A Genetic Algorithm for VLSI Floorplanning

Christine L. Valenzuela (Mumford)[1] and Pearl Y. Wang[2⋆]

[1] Cardiff School of Computer Science & Informatics, Cardiff University, UK.
C.L.Mumford@cs.cardiff.ac.uk
[2] Department of Computer Science MS4A5, George Mason University, Fairfax, VA 22030-4444, USA.

Abstract. We present a genetic algorithm (GA) which used a normalized postfix encoding scheme to solve the VLSI floorplanning problem. We claim to have overcome the representational problems previously associated with encoding postfix expressions into GAs, and have developed a novel encoding scheme which preserves the integrity of solutions under the genetic operators. Optimal floorplans are obtained for module sets taken from some MCNC benchmarks. The slicing tree construction procedure, used by our GA to generate the floorplans, has a run time scaling which compares very favourably with other recent approaches.

1 Introduction

One of the most important stages in the physical design of VLSI circuits is floorplan design: the placement of a set of rectangular circuit modules on a chip so as to minimize the total area and the total interconnecting wire length. When placing circuit modules (or *macro cells*) many of the modules are themselves not yet fully designed and frequently have some flexibility in their shape. For example a circuit module made up from 12 identical components may have them placed in one row of 12 components, 2 rows of 6 components, 3 rows of 4 components etc., offering the floorplan designer a range of possible shapes for that module. Using a technique based on a *slicing floorplan*, which can be obtained by recursively dividing a rectangle into two parts with either a vertical or a horizontal line, it is possible to fully exploit the available flexibility in the circuit modules and efficiently combine module placement and area optimization into a single algorithm. The purpose of our paper is to present a genetic algorithm (GA) which appears to be very effective at breeding good (often optimal) slicing floorplans. We believe that with our GA we have overcome most of the representational problems usually associated with encoding floorplans, and claim that our solution is elegant as well as effective.

We now summarize some important definitions based on [13]. Let a given rectangle, R, have height $h(R)$, width $w(R)$ and area $A(R)$. The *aspect ratio* of R is the ratio $h(R)/w(R)$. A *soft rectangle* is one that can have different shapes as long as the area remains the same. The *shape flexibility* of a soft rectangle

⋆ This work was partially supported by NASA grant NAG-5-4868.

specifies the range of its aspect ratio. A soft rectangle of area $A(R)$ is said to have a shape flexibility r if its aspect ration can take on any value between $1/r$ and r. The shape flexibility of modules provides a continuous range of candidate aspect ratios for our soft modules.

An alternative to the slicing floorplan, which is favoured by many researchers, is the *non-slicing floorplan*. In the non-slicing floorplan there is no requirement for recursive construction, and tighter packings are often possible using this approach. Our main motivation for using a slicing floorplan approach is that by considering only slicing floorplans, we are able to massively reduce the size of the search space. Stockmeyer [9], examining cases where each subcircuit may have different layout alternatives in a floorplan, showed that there is an efficient polynomial time area optimization algorithm for slicing floorplans whereas the area optimization problem for non-slicing floorplans is NP-Hard.

A recent non-slicing placement technique, called the sequence-pair method [5], has been extended to handle soft modules and area optimization [4]. However the sequence-pair method has to solve expensive convex programming problems in order to determine the exact shape of each module, and this results in a very long runtime. Another relatively new technique, called the Bounded Sliceline Grid (BSG) packing algorithm [6], has proved successful in the placement of hard rectangles (i.e. rectangles with no flexibility in their shape). On the downside, though, a single application of BSG packing algorithm scales at $T(n) = O(n^2)$ for hard modules, given the n x n grid size suggested by the authors. This compares with $T(n) = O(n)$ for hard modules using a slicing floorplan approach. It is our belief that the average case run time scaling of the combined placement and area optimization algorithm for slicing floorplans with soft modules at $T(n) = O(n \lg n)$ will be hard to beat.

To the best of our knowledge the slicing structure representation used in our GA is novel. Although there are several other examples of genetic algorithms applied to slicing tree structures in the literature, it would appear that none of them uses a GA to manipulate encoded normalized postfix strings. Schnecke and Vornberger, for example, used a GA to manipulate the slicing tree directly [8]. However, to facilitate their crossover involves complex repair mechanisms simply to ensure that the final product (or offspring) represents a legal slicing floorplan, with no duplications or deletions of modules. Another approach [2], in what is probably the best known study of its type, used a collection of four different crossovers and applied them to postfix expressions that were not normalized (using non-normalized expressions greatly increases the search space).

We view our main contribution to the field of slicing tree optimization as the extension of the ideas of Wong, Leong and Lui [12]: they used a normalized postfix representation for simulated annealing, and, through the addition of a novel encoding, we have adapted the approach to produce a simple but effective genetic algorithm. We test our GA on soft modules from the benchmark MCNC data sets, *ami33* and *ami49*. The objective of our present study is a 'proof of concept' and we limit our objective function to the construction a floorplan of

minimum area. Other elements, such as the minimization of the total wire length, will be included in the cost function at a later date.

In section 2 we begin with a review of slicing floorplans and their postfix representations, and then we briefly describe our approach to the addition of shape-curves for the combination of soft modules. Section 3 describes the representation we use to encode our slicing floorplans, and also the decoder which interprets these structures as normalized postfix expressions. In section 4 we describe our genetic algorithm and section 5 presents our results. We conclude in section 6 with a summary of our achievements and an outline of our plans for future work.

2 Slicing Structures and Postfix Representations

A *slicing floorplan* is a rectangular floorplan with n basic rectangles that can be obtained by recursively *cutting* a rectangle into smaller rectangles using a series of vertical and horizontal guillotine cuts. A slicing floorplan can be represented in the form of a binary tree, called a *slicing tree*, in which each internal node of the tree is labelled either * or +, corresponding to a vertical or a horizontal cut respectively. Each leaf represents a basic rectangle and is labelled between 1 and n, where n is the total number of basic rectangles. Each slicing tree can be represented, alternatively, using a postfix expression. The postfix expression is derived by carrying out a post-order traversal.

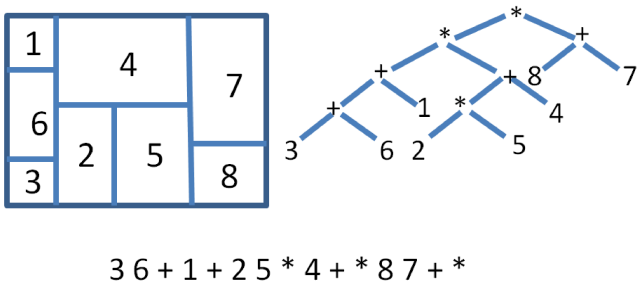

Fig. 1. Slicing floorplan, skewed slicing tree and corresponding normalized postfix expression

There is a one-to-many relationship between slicing floorplans and slicing tree representations for slicing floorplans. If we restrict our representations to skewed slicing trees, however, we obtain unique depictions for slicing floorplans [12]. A *skewed slicing tree* is a slicing tree in which no node and its right child have the same label in {*, +} and it is obtained by making consecutive vertical cuts from right to left, and making consecutive horizontal cuts from top to bottom. The postfix expression derived from a skewed slicing tree is called a

normalized postfix expression, and provides a linear form of the representation. Figure 1 illustrates a typical slicing floorplan, (b) and the corresponding skewed slicing tree and normalized postfix expression. A normalized postfix expression is obtained by traversing a skewed slicing tree in post-order and is characterized by chains of {*, +} operators in which the operators alternate. For example the postfix expression 1 2 3 + * 4 * is normalized, but the expression 1 2 3 + + 4 * is not (because of the two adjacent + symbols). A slicing floorplan with $(n-1)$ cuts will produce n basic rectangles. Thus a postfix expression consists of exactly $2n-1$ entries.

A normalized postfix expression which characterizes a slicing floorplan can be written: $\pi_1 c_1 \pi_2 c_2 \pi_3 c_3 \pi_4 c_4, \ldots, \pi_n c_n$ where $\pi_1 \pi_2 \pi_3 \pi_4 \ldots \pi_n$ represent a permutation of the $1, 2, \ldots, n$ basic rectangles, and the c_i's are chains of operators, either $+*+*+*\ldots$, or $*+*+*\ldots$ If we let $l(c_i)$ represent the length of the chain, c_i, then $\sum_i l(c_i) = n-1$, and $l(c_1) = 0$. Also for any position, $i : 1 \leq i \leq n$, $\sum l(c_i) \leq i-1$ (this is often referred to as the *balloting property*).

2.1 Circuit Module Placement

In the discussion so far, we have viewed a slicing tree as a top down description of a slicing floorplan, in which the slicing tree specifies how a given rectangle is cut into smaller rectangles by vertical and horizontal cuts. An alternative is to view a slicing tree as a description of a bottom up procedure. From a bottom up point of view the slicing tree describes how pairs of smaller rectangles can be combined recursively to yield larger rectangles. Figure 2 shows the actions of the binary operations + and * on the two rectangles A and B: '+' puts B on top of A, and '*' puts B on the right of A. In the example depicted in Figure 2, the two rectangles A and B have combined under + or *, the combined module is replaced by the smallest enclosing rectangle, resulting in the creation of dead space (or waste) in the floorplan.

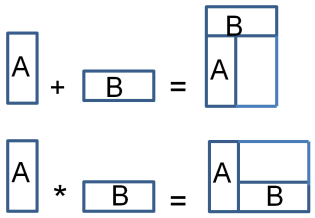

Fig. 2. Binary operations for combining rectangles

2.2 Area Optimization Using soft Modules

Various vertical (y coordinate) and horizontal (x coordinate) dimensions are possible for a soft module with aspect ratio, ρ, such that $1/r \leq \rho \leq r$, and these

can be modelled by a shape curve, Γ. Γ is a smooth, continuous curve, lying entirely within the first quadrant, such that the x and y coordinates of points lying on or above the curve define the *feasible region*.

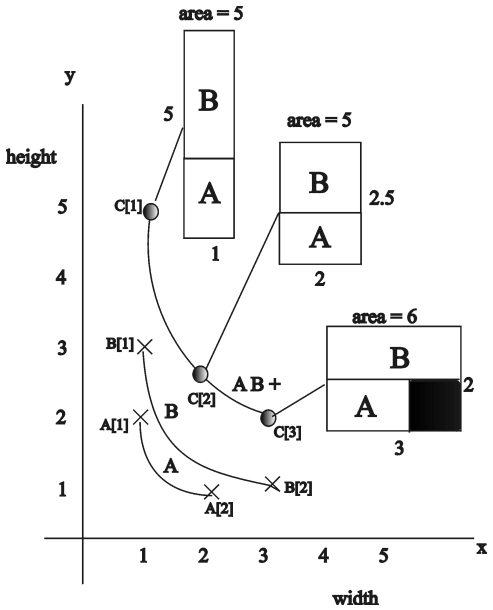

Fig. 3. Shape curve addition for vertical combination. Shape flexibility for A is 2 and B is 3.

Pairs of soft modules, A and B, can be combined by adding their shape curves; AB +, by adding along the y direction and AB *, by adding along the x direction. Figure 3 illustrates a vertical (+) combination of a pair of modules. The shape curves depicted in the diagram indicate possible height and width dimensions for A, B and for the enclosing rectangle A B +. The two points at either end of each of the curves mark the limits of flexibility for the rectangles, which means that the rectangles can only be made taller or wider than this by the addition of dead space. When a pair of soft modules is combined, the new shape curve can be computed simply by adding together the so-called 'corners' of the curves for the component modules. The diagram shows clearly that the shape curves for basic modules of fixed orientation (i.e. no rotation is allowed) are each completely characterized by two 'corners'. (Note: a hard module of fixed orientation is completely characterized by a single point or 'corner').

So much for combining basic rectangles in pairs. In order to produce a slicing floorplan from a postfix expression it is necessary to create a recursive process which combines together super-modules, as well as basic modules, adding together their shape curves in a bottom-up fashion. Fortunately the process of adding shape curves for super-modules is essentially the same as the procedure

for combining two basic modules, only with more 'corners' to add. Although full details of our routines for combining shape curves are omitted from the present paper, to save space, they are published elsewhere [11]. Essentially our approach makes use of our observation that the area of a module/super-module varies uniformly between any two adjacent corners, and we perform a full evaluation of all the points. Since a basic module of fixed orientation has a maximum of two corners, a combined module, produced from two basic modules, will have at most four corners on its shape curve. Combining n modules following an arbitrary slicing tree, where the number of corners may double at each level as the algorithm combines more modules and moves up the tree, gives an average case run time of $T(n) = O(n\lg n)$ [9]. Despite scaling well the run time for our shape curve combination routine is currently rather long, because we have not as yet, incorporated approximations, as suggested by Wong [12], to reduce the number of corners accumulated by our shape curves.

3 The Representation and Decoder

Our representation is order based and consists of an array of records, with one record for each of the basic rectangles of the data set. Each record contains three fields:

- *a rectangle ID field*: this identifies one of the rectangles from the set {1, 2, 3,..., n}
- *an op-type flag*: this boolean flag distinguishes two types of normalized postfix chains, T = + * + * + * +...., and F = * + * + * + *.....
- *a chain length field*: this field specifies the maximum length of the operator chain consecutive with the rectangle identified in the first field.

Algorithm 1 Outline decoder algorithm

1) Examine next (first) record; print the rectangle ID.
2) Generate a chain of alternating operators of op-type specified in the op-type flag. this chain should have length defined in the length field.
3) Print operators, in sequence, from the chain generated in 2) until either you get to the end of the chain or the addition of more operators would violate the balloting property.
4) If there are more records left to process, then go to 1) else complete the normalized postfix expression by printing further operators at the end of the postfix string until the number of operators is one less than the total number of rectangles in the expression.

Our decoder converts a given instantiation of the array of records into a legal normalized postfix expression by writing down the rectangle IDs in the order given, and inserting the type of normalized chain of operators indicated by the

op-type flag immediately following each rectangle number. The maximum length of each chain of operators given in the chain length field is allocated provided that the balloting property is not violated by doing so (i.e. if we are currently processing the i^{th} rectangle in the list the total number of operators in the postfix expression constructed so far must be less than or equal to $(i-1)$). If the decoder reaches the end of the sequence of records and the resulting postfix string has insufficient operators (less than $n-1$), extra operators are added on to the end of the string maintaining the normalized pattern of ..+ * + *... etc. The decoder algorithm is presented in **Algorithm 1**. Below is an example showing an encoded string and its normalized postfix interpretation:

rect5	rect2	rect4	rect1	rect3
op-type*	op-type+	op-type*	op-type*	op-type+
length 2	length 1	length 0	length 2	length 0

Postfix expression generated: 5 2 + 4 1 * + 3 +

4 The Genetic Algorithm

The simple genetic algorithm (GA) used here is derived from the model of [3] and is an example of a 'steady state' GA (based on the classification of [10]. It is It uses the 'weaker parent replacement strategy' first described in [1]. The GA applies the genetic operators to permutations of rectangle records. The fitness values are based on the amount of dead space produced in each floorplan defined by the individual normalized postfix expressions encoded in the population. The first parent is selected deterministically in sequence, but the second parent is selected in a roulette wheel fashion, the selection probabilities for each genotype being calculated using the following formula:

$$selection\ probability = (Rank)/\sum Ranks$$

where the genotypes are ranked according to the values of the dead space that they have produced, with the worst ranked 1, the second worst 2 etc. and the best ranked highest. The GA breeds permutations of records from which our decoder produces normalized postfix strings. These strings are, in turn, processed by a stack to generate a floorplan. For each horizontal or vertical combination, the shape curves are added as described in section 2. As each floorplan design is generated, the dead space is calculated and recorded. The initial population consists of random permutations of records with each basic rectangle represented exactly once in each list. The op-type flag for each record is set to '+' or '*' with equal probability, and the value in the length field is generated in two stages:

- Stage 1: length = 0, with a probability of 0.5
- Stage 2: if the length is not set to zero, then it is generated from a Poisson distribution with mean 3.

4.1 Genetic operators for permutations

We use three different mutation operators, one for each of the fields in our encoding structure (rectangle ID, op-type, and op length):

- **M1** Swap positions of two rectangle IDs.
- **M2** Switch op-type flag, + to * or vice versa.
- **M3** Mutate length by incrementing or decrementing (i.e. length = length + 1, length = length - 1) with equal probability. (If length is zero we increment).

M1 and M2 produce an identical effect to the M1 and M2 operators defined in chapter 3 of [12]. Our M3 operator, on the other hand, is different from the M3 described in Wong *et al*, although its effect is similar. Our M3 will always produce legal postfix expression. In the very early stages of our study we chose some non-problem specific permutation crossovers for testing and carried out some extensive comparisons to test the performance of four crossovers on our data sets. Overall Cycle Crossover (CX) [7] came out best and was selected for our study. Our implementation of CX is efficient and runs in linear time.

For our GA we choose a population size of $20n$, where n is the number of modules in the problem. We chose this rather large size for our population because it matched the number of evaluations undertaken at each temperature by the simulated annealing algorithm of Young and Wong in their recent papers [14, 15]. The GA is halted when 40 generations have passed since the last improvement was recorded in the best-so-far. We set the limits on the aspect ratio for the final enclosing rectangle (i.e. the chip aspect ratio) to those used in 1988 by [12]: $1/2 \leq$ *chip aspect ratio* ≤ 2. Unfortunately there does not appear to be a way to predict a chip aspect ratio in advance of a full evaluation of a postfix expression. To ensure that we obtain acceptable solutions, we simply reject all offspring in which the best point on the final shape curve does not correspond to a legal chip aspect ratio, and try generating them again.

5 Results

Table 1. Means of 5 replicate runs for % dead space of the genetic algorithm

Problem	Shape flexiblity	Genetic algorithm			
		Mean % dead space	# evaluations	Mean run time mins:secs	Best % dead space
ami33	2	0	66734	2:33	0
ami33	3	0	39183	1:33	0
ami33	4	0	28919	1:11	0
ami49	2	0.07	197934	11:42	0
ami49	3	0	101405	6:08	0
ami49	4	0	72913	4:31	0

The results for our GA are summarized in Table 1. We use the modules from the benchmark data sets *ami33* and *ami49* with 33 and 49 modules respectively. Column 2 gives the shape flexibility of the basic rectangles for the experiments in the rows and columns 3 to 6 summarize the dead space obtained by the GA. The percentage of dead space for the best floorplan is recorded over five replicate runs of the GA in column 3, and the overall best for from the five runs is shown in column 6. The total number of postfix expression evaluations averaged over the five replicate runs is noted in column 4, and the average run time appears in column 5. (The individuals rejected because of illegal chip aspect ratios are counted amongst the evaluations.) As we mentioned earlier, we have not as yet incorporated any approximations to reduce the number of 'corners' accumulated during the floorplan construction process. Thus we could surely improve our run times. Because we have used precise calculation we observe vastly inflated run times due to the large number of 'corners' accumulating during the construction process. From Table 1 it is clear that the GA frequently produces optimal results. Figure 4 shows a GA frequently produces optimal results. Figure 4 shows a typical floorplan found by our GA for *ami49*.

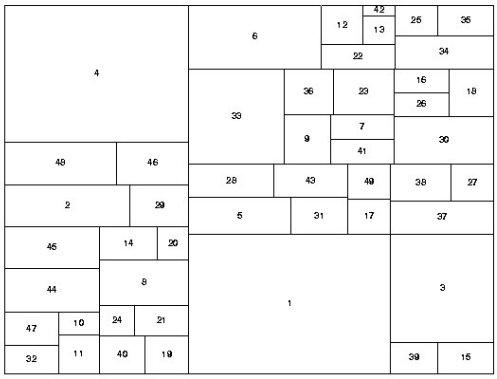

Fig. 4. Ami49 floorplan with shape flexiblity 2 and 5 0 % dead space

6 Conclusions and Further Work

This paper describes a genetic algorithm (GA) which uses a novel encoding to breed normalized postfix expressions for macro cell placement and area optimization in VLSI floorplan design. Our experiments confirm that simple order based genetic operators are effective in guiding the genetic search when our encoding is used.

The only recent related work on VLSI, of which we are aware, is that reported by Young and Wong [13–15]. Young and Wong use soft modules and normalized postfix expressions with their simulated annealing algorithm. In the introduction to [15] the authors report results of less than 1% dead space for several MCNC benchmarks including *ami33* and *ami49*. Although we achieve 0 % dead space for the same problems, we must be cautious about making direct comparisons. We are not yet include wire length in our optimization, and this is usually accounted for by Young and Wong.

We believe that our new approach has opened up some interesting possibilities for genetic algorithms applied to slicing floorplans. Not only does our algorithm produce excellent results but the slicing tree construction process used to generate the floorplans has a run time scaling of $O(n)$ for hard modules and $O(n \lg n)$ for soft modules. This compares very favourably with the Bounded-Sliceline Grid (BSG) that has been used in GAs and Simulated Annealing for VLSI placement by several authors of recent papers. Work in progress is currently focussed on incorporating some simple heuristics into our approach and extending the GA to allow rotation of the modules and super-modules. We also plan to incorporate a cost for wire length into the objective function in the near future, and extend to much larger problems.

References

1. D. J. Cavicchio. *Adaptive search using simulated evolution.* Unpublished doctorial dissertation, University of Michigan, Ann Arbor, 1970.
2. J. P. Cohoon, S. U. Hedge, W. N. Martin and D. S. Richards. *Distributed Genetic Algorithms for the Floorplan Design Problem.* IEEE Transactions on Computer Aided Design, Vol. 10, No. 4, April 1991, pages. 483-492.
3. J. H. Holland. *Adaptation in natural and artificial systems.* Ann Arbor: The University of Michigan Press, 1975.
4. H. Murata and Ernest S. Kuh, *Sequence-pair based placement method for hard/soft /pre-placed modules*, International Symposium on Physical Design, pages 167-172, 1998.
5. S. Nakatake, H. Murata, K. Fujiyoushi. and Y. Kajitani. *Rectangle-packing-based module placement.* Proceedings IEEE International Conference on Computer-Aided Design, pages 143-145, 1995.
6. S. Nakatake, K. Fujiyoshi, H. Murata and Y. Kajitani. *Module Placement on BSG-Structure and IC Layout Applications*, Proceedings of ICCAD, pages 484-491, 1996.
7. I. M. Oliver, D. J. Smith, and J. R. C. Holland, *A study of permutation crossover operators on the traveling salesman problem.* Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms, pages 224-230, 1987.
8. V. Schnecke. and O. Vornberger, *Genetic Design of VLSI-Layouts*, the First International Conference in Genetic ALgorithms in Engineering Systems: Innovations and Applications (GALESIA), IEE 1995, pages 430-435.
9. L. Stockmeyer, *Optimal Orientations of Cells in Slicing floorplan Designs, Information and Control*, vol. 59, pages 91-101, 1983.

10. G. Syswerda, *Uniform Crossover in Genetic algorithms*. Proceedings of the Third International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates, 1989.
11. C. L. Valenzuela and P. Y. Wang. *VLSI Placement and Area Optimization Using a Genetic Algorithm to Breed Normalized Postfix Strings*. IEEE Transactions on Evolutionary Computations, Vol 6 (4) pp 390-401, 2002.
12. D. F. Wong, H. W. Leong and C. L. Liu, *Simulated Annealing for VLSI Design*, Kluwer Academic Press, Boston MA, 1988.
13. F.Y. Young and D.F. Wong, *How Good are Slicing floorplans*, Integration, the VLSI Journal, Vol 23, pages 61-73, 1997.
14. F. Y. Young and D. F. Wong, *Slicing floorplans with pre-placed modules*. Proceedings IEEE International Conference on Computer-aided Design, pages 252-258, 1998.
15. F. Y. Young and D. F. Wong, *Slicing floorplans with range constraints*. International Symposium on Physical Design, pages 97-102, 1999.