Slicing Floorplans with Boundary Constraints

F. Y. Young, D. F. Wong, and Hannah H. Yang

***Abstract*—In floorplanning of very large scale integration design, it is useful if users are allowed to specify some placement constraints in the packing. One particular kind of placement constraints is to pack some modules on one of the four sides: on the left, on the right, at the bottom, or at the top of the final floorplan. These are called boundary constraints. In this paper, we enhanced a well-known slicing floorplan algorithm [10] to handle these boundary constraints. Our main contribution is a necessary and sufficient characterization of the Polish expression, a representation of the intermediate solutions in the simulated annealing process, so that we can check these constraints efficiently and can fix the expression in case the constraints are violated. We tested our algorithm on some benchmark data and the performance is good.**

***Index Terms*—Floorplanning, placement constraints, simulated annealing, slicing.**

I. Introduction

Floorplan design is an important step in physical design of very large scale integration circuits. It is the problem of placing a set of circuit modules on a chip to minimize total area and interconnection cost. In this early stage of physical design, most of the modules are not yet designed and thus are flexible in shape (soft modules) and are free to move (free modules).

Many existing floorplanners are based on slicing floorplans [1], [10], [2], [6], [9] and it has been shown theoretically that slicing floorplans can pack modules tightly [11]. There are several advantages of using slicing floorplans. First, focusing only on slicing floorplans significantly reduces the search space and this leads to fast runtime. Second, the shape flexibility of the soft modules can be fully exploited to pack modules tightly using an efficient shape curve computation technique [8], [7]. As a result, existing floorplanners that use slicing floorplans are usually very efficient in runtime and yet can pack modules tightly.

Recently, there are some interesting research activities in the direction of nonslicing floorplans. Two methods, bound-sliceline-grid (BSG) [5] and sequence-pair (SP) [3], are proposed. These methods are originally designed for placement of modules which have no flexibility in shape (hard modules). The sequence-pair method is recently extended to handle soft modules [4]. In order to handle soft modules, it needs to solve an expensive convex programming problem to determine the exact shape of each soft module numerous times, and this results in long runtime. Note that for the same set of benchmark data (apte, xerox, hp, ami33, and ami49) in [4], we run the slicing floorplan algorithm in [10] and can obtain comparable results using only a fraction of the runtime. In fact, we have less than 1% dead space using no more than 7 s for all the test problems.

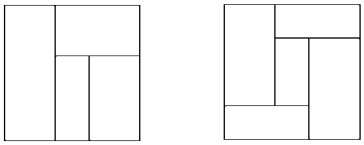
In floorplanning, it is useful if users are allowed to specify some placement constraints in the final packing. We did some previous work on floorplanning with preplaced modules [12]. A preplaced module is fixed in position, height and width. We solved this problem by a novel shape curve computation procedure which takes the positions of the preplaced modules into consideration.

Manuscript received November 17, 1998; revised February 5, 1999. This work was supported in part by the Texas Advanced Research Program and in part by a grant from the Intel Corporation. This paper was recommended by Associate Editor C.-K. Cheng.

F. Y. Young and D. F. Wong are with the Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712-1188 USA.

H. H. Yang is with the Intel Corporation, Hillsboro, OR 97124-5961 USA.

Publisher Item Identifier S 0278-0070(99)06618-X.



(a) (b)

Fig. 1. A (a) slicing floorplan and (b) nonslicing floorplan.

The placement constraint we consider here is called boundary constraint: some modules are constrained to be packed on one of the four sides: on the left, on the right, at the bottom, or at the top of the final floorplan. This is useful because designers may want to place some modules along the boundary for input-output connections. Besides, floorplanning is usually done hierarchically in which modules are grouped into different units and floorplanning is done independently for each unit on the chip. It will help if some modules are constrained to be packed along the boundary of the unit so that they can abut with some other modules in the neighboring units. We extend a well-known slicing floorplan algorithm by Wong and Liu [10] to handle these constraints. Our main contribution is a necessary and sufficient characterization of the Polish expression, a representation of the intermediate solutions in the simulated annealing process, so that we can check these boundary constraints efficiently and can fix the expression in case the constraints are violated. We tested our algorithm with some benchmark data and the performance is good.

The rest of the paper is organized as follows. We first define the problem formally in Section II. Section III provides a brief review of the Wong–Liu algorithm. The new work is presented in Section IV and the experimental results are shown in Section V.

II. Problem Definition

A module A is a rectangle of height $h(A)$, width $w(A)$, and area $area(A)$. The aspect ratio of A is defined as $h(A)/w(A)$. A soft module is a module whose shape can be changed as long as the aspect ratio is within a given range and the area is as given. A floorplan for n modules consists of an enveloping rectangle R subdivided by horizontal lines and vertical lines into n nonoverlapping rectangles such that each rectangle must be large enough to accommodate the module assigned to it. There are two kinds of floorplans: slicing and nonslicing (Fig. 1). A slicing floorplan is a floorplan which can be obtained by recursively cutting a rectangle into two parts by either a vertical line or a horizontal line. A nonslicing floorplan is a floorplan which is not slicing.

In our problem, we are given two kinds of soft modules $M = F \cup B$. The modules in F are free to move while the modules in B are constrained to be packed on one of the four sides of the final floorplan. A feasible packing is a packing in the first quadrant such that the width and height of all the modules are consistent with their aspect ratio constraints and their area constraints, and all the modules in B are placed on the boundaries as required (Fig. 2). Our objective is to construct a feasible floorplan R to minimize $A + \lambda W$ where A is the total area of the floorplan R, W is an estimation of the interconnect cost and λ is a constant that controls the relative importance of A and W. We require that the aspect ratio of the final packing is between two given numbers $r_{\min}$ and $r_{\max}$.

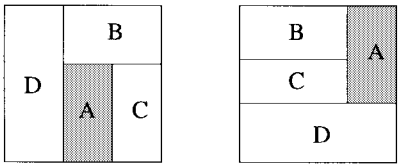


Fig. 2. An example of a feasible floorplan. Suppose module A is constrained to be packed along the right boundary. Then the packing in (a) is infeasible but the packing in (b) is feasible.

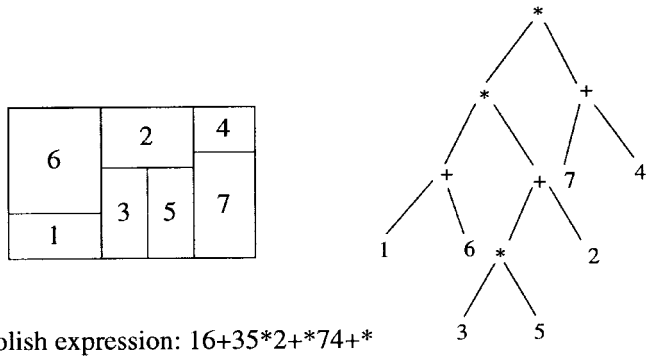


Fig. 3. Slicing tree representation and Polish expression representation of a slicing floorplan.

III. Wong–Liu Algorithm

A slicing floorplan can be represented by an oriented rooted binary tree, called a slicing tree (Fig. 3). Each internal node of the tree is labeled by a * or a $+$, corresponding to a vertical or a horizontal cut respectively. Each leaf corresponds to a basic module and is labeled by a number from 1 to n. No dimensional information on the position of each cut is specified in the slicing tree. If we traverse a slicing tree in postorder, we obtain a Polish expression and a Polish expression is said to be *normalized* if there is no consecutive *'s nor consecutive $+$'s in the sequence. It is proved in [10] that there is a one-to-one correspondence between the set of normalized Polish expressions of length $2n-1$ and the set of slicing floorplans with n modules.

In [10], Wong and Liu used the set of all normalized Polish expressions as the solution space for the simulated annealing method. In order to search the solution space efficiently, they defined three types of moves (M1, M2, and M3) to transform a Polish expression into another. They can make use of the flexibility of the soft modules to select the "best" floorplan among all the equivalent ones represented by the same Polish expression. This is done by carrying out an efficient shape curve computation [7], [10] whenever a Polish expression is examined. The cost function is $A+\lambda W$ where A is the total packing area and W is the interconnect cost. This algorithm is very efficient and the performance is very good.

However, their method does not consider any placement constraint and there is actually a simple and natural way to handle boundary constraint in the Polish expression representation. We will describe it in Section IV.

IV. Our Method

In the simulated annealing process, we check the normalized Polish expression in each iteration to see whether the boundary constraints are satisfied. This can be done efficiently in linear time by scanning the expression once. Then we fix the violated constraints as much as possible, and include in the cost a boundary constraint term to penalize the remaining violations.

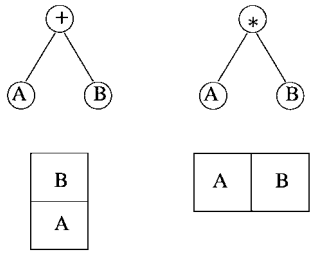


Fig. 4. Relative positions of modules denoted by slicing trees.

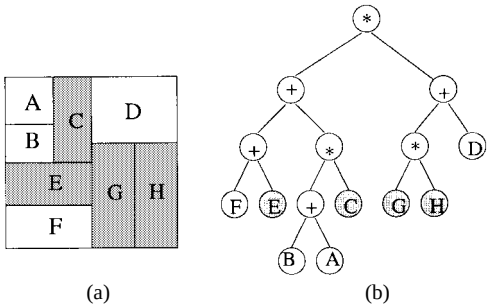


Fig. 5. Characterization of slicing trees for different boundary constraints. Module E is on the left boundary of R, so it must be in the left subtree of any internal node in T labeled "*," module H is on the right boundary of R, so it must be in the right subtree of any internal node in T labeled "*," module C is on the upper boundary of R, so it must be in the right subtree of any internal node in T labeled "+," module G is on the lower boundary of R, so it must be in the left subtree of any internal node in T labeled "+."

A. Checking the Boundary Constraints

The slicing trees and Polish expressions have orientation. In Fig. 4, the slicing tree on the left corresponds to a Polish expression $AB+$, which means that module A is below module B. The slicing tree on the right corresponds to the expression $AB*$, which means that module A is on the left of module B. Therefore, if we want to pack a module A on the right (left) boundary of the final floorplan, the slicing tree T should be such that A is not in the left (right) subtree of any internal node of T labeled $*$. Similarly, if we want to put a module A at the top (bottom) of the floorplan, the slicing tree T should be such that A is not in the left (right) subtree of any internal node of T labeled $+$. An example is shown in Fig. 5. Lemma 1 summarizes the above observations.

Lemma 1: Given a slicing tree T, a module in T is on the right boundary of the floorplan R corresponding to T if and only if it is not in the left subtree of any internal node in T labeled $*$. A module is on the left boundary of R if and only if it is not in the right subtree of any internal node in T labeled $*$. A module is on the upper boundary of R if and only if it is not in the left subtree of any internal node in T labeled $+$. A module is on the lower boundary of R if and only if it is not in the right subtree of any internal node in T labeled $+$.

In the annealing process, we use Polish expressions to represent the slicing trees. It will be inefficient if we build a slicing tree in each iteration to check the conditions in Lemma 1. Actually we can check the necessary and sufficient conditions in Lemma 1 efficiently by scanning the Polish expression once. This is done by keeping a stack when scanning the expression from right to left. Each stack element x has four bits: $x.left$, $x.right$, $x.top$, and $x.bottom$. We push an element onto the stack whenever we see an operator α in the expression. This stack element represents the subfloorplan X represented by the children subtrees of α in the slicing tree T. The four bits indicate whether there are modules above X, below X, on the right of X and on the left of X, e.g., $x.left = 1$ if and only if there is at least one module on the left of X in the floorplan.

We scan the Polish expression from right to left. When we scan a $+$, we push a new element x onto the stack. The four bits of x are copied from the previous stack top element, except that $x.bottom$ is assigned to 1. Similarly, we push a new element onto the stack whenever we scan a $*$ but now we assign $x.left$ to 1 and copy the other three bits from the previous stack top element. The invariant is that whenever we scan a module A in the expression, the four bits at the top of the stack will indicate whether there are modules above A, below A, on the right of A and on the left of A, and we can copy these information to $A.above$, $A.below$, $A.right$, and $A.left$. These four bits, when attached to a module name, indicate whether there are modules lying above, below, on the right and on the left of that module in the final floorplan. Finally, we can check the boundary constraints with these information, e.g., a module A constrained to be placed at the top of the floorplan should have $A.top = 0$. Since we scan the Polish expression from right to left, we are doing a reversed postorder traversal in the slicing tree. Each element x in the stack represents an internal node v in the tree, and the flag $x.flag$ tells whether we have backtracked from the right subtree of v in the traversal. $x.flag = 0$ if we are still in the right subtree of v, and $x.flag = 1$ if we are in the left subtree of v.

Algorithm Check-Boundary-Constraints
Input: A Polish expression $\alpha = \alpha_1\alpha_2\cdots\alpha_{2n-1}$
Output: For each module A, decide whether there are modules lying above A, below A, on the right of A and on the left of A in the final floorplan.

1. *top* $= 0$. Assign 0 to all four bits of *stack*[*top*]
2. For $i = 2n-1$ downto 1:
3. If α_i is a $*$ operator:
4. Push a new element x onto the stack
5. $x.left = 1$
6. Copy $x.right, x.above$, and $x.below$ from $stack[top-1]$
7. $x.flag = 0$; $x.op = *$
8. If α_i is a $+$ operator:
9. Push a new element x onto the stack
10. $x.below = 1$
11. Copy $x.left, x.right$, and $x.above$ from $stack[top-1]$
12. $x.flag = 0$; $x.op = +$
13. If α_i is a module name:
14. Copy the four bits from $stack[top]$ to α_i
15. While $stack[top].flag = 1$ and $top > 0$
16. Pop stack
17. If $top > 0$:
18. $stack[top].flag = 1$
19. If $(stack[top].op = *)$
20. $stack[top].right = 1$
21. $stack[top].left = stack[top-1].left$
22. If $stack[top].op = +$
23. $stack[top].above = 1$
24. $stack[top].below = stack[top-1].below$

Proof of correctness: Consider any subtree t in the slicing tree T, we want to prove by induction that if the stack top element shows correctly the boundary conditions for the subfloorplan represented by t when we first visit the root of t, the algorithm will assign correct boundary conditions to all the basic modules in t. Notice that if this statement is true, we can put t as the whole slicing tree T. Since all four bits of the stack top element are initialized to zero at the beginning, which are the correct boundary conditions for the entire floorplan, we can conclude that the algorithm will assign correct boundary conditions to all the basic modules in T.

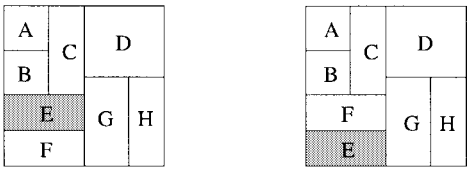


Fig. 6. An example of fixing a Polish expression.

We prove by induction on the depth of the subtree t. Lets consider the base case when the depth of t is one, i.e., t consists of a single basic module only. We assume that the stack top element shows correctly the boundary conditions for the subfloorplan represented by t when we first visit the root of t, this implies trivially that the stack top element shows correctly the boundary conditions for this basic module when we visit it. Thus the algorithm will assign correct boundary conditions to this module and the statement is true.

Now, we assume that the statement is true for any subtree t which has a depth less than or equal to k where $k \geq 1$. Lets consider a subtree t of depth $k+1$. We assume that the stack top element shows correctly the boundary conditions for the subfloorplan represented by t when we first visit the root of t. We consider two different cases. In the first case, we assume that the root of t is a $*$ operator. Let t_1 and t_2 be the right and left subtree at the root of t, respectively. The next step after visiting the root of t is to visit t_1 (a reversed postorder traversal). The boundary conditions for the subfloorplan R_{t_1} represented by t_1 is the same as that of t except that we are sure there must be at least one module on the left of R_{t_1}, since t_2 is on the left of t_1. In the algorithm, we push a new element x onto the stack before visiting t_1, and copy the bits from the previous stack top element to this new element except that we put $x.left = 1$. This stack top element thus shows correctly the boundary conditions for R_{t_1} when we start to visit t_1. According to the inductive hypothesis, the algorithm will assign correct boundary conditions to all the basic modules in t_1 since the depth of t_1 is less than or equal to k. When we backtrack from t_1, we modify the bits of x before visiting t_2. We copy $x.left$ from the element below and put $x.right = 1$ because t_1 is on the right of t_2 and there is at least one module on the right of the subfloorplan R_{t_2} represented by t_2. This stack top element will thus show correctly the boundary conditions for R_{t_2} when we start to visit t_2. According to the inductive hypothesis, the algorithm will assign correct boundary conditions to the basic modules in t_2 since the depth of t_2 is less than or equal to k. In the second case, we assume that the root of t is a $+$ operator. We can argue similarly as above to show that the algorithm will assign correct boundary conditions to all the basic modules in t.

Therefore by induction, the statement is true and we can conclude that the algorithm will assign correct boundary conditions to all the basic modules in T.

2) Analysis: Let n be the number of modules. The length of the Polish expression will be $2n-1$: n modules and $n-1$ operators. In the algorithm, we will scan the Polish expression once from right to left. When we see an operator, we will push an element onto the stack and set the flag of the element to zero. When we see a module name, we will pop the stack until we see an element of flag 0, then we will reset the flag to one. We do a constant amount of work

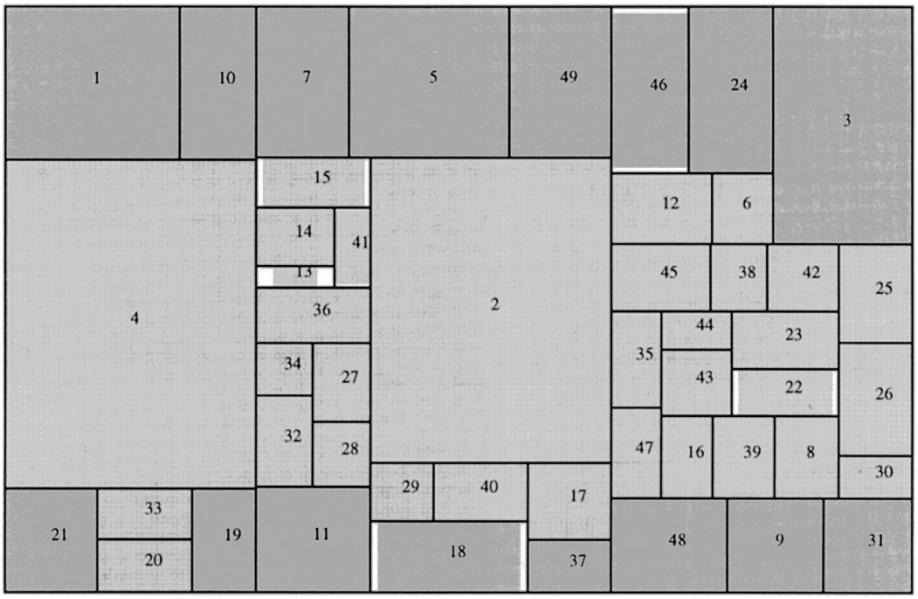


Fig. 7. Another result packing of ami49. Modules 1, 3, 5, 7, 10, 24, 46, and 49 are constrained to the upper boundary. Modules 9, 11, 18, 19, 21, 31, 37, and 48 are constrained to the lower boundary.

for each module name and operator in the Polish expression and the complexity of the algorithm is thus $O(n)$.

B. Fixing a Polish Expression

If a Polish expression does not satisfy the boundary constraints, we can fix it as much as possible by shuffling the modules. An example is shown in Fig. 6. In the figure, boundary constraint is violated in Fig. 6(a) since module E is not packed at the bottom, as required. To fix this, we exchange E with F where F is the module closest to E in the Polish expression and that F is packed on the lower boundary. In general, if a module A is not packed along the boundary as required, we will shuffle it with another module B which is closest to A in the Polish expression and that B's position satisfies the boundary constraint of A. The complexity of this procedure is $O(mn)$ where m is the number of constrained modules and n is the total number of modules. In the worst case, for each constrained module A, we need to scan all the other modules to find the closest one which satisfies the constraint of A. However, if we start the scanning process from the position of A, we usually do not need to scan all the modules to find the closest one. Besides the number of modules which violate their constraints decreases rapidly during the annealing process. Therefore, the time taken is actually much faster in practice.

It is possible that some constraints are still violated after all the possible shufflings, since a Polish expression may correspond to a floorplan which does not have enough positions along the boundaries to satisfy all the required constraints. We include a boundary constraint term in the cost function to penalize the remaining violated constraints. All violations will be eliminated as the annealing process proceeds because of this boundary constraint penalty term.

C. Cost Function

The cost function is defined as $A + \lambda W + XD$ where A is the total area of the packing obtained from the shape curve at the root of the slicing tree, W is the half-perimeter estimation of the interconnect cost, and D is the penalty term for the boundary constraint. The penalty term D is the total distance of the modules from the boundaries of the floorplan along which they should be packed. For instance, if module A is constrained to be packed on the right, the penalty term for A will be the distance between the right side of A and the right boundary of the final floorplan. The penalty terms are similarly defined for modules constrained to be packed on the left, at the top and at the bottom. λ and X are constants which control the relative importance of the three terms. λ is usually set such that the area term and the interconnect term are approximately balanced. The boundary constraint terms D will drop to zero as the process proceeds.

TABLE I
RESULTS OF TESTINGS WITH MCNC EXAMPLES

Data	Dead space (%)	Time (sec)
ami33-bc1	1.81	4.58
ami33-bc2	1.33	4.53
ami33-bc3	1.62	4.41
ami33-bc4	1.86	4.28
ami33-bc5	1.51	4.01
ami49-bc1	1.51	37.77
ami49-bc2	3.17	36.98
ami49-bc3	4.65	39.76
ami49-bc4	3.48	36.03
ami49-bc5	4.25	37.60
playout-bc1	3.36	41.70
playout-bc2	2.46	42.83
playout-bc3	2.51	41.74
playout-bc4	2.02	41.72
playout-bc5	5.20	42.06

V. EXPERIMENTAL RESULTS

We tested the above method on three MCNC building blocks examples: ami33, ami49, and playout. ami33 has 33 modules and 123 nets. ami49 has 49 modules and 408 nets. playout has 62 modules and

1611 nets. We pick 12 modules from ami33, 16 modules from ami49 and 20 modules from playout, and require them to be packed along the boundaries evenly. We tested the floorplanner with 15 data sets which are derived from the MCNC examples by imposing different boundary constraints on the selected module. The starting temperature is decided such that an accepting ratio is 100% at the beginning. The temperature is lowered at a constant rate (0.9), and the number of iterations at one temperature step is twenty times the number of modules. All the experiments were carried out on a 300-MHz Pentium II Intel processor.

Table I shows the experimental results. All the boundary constraints are satisfied in each data set. Both the packing quality and efficiency are satisfactory. Fig. 7 is a result packing of ami49 in which we require modules 1, 3, 5, 7, 10, 24, 46, and 49 to be packed at the top and modules 9, 11, 18, 19, 21, 31, 37, and 48 at the bottom. The packing is very tight and all the boundary constraints are satisfied.

REFERENCES

[1] K. Bazargan, S. Kim, and M. Sarrafzadeh," Nostradamus: A floorplanner of uncertain design," in *Proc. Int. Symp. Physical Design*, 1998, pp. 18–23.

[2] D. P. Lapotin and S. W. Director, "Mason: A global floorplanning tool," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 1985, pp. 143–145.

[3] H. Murata, K. Fujiyoushi, S. Nakatake, and Y. Kajitani, "Rectangle-packing-based module placement," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 1995, pp. 472–479.

[4] H. Murata and E. S. Kuh, "Sequence-pair based placement method for hard/soft/preplaced modules," in *Proc. Int. Symp. Physical Design*, 1998, pp. 167–172.

[5] S. Nakatake, K. Fujiyoushi, H. Murata, and Y. Kajitani, "Module placement on BSG-structure and IC layout applications," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 1996, pp. 484–491.

[6] R. H. J. M. Otten, "Automatic floorplan design," in *Proc. 19th ACM/IEEE Design Automation Conf.*, 1982, pp. 261–267.

[7] ——, "Otten efficient floorplan optimization," in *IEEE Int. Conf. Computer Design*, 1983, pp. 499–502.

[8] L. Stockmeyer, "Optimal orientations of cells in slicing floorplan designs," *Inform. Contr.*, vol. 59, pp. 91–101, 1983.

[9] T. Tamanouchi, K. Tamakashi, and T. Kambe, "Hybrid floorplanning based on partial clustering and module restructuring," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 1996, pp. 478–483.

[10] D. F. Wong and C. L. Liu, "A new algorithm for floorplan design," in *Proc. 23rd ACM/IEEE Design Automation Conf.*, 1986, pp. 101–107.

[11] F. Y. Young and D. F. Wong, "How good are slicing floorplans?," *Integration, the VLSI J.*, vol. 23, pp. 61–73, 1997.

[12] ——, "Slicing floorplans with preplaced modules," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 1998, pp. 252–258.

Multilevel Spectral Hypergraph Partitioning with Arbitrary Vertex Sizes

J. Y. Zien, M. D. F. Schlag, and P. K. Chan

***Abstract*—This paper presents a new spectral partitioning formulation which directly incorporates vertex size information by modifying the Laplacian of the graph. Modifying the Laplacian produces a generalized eigenvalue problem, which is reduced to the standard eigenvalue problem. Experiments show that the scaled ratio-cut costs of results on benchmarks with arbitrary vertex sizes improve by 22% when the eigenvectors of the Laplacian in the spectral partitioner KP are replaced by the eigenvectors of our modified Laplacian. The inability to handle vertex sizes in the spectral partitioning formulation has been a limitation in applying spectral partitioning in a multilevel setting. We investigate whether our new formulation effectively removes this limitation by combining it with a simple multilevel bottom-up clustering algorithm and an iterative improvement algorithm for partition refinement. Experiments show that in a multilevel setting where the spectral partitioner KP provides the initial partitions of the most contracted graph, using the modified Laplacian in place of the standard Laplacian is more efficient and more effective in the partitioning of graphs with arbitrary-size and unit-size vertices; average improvements of 17% and 18% are observed for graphs with arbitrary-size and unit-size vertices, respectively. Comparisons with other ratio-cut based partitioners on hypergraphs with unit-size as well as arbitrary-size vertices, show that the multilevel spectral partitioner produces either better results or almost identical results more efficiently.**

***Index Terms*—Eigenvalues, multilevel, multiway partitioning, partitioning, ratio-cut metric, spectral method.**

I. INTRODUCTION

Hypergraph partitioning is an important problem with a variety of diverse and practical applications, including circuit partitioning, network performance analysis, database storage optimization, and parallel processing. Two general classes of algorithms have emerged in the research community: spectral algorithms, and iterative refinement algorithms. A spectral partitioning algorithm uses the eigenvectors of a graph for generating partitions of a graph. The eigenvectors provide an optimal solution to a relaxed version of the ratio-cut partitioning problem. In ratio-cut partitioning, no constraints are placed on the partition sizes; instead the objective function favors more balanced partitions [1], [2]. Ratio-cut partitioning excels in situations where finding natural clusters is preferable to imposing constraints [3]. Iterative refinement algorithms are based on defining a cost function and then performing a sequence of moves to reach a locally optimal solution. Combining the two approaches, using a spectral algorithm to find initial partitions, and then performing iterative refinement, provides a deterministic algorithm resulting in solutions whose ratio-cut costs are better than those produced by either method alone.

Except for the work of Hendrickson and Leland [4] on load balancing for hypercube multiprocessors, previous spectral algorithms for partitioning graphs and hypergraphs have been limited by the

Manuscript received September 3, 1998; revised January 28, 1999. This work was supported by the National Science Foundation (NSF) under Grant MIP-9223740. This paper was recommended by Associate Editor R. Gupta.

J. Y. Zien was with the Computer Engineering Department, University of California, Santa Cruz, CA 95064 USA. He is now with the IBM Almaden Research Center, San Jose, CA 95120 USA.

M. D. F. Schlag and P. K. Chan are with the Computer Engineering Department, University of California, Santa Cruz, CA 95064 USA.

Publisher Item Identifier S 0278-0070(99)06628-2.