A new cell placement algorithm for optimal linear layout

Eric Gautrin, Oumarou Sié
IRISA
Campus de Beaulieu
35042 Rennes Cédex
FRANCE
Phone: (33) 99 84 71 00
Email: gautrin@irisa.fr, sie@irisa.fr

keywords: placement, linear layout, graph cut, regular array.

summary: This paper presents insights gained from an experience with the optimal linear layout of processors in regular arrays. In such design styles, the area optimization is equivalent to a graph mincut computation. We propose a heuristic based on the Gurari and Sudborough equivalent relation and we experiment it on a graph representation adapted to the *mincut* computation.

Abstract

This paper presents insights gained from an experience with the linear layout of processors in regular arrays. In such structures, the processors may be replicated several hundred times. In a linear layout design style, the area optimization is equivalent to a net congestion minimization in the routing area. This problem is similar to the computation of the mincut of a graph or a hypergraph. As a processor is composed of a few cells (less than one hundred), we decided to experiment on algorithms with even a high complexity.

To find an optimal placement, we use first the algorithm of Gurari and Sudborough, because it has the lowest complexity. They propose a polynomial decision algorithm which determines whether a graph has a cut value less than a given constant. Used iteratively from a cut lower bound, the algorithm stops at the first positive answer and delivers a vertex placement respecting the exact *mincut* value.

This decision algorithm has an $O(N^k)$-time complexity where N is the number of vertices and k the cut value. We propose a heuristic based on the same equivalence relation among the partial ordering, but which limits the choice to only the placed vertex successors when extending a partial ordering. For many sample cases, this heuristic gives the exact *mincut* value with better run times.

However, the *mincut* value is not guaranteed to be determined using the heuristic algorithm. We investigate the use of the original decision algorithm to check and decrease this upper bound of the *mincut* value.

For the implementation of these different algorithms, we face two problems. First, the data structures initially proposed by Gurari and Sudborough can not be implemented due to the memory storage requirement. Second, the algorithms use graphs when the natural representation of a circuit is a hypergraph. We propose a graph representation adapted to the *mincut* computation.

1 Introduction

Many signal or image processing algorithms demand regular computations, which can be efficiently implemented on massively parallel architectures such as systolic or regular arrays. These structures are composed of simple processors connected to their nearest neighbors [8]. To produce compact layouts, the automatic generation of the layout is done in two steps: processor layout generation; and array assembly. Array assembly is addressed by the MADMACS system [4]. In this paper, we focus on the processor layout generation.

Processors in such structures are made of few cells, generally less than one hundred. The generation can be done efficiently by a linear layout of predefined

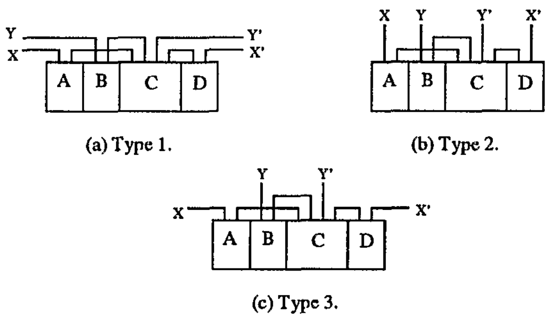


Figure 1: Different types of connector placement constraints

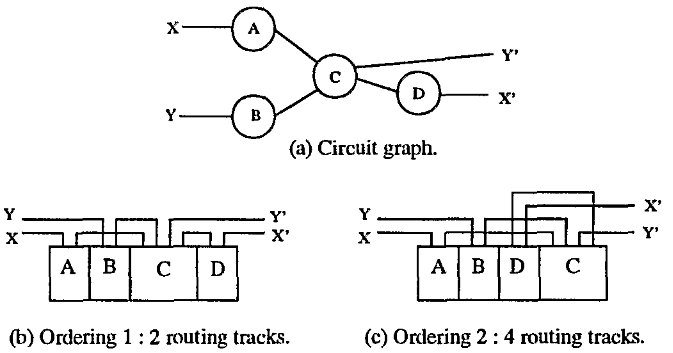

X
A
Y
B
C
D
Y'
X'
(a) Circuit graph.
Y
X
A
B
C
D
Y'
X'
(b) Ordering 1 : 2 routing tracks.
Y
X
A
B
D
C
X'
Y'
(c) Ordering 2 : 4 routing tracks.

Figure 2: Vertex ordering effect on the *cutwidth*.

cells. This layout style is common in circuit design; we can cite gate matrices [2, 14], Weinberger arrays [13], datapath generators [12], and single row of standard cells [7, 11]. However, the processor layout generation must be constrained in order to retain the regularity at the array level. As presented in figure 1, these constraints are related to the processor connector positions.

As a cell has a single layout in the library and also a fixed size, minimization of the global layout area will be achieved by routing area minimization. For such linear layouts, a good approach is to globally reduce the net congestion [1, 6], which corresponds to the routing tracks necessary to achieve the routing. This problem is known as Mincut Linear Arrangement, Backboard Permutation or Optimal Linear Arrangement [9, 5, 10]. Algorithms seek a hypergraph vertex linear ordering which minimizes the maximal cut (also named *cutwidth* or *mincut*) between successive nodes, as illustrated in figure 2.

The *mincut* computation is an NP-complete problem[3]. Many heuristics have been proposed in the past to solve this problem. These approaches are not guaranteed to deliver the exact *mincut* value. However, some authors have proposed decision algorithms which deliver one possible hypergraph linear ordering, with a $cut \leq k$ (where k is a constant integer). By iteratively applying these algorithms, one can find the exact *mincut* of a hypergraph. However, this approach is time-consuming. For example, the Miller and Sudborough decision algorithm [10] has a polynomial-time complexity in $O(N^{k^2+3k+3})$, where N is the vertex number and k the cut value. Gurari and Sudborough [5] propose a decision algorithm with an $O(N^k)$-time complexity where N is the vertex number and k the cut value. For small problems (less than one hundred cells), this algorithm is attractive. However, the direct representation of a circuit is not a graph, but a hypergraph.

In regular arrays, a processor may be replicated several hundred times. A simple width minimization of the processor, for example, in the range of a 5 micron saving in the layout of a routing track, will lead to a 0.5 mm width minimization of a 100 processor wide array. For these reasons, we seek an algorithm which gives us the exact mincut value. As a processor is composed of few cells (less than one hundred), an algorithm with even a high complexity can be implemented if it has acceptable run time on small problems.

We investigate the results of Gurari and Sudborough because their original algorithm has the lowest complexity. Different algorithms based on their equivalence relation are implemented. This equivalence relation is among the partial orderings and guarantees that every partial layouts of a class will be completed in total layouts with the same cutwidth.

The first algorithm is an iterative implementation of their original decision algorithm. Starting from a mincut lower bound, the algorithm is used iteratively until it delivers an answer. However, this algorithm has an $O(N^k)$-time complexity.

We propose a second algorithm which uses a heuristic for induction on the number of vertices. In fact, the heuristic limits the vertex to extend a partial ordering to be one of the successors of the placed vertices. This algorithm is faster and gives, in many cases, similar *mincut* values to those of the original algorithm. However, it does not guarantee finding the mincut value.

The third algorithm combines the two previous ones. First, starting from a lower bound, the heuristic algorithm is used iteratively to approach the mincut value, which is in fact an upper bound. Then, in order to find the exact value, we use the exact algorithm to check and decrease the mincut upper bound.

These three algorithms have been implemented and evaluated. For their implementations, we face two problems. First, the data structures that the authors propose are not reasonnable due to the memory storage requirement. We propose

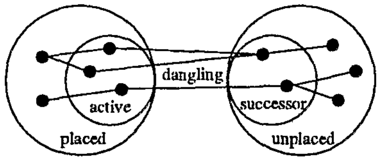


Figure 3: Different subsets in vertices and edges.

other ones. Second, the natural representation of a circuit is a hypergraph and their algorithm treats only graphs. To resolve these problems, we propose a graph representation of a circuit and modify their original algorithm.

Section 2 gives some definitions and presents the equivalence relation among partial orderings proposed by Gurari and Sudborough. Section 3 details the different algorithms. Section 4 presents their implementation. Results are gathered in section 5.

2 Equivalence relation

In this section, we first give some definitions and then present the equivalence relation among the partial orderings, on which the original algorithm of Gurari and Sudborough is based.

2.1 Definitions.

Let $G = (V, \Gamma)$ be a finite undirected graph. A one-to-one mapping function $L : V \rightarrow \{1, \ldots\ldots, card(V)\}$ is a linear ordering of G called a total layout L. A partial layout L' of the graph $G = (V, \Gamma)$ is a one-to-one mapping function $L' : V' \subseteq V \rightarrow \{1, .., card(V')\}$.

Given a partial layout L', the *placed* set of this partial layout is the set of placed vertices and the *unplaced* set is the set of unplaced vertices:

- $placed(G, L') = V'$;
- $unplaced(G, L') = V \setminus V'$;

Given the partition of a partial layout into its *placed* set on one side, and its *unplaced* set on the other side (see figure 3), there exist edges between one vertex of the *placed* set and one vertex of the *unplaced* set. The subset of vertices *placed* is called the *active* set, the subset of vertices *unplaced* is called the *successor* set, and the set of edges between *placed* and *unplaced* is called *dangling*.

- $active(G, L') = \{v \in V' \mid \exists \gamma \in \Gamma \wedge \gamma = (v, v') \wedge v' \in unplaced(G, L')\}$;
- $dangling(G, L') = \{\gamma \in \Gamma \mid \gamma = (v, v') \wedge v \in active(G, L') \wedge v' \in unplaced(G, L')\}$;
- $successor(G, L') = \{v \in unplaced(G, L') \mid v' \in active(G, L') \wedge (v', v) \in \Gamma\}$;

Given a partial layout, we define the *netcut* between two successives vertices as the number of edges between the left and right parts. The *cutwidth* of the layout is the maximum of its *netcuts*, and the *mincut* of the graph is the minimum of the *cutwidths* of total layouts.

- $netcut(G, L', i) = card(\{\gamma \in \Gamma \mid \gamma = (v, v') \wedge L'(v) \leq i \wedge L'(v') > i\})$;
- $cutwidth(G, L') = max_i(netcut(G, L', i))$ where $i \leq card(V')$;
- $mincut(G) = cutwidth(G) = min_L(cutwidth(G, L))$;

2.2 Equivalence relation

To limit the tree search of partial and total layouts, Gurari and Sudborough [5] propose an equivalence relation $\Re$ among the partial layouts of a graph $G(V, \Gamma)$. Two layouts $L1$ and $L2$ are equivalent if:

$$\begin{aligned} cutwidth(G, L_1) &= cutwidth(G, L_2)\,; \\ active(G, L_1) &= active(G, L_2)\,; \\ dangling(G, L_1) &= dangling(G, L_2)\,; \end{aligned}$$

Let L_1 and L_2 two equivalent partial layouts, then:

$$\begin{aligned} placed(G, L_1) &= placed(G, L_2)\,; \\ unplaced(G, L_1) &= unplaced(G, L_2)\,; \\ successor(G, L_1) &= successor(G, L_2)\,; \end{aligned}$$

If L_1 and L_2 are equivalent, every layout of the vertices of $unplaced(G, L_1)$ completes L_1 and L_2 into total layouts of G with the same cutwidth. So, for each equivalence class, only one partial layout must be completed and memorized.

3 Algorithms

In this section, we present the two algorithms. Basically, they determine only if $mincut(G) \leq k$. To compute the mincut value, they must be used iteratively by either increasing or decreasing the k value. We choose the first solution to reuse the classes examined during the former iterations. To reduce the number of iterations needed, k is initialized with the maximum of the *Left* degree, of the *Right* degree and of the greater half-degree of the other vertices.

3.1 Original algorithm

Gurari and Sudborough propose a decision algorithm based on the former equivalence relation. The algorithm uses two data structures: a boolean array $tabClasses$ to mark the equivalence classes with $cutwidth \leq k$ and a stack Q to memorize one partial layout for each marked equivalence class. Q is initialized by L_0, the empty layout. The algorithm is as follows:

```
While $Q \neq \emptyset$ do
    Take a partial layout $L$ in $Q$;
    For each vertex $v$ in $unplaced(G, L)$ do
        $L' = L + v$;
        If $L'$ is a total layout then
            If $cutwidth(G, L') \leq k$ then
                return($G$ has a mincut $\leq k$, $L'$ is a linear ordering);
        If $L'$ is a partial layout then
            If $cutwidth(G, L') \leq k$ then
                If $tabClasses(L')$ unmarked then
                    Mark $tabClasses(L')$;
                    $Q = Q \cup L'$;
    EndFor
EndWhile
return($G$ has a mincut $> k$);
```

To find the exact $mincut$ value, this algorithm must be used iteratively by increasing a lower bound, or decreasing an upper bound.

3.2 Heuristic algorithm

To reduce the run time, it is necessary to limit the tree search by means of heuristic. To extend a partial layout L, the possible vertices are chosen among the vertices of $successor(G, L)$, as $card(successor(G, L)) \leq card(unplaced(G, L))$. This choice limits locally the cutwidth increase. The previous algorithm is simply modified by replacing the primitive instruction:

For each vertex x in $unplaced(L)$ **do** $\longrightarrow$ **For** each vertex x in $successor(L)$ **do**

This approach can be understood as the generalization of the greedy algorithm presented by Kang [7], where the vertex to extend a layout L is in $successor(G, L)$ and induces the minor increase of the graph cut. But the choice is definitive. Like partition-based heuristics [2], this search approach can be trapped into a local optimal solution, while ours does not, thanks to the backtracking.

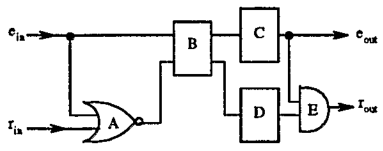
The present heuristic does not guarantee a better complexity in comparison to the Gurari and Sudborough algorithm. For instance, if G is a complete graph, the time complexity and the number of examined classes will be the same. However, as shown on practical examples (see next section), the number of examined classes and the run times are reduced using the heuristic approach.

3.3 Heuristic & Check algorithm

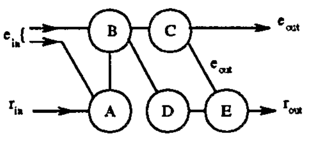
The heuristic actually gives an upper bound of the $mincut$ value even if, in many cases, this value is exact. We have investigated using the original algorithm to check and eventually decrease the value delivered by the heuristic algorithm. This approach is attractive. As the heuristic is faster than the original algorithm, and the heuristic $mincut$ value is exact in many cases, the original algorithm will be used only to check and not to decrease iteratively this $mincut$ value. Furthermore, the results show that the heuristic value is closer to the exact $mincut$ value than the lower bound used to initialize the iteration.

4 Implementation

When implementing these algorithms, we face two problems: the original data structures are not adapted in term of storage requirements; and the natural representation of a circuit is a hypergraph, while these algorithms work only on graphs.

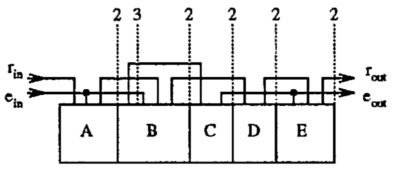


(a) Example of circuit.

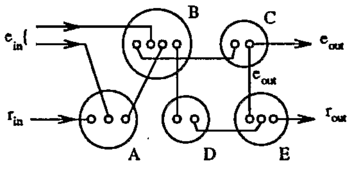


(b) Net representation.

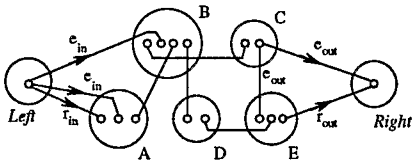
Figure 4: Net representation.



(a) Symbolic layout.



(b) Cell representation.



(c) Final graph.

Figure 5: Cell representation.

4.1 Cell representation

One problem in using the previous algorithm is the representation of a circuit with a graph. In the graph model, each cell is usually represented by one vertex and each net by one or several edges. But this model lacks accuracy for the mincut computation.

Net representation: A single net can connect P inputs and/or outputs. Such a net can be represented by $(P-1)$ edges with a common vertex. But they must be marked as a single multi-edge to be counted as 1 during the cut computation, as they will be laid out by a single routing wire. This representation is illustrated in figure 4.b.

Cell representation: The *mincut* computation just considers the edges between successive vertices. So, the symbolic layout (see figure 5.a), has a mincut of 2.

But on B itself, the cut is 3. In this example, the cell B input/output ordering increases the cut. This is a real problem as a cell has generally a single layout representation in the library with a fixed input/output ordering, for example $E1$ on the left, $E2$ in the middle, and Y on the right. However, this problem has to be considered on cells with 4 or more inputs/outputs only. With 2 or 3, a horizontal mirror solves this problem.

To compute the real cut, inputs/outputs are represented by interdependent vertices. Figure 5.b illustrates this circuit graph model. To place such a vertex, all the interdependent vertices will be successively placed according to the input/output ordering. If the new graph has additional vertices, the complexity does not increase as the placement is predefined, but the cut computation is more time-consuming.

Circuit cells have some structural properties that can be used to decrease the netcut on the cell itself. It is possible to examine the horizontal mirror of the cell (in fact the reverse input/output predefined ordering). In addition, some cell inputs are commutative, for example, "Nand" gate inputs. This property can be used to modify the input netlist which is, in some sense, equivalent to changing the input/output predefined ordering.

External signals: A circuit has external inputs/outputs which must be considered in the $mincut$ computation. In the case of $type$ 1 processor, two vertices, named *Left* and *Right*, are added and connected to the external signals (see figure 5.c). These *Left* and *Right* vertices will be initially placed on the leftmost and rightmost of the linear ordering. Then, the initial value of the stack Q is the *Left* vertex rather than the empty layout L_0, and every vertex of $unplaced(L)$ – *Right* can be chosen. Indeed, the *Right* vertex placed in the rightmost position does not modify the cutwidth.

For $type$ 2 or $type$ 3 processors (see figure 1.b and c), the vertical part of the connectors are to be ignored. So, the $mincut$ computation for $type$ 3 is similar to the $type$ 1 one. On the other hand, $type$ 2 have no *Right* vertex and morover, their *Left* vertex is not fixed: during the tree search, the different graph vertices will successively represent the *Left*.

Graph reduction: Makedon and Sudborough [9] have proposed removing every vertex with a degree of 2 and merging the edges. They prove that this new graph has the same mincut. Moreover, we propose to remove every edge (or multi-edge) between the *Left* and *Right* vertices. Such an edge, the clock for example, is a diffusion signal, and will be laid out on a single routing track.

Circuit	Cells	Mincut	tabClasses entries	Stocked classes	Max for a list
C432	150	41	6150	82131	21852
C208	103	12	1236	1855	151
C77	77	15	1155	89113	8140
C50	50	21	1050	49	1
C40	40	12	480	98771	10865
C29	29	8	232	31	2
C11	11	7	77	43	8
C8	8	4	32	7	1
C7	7	4	28	7	2

Table 1: *tabClasses* lists study.

4.2 Data structures

A direct implementation of Gurari and Sudborough algorithm is unrealistic. As there are $O(N^k)$ possible classes, the porposed *tabClasses* boolean array data structure would require too much storage and the computer used to execute this algorithm would spend its time "swapping" the memory. Instead, we use a bidimensional array $[1..N, 1..k]$ of list pointers. This array is indexed by $card(placed(G, L))$ and $card(dangling(G, L))]$. Each list contains all the examined equivalence classes with the same number of vertices and the same rightmost netcut. Moreover, to optimize the search of a class in a list, the lists are sorted in lexical order. Such structures are computationally practicable, however, the maximum number of elements in each list is in theory $O(N^k)$. Experience shows that the number of list elements is lower (see Table 1).

5 Results

In this section, we give two comparative tables: one on the run times of the algorithms, and one on the computed and memorized classes by the algorithms.

Circuit description: Different circuits are used to compare these algorithms: $C432$, $C208$, $C29$ from the ISCCAS85 benchmarks, $C77$ a counter, $C50$ an elliptical fifth order filter, $C40$ a clock controller, $C11$ a systolic correlator processor, $C8$ a systolic convolver processor, and $C7$ an example which shows the heuristic limitation. Table 2 gives the characteristics of these different circuits: number of cells, number of hyperedges, number of edges, maximum of the $Left$ degree, of

Circuit	Cells	Hyperedges	Edges	Degree
C432	150	186	333	36
C208	103	113	181	10
C77	77	90	157	12
C50	50	71	105	21
C40	40	44	86	10
C29	29	37	63	8
C11	11	16	23	5
C8	8	11	18	3
C7	7	12	12	3

Table 2: Characteristics of the circuit examples.

the $Right$ degree and of the greater half-degree of the other vertices. This last value is used to initialize k in the iteration.

Algorithm comparison: We compare the following algorithms:

- **G. & S.**, the original algorithm,
- **Heuristic**, the heuristic algorithm,
- **Heuristic & Check**, which combines the heuristic and original algorithms.

These different algorithms were coded in C. We used a SUN4/50 with 32 Mo RAM. Table 3 summarizes the run time, the $mincut$, and *Loop* the number of iterations for each algorithm. This table also gives the *cut* found with an implementation of Kang *"In-Act-Out"* algorithm [7].

The heuristic provides good results for both run time and $mincut$ value. For all the examples studied with the exception of the *counterexemple C7* built to emphasize the heuristic limitation, the same number of iterations is required by **G. & S.** and **Heuristic** algorithms.

However on some examples, the **Heuritic & Check** algorithm gives greater run times in comparison to the **G. & S.** algorithm. There is different reasons to explain these results. Firstly on the small examples, the results are not significative. Secondly, the **Check** step corresponds to the most time-consuming iteration of the **G. & S.** algorithm. Finally, the **Check** step does not used actually the classes computed during the **Heuristic** step. Using them, this step would be improved by a factor 2 (see the table 4 especially on $C432$ and $C40$).

	G. & S.			Heuristic			Heuristic & Check			Kang
Circuit	Time	Cut	Loop	Time	Cut	Loop	Time	Cut	Loop	Cut
C432	20258.6s	41	6	13978.5s	41	6	32631.44s	41	7	53
C208	128.3s	12	3	14.5s	12	3	19.04s	12	4	22
C77	7574.4s	15	7	1269.3s	15	7	3295.30s	15	8	23
C50	0.25s	21	1	0.25s	21	1	0.28s	21	2	24
C40	3730.2s	12	3	1682.21s	12	3	4908.52s	12	4	13
C29	0.17s	7	3	0.13s	7	3	0.14s	7	4	13
C11	0.16s	7	3	0.11s	7	3	0.20s	7	4	8
C8	0.05s	4	2	0.02s	4	2	0.04s	4	3	4
C7	0.04s	4	2	0.03s	5	3	0.04s	4	5	5

Table 3: Comparison of algorithms.

	G. & S.		Heuristic	
Circuit	Computed layouts	Memorized classes	Computed layouts	Memorized classes
C432	11957829	82131	5924480	74918
C208	120643	1855	14778	702
C77	3381844	89113	210884	13505
C50	75	49	166	49
C40	3203985	98771	1453474	65663
C29	243	31	120	29
C11	348	43	246	43
C8	13	7	15	7
C7	16	7	13	6

Table 4: Computed layouts and memorized classes.

Computed layouts and classes: The heuristic limits the number of examined layouts and of classes memorized in $tabClasses$. A limitation of memorized classes is important to avoid memory "swapping" and reduce run time. Table 4 summarizes the differences on the last iteration, between **Heuristic** and **G. & S.**

As shown in table 4, the heuristic reduces the number of computed layouts and memorized classes, especially on large circuits. On $C50$ and $C8$, there is no reduction. In fact on these examples, the vertex ordering is close to the optimal linear ordering. We investigated the use of initial vertex sorting (a topological sorting, result of Kang *InActiveOut* algorithm, ...) before processing, but without success in terms of reduced run times.

6 Conclusion

This paper has presented an approach for the linear layout of processors in regular arrays. In such structures, the processors are often replicated several hundred times. A few microns of optimization on a single processor will lead to a large optimization on the array. In a linear layout design style, the area optimization comes down to a net congestion minimization in the routing area. This problem is similar to the computation of the mincut of a graph or a hypergraph. As a processor is composed of a few cells (less than one hundred), even an algorithm with a high complexity can be implemented.

We investigate the decision algorithm proposed by Gurari and Sudborough, to find a processor linear placement respecting the exact *mincut* value of a graph. To efficiently implement this algorithm, we propose a new data structure to memorize the partial linear ordering, and an accurate graph model of a circuit which ensures the identity between the graph's *mincut* and the number of routing tracks needed for the layout.

As the iterative implementation of the decision algorithm is time-consuming, we investigate a heuristic based on the equivalence relation among the partial linear ordering, initially proposed by Gurari and Sudborough. This heuristic limits the possible vertices to extend a partial linear placement, to be one of the successors of the placed ones. While this new algorithm in theory gives an upper bound of the *mincut* value, in many cases, it gives the exact value but with better run times in comparison to the decision algorithm.

The heuristic does not guarantee an exact *mincut* value. We study a step to check and decrease the value determined by the heuristic, using the decision algorithm. In some cases, the **Heuristic + Check** times are greater than these found directly with the decision algorithm. This can be explained by the fact that we do not use the classes computed by the former **Heuristic** algorithm in the **Check** step. We think, using these classes would improve the **Check** step by a factor 2.

Actually, we get the exact *mincut* value with good run times on processors with less than one hundred cells. Such processor sizes are large enough for many regular array cases. On larger problems, these algorithms are actually too time-consuming or do not run to completion. We continue investigations to reduce the run times: improving the data structures, looking for other graph reductions based on structural properties of circuits.

References

[1] C. M. Fiduccia & R. M. Matheyses. A Linear-time Heuristic for Improving Network Partitions. Proc. 19th Design Automation Conference (1986), pp.622-629.

[2] T. Fuji, H. Horikawa, T. Kikuno & N. Yoshida. A Heuristic Algorithm For Gate Assignment in One-Dimensional Array Approach. IEEE Trans. on CAD, vol.CAD-6 n^o2 (March 1987), pp. 159-164.

[3] M. R. Garey & D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, San Francisco (1979).

[4] E. Gautrin & L. Perraudeau. Madmacs : A Tool for the Layout of Regular Arrays. WG 10.5 IFIP Workshop on Synthesis, Generation and Portability of Library Blocks for ASIC Design (1992), pp. 212-221.

[5] E. M. Gurari & I. H. Sudborough. Improved Dynamic Programming Algorithms for Bandwidth Minimization And The MinCut Linear Arrangement Problem. Journal of Algorithms, 5 (1984), pp. 531-546.

[6] A. G. Hoffman. Towards Optimizing Global Min Cut Partitionning. Proc. The European Conference on Design Automation (1991), pp. 1167-1171.

[7] S. Kang. Linear ordering and application to placement. Proc. 20th Design Automation Conference (1983), pp. 457-464.

[8] H.T. Kung. Why systolic architectures? *IEEE Computer*, Vol 15, pp. 37, 1982.

[9] F. S. Makedon & I. H. Sudborough. Minimizing Width in Linear Layouts. Proc. of International Conference on Automata, Languages, and Programming(ICALP), Lecture Notes in Computer Science vol.154, Springer-Verlag (1983), pp. 478-490.

[10] Z. Miller & I. H. Sudborough Polynomial Algorithm for Recognizing bounded Cutwidth in hypergraphs. Math. Systems Theory 24, Springer-Verlag (1991), pp. 11-40.

[11] P. Ramyalal Suaris & G. Kedem. A Quadrisection-Based Combined Place and Route Scheme for Standard Cells. IEEE Transactions on Computer-Aided Design, vol.8, n^o.3 (March 1989).

[12] H. E. Shrobe. The Datapath Generator. CompCon82 High Technology in the Information Industry, pp. 340-344, IEEE Computer Society (1982).

[13] A. Weinberger. Large Scale Integration of MOS Complex Logic : A Layout Method. IEEE J. Solid State Circuits SC-2 (1967), pp. 182-190.

[14] O. Wing, S. Huang & R. Wang. Gate Matrix Layout. IEEE Trans. on CAD, vol.CAD-4 n^o3 (July 1985), pp. 220-231.