{14} & c_{11} & c_{16} & c_{15} & c_{18} & c_{19} \\ c_{67} & c_{62} & c_{63} & c_{64} & c_{61} & c_{66} & c_{65} & c_{68} & c_{69} \\ c_{57} & c_{52} & c_{53} & c_{54} & c_{51} & c_{56} & c_{55} & c_{58} & c_{59} \\ c_{87} & c_{82} & c_{83} & c_{84} & c_{81} & c_{86} & c_{85} & c_{88} & c_{89} \\ c_{97} & c_{92} & c_{93} & c_{94} & c_{91} & c_{96} & c_{95} & c_{98} & c_{99} \end{pmatrix}$$

Figure 5: C-Matrix after Step 2

$$\begin{pmatrix} c_{55} & c_{52} & c_{53} & c_{54} & c_{51} & c_{56} & c_{57} & c_{58} & c_{59} \\ c_{25} & c_{22} & c_{23} & c_{24} & c_{21} & c_{26} & c_{27} & c_{28} & c_{29} \\ c_{35} & c_{32} & c_{33} & c_{34} & c_{31} & c_{36} & c_{37} & c_{38} & c_{39} \\ c_{45} & c_{42} & c_{43} & c_{44} & c_{41} & c_{46} & c_{47} & c_{48} & c_{49} \\ c_{15} & c_{12} & c_{13} & c_{14} & c_{11} & c_{16} & c_{17} & c_{18} & c_{19} \\ c_{65} & c_{62} & c_{63} & c_{64} & c_{61} & c_{66} & c_{67} & c_{68} & c_{69} \\ c_{75} & c_{72} & c_{73} & c_{74} & c_{71} & c_{76} & c_{77} & c_{78} & c_{79} \\ c_{85} & c_{82} & c_{83} & c_{84} & c_{81} & c_{86} & c_{87} & c_{88} & c_{89} \\ c_{95} & c_{92} & c_{93} & c_{94} & c_{91} & c_{96} & c_{97} & c_{98} & c_{99} \end{pmatrix}$$

Figure 6: C-Matrix after SHUFFLE

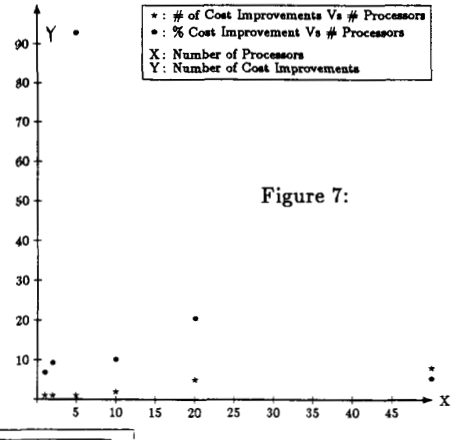

Figure 7:

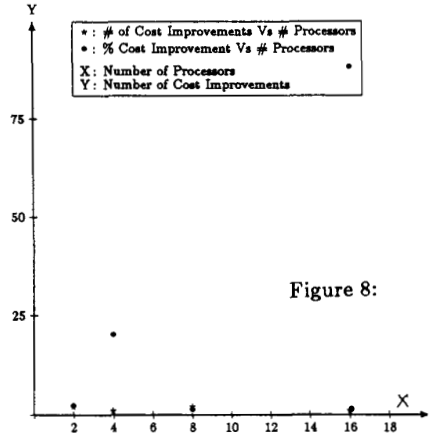

Figure 8:

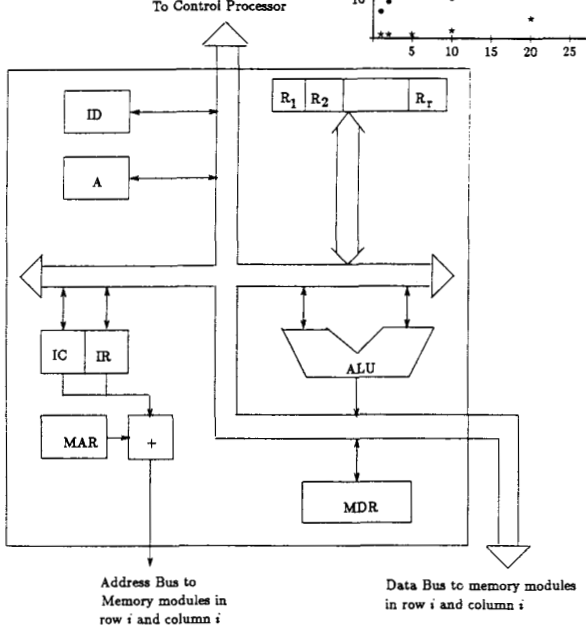

Figure 2: Internal Organization of PE_i

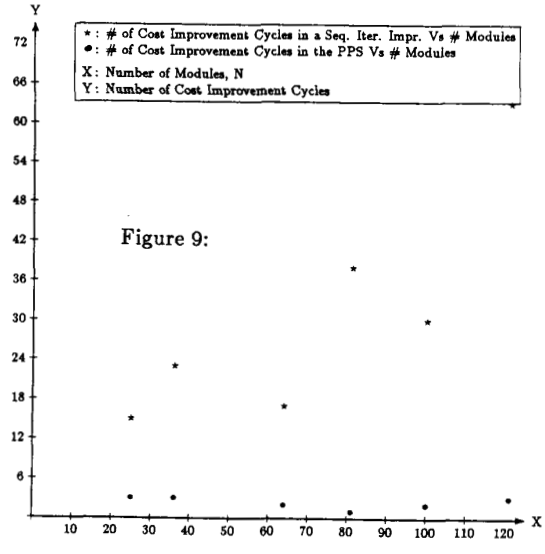

Figure 9: