AN ESCAPE ROUTING FRAMEWORK FOR DENSE BOARDS WITH HIGH-SPEED DESIGN CONSTRAINTS

ABSTRACT

Shrinking transistor sizes, increasing circuit complexities, and high clock frequencies bring new board routing challenges that can not be handled effectively by traditional routing algorithms. Many high-end designs in the industry today require manual routing efforts, which increases the design cycle times considerably. In this paper, we propose an escape routing algorithm to route nets within multiple dense components simultaneously so that the number of crossings in the intermediate area is minimized. We also show how to handle high-speed design constraints within the framework of this algorithm. Experimental comparisons with a recently proposed algorithm [10] show that our algorithm reduces the via requirements of industrial test cases on average by 39%.

1. INTRODUCTION

Due to shrinkage of transistor sizes and increasing functional complexities, the densities of VLSI circuits have been increasing rapidly. In parallel to these developments, boards and packages have been reducing in size, while the pin counts have been increasing [1, 6]. For example, a multichip module (MCM) used in IBM eServer z900 [4] (introduced in 2000) contains twenty processor chips, eight L2 cache chips, two system control chips, four memory bus adapter chips, and a clock chip. On the bottom of this MCM, there are 4224 I/O pins, within an area of 127-mm $\times$ 127-mm. In the subsequent generation of the same series, IBM eServer z990 [13] (introduced in 2003), the corresponding number of pins in an MCM has increased about 20 percent, with a decrease of almost 50 percent in the substrate area. With increasing pin densities of this pace, routing nets on boards beneath the component areas (escape routing) is increasingly becoming the main bottleneck in terms of overall routability [13]. Since the traditional routing algorithms can not handle designs with such complexities, many high-end boards in the industry today require manual efforts for routing, which take about a month in a typical design cycle [7]. So, new routing algorithms that can handle recent challenges are needed to significantly reduce this time.

In a typical high end board, there are a number of components, corresponding to MCM, memory, and I/O modules. A component is typically in the form of a 2-D pin array, with routing tracks between adjacent rows and columns of pins. The routing resources within a component are extremely limited due to blockage of pins, and tight clearance rules. Furthermore, a large number of nets need to be routed from their terminal pins to the component boundaries using these limited resources. On the other hand, there are relatively few blockages in the area between different components, and the amount of available routing resources is relatively larger.

In a recent work [10], a problem decomposition has been given to distinguish two different problems: (1) routing nets from pin terminals to component boundaries *(escape routing)*, and (2) routing nets between component boundaries *(area routing)*. Here, escape routing problems for different components need to be considered simultaneously to reduce the number of crossings in the intermediate area. In other words, we can not just apply a traditional escape routing algorithm [2, 3, 5, 12, 14] on each component independently. The reason is that such an approach would ignore the connections between different components, and increase the buried via requirements of the next routing stage (area routing) substantially. Especially in high-speed designs, these vias seriously degrade signal characteristics, add additional delay, decrease routing area, and lower the manufacturing yields. Furthermore, for some board designs, no buried vias are allowed, for the purpose of limiting manufacturing costs [7]. For such designs, the nets need to be routed in a planar fashion on every layer. Hence, an escape routing algorithm that tries to minimize (or completely avoid) crossings in the intermediate areas is crucial to handle the recent challenges encountered in board routing problems.

In this paper, we propose an algorithm for solving the escape routing problem in multiple components simultaneously, so that the number of crossings in the intermediate area is minimized, and high-speed design constraints are satisfied. Figure 1 illustrates a one-layer escape routing solution for two components. In this figure, nets have been routed from their terminal pins to the corresponding component boundaries. Here, only one net (net D) crosses with the others in the intermediate area. For this net, the area router will need to use a via to resolve the crossing. We can say that the number of crossings in the intermediate area is a good measure for the via requirements of an escape routing solution.

Compared to the algorithm proposed in [10], our algorithm has three main advantages: (1) more general escape patterns are considered within the framework, instead of simple straight connections (Section 3), (2) an improved maximal planar route selection algorithm is proposed, which is general enough to handle multi-capacity escape slots, and high-speed design constraints (Section 4), (3) explicit discussion about how to handle various high-speed design constraints is given for this framework (Section 5). Our experiments in Section 6 show that our algorithm reduces the via requirements of industrial test cases on average by 39%, compared to the recent algorithm [10].

The rest of this paper is organized as follows. We give a formal description of this problem in Section 2. Our methodology to solve this problem is based on generating a number of different routing alternatives for each net, and selecting the maximum planar subset of escape patterns on each layer. In Section 3, we propose an algorithm to generate escape patterns based on congestion levels within the components, and the crossings in the intermediate area. Then, we propose a randomized algorithm in Section 4 for the problem of maximum planar route selection. In Section 5, we discuss how

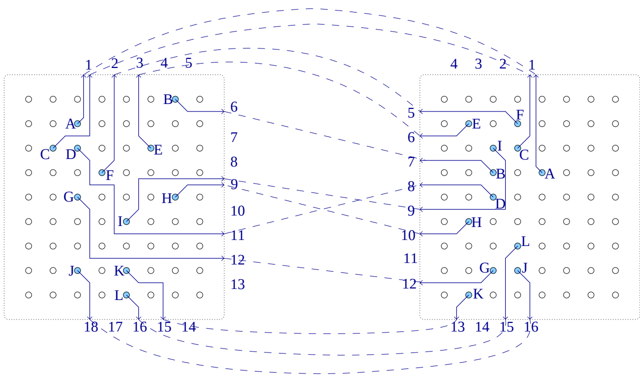

Figure 1: An escape routing solution for 12 nets. The escape slots are identified on the boundaries of components. The connections in the intermediate area are shown by dashed lines.

to handle high-speed design constraints within the framework of this algorithm. Finally, we demonstrate the effectiveness of this algorithm on industrial test cases in Section 6.

2. PROBLEM FORMULATION AND METHODOLOGY

Let a *component* be defined as a 2-D array of pins, where each pin spans multiple layers, and routing tracks are defined on each layer between adjacent rows and columns of pins. An *escape segment* is defined to be a route from a pin inside the component to an *escape slot* on the component boundary. For a component, a set of *escape slots* are defined on its boundary, defining the permissible end-points of escape segments originating from the pins. Due to limited routing resources, buried vias are not allowed inside the components. So, routing within the component area needs to be planar on every layer. Two escape segments corresponding to two different nets are defined to have a *conflict* inside the component if they can not be routed together on the same layer in a planar fashion. The number of routing tracks at each row and column is predetermined based on the pin diameters, wire widths, pin spacings, and clearance constraints. In a feasible solution, the number of escape segments passing through a row/column of the component can not exceed the corresponding capacity of that row/column.

Let us assume that the input problem consists of only two components, which are denoted as *left* and *right* components, respectively, for simplicity of presentation. A *net* is assumed to have two terminals, one in each component. An escape pattern P_i for net i is defined to be the combination of two escape segments originating from the terminals of net i in the left and right components. Two escape patterns P_i and P_j corresponding to nets i and j are defined to have a *conflict* iff their escape segments have conflicts within at least one component. Note here that a pair of non-conflicting escape patterns P_i and P_j can have a *crossing* in the intermediate area between components, depending on the relative ordering of their escape slots. Since buried vias are allowed in the intermediate area between components, these crossings are allowed in a feasible solution. However, the number of crossings need to be minimized for the objective of via minimization.

Based on these definitions, the simultaneous escape routing problem for a set of nets can be stated as follows: *Find an escape pattern P_i for each net i, and assign it to a layer such that: (1) no pair of escape patterns on the same layer conflict with each other, (2) the capacity constraints on all rows and columns of the components are satisfied, and (3) the number of crossings in the intermediate area is minimized.*

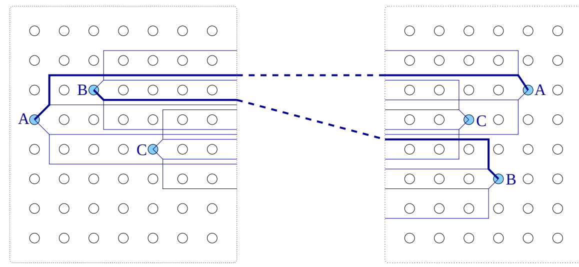

Figure 2: The output of a simple pattern generation technique for 3 nets. The maximal planar subset is highlighted with bold lines.

Figure 1 illustrates a sample one layer solution for 12 nets. The number of escape slots in the left and right components are 18 and 16, respectively. While the slots on the left component are numbered increasing in the clockwise direction, the slots on the right component are numbered increasing in the counter clockwise direction. Among the 12 nets routed on this layer, only one of them (net D) crosses with the others in the intermediate area. Some of the escape slots (slots 1 and 9 in the left component, slot 1 in the right component) are used by more than one escape segments. This is allowed in a feasible solution as long as the capacity constraints are not violated.

Our methodology to solve this problem is to process one layer at a time, and route as many number of noncrossing nets as possible on each layer. After finding a maximal planar routing solution for all layers, we distribute the remaining nets to available layers, this time allowing crossings in the intermediate area. In the rest of the paper, we will focus on the problem of maximal planar routing. The details of the postprocessing phase to distribute the remaining nets to available layers will not be given due to page limitations.

Our algorithm for maximal planar routing consists of two phases: (1) Generate a number of different routing alternatives for each net, and (2) select the maximal subset of routing patterns that will give a feasible planar routing solution for the current layer. A similar algorithm has been proposed for this purpose recently [10]. Our main contributions in this paper can be summarized as follows. For the first phase, we propose an algorithm to generate routing patterns based on the congestion levels inside the components, and the number of crossings in the intermediate area (Section 3). For the second phase, we propose a more sophisticated randomized algorithm, which can also be generalized to handle high-speed design constraints (Sections 4 and 5).

3. ESCAPE PATTERN GENERATION

3.1. Motivation

In this section, we describe an algorithm to generate a number of different routing alternatives for each net. In a recent work [10], a simple pattern generation methodology has been used for this purpose. In particular, 4 escape segments are generated for each net within each component, for a total of $4 \times 4 = 16$ escape patterns. The escape segments generated in that algorithm have vertical spans of at most 2 rows, as illustrated in Figure 2. The

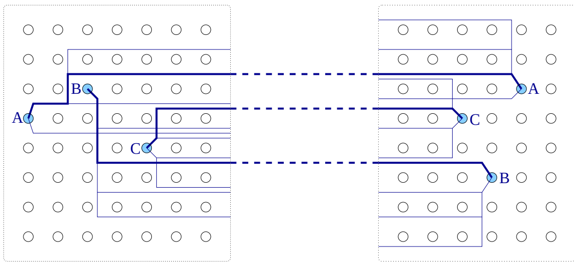

Figure 3: Pattern generation with the objective of low congestion levels, and small number of crossings. The maximal planar subset is highlighted with bold lines.

justification here is that escape patterns with large vertical spans block other patterns; so small vertical spans are needed for maximal planar routing. However, non-regular escape patterns with larger vertical spans, as illustrated in Figure 3, may be helpful in some situations. For example if we use the simple pattern generation technique of the recent work [10], as in Figure 2, then only 2 out of 3 nets will be routed in a planar fashion, as highlighted in the figure. However, if we use a more intelligent pattern generation algorithm as in Figure 3, we can route all 3 nets in a planar fashion. With this motivation, we propose an algorithm in this section that generates escape patterns based on congestion levels within the components, and the crossings in the intermediate area.

Our objective here is to generate escape patterns with low congestion levels inside the components, and small number of crossings in the intermediate area. However, the patterns generated need to satisfy the following two properties:

Consider any pair of escape segments S_i and S_j generated within the same component. Let V be a constant input parameter.

Property 3.1. *If S_i and S_j correspond to the same net (i.e. S_i and S_j originate from the same terminal), then it must be the case that $|S_i.slot - S_j.slot| < V$.*

Property 3.2. *If S_i and S_j have a* conflict*, then it must be the case that $|S_i.slot - S_j.slot| < V$.*

Here, the notation $S.slot$ denotes the index of the *escape slot* of segment S, as defined in Section 2. Intuitively, the segments belonging to the same net, and the conflicting segments must escape to slots that are *close* to each other on the component boundary. In the examples of Figure 2 and 3, these two properties hold for $V = 4$. As will be discussed in detail in Section 4, our *maximal planar route selection* algorithm will be based on the assumption that these two properties hold. Furthermore, we will show in Section 4 that a polynomial-time optimal algorithm exists for *maximal planar route selection* problem if these two properties hold for a constant V value.

3.2. The Algorithm

Given a simultaneous escape routing problem, as defined in Section 2, we start with sorting all the net terminals based on their distances to the closest escape slots on the component boundaries. Then, we process these terminals in sorted order, starting from the terminal closest to an escape slot. Originating from each terminal, we generate a number of escape segments, by using the algorithm outlined in Figures 4 and 5.

GENERATE-ESCAPE-SEGMENTS($G, t, V, \mathcal{T}$)
  // *G: the grid graph corresponding to the component*
  // *t: the terminal from which the segments will be generated*
  // *V: the input parameter*
  // *$\mathcal{T}$: the set of target escape slots*
  for $index \leftarrow 1$ to V do
    $S \leftarrow$ GENERATE-ONE-ESCAPE-SEGMENT($G, t, \mathcal{T}$)
    add S to the set of escape segments originating from t
    $\mathcal{T} \leftarrow \mathcal{T} \cap (\{s : S.slot - V < s < S.slot + V\} \setminus \{S.slot\})$
      // *limit the target slot range to satisfy Property 3.1*

Figure 4: High level algorithm to generate a number of V escape segments originating from terminal t.

Figure 4 displays the high level algorithm used to generate a number of routing segments originating from a given terminal t. Here, graph G is used to model the routing resources of the input component. As described in Section 2, a component is assumed to be a 2-D array of pins, with rows and columns of routing tracks between adjacent pins. Also, a set of target *escape slots* $\mathcal{T}$ is specified for terminal t as input to the algorithm of Figure 4. Although $\mathcal{T}$ can be set such that it contains all escape slots on the component boundary, it is also possible to set it based on the length constraints of the corresponding net, as will be discussed in Section 5. Observe in Figure 4 that after an escape segment S is generated from terminal t, the set $\mathcal{T}$ is restricted to the escape slots that are within the neighbourhood of escape slot of S. The purpose here is to make sure that Property 3.1 is maintained for the segments generated from terminal t.

Before describing the low level algorithm, we need to make the following definition:

Definition 3.1. *Let $v.segments$ denote the set of escape segments passing through vertex v. An escape slot s is defined to be* reachable *from vertex v iff for each escape segment $S \in v.segments$, it is the case that $|S.slot - s| < V$, where V is the input parameter specified in Property 3.2. The set of* reachable *escape slots from vertex v is denoted as $v.reachableSlots$*

Remark 3.1. *Consider a path P in grid graph G that starts at terminal t, and ends at escape slot s. If $s \in v.reachableSlots$ for each $v \in P$, then it is guaranteed that path P satisfies Property 3.2.*

The low level algorithm used to generate one escape segment is given in Figure 5. This is basically a variant of Dijkstra's shortest path algorithm [3]. As an additional constraint, we make sure that Property 3.2 is satisfied, by restricting the target slot range when a conflict with an existing pattern is possible. The cost of edge $(u \rightarrow v)$ is computed by the following formula:

$$cost(u \rightarrow v) = \alpha.dist(u \rightarrow v) + \beta.cong(v) + \gamma.cross(v) \quad (1)$$

Here, α, β, and γ are scaling factors for *distance*, *congestion*, and *crossing* cost metrics, respectively. Congestion cost for vertex v is computed based on the number of escape segments passing through v. Before generating any escape pattern, we first estimate the congestion values for individual vertices through path analysis. As we generate escape segments, we gradually replace these estimations with actual congestion values. If v is a vertex corresponding to an escape slot, we also compute a crossing cost based on the estimated number of crossings in the intermediate area.

GENERATE-ONE-ESCAPE-SEGMENT(G, t, $\mathcal{T}$)
```
pQ ← an empty priority queue
for each vertex v ∈ G that is adjacent to terminal t do
  v.label ← 0
  v.targetSlots ← 𝒯
  pQ ← pQ ∪ {v}
while pQ not empty do
  u ← pQ.extractMin()
  if u corresponds to an escape slot then terminate loop
  for each edge (u → v) ∈ G do
    if (u.targetSlots ∩ v.reachableSlots ≠ ∅)
        && (u.label + cost(u → v) < v.label) then
      v.label ← u.label + cost(u → v)
      v.targetSlots ← u.targetSlots ∩ v.reachableSlots
        // limit the target slot range to satisfy Property 3.2.
      v.parent ← u
      pQ ← pQ ∪ {v}
construct escape segment S by backtracking from escape slot u
```

Figure 5: Low-level algorithm to generate one escape segment originating from terminal t.

4. MAXIMAL PLANAR ROUTE SELECTION

In this section, it is assumed that a number of escape patterns that maintain Property 3.1 and Property 3.2 have been generated. The objective now is to select the maximum number of escape patterns such that: (1) at most one pattern for each net is selected, (2) the segments of the selected patterns do not *conflict* with each other within components (i.e. they are routable in a planar fashion on the same layer), and (3) the selected patterns have no crossing in the intermediate area.

4.1. Problem Modeling

Let $P.slotL$ and $P.slotR$ denote the escape slots of escape pattern P in the left and right components, respectively. Furthermore, let us assume that a unique rank is assigned for each escape segment within a component, indicating the relative ordering between different segments. As an example, consider the segments of nets I and H in the left component of Figure 1. Although these two segments escape to the same slot (slot 9), the rank of net I's segment must be less than the rank of the corresponding segment of net H. Let $P.rankL$ and $P.rankR$ denote the rank of pattern P in the left and right components, respectively. For simplicity of presentation, we will first consider the problem with unit slot capacities in the following definitions.

Definition 4.1. *The* less-than predicate *for two escape patterns is defined as follows:* $P_i \prec P_j$ *iff* $P_i.rankL < P_j.rankL$ *and* $P_i.rankR < P_j.rankR$.

Note here that the precedence relation defined above is *transitive*, i.e. if $P_i \prec P_j$ and $P_j \prec P_k$, then $P_i \prec P_k$. Based on this property, we can give the following definition:

Definition 4.2. *A pattern sequence $\mathcal{S}$ is defined to be an ordered set of patterns* $\{P_1, P_2, ..., P_n\}$ *such that if* $i < j$ *then* $P_i \prec P_j$.

Definition 4.3. *A pattern sequence $\mathcal{S}$ is defined to be* permissible *iff it contains no pair of conflicting patterns*[1].

[1]We denote two patterns P_i, P_j as *conflicting* iff they can not occur together in a valid planar escape routing solution.

PLANAR-ROUTE-SELECTION($\mathcal{P}$: a set of patterns)

map each pattern in $\mathcal{P}$ to a cell of checkerboard $\mathcal{C}$
rowwise partition $\mathcal{C}$ into subproblems with $V-1$ rows each
randomly generate a set of subsequences in each subproblem
create a graph $\mathcal{G}_R$ as follows:
- A vertex v_j^i exists in $\mathcal{G}_R$ corresponding to each subsequence S_j^i in subproblem i.
- The weight of v_j^i is equal to the number of patterns in S_j^i.
- Let x_j^i and x_k^{i+1} denote the x coordinates of the checkerboard cells of the last patterns in subsequences S_j^i and S_k^{i+1}.
  An edge from v_j^i to v_k^{i+1} exists in $\mathcal{G}_R$ iff:
  (1) all patterns in S_j^i are to the left of all patterns in S_k^{i+1}
  (2) $x_k^{i+1} > x_j^i + V - 2$,
  (3) no pattern in S_j^i conflict with a pattern of S_k^{i+1}.

return the longest path in $\mathcal{G}_R$

Figure 6: High level description of the randomized planar route selection algorithm

Theorem 4.1. *For a given set of escape patterns, the longest permissible pattern sequence $\mathcal{S}$ is equivalent to the maximum subset of patterns that can be routed on one layer in a planar fashion.*

Theorem 4.2. *For a given set of escape patterns, assume that Property 3.1 and Property 3.2 are satisfied for a constant V value. Then, there is a polynomial-time optimal algorithm to solve the maximal planar route selection problem.*

PROOF. Due to page limitations, the detailed proof is omitted here. However, it is based on showing that a dynamic programming algorithm with time complexity $o(n^{V+1})$ can solve the problem of longest permissible sequence if Properties 3.1 and 3.2 are satisfied. □

Although a polynomial-time optimal algorithm exists for this problem, its high time complexity makes it impractical for large circuits. For this reason, we propose a much faster randomized algorithm in the next subsection, which gives solutions that are very close to optimum in practice. As mentioned before, we will also discuss how to handle high-speed design constraints within the framework of this algorithm in Section 5. The algorithm proposed in the next subsection can also handle multi-capacity slots, as given by the following definition.

Definition 4.4. *Assume that each escape slot is defined to have a particular capacity, as defined in Section 2. A sequence $\mathcal{S}$ is defined to be* capacity constrained *iff the number of patterns in $\mathcal{S}$ that use a particular escape slot is less than or equal to the corresponding slot capacity.*

4.2. A Randomized Algorithm

In this section, we propose a randomized algorithm to solve the *capacity-constrained longest permissible sequence* problem for a given set of escape patterns. The high level algorithm is given in Figure 6. Compared to the algorithm given in [10], the main improvement is our randomized subsequence generation algorithm, as given in Figure 9. This algorithm not only improves the routing results considerably (Section 6), but also is general enough to handle multi-capacity escape slots, and typical high-speed design constraints (Section 5). We also prove later in this section that the

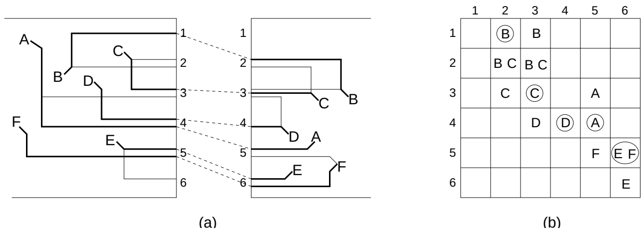

Figure 7: (a) A set of routing patterns defined for 6 nets. (b) The corresponding checkerboard model. For clarity, only one or two escape segments are illustrated for each net. The maximum planar subset is highlighted in both figures.

average-time complexity of this algorithm is linear in the component sizes (Theorem 4.6).

The (conceptual) *checkerboard* model introduced in the algorithm of Figure 6 is defined as follows [10]:

Definition 4.5. *Let $\#sL$ and $\#sR$ denote the number of escape slots defined on the left and right components, respectively. Let $\mathcal{C}$ be a (conceptual) checkerboard with $\#sL$ rows and $\#sR$ columns. An escape pattern P is defined to be* mapped *to cell (i, j) of checkerboard $\mathcal{C}$ iff $P.slotL = i$ and $P.slotR = j$.*

Figure 7 illustrates a sample escape problem, and the corresponding checkerboard model. Let us consider two patterns P_i and P_j on this checkerboard. If P_j is below and to the right of P_i (e.g. $P_i = B12, P_j = C33$), then $P_i \prec P_j$, as defined in Definition 4.1. If P_j is above and to the right of P_i (e.g. $P_i = D43, P_j = A35$), then neither $P_i \prec P_j$, nor $P_j \prec P_i$. Otherwise, if P_i and P_j are on the same row (e.g. $P_i = D44, P_j = A45$), or the same column (e.g. $P_i = C33, P_j = D43$), or the same cell (e.g. $P_i = E56, P_j = F56$), then we need to check the ranks of P_i and P_j to determine the relationship between these patterns. For instance, ranks of $E56$ in both left and right components are less than those of $F56$ (since the corresponding escape segments of net E are above those of net F); hence $E56 \prec F56$.

After mapping each pattern to a checkerboard cell, $\mathcal{C}$ is rowwise partitioned into subproblems. Then a set of *capacity-constrained permissible* subsequences is randomly generated within each subproblem, as will be described in detail later in this section. After that, these subsequences are merged together to obtain the *capacity-constrained longest permissible sequence*. For this purpose, a graph model $\mathcal{G}_R$ is defined in Figure 6, which satisfies the following lemma.

Lemma 4.3. *Consider two subsequences S_j^i and S_k^l in subproblems i and l, respectively. If there is a path between the corresponding vertices v_j^i and v_k^l in $\mathcal{G}_R$, then it is guaranteed that S_j^i and S_k^l contain no patterns that conflict with each other.*

A more restricted version of this lemma has been given previously for a similar graph model [10]. Based on this lemma, we can compute the longest path in acyclic graph $\mathcal{G}_R$ to find the best combination of subsequences generated in different subproblems. Then, we can merge these subsequences to obtain the longest permissible sequence. Figure 8 illustrates the execution of the randomized algorithm on a sample board. Here, assume that a number of patterns have already been mapped to this checkerboard, and the conflicts between patterns are as listed in this figure. A small set of randomly generated subsequences[2] is shown for each subproblem on the right. Corresponding to each subsequence, there is a vertex in $\mathcal{G}_R$, and edges between them are created based on the rules defined in Figure 6. For instance, there is no edge from $\{A, B, C, E\}$ to $\{I, J, L, M\}$ because patterns E and I are conflicting. Similarly, there is no edge from $\{A, B, F, G\}$ to $\{I, J, L, M\}$, because it violates rule (2) in Figure 6. The longest path in $\mathcal{G}_R$, corresponding to the *capacity-constrained longest sequence* is also highlighted in this figure.

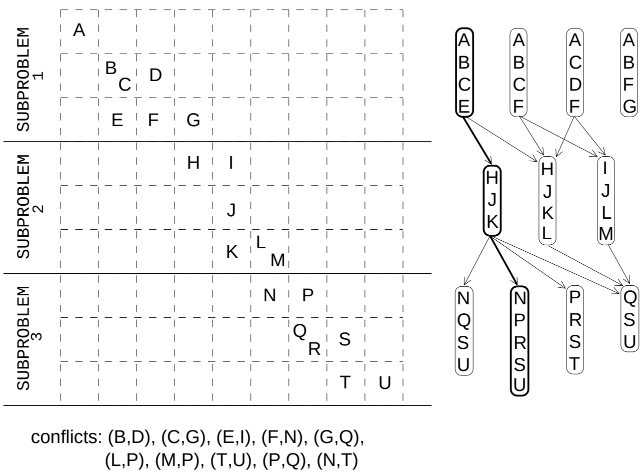

Figure 8: Illustration of the randomized algorithm given in Figure 6 on a sample checkerboard. For clarity, ranks of the patterns are not displayed. The set of subsequences generated for each subproblem are shown on the right, together with the corresponding graph $\mathcal{G}_R$, and the (highlighted) longest path. It is assumed here that each escape slot has a capacity of two.

The algorithm we use to generate a set of random subsequences is outlined in Figure 9. In the beginning, this recursive function is called for each cell on the first row of the given subproblem, with argument $subseq$ set to $\emptyset$. In one recursive call, first it is checked whether the partial subsequence generated so far is good enough to store. This decision is made by comparing the weight of the current subsequence $subseq$ with the weights of the subsequences already stored for this subproblem. Let tx denote the x-coordinate of the checkerboard cell corresponding to the last pattern in $subseq$. The weight of $subseq$ is compared with only the subsequences that end at column tx of the checkerboard. An input parameter determines how many subsequences can be stored corresponding to each column[3]. If the partial subsequence *subseq* is to be stored, a previously stored subsequence with less weight may need to be replaced. Note here that our purpose is to generate a large variety of *good* subsequences for the given subproblem, instead of generating only the *best* ones. The variety among subsequences is obtained by making sure that a subsequence ending at a particular checkerboard column does not replace another subsequence ending at a different column.

[2] For clarity, only 3 or 4 subsequences are given in this example. Normally, hundreds or even thousands of subsequences are generated for each subproblem to obtain a good variety.

[3] In our experiments, the maximum number of subsequences that can be stored corresponding to each column is set to 50.

GENERATE-SUBSEQ($x, y, subseq$)
// *(x,y): coordinate of the current checkerboard cell*
// *subseq: the partial subsequence generated so far*
if cell (x, y) is not within subproblem boundaries
  terminate recursion
if $subseq$ is *good*
  store $subseq$ in candidate set of the subproblem
Let P' be the last pattern in $subseq$
$\mathcal{T} \leftarrow \{P : P' \prec P$ (see Definition 4.1) AND
    $((x \leq P.slotR \leq x + \Delta$ AND $P.slotL = y)$ OR
    $(y \leq P.slotL \leq y + \Delta$ AND $P.slotR = x))$ AND
    capacity of $(P.slotR, P.slotL)$ not fully used AND
    P has no conflict with $subseq\}$
for each pattern $P \in \mathcal{T}$ do
  randomly determine whether to *accept* or *reject* P
  if P is *accepted*
    GENERATE-SUBSEQ($P.slotR, P.slotL, subseq \cup \{P\}$)
GENERATE-SUBSEQ($x + 1, y + 1, subseq$)

Figure 9: Algorithm to generate a set of random subsequences

The next step of the recursive algorithm is to find the set of patterns $\mathcal{T}$ that can be added to the partial subsequence $subseq$. Here, this selection is done based on the invariant that $subseq$ remains *permissible* (Definition 4.3), and capacity constrained (Definition 4.4). In one recursive iteration, we consider the patterns that are (1) on cell (x, y), (2) on column x, and (3) on row y of the checkerboard. To limit the search space, we only consider patterns that are within Δ-neighbourhood of (x, y), where Δ is an input parameter, typically set to a value less than five. Figure 10 illustrates the physical meaning of selecting patterns from the same cell, row, or column of the checkerboard.

After finding the candidate pattern set $\mathcal{T}$, we consider each P in $\mathcal{T}$, and randomly decide whether to accept or reject P. Here, the probability of *accepting* pattern P is set so that the expected number of escape patterns that can be selected from set $\mathcal{T}$ is equal to a fixed input parameter[4]. In other words, this probability is inversely proportional to the number of candidate patterns in $\mathcal{T}$. If P is *accepted*, then another recursive call is made starting from the current checkerboard cell. After all patterns in $\mathcal{T}$ are considered, a recursive call to cell $(x + 1, y + 1)$ is made to continue subsequence generation without selecting any pattern from the current level. the main purpose here is to have a good variety in the generated subsequences.

For the following complexity analysis, we assume that parameter V given in Properties 3.1 and 3.2, and all the slot capacities are constants (i.e. have complexity $O(1)$).

Lemma 4.4. *Let $\mathcal{R}$ be the recursion tree of the function GENERATE-SUBSEQ given in Figure 9. The following two properties hold for $\mathcal{R}$: (1) The maximum depth of $\mathcal{R}$ is $O(1)$. (2) The number of recursive calls made from a node in $\mathcal{R}$ is $O(1)$ on the average.*

PROOF. At each recursive call, either a pattern P is added to the partial subsequence, or the x and y coordinates are both incremented by 1. Since each subproblem consists of V rows, and escape slot capacities are constants, the maximum length of any subsequence is $O(1)$. Hence, the maximum recursion depth is $O(1)$. Furthermore, we randomly decide whether to accept or reject pattern P such that the expected number of patterns selected in each iteration is constant. As a result, the number of recursive calls made from a node in $\mathcal{R}$ is $O(1)$ on the average. □

[4]We have set the expected number of patterns that can be selected at each recursive iteration to 7 in our experiments. The execution time of the subsequence generation phase can be controlled by this parameter.

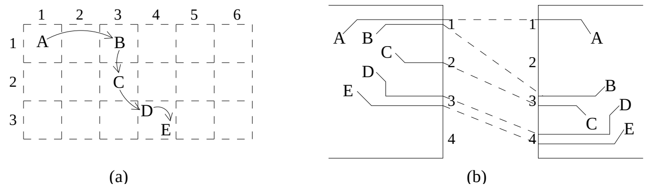

Figure 10: (a) A subsequence on the checkerboard, and (b) the corresponding escape patterns.

Lemma 4.5. *The recursive function GENERATE-SUBSEQ(x,y,subseq) is invoked only a constant number of times for each checkerboard cell (x, y).*

PROOF. Our proof is based on induction on the depth of the recursive tree $\mathcal{R}$. Obviously, the checkerboard cell at the root of $\mathcal{R}$ is called only a constant number of times (base case). Let us consider a grid cell (x, y), and let us assume that the induction hypothesis holds for all cells called before (x, y). From the algorithm of Figure 9, we know that only the cells that are in the Δ-neighbourhood of (x, y) can make a call to (x, y). Since Δ is constant, the lemma follows due to the induction hypothesis. □

Theorem 4.6. *The total average-time complexity of subsequence generation for all subproblems is $O(n + s^2)$, where n is the number of nets, and s is the number of escape slots on the component boundaries.*

PROOF. We will first prove that the average-time complexity for subproblem i is $O(n_i + s)$, where n_i is the number of patterns mapped to a cell within subproblem i. In one recursive call, all patterns P mapped to cells in the Δ-neighborhood of cell (x, y) are processed to determine set $\mathcal{T}$. Since Δ is constant, and due to Lemma 4.5, each pattern is processed only a constant number of times. Furthermore, the average number of nodes in a recursion tree $\mathcal{R}$ is $O(1)$, due to Lemma 4.4. Since there are s separate recursion trees (each root corresponding to a cell on the first row of the current subproblem), the average-time complexity for one subproblem is $O(n_i + s)$. Based on this, the total average-time complexity for all subproblems can be written as $\sum_{1 \leq i \leq s/V} O(n_i + s) = O(n + s^2)$. □

Theorem 4.7. *Let K denote the maximum number of subsequences that can be stored for each subproblem. The average time complexity for the proposed randomized planar route selection algorithm is $O(n + s^2 + K^2 s)$, where n is the number of nets, and s is the number of escape slots on component boundaries.*

PROOF. In graph $\mathcal{G}_R$ (defined in the beginning of this section), there is a vertex corresponding to each subsequence generated. Since there are $s/V = O(s)$ subproblems, the number of vertices in $\mathcal{G}_R$ is $O(Ks)$. The edges in $\mathcal{G}_R$ are only between vertices that correspond to adjacent subproblems. Hence, the number of edges in $\mathcal{G}_R$ is $O(K^2 s)$. Since, $\mathcal{G}_R$ is acyclic, computing the longest path has linear time complexity in the graph size [3], which is $O(K^2 s)$. As given in Theorem 4.6, the average time complexity of subsequence generation is $O(n + s^2)$; so the proof is complete. □

5. HANDLING HIGH-SPEED DESIGN CONSTRAINTS

In the following subsections, we discuss how to generalize the algorithms given in Sections 3 and 4 to handle different high speed design constraints.

5.1. Maximum Length Constraints

Board designers specify maximum length constraints for critical nets to limit the maximum arrival times. We can handle these constraints during pattern generation phase of our framework. Specifically, we can restrict the set of target escape slots (parameter $\mathcal{T}$ in Figure 4) such that the escape segments with long detours are avoided. Furthermore, remember that an escape pattern is created by merging two escape segments from the left and right components. It is possible to check the maximum length constraints during this step, and eliminate the patterns that violate the corresponding constraints.

5.2. Minimum Length Constraints

Minimum length constraints are typically enforced for nets belonging to a bus structure, with the objective of matching the signal arrival times. Recently, routing algorithms have been proposed to handle these constraints [8, 9, 11]. Typically, the length of a *short* net need to be extended to satisfy its min-length constraint. Since the routing resources within components are extremely limited, it makes more sense to perform length extension in the intermediate area between components, in a later stage of the routing system. However, we can also modify our randomized planar route selection algorithm (Section 4) such that the patterns that satisfy min bounds are preferred over the others. For this purpose, we can assign a weight to each escape pattern, based on its length and the corresponding min length constraint. Then, the randomized algorithm given in Section 4 can be used to select the permissible pattern sequence with the largest weight.

5.3. Adjacency Constraints for Noise Avoidance

Adjacency constraints between different nets are defined by designers to avoid crosstalk problems. A typical adjacency constraint between nets n_i and n_j can be stated as follows [7]: *If n_i and n_j are routed adjacent to each other on the same layer, then their routes need to be separated by at least k routing tracks.* Such a constraint is enforced typically on signal nets that belong to different bus structures. In the context of the model defined in Section 4.1, we can restate this constraint as follows: *If the patterns corresponding to n_i and n_j are adjacent in a permissible pattern sequence $\mathcal{S}$, then the escape slots of these patterns need to be separated by at least k routing tracks.* This constraint can be handled effectively in the subsequence generation algorithm given in Figure 9 by comparing the last pattern in the partial subsequence $subseq$ with the candidate pattern P. Specifically, the following line needs to be added immediately after set $\mathcal{T}$ is defined in Figure 9:

$\mathcal{T} \leftarrow \mathcal{T} \cap \{P :$ if (P', P) has a k-adjacency constraint, then there are k empty tracks between P' and P in both left and right components

By adding this line, we make sure that only the subsequences that do not violate adjacency constraints are generated. In addition, we also need to check these constraints for subsequences in neighbouring subproblems. Specifically, we need to add the following rule while defining the edges of $\mathcal{G}_R$ in Figure 6:

Let v_j^i and v_k^{i+1} denote two vertices in $\mathcal{G}_R$ corresponding to subsequences S_j^i and S_k^{i+1}, which have been generated in subproblems i and $i+1$, respectively. Let P_j^i denote the last pattern in subsequence s_j^i, and P_k^{i+1} denote the first pattern in subsequence S_k^{i+1}. If (P_j^i, P_j^{i+1}) have an adjacency constraint of at least k tracks, then an edge from v_j^i to v_k^{i+1} (in $\mathcal{G}_R$) exists only if there are at least k tracks between P_j^i and P_k^{i+1} in both components.

These two modifications are sufficient to ensure that the output of our algorithms satisfy all adjacency constraints.

5.4. Differential Pairs

A differential pair is a complementary pair of nets that provide noise immunity. The two nets within a differential pair need to be routed parallel to each other, separated by a specific distance as long as possible. Let us consider two nets n_i and n_j that belong to a differential pair. During pattern generation, we can identify the pairs of escape segments corresponding to n_i and n_j that adhere to these constraints. In the context of the model defined in Section 4.1, a pattern corresponding to n_i can exist in a permissible sequence $\mathcal{S}$ only if it is *adjacent* to an acceptable segment of n_j. This constraint can be explicitly checked in the subsequence generation algorithm of Figure 9 by comparing the last pattern in the partial subsequence $subseq$ with the candidate pattern P. Specifically, the following code segment needs to be added immediately after set $\mathcal{T}$ is defined in the algorithm of Figure 9:

Let P'' be the second-to-last pattern in subseq
 if P' belongs to a differential pair AND
  (P'', P') is not a differential pair then
   $\mathcal{T} \leftarrow \mathcal{T} \cap \{P : (P', P)$ is a differential-pair$\}$

By adding these lines, we make sure that patterns belonging to a differential pair always occur together in a subsequence. However, we also need to check differential pairs that are in two adjacent subproblems. For this purpose, we need to add the following rule while defining the edges of $\mathcal{G}_R$ in Figure 6:

Let P_{j-1}^i and P_j^i denote the second-to-last and last patterns in subsequence s_j^i; let P_k^{i+1} denote the first pattern in subsequence S_k^{i+1}. Assume that P_j^i is part of a differential pair, and (P_{j-1}^i, P_j^i) is not a differential pair. If this is the case, then an edge from v_j^i to v_k^{i+1} (in $\mathcal{G}_R$) exists only if (P_j^i, P_k^{i+1}) is a differential pair.

These two modifications are sufficient to handle the differential pair constraints.

6. EXPERIMENTAL RESULTS

We have performed experiments on escape problems extracted from a real industrial board design, for which current industrial tools fail to produce a routing solution. We have implemented our algorithms in C++, and performed the experiments on an Intel Pentium 4 2.4GHz system with 1GB memory, and a Linux operating system. The input parameter V given in Properties 3.1 and 3.2 is set to 4 in our experiments.

As mentioned in Section 4.1, the optimal algorithm for *maximal planar route selection* has a high time complexity, and it is

Table 1: Comparison of our framework with a recently proposed approach

Input	# layers	# nets	OUR ALGORITHM nonplanar nets	OUR ALGORITHM time (m:s)	ALGORITHM IN [10] nonplanar nets	ALGORITHM IN [10] time (m:s)
IBM1	5	426	25	1:56	50	0:37
IBM2	5	428	35	0:58	44	0:30
IBM3	4	352	22	1:04	48	0:33
IBM4	5	312	47	1:22	68	0:54
IBM5	3	226	6	0:25	18	0:18
IBM6	5	441	35	1:01	50	0:32

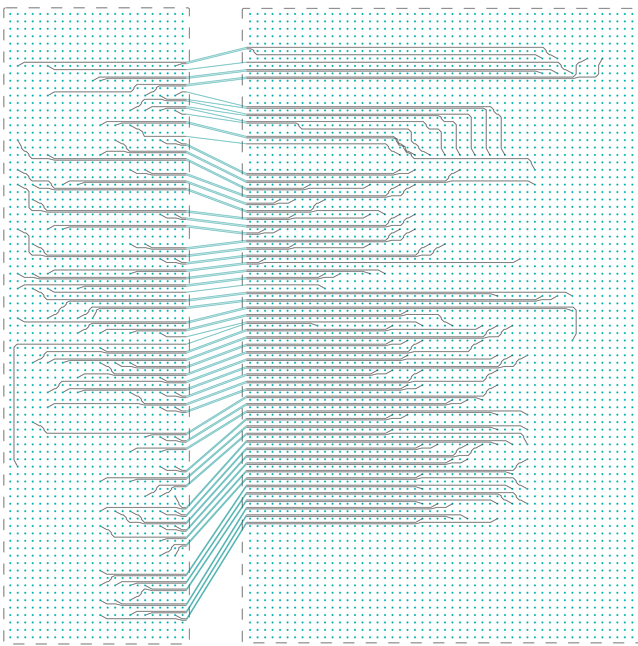

Figure 11: A planar escape routing solution is illustrated for two components. 111 nets have been routed on this layer. The connections in the intermediate area are shown as straight lines between components.

impractical for large circuits. Our experiments on relatively small sized problems indicated that the solutions given by our randomized algorithm are within about 3% of the optimal solution in terms of the number of planar routes selected. However, the optimal algorithm was not applicable to larger problems due to its large time complexity. For this reason, we have used the algorithm proposed in [10] for comparison with the algorithm we propose in this paper. Table 6 gives the results obtained on industrial test cases. As mentioned before, layers are processed one by one, and the maximal planar routing solution is found for each layer. The number of nets that could not be routed in a planar fashion is given for each problem under columns *nonplanar nets*. These nets will be distributed to available layers later, allowing crossings in the intermediate channel. As discussed before, a *crossing* net will need to use a via during the later stages of the routing system. The results in Table 1 indicate that our algorithm reduces the via requirements on the average by 39%, for the given industrial test cases. A sample output of our maximal planar routing algorithm for one layer is illustrated in Figure 11.

7. CONCLUSIONS

We have proposed an algorithm to solve the escape routing problem in multiple components simultaneously. Compared to the algorithm proposed in [10], our main contributions can be summarized as follows. First, we propose a more intelligent pattern generation algorithm that is based on congestion levels in the components, and the number of crossings in the intermediate area. Then, we propose a more sophisticated randomized algorithm for the maximal planar routing problem. We also show how to handle typical high speed design constraints within the framework of this algorithm. Our experiments show that our algorithm can reduce the via requirements significantly.

8. REFERENCES

[1] W. D. Brown. *Advanced Electronic Packaging with Emphasis on Multichip Modules*. IEEE Press Series on Microelectronic Systems, 1999.

[2] W.-T. Chan, F. Y. L. Chin, and H.-F. Ting. Escaping a grid by edge-disjoint paths. In *Proc. of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 726–734, 2000.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

[4] H. Harrer, H. Pross, W. T.-M, W. D. Becker, H. I. Stoller, M. Yamamoto, S. Abe, B. J. Chamberlin, and G. A. Katopis. First- and second-level packaging for the IBM eServer z900. *IBM Journal of Research and Development*, 46:397–420, 2002.

[5] J. Hershberger and S. Suri. Efficient breakout routing in printed circuit boards. In *Proc. of the 13th Annual Symposium on Computational Geometry*, pages 460–462, 1997.

[6] E. H. Laine and P. M. O'Leary. IBM chip packaging roadmap. In *Proc. of Int'l. Packaging Strategy Symposium, SEMICON West*, 1999.

[7] J. Ludwig. IBM Systems Group. Private communication, 2004.

[8] M. M. Ozdal and M. D. F. Wong. Length matching routing for high-speed printed circuit boards. In *Proc. of IEEE/ACM Intl. Conf. on Computer-Aided Design (ICCAD)*, Nov. 2003.

[9] M. M. Ozdal and M. D. F. Wong. A provably good algorithm for high performance bus routing. In *Proc. of IEEE/ACM Intl. Conf. on Computer-Aided Design (ICCAD)*, Nov. 2004.

[10] M. M. Ozdal and M. D. F. Wong. Simultaneous escape routing and layer assignment for dense pcbs. In *Proc. of IEEE/ACM Intl. Conf. on Computer-Aided Design (ICCAD)*, Nov. 2004.

[11] M. M. Ozdal and M. D. F. Wong. A two-layer bus routing algorithm for high-speed boards. In *Proc. of IEEE Intl. Conf. on Computer Design (ICCD)*, Oct. 2004.

[12] A. Titus, B. Jaiswal, T. Dishongh, and A. N. Cartwright. Innovative circuit board level routing designs for bga packages. *IEEE Trans. on Advanced Packaging*, 27, 2004.

[13] T.-M. Winkel, W. D. Becker, H. Harrer, H. Pross, D. Kaller, B. Garben, B. J. Chamberlin, and S. A. Kuppinger. First- and second-level packaging of the z990 processor cage. *IBM Journal of Research and Development*, 48:379–394, 2004.

[14] M. Yu and W. W. Dai. Single-layer fanout routing and routability analysis for ball grid arrays. In *Proc. of Intl. Conf. on Computer-Aided Design (ICCAD)*, pages 581–586, 1995.