VLSI Placement using Uncertain Costs

Cheryl L. Harkness
Daniel P. Lopresti

Brown University
Dept. of Computer Science
Providence, Rhode Island 02912

Technical Report No. CS-90-12
May 1990

VLSI Placement using Uncertain Costs

Cheryl L. Harkness

Daniel P. Lopresti †

Department of Computer Science

Brown University

Providence, RI 02912

Abstract

Many objective functions used to evaluate placement quality contain uncertain parameters. While these values (e.g., channel width, cell area, wire length, and circuit delay) can be estimated, they cannot be precisely known until the layout is finished or sometimes until the chip is fabricated. As a result, current automatic placement algorithms must use "expected" values in their objective functions providing one possible estimate of placement quality. An algorithm that uses the full range of potential values when computing placement cost can yield a more credible prediction of placement quality and reveal more about the structure of optimal configurations. In this paper, we propose an interval-based approach to modeling uncertainty in automatic placement algorithms. We illustrate our methodology by implementing an interval branch and bound placement algorithm. Using this example, we demonstrate how interval operators can be used to compute bounds on placement cost and to determine minimum-cost configurations. We also test the performance of our algorithm and analyze the results.

†Daniel Lopresti is supported in part by the Defense Advanced Research Projects Agency under ONR contract number N00014-83-K-0146.

1 Introduction

Rapid increases in circuit size and complexity in the past ten years have made automatic placement an essential element of the VLSI design tool suite. The goal of automatic placement is to find an arrangement of the circuit components that facilitates routing and optimizes certain aspects of the circuit. These goals are encoded within an *objective function*, which provides a measure of the quality of each configuration. [Prea86] presents an overview of the most common automatic placement techniques. These can be divided into two major categories: constructive and iterative. The constructive approaches begin with a partial placement and build a complete placement from it. Typical examples of these algorithms include cluster growth [Dai89], min-cut partitioning [Breu77, Fidu82, Kern70, MS89, Tric86], global placement [Jack86], and branch and bound [Dai89, Prea79, Tric86]. Iterative techniques, such as pairwise interchange and simulated annealing [Kirk83], start with a complete placement and attempt to improve it. In spite of their differences, all placement algorithms share a number of common features. They all evaluate an objective function for each placement generated, compare the values returned by the objective functions to chose the best configuration of elements, and save the best result found.

While there are almost as many different objective functions as placement programs, the most common measures of placement quality are circuit area, channel density, maximum wire length, and circuit delay. Many of these objective functions contain parameters whose exact values cannot be predicted in advance, such as cell area, routing area, wire length, and delay. In top-down or hierarchical placement, the dimensions of higher-level cells are determined by placements at lower-levels in the hierarchy. If the lower-level cells are not completely placed and routed, the exact dimensions of the higher-level cells containing them may not be known. Similarly, channel width and wire length can only be accurately determined by doing a complete routing. This calculation is computationally expensive and thus is practical only for the final placement configuration. While exploring placement alternatives, the routing area and wire length are usually estimated. The circuit delays used in timing-driven placement are also subject to variation. While these uncertain parameters can be estimated, they cannot be precisely known before the layout is finished or the chip is fabricated.

Despite the prevalence of uncertainty in placement metrics, current placement programs employ only "expected" values in their objective functions, returning one possible cost for each configuration. Using these values, the algorithm returns the configuration whose cost in minimum. Instead of ignoring parameter variation, systems incorporating uncertainty use the full range of possible values for each variable to compute bounds on the cost of a configuration. Using these cost ranges, it returns a *set* of placements, each of which is optimal for some combination of input values.

Placement programs that account for uncertainty offer several advantages over current systems. One advantage is a more comprehensive measure of placement quality. Since the parameters used in objective functions have variable values, different combinations of their values will result in different placement costs and possibly a different "best" configuration. Algorithms incorporating uncertainty return a set of placements, each of which is optimal for some assignment of values. As the placement process continues, the values of these parameters may be refined, allowing the designer to chose one placement from the set of potentially optimal placements. Since it is unlikely that the final values of each parameter will exactly match the precomputed single-valued estimates, the final layout may be quite different from the one generated using a conventional placement algorithm.

Additional information about the best layout is the second advantage to including uncertainty. Systems that return a single configuration give no details about the features of the layout that make it optimal. By comparing the layouts from a set of best configurations, we begin to discover the similarities between them. These similarities reveal the factors that produced the best layouts for a particular circuit. For instance, certain components may appear in the same location in all potentially optimal placements. Also, strongly connected components may always appear in the same relative position. In addition to increasing our understanding of the design, these invariants may also indicate critical areas on the chip, which can be changed to further improve the placement.

A third advantage of incorporating uncertainty is that it facilitates iterative improvement without costly recomputation. As placement proceeds, the values of some uncertain parameters may be refined or may change. For instance, in hierarchical placement, the dimensions of the blocks at higher-levels will be refined as the layouts are completed at the lower-levels of the hierarchy. Similarly, channel width and wire length estimates are updated after full routing. Using current placement algorithms, the designer must re-execute the complete algorithm each time these values change. Since finding a good placement is a computationally expensive process, this is not practical. Using an algorithm that models uncertainty, we may not have to completely recompute the layout after these changes. As long as the updated values or ranges lie within the ranges initially given for each parameter, the previous placement results are still valid. Instead of running the placement algorithm again, we simply recompute the cost of the placements in the optimal set and discard those that no longer belong.

For these reasons, we propose a framework for placement using uncertain objective functions. We represent placement costs as intervals and create an algebra for manipulating these uncertain values. We then incorporate this algebra within an existing placement algorithm [Prea79] to create one that handles uncertain objective functions. Although a branch and bound placement algorithm was chosen, the general principles employed apply to other placement strategies. In particular, we address

the common issues of evaluating an uncertain objective function, comparing two placements with uncertain costs to find the best configurations, and storing the resulting minimum-cost placements.

2 A Branch and Bound Placement Algorithm

In [Prea79], Preas and vanCleemput present a branch and bound placement algorithm for arbitrarily-sized blocks. Because of its exponential computational complexity, the branch and bound technique is only practical for circuits with relatively few components (i.e., fewer than 20). Hence the placement algorithm begins by partitioning the circuit to create several smaller layout problems of a suitable size. The result of top-down partitioning is a hierarchical circuit description, where each level contains a collection of rectangular circuit components and the interconnections between them. Beginning at the bottom levels of the hierarchy, a constructive branch and bound algorithm is applied recursively to compute an initial placement. Because of the size of the search space for some larger circuits, the constructive algorithm may require an unreasonably long time to find the optimal solution. For these cases, Preas and vanCleemput propose an iterative improvement scheme based on the same branch and bound strategy.

2.1 The Objective Function

The goal of this placement algorithm is to minimize circuit area; thus, we must estimate the size of each trial placement. Preas and vanCleemput begin by creating the *horizontal* and *vertical channel position graphs* that define the layout. A channel position graph is a directed acyclic graph whose nodes represent the boundaries between circuit components and channels and whose edges represent the channels and circuit blocks themselves. There are two special nodes in each graph depicting the right and left (top and bottom) boundaries of the layout. Associated with each edge is a *length*, representing the physical dimensions of the corresponding block or channel. Once the channel position graphs for a particular placement are created, we traverse the graphs to find the longest paths between the boundary nodes. The length of this longest path gives the minimum length (or width) of the layout. These dimensions are then multiplied to determine circuit area.

2.2 Branch and Bound Placement

Given a set of circuit blocks and their interconnections, Preas and vanCleemput employ branch and bound techniques to find the configuration with minimum area. Branch and bound is a well-known potentially-exhaustive search strategy [Aho83, pp. 330-336] that traverses a search tree to locate the minimum cost solution. For the placement problem, the internal nodes of the search tree represent

partial placements and the leaf nodes represent complete placements. The root of the tree contains a *seed* placement, consisting of one or more blocks and their positions relative to each other. By adding an unplaced block to a partial placement in all possible positions, we generate its child configurations. At each successive level of the tree, unplaced blocks are added, until each leaf node contains a different complete placement. Associated with each node in the search tree is a number, indicating its cost. In this case, the cost of each (partial or complete) placement is its area. When visiting a node, we compare its cost to that of the current best complete configuration. If the cost of the node is greater than or equal to the current minimum cost, we prune that solution from the search tree; otherwise, we consider each of its children and compute their costs. At the leaf nodes, we locate and save the one with the minimum area. When all the nodes have been either visited or pruned, the search is complete and the best solution is returned. This branch and bound placement algorithm is summarized in Figure 1.

Figure 2 presents the results of a layout computation on a circuit with ten cell blocks[1]. The average dimensions (in mils) of each block are displayed in Table 1. After generating and examining more than 180,000 partial and complete configurations, the algorithm returned the minimum-cost placement displayed in Figure 2. The resulting configuration has an area of 1998 square mils. Below the layout, the horizontal and vertical channel position graphs defining this optimal placement are displayed. The graph edges corresponding to circuit components are labeled with block identifiers and edges corresponding to channels are labeled with the letter H (horizontal channels) or V (vertical channels). Each graph edge is also labeled with the estimated length or width of its associated block or channel in parentheses. The longest paths through the horizontal and vertical channel position graphs are 37 and 54 mils respectively.

3 An Interval Placement Algorithm

Having described the branch and bound placement algorithm, we now discuss the changes that must be made to this algorithm to incorporate uncertain objective functions. While its general structure remains unaffected, three aspects of the algorithm do change. They are

- Computing uncertain cost functions
- Comparing values returned by these cost functions
- Saving minimum-cost placements

[1] This circuit was adapted from an example in [Cies84]

```
Initialize minimum cost
Generate seed placement
Add placement to search tree
While there are unvisited placements on search tree
    Choose placement from search tree
    If placement cost ≥ minimum cost
        Prune placement from search tree
    Else if placement is complete
        Minimum cost = placement cost
        Save placement as current best
    Else (placement is partial and unpruned, so expand it)
        Choose unplaced block
        For each possible position in partial placement
            Create child placement by adding unplaced block at position
            Compute child placement cost
            If child cost ≥ minimum cost
                Prune child from search tree
        End for
End while
Return the minimum placement
```

Figure 1: Branch and Bound Placement Algorithm (Preas and vanCleemput)

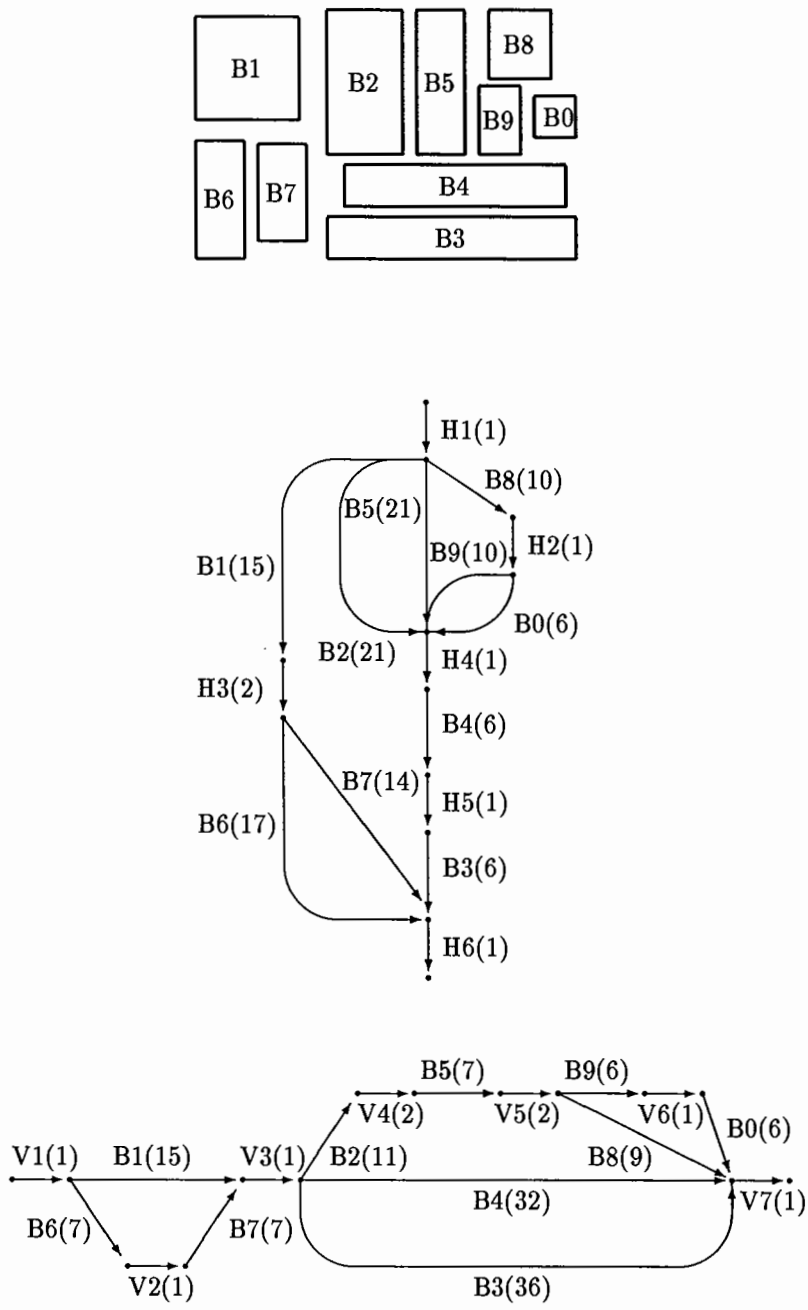


Figure 2: Resulting Minimum-Cost Placement with Channel Position Graphs

3.1 Computing an Interval Objective Function

The first of these issues is how to calculate an interval objective function. Since there are many different objective functions in current placement programs, it would be impractical to enumerate them all. Instead, we present a general methodology for creating an interval objective function and then illustrate it with a single example (circuit area).

To compute an interval objective function, we represent all uncertain parameters (e.g., channel widths, component sizes, wire lengths, component delays) in the cost function as intervals. An interval $[a, b]$ over the set of real numbers $\Re$ is defined as the set $\{x \mid x \in \Re, a \leq x \leq b\}$. A *degenerate* interval $[a, a]$ is an interval which contains only the one element a. We then create an application-specific interval algebra for manipulating these ranges. The interval operators in this algebra are the logical interval extensions of the discrete operators employed to compute the objective function. These interval operators F are derived from their non-interval counterparts f using the following standard formula:

$$F(\hat{x}, \hat{y}) = \{f(x, y) \mid x \in \hat{x} \text{ and } y \in \hat{y}\}\text{[Moor79]} \tag{1}$$

where $\hat{x}$ and $\hat{y}$ denote intervals. Having derived the equivalent interval operators, we then substitute them into the objective function to create an interval version of it. The resulting interval objective function returns bounds on its value when given interval inputs.

To illustrate this method, suppose that our objective function is layout area. Since cell dimensions and channel widths may not be known exactly, we represent their values as intervals. Now we must create an interval algebra for manipulating these ranges. The non-interval objective function described in section 2.1 employs three operations to compute layout area: $+$ to sum the edge lengths in a channel position graph, max to compare path lengths to determine the longest one, and $\cdot$ to multiply the longest path lengths from each graph to calculate the area. For the interval version, we derive the interval extensions $\oplus$ (addition), MAX (maximum), and $\odot$ (multiplication) of these operators using Equation 1. The resulting interval operators are defined as follows:

$$[a, b] \oplus [c, d] \equiv [a + c, b + d] \tag{2}$$

$$[a, b] \odot [c, d] \equiv [min(ac, ad, bc, bd), max(ac, ad, bc, bd)] \tag{3}$$

$$MAX([a, b], [c, d]) \equiv [max(a, c), max(b, d)] \tag{4}$$

To compute longest path length through a channel position graph, we can use a standard shortest-path algorithm [Aho74, pp. 195-201]. This algorithm is defined over the closed semi-ring $(\Re, max, +, -\infty, 0)$. By replacing these discrete operators with their interval counterparts, we create an interval version of the algorithm defined over the closed semi-ring $(\hat{\Re}, MAX, \oplus, [-\infty, -\infty], [0, 0])$,

where $\hat{\Re}$ represents the set of intervals over $\Re$. This interval longest-path algorithm returns bounds on longest path length. Once bounds on the length and width of the layout are found, the $\odot$ operator can be used to compute bounds on circuit area.

3.2 Comparing Placement Cost

Comparing the values returned by the objective functions is the second issue that we must address. Placement costs are used to prune unpromising placements from the search and to choose the best placement. To perform these tasks, we must be able to compare two interval placement costs to determine which is smaller.

In the first instance, we compare a placement cost to the current minimum cost, pruning the placement from the search if its cost exceeds the current minimum value. For pruning, we derive the interval comparator $\rhd$ (greater than) from its non-interval equivalent $>$ as follows:

$$[a, b] \rhd [c, d] \Longleftrightarrow a > d. \tag{5}$$

According to this definition, interval $[a, b]$ is greater than $[c, d]$ if and only if each member of the first interval range is greater than all members of the second. In our interval placement problem, a placement is thus pruned from the search tree if and only if all members of its cost range are greater than all members of the current minimum bound. This guarantees that we already have a better solution for all possible placement cost values; therefore, this placement cannot be a best placement and thus can be discarded.

Second, these metrics are used to compute the minimum placement cost. For our interval placement algorithm, we need an interval MIN operator to calculate the minimum of two placement costs. This operator is derived from the discrete operator *min* and is defined as follows:

$$MIN([a, b], [c, d]) \equiv [min(a, c), min(b, d)]. \tag{6}$$

The interval returned by this operator represents the set of minimum costs obtained by comparing two placements over all possible combinations of their values.

3.3 Saving the Minimum-Cost Results

It is not enough to know the minimum placement cost, we must also save the placement configurations themselves. The original placement algorithm computes the minimum placement cost $a \in \Re$ and returns a single representative configuration with cost a. In reality, there may be several configurations with minimum or close to minimum cost, any one of which could be used in the final design. Our interval version of the placement algorithm returns bounds $\hat{a} \in \hat{\Re}$ on minimum placement cost

and a *set* of configurations such that each placement in this set has a cost that lies within $\hat{a}$. Under some scenario, any of these placements could be optimal. Since each of these placement represents a potential best configuration, we must modify the algorithm to save them. Whenever we locate a complete placement with cost $\hat{c}$ such that $\hat{a} \cap \hat{c} \neq \emptyset$, we first update the minimum placement cost using the MIN operator described above and add the new placement to the set. Since the minimum placement cost may have changed, we must update the set of optimal placements by deleting all placements whose costs now exceed the new minimum cost bound. To perform this function, we use the previously-defined $\triangleright$ operator. For example, let $\hat{b}$ represent the cost of a particular configuration in the optimal set and let $\hat{a}$ represent the updated minimum-cost bounds. If $\hat{b} \triangleright \hat{a}$, then for each element of $\hat{b}$ there exists another configuration in the optimal set with smaller cost. Since this placement cannot be an optimal placement under any scenario, we delete it from the set.

The set of optimal configurations can be implemented in a variety of ways. One possibility is a linear linked list sorted in decreasing order by the lower bound on placement cost. Inserting a placement in this list has a worst-case time complexity of $O(n)$, where n is the number of configurations in the set. Deleting an item is much more efficient, since the placements with the largest areas are located at the front of the list and can be retrieved in $O(1)$ time. A better implementation is a heap with the condition that the minimum cost of each parent configuration is greater than those of its children. The cost of inserting a placement in this structure is $O(\log n)$. Since the placement with the greatest cost is kept at the top of the heap, this item can be retrieved in $O(1)$ time. Restoring the heap property after a deletion requires $O(\log n)$ time. As a result, the total cost of deleting a placement is also $O(\log n)$.

3.4 The Interval Algorithm

Incorporating the changes described above yields a new interval branch and bound placement algorithm, as summarized in Figure 3.

To illustrate this algorithm, we re-examine the ten-cell circuit in Figure 2. Instead of using only average values, we represent uncertain cell and channel sizes by their interval ranges. The interval dimensions of each block are displayed in Table 1. In the course of the search, the algorithm examined over 3 million partial and complete placements and returned a set of five optimal configurations for the circuit. The results of this computation are displayed in Figure 4. As shown, the best layouts had an area bound of $[1980.2, 2016.4]$ square mils. A closer examination of the set of minimum-cost placements reveals many similarities. For instance, elements B2, B3, B4 and B5 remain in the same relative position in all best placements. Other elements tend to cluster together, such as B8, B9 and B0. Some components appear in the same position in all five configurations. For example, block B1

```
Initialize minimum cost bound
Generate seed placement
Add placement to search tree
While there are unvisited placements on search tree
    Choose placement from search tree
    If placement cost ▷ minimum cost
        Prune placement from search tree
    Else if placement is complete
        Minimum cost = MIN(minimum cost, placement cost)
        Prune minimum list of all placements whose
            place cost ▷ minimum cost
        Add new placement to list of minimum placements
    Else (placement is partial and unpruned, so expand it)
        Choose unplaced block
        For each possible position in partial placement
            Create child placement by adding unplaced block at position
            Compute child placement cost
            If child cost ▷ minimum cost
                Prune child from search tree
        End for
End while
Return the minimum placements
```

Figure 3: Interval Branch and Bound Placement Algorithm

Block	Length			Width		
	min	max	avg	min	max	avg
B1	14.92	15.08	15.0	14.92	15.08	15.0
B2	10.94	11.06	11.0	20.89	21.11	21.0
B3	35.82	36.18	36.0	5.97	6.03	6.0
B4	31.84	32.16	32.0	5.97	6.03	6.0
B5	6.96	7.04	7.0	20.89	21.11	21.0
B6	6.96	7.04	7.0	16.91	17.09	17.0
B7	6.96	7.04	7.0	13.93	14.07	14.0
B8	8.95	9.05	9.0	9.95	10.05	10.0
B9	5.97	6.03	6.0	9.95	10.05	10.0
B0	5.97	6.03	6.0	5.97	6.03	6.0

Table 1: The Dimensions of the Circuit Blocks

is in the upper left-hand corner in all optimal solutions. These invariants indicate areas on the chip which can be improved to reduce the area of all optimal configurations. For example, blocks B2, B3, B4 and B5 all lie on critical paths in the horizontal channel position graphs of each configuration; therefore, these blocks and the channels between them define the width of the layout in all instances. By reducing the area of these blocks or by improving the routing between them, we can reduce the area of all five solutions.

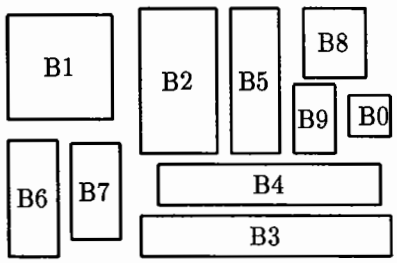
4 Performance

Both the interval and non-interval branch and bound placement algorithms presented in this chapter were implemented in the C programming language and run on a SUN SPARCstation. In this section, we discuss and compare the performance of these two approaches.

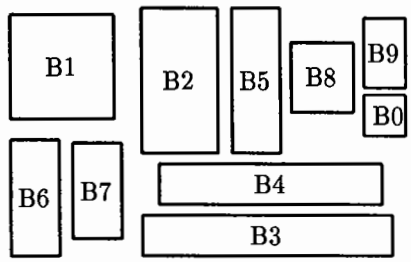
4.1 Speed

We begin by analyzing the operating speed of both implementations. Because it is an exhaustive search strategy, the original branch and bound algorithm may generate all possible configurations of the circuit components before finding the best solution; therefore, this algorithm has worst-case running time that is exponential in the number of circuit components. Since the interval solution is based on the same approach, it too will have an exponential worst-case running time.

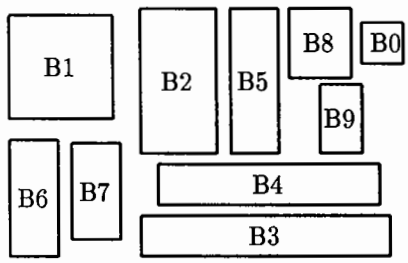
The Minimum-Cost Placements have cost bound = [1980.2, 2016.4]
The set of these placements follows:



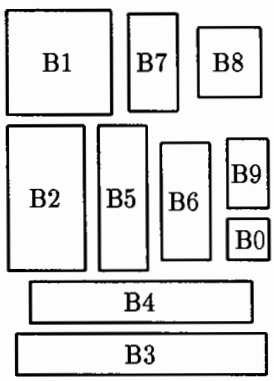
Cost = [1980.2, 2016.4]



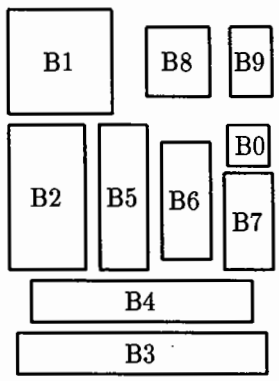
Cost = [1981.9, 2016.4]



Cost = [1982.2, 2016.4]



Cost = [1995.4, 2032.7]



Cost = [1995.6, 2032.7]

Figure 4: The Set of Minimum-Cost Placements for a Circuit with 10 Components

The interval algorithm differs from the non-interval version in two main areas: the use of interval operations and the storage of minimum-cost solutions. First, the interval solution employs interval operations to perform its calculations and comparisons. For each addition, multiplication, minimum, and maximum operation performed in the original version, we perform two such operations in the interval version. This is particularly evident in the longest path computation used to calculate placement cost; hence, we expect that this function will require twice the time in the interval implementation.

Computing the minimum-cost solution is the second major difference between the interval and non-interval placement algorithms. Instead of returning a single configuration, the interval version returns a set of potential minimum-cost placements. Maintaining this set adds to the operating speed of the interval algorithm. Specifically, whenever the minimum-cost bound is updated in the course of the search, we must insert and delete items from this set. Using a heap to implement the set requires $O(\log n)$ time for each insertion and deletion, where n is the number of placements in the set. The total number of possible complete configurations is exponential in m, the number of circuit elements. In the worst-case, the set of minimum-cost placements could contain all of these possible configurations, yielding insertion and deletion times of $O(m)$. This decreases the operating speed of the interval algorithm by a factor of m in the worst-case. Fortunately, the list of minimum-cost placements is typically small, containing a fraction of the total number of possible complete placements. For example, consider the circuit in Figure 4. Of the several billion possible complete placements, the optimal solution contained only five configurations.

We also tested both the interval and non-interval implementations on identical circuit descriptions and recorded their response times. The circuits tested contained up to ten components. For the larger circuits, the programs generated more than 180,000 distinct partial and complete configurations in the search for the best solution. Even so, this figure represents a significantly pruned portion of the almost 98 billion possible configurations in the search space for that problem. In spite of the large numbers of configurations explored, these experiments showed no appreciable difference in the operating speeds of the two algorithms. The apparent similarity in running times is attributed to the observation that most of the computation time is spent generating child placements, rather than evaluating them. The routines for generating child configurations are identical in both implementations.

4.2 Interval Width

Since an interval cost is used to prune the search tree in the branch and bound placement algorithm, the width of these intervals will effect the efficiency of the search. Recall that the algorithm prunes

a partial placement from the tree when its cost exceeds the current minimum cost. In the interval version, placement cost is greater than minimum cost if and only if the lower bound on placement cost is greater than the upper bound on minimum cost. Obviously, wider intervals yield larger upper bounds. With a larger upper bound on minimum layout cost, we prune fewer nodes from the search tree and increase time needed to find the optimal solutions. Even in the non-interval algorithm, pruning efficiency is heavily dependent on circuit structure. In circuits with many blocks of similar shape and size, the standard algorithm may not be able to prune many nodes from the search tree, resulting in a nearly exhaustive enumeration of placement alternatives. For circuit descriptions containing blocks with widely varying sizes, the algorithm can prune a larger portion of the nodes from the search tree. Because pruning efficiency also depends on relative block size, a precise formula indicating the relationship between interval width and pruning efficiency is difficult to determine. We can, however, present several examples illustrating this correlation.

Figure 5 depicts the effects of interval width on search efficiency for several circuit descriptions containing five elements. In this graph, interval width is measured by the percent variance from the center of the interval (i.e., interval = center $\pm$ center $\cdot$ variance). Search efficiency is measured by the percentage of the 6565 possible partial and complete configurations generated and examined by the placement algorithm. In this experiment, we started with degenerate intervals and counted the number of partial and complete placements examined in the course of the search. For each circuit description, we varied the width of the intervals by a given percentage and recorded the resulting number of placements examined in locating the optimal solution. The graph displays these results for three different circuit descriptions. The first circuit (denoted by dots with dashed edges) contains blocks with a wide variation of sizes, the second circuit (denoted by triangles and solid lines) contains five blocks with three different sizes, and the third (denoted by diamonds and dotted edges) contains five identical blocks. As the graph shows, the increase in the number of configurations produced was modest for small cost interval widths ($\pm$5%). For large interval widths ($\pm$25%), the algorithm degenerated to an exhaustive enumeration of placement alternatives. Therefore, this algorithm should not be used, if the intervals given for circuit parameters are too wide.

As expected, interval width also affects the size of the optimal placement set. Wider intervals yield wider minimum cost bounds. With a larger upper bound on minimum layout cost, we prune fewer configurations from the optimal solution set. Figure 6 presents a graph illustrating the effects of interval width on the size of the optimal placement set for our three circuit examples. In each case, we varied the width of the cost intervals by the given percentage and recorded the number of placements in the final solution set. As shown in the graph, the solution sets for small cost interval widths ($\pm$5%) contain few configurations. For larger cost intervals ($\pm$25%), the solution sets become

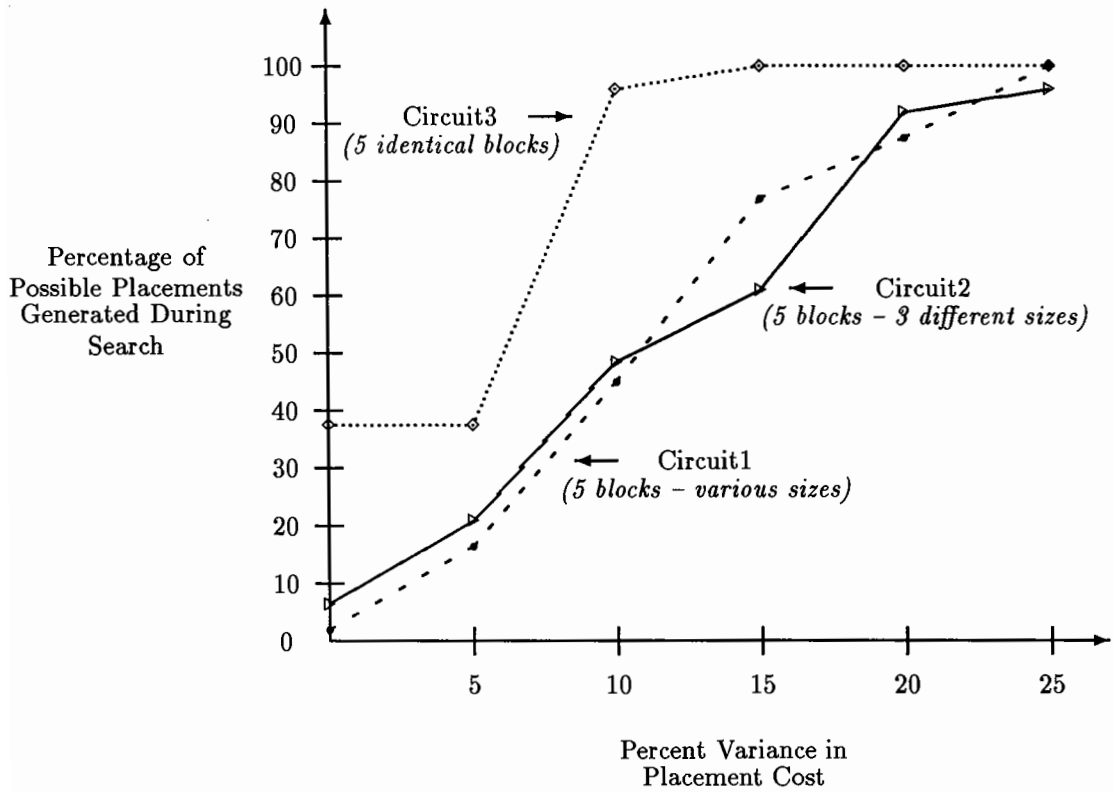


Figure 5: Cost Interval Width vs. Pruning Efficiency

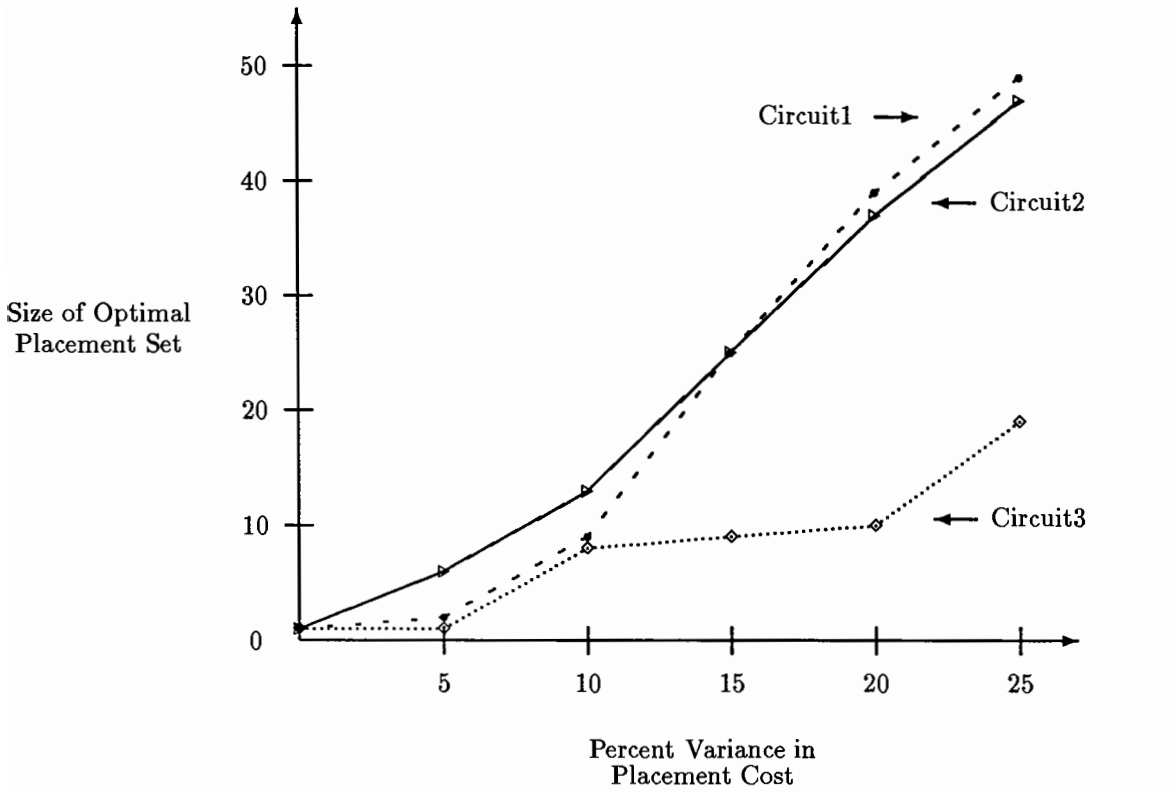


Figure 6: Cost Interval Width vs. Optimal Placement Set Size

too large to be useful. Not surprisingly, the less we know about the circuit, the less we can conclude about the optimal layout.

5 Summary

In this paper, we discussed the problem of uncertain cost functions in automatic placement algorithms. We noted that many objective functions used to measure placement quality contain uncertain parameters, but that current placement algorithms use only single-valued estimates in their computations. We argued that using the full range of possible values when computing placement costs yields a more credible prediction of placement quality and reveals more about the structure of the optimal configurations. For these reasons, we proposed an interval-based approach for modeling uncertainty in placement algorithms. Given the interval range for each uncertain parameter in an objective function, the interval placement algorithm computes and returns bounds on minimum placement cost and a set of placements, each of which is optimal for some combination of input

values.

We illustrated this approach by implementing an interval branch and bound placement algorithm. We showed how interval operators could be used to compute the interval cost of a placement and to determine the minimum-cost configuration. Although a branch and bound algorithm was chosen, the general principles employed also apply to other placement methods.

To test the performance of our interval approach, we implemented interval and non-interval versions of the placement algorithm. Analysis suggests that the interval program may be slower than the non-interval one by a factor of m, the number of circuit components, in the worst-case. Experiments on several circuits, however, showed no appreciable difference in the operating speeds of the two implementations. We also observed that the width of the intervals affects the efficiency of the search, since wider placement cost bounds reduce the number of placements that are pruned from the search tree. Wider intervals also increase the size of the solution set, as fewer placements can be pruned from the optimal set.

References

[Aho74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[Aho83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, 1983.

[Breu77] Melvin A. Breuer. A class of min-cut placement algorithms. In *14th IEEE Design Automation Conference*, pages 284–290, 1977.

[Cies84] Maciej J. Ciesielski and Edwin Kinnen. Digraph relaxation for CAD layouts of cell based integrated circuits. In *1983-84 Computer Science and Computer Engineering Research Review.* University of Rochester, 1983-84.

[Dai89] Wayne Wei-Ming Dai, Bernhard Eschermann, Ernest S. Kuh, and Massoud Pedram. Hierarchical placement and floorplanning in BEAR. *IEEE Transactions on Computer-Aided Design*, 8(12):1335–1349, December 1989.

[Fidu82] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th IEEE Design Automation Conference*, pages 175–181, 1982.

[Jack86] Micheal Jackson and Ernest Kuh. Performance-driven placement of cell based IC's. In *26th IEEE Design Automation Conference*, pages 370–375, 1986.

[Kern70] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.

[Kirk83] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[Moor79] Ramon E. Moore. *Methods and Applications of Interval Analysis.* SIAM, Philadelphia, 1979.

[MS89] Malgorzata Marek-Sadowska and Shen P. Lin. Timing driven placement. In *IEEE International Conference on Computer-Aided Design*, pages 94–97, November 1989.

[Prea79] B.T. Preas and W.M. van Cleemput. Placement algorithms for arbitrarily shaped blocks. In *16th IEEE Design Automation Conference*, pages 474–480, 1979.

[Prea86] Bryan T. Preas and Patrick G. Karger. Automatic placement: A review of current techniques. In *23rd IEEE Design Automation Conference*, pages 622–629, 1986.

[Tric86] Micheal T. Trick and Stephen W. Director. LASSIE: Structure to layout for behavioral synthesis tools. In *26th IEEE Design Automation Conference*, pages 104–109, 1986.