An Efficient Zero-Skew Routing Algorithm

Masato Edahiro
C&C Research Laboratories, NEC Corporation
Miyazaki, Miyamae-ku, Kawasaki 216, Japan
Department of Computer Science, Princeton University
Princeton, NJ 08544-2087, USA

Abstract— **A bucket algorithm is proposed for zero-skew routing with linear time complexity on the average. Our algorithm is much simpler and more efficient than the best known algorithm which uses Delaunay triangulations for segments on Manhattan distance. Experimental results show that the linearity of our algorithm is accomplished. Our algorithm generates a zero-skew routing for 3000-pin benchmark data within 5 seconds on a 90MIPS RISC workstation.**

I. Introduction

With scaling down the dimensions of devices, zero-skew clock routing technique becomes more critical for high performance VLSI's because wiring delay grows comparable to (or even dominates) gate switching delay. Even though exact zero skew was achieved [11], more efficient delay minimization algorithms are still desired.

There have been four types of algorithms proposed for the total-wire-length minimization in zero-skew routing. Matching-based [3] and partitioning-based algorithms [2] were the first practical algorithms for this problem whose time complexity is $O(n \log^2 n)$. A bottom-up construction algorithm [8] reduced the total wire length by 15%, though its time complexity is $O(n^2 \log n)$. A clustering-based algorithm [4] improved the time complexity to $O(n \log n)$ without any increase of the total wire length. Thus, the clustering-based algorithm is currently the most efficient in a theoretical sense. In practice, however, the clustering-based algorithm might not be efficient because it is required to construct Delaunay triangulations for segments on Manhattan distance, for which no practical algorithm has been proposed.

As for the total delay minimization, Edahiro [5] recently proposed delay time estimation and optimum wire-sizing methods. He proposed a minimum-delay zero-skew routing algorithm by combining these methods with the clustering-based algorithm [4].

In this paper, we propose a zero-skew routing algorithm that is much simpler and practically more efficient than the clustering-based algorithm. The time complexity of our algorithm is linear on the average. In this algorithm, we introduce a relation graph to represent the nearness between nodes on clock trees. In addition, we define an independent edge set on the relation graph that is used to construct minimum-delay zero-skew routings in a bottom-up fashion.

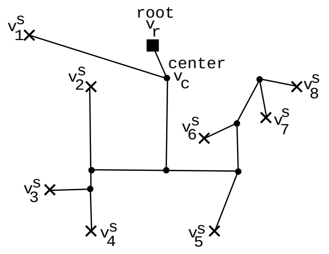

Figure 1: Clock Tree with a root v_r and leaves $S = \{v_1^s, v_2^s, \ldots, v_8^s\}$.

We use a bucket algorithm to construct relation graphs. The bucket algorithm finds 'nearness' between nodes in linear time on the average. In order to search the independent edge set, a depth first traverse is utilized on the relation graphs, which is performed in linear time. As a result, linearity of our algorithm is accomplished.

Experimental results show that, while the delay time increases 3% in our algorithm compared with the best known algorithm [4, 5], our algorithm is much simpler and more efficient. Our algorithm generates a zero-skew routing for 3000-pin benchmark data within 5 seconds on a 90MIPS RISC workstation.

II. Zero-Skew Routing

Given a fan-out terminal v_r and a set of n fan-in terminals $S = \{v_1^s, v_2^s, \ldots, v_n^s\}$, a *clock tree* is defined by a tree rooted by v_r whose n leaves are S (Fig. 1). We call the fan-out terminal *root* and fan-in terminals *leaves*. A set of leaves in the subtree rooted by a node v is called *leaves connecting to* v and denoted by S_v. In this paper, clock trees are always binary, though nodes may degenerate and they do not look binary in some cases. Also, we call the nearest internal node to the root the *center* denoted by v_c.

We assume that the *load capacitance* $C(v_i^s)$ is given for each leaf v_i^s, which is usually the gate capacitance of transistors associated with the leaf. Also, the *load capacitance* $C(v)$ for an internal node v is defined by the total capacitance in S_v that includes wire capacitance as well as gate capacitance.

Then, a *zero-skew routing* for the given root and leaves is defined by a clock tree in which all delay time from the root to all leaves is equal. From this definition, it is clear that, for any node in the zero-skew routing, all delay from the node v to leaves in S_v should be equal, which is called *delay time* $\tau(v)$ *for* v. For leaves v_i^s, $\tau(v_i^s) = 0$.

An exact zero-skew routing can be constructed in a bottom-up fashion by repeatedly calculating the position of an internal node v from the positions of children v_1 and v_2 of v using the following equations derived from π-model [5]:

$$\begin{aligned}\tau(v) &= \frac{rl_1}{w_1}\left(\frac{cl_1w_1}{2} + C(v_1)\right) + \tau(v_1) \\ &= \frac{rl_2}{w_2}\left(\frac{cl_2w_2}{2} + C(v_2)\right) + \tau(v_2), \\ C(v) &= C(v_1) + C(v_2) + c(l_1w_1 + l_2w_2),\end{aligned}$$

where l_1 (l_2) and w_1 (w_2) are length and width of the wire from v to v_1 (v_2), and r and c are wire resistance and capacitance for an unit length and width wire.

This operation to determine the location of a node v is called the *zero-skew merge*. At a zero-skew merge, it is clear that $l_1 + l_2 \geq l$, where l is the distance between v_1 and v_2. Therefore, in order to minimize the delay time, it is desired to find v such that $l_1 + l_2 = l$. If there is no such a v, zero-skew routing algorithms should use a *detour*. Although the number of detours depends on the algorithms and the input data, detours hardly appear in actual layouts on 'good' algorithms.

Note that, in Manhattan distance, a set of feasible points for v satisfying the above equations forms a diagonal segment in general. This segment (or point for a leaf) is called the *segment for* v or simply v. The segment for v can be calculated in constant time even if children are also expressed by diagonal segments [2, 6].

III. Definitions

In this section, we define the relation graph and the independent edge set on the relation graph, which play important roles in our algorithm.

A. Relation Graph

First, we define the relation graph with a relation function $f(v_i, v_j)$ of segments v_i and v_j. Examples of relation functions will be described later. Let K be a set of segments. The *relation graph* $G(K, E)$ for K is defined by the weighted directed graph such that

i) Each node $v \in K$ has exact one out-going edge, so that $|E| = |K|$,

ii) If $(v_i, v_j) \in E$ for $v_i, v_j \in K$, $\forall v_k \in K - \{v_i\}$, $f(v_i, v_j) \leq f(v_i, v_k)$ (if there is a tie, one of the tie edges is arbitrarily selected),

iii) Each edge $e = (v_i, v_j) \in E$ has a weight $w(e) = f(v_i, v_j)$.

An example of relation graph is depicted in Fig. 2. As we explain in the following sections, some edges in relation graphs are selected at each step of our algorithm, and, for each selected edge, its two endpoints are zero-skew merged. We say that the zero-skew merge is *associated with* the edge.

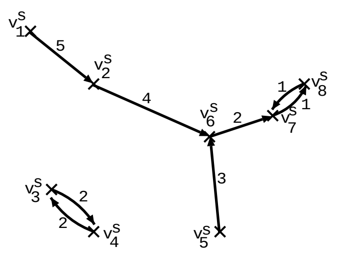

Figure 2: Relation Graph for $K = \{v_1^s, \ldots, v_8^s\}$ in Fig. 1.

We use two types of relation function. The *nearest-neighbor relation function* calculates the Manhattan distance between segments for v_i and v_j. Note that the relation graph with this function is equivalent to the *nearest neighbor graph* [4, 9]. Since the distance between segments for v_i and v_j turns out the increase of the total wire length when v_i and v_j are zero-skew merged, the nearest neighbor relation tries to minimize the total wire length.

The other is called the *minimum-delay relation function.* Since the total delay t_d is well-estimated by the following formula [5]:

$$t_d \approx 1.85\frac{C(v_r)}{\beta V_{DD}} + 0.7\tau(v_r),$$

where β is the MOS transistor gain factor, we define the minimum-delay relation function by:

$$f(v_i, v_j) = 1.85\frac{C(v)}{\beta V_{DD}} + 0.7\tau(v),$$

where v is the segment obtained by the zero-skew merge between v_i and v_j. It is clear that this relation function tries to minimize the total delay. Note that it is not difficult that the minimum-delay relation function includes the wire width optimization technique proposed in [5].

In this paper, we analyze properties of the relation graphs and our routing algorithm only for the nearest-neighbor relation function because the minimum-delay relation function is too complicated to look into. Fortunately, since short wire length tends to cause short delay time, these two relation functions have similar behavior in zero-skew routing algorithms.

Now, we show a property for the nearest neighbor graph [4]. Let the sequence $\{e_0, e_1, \ldots, e_{|K|-1}\}$ be the edges in E sorted by their weights in non-decreasing order. Then, the following property characterizes the weight of e_i.

Property 1 *For* $0 \leq \forall i < |K| - 1$,

$$w(e_i) \leq \frac{D}{\sqrt{|K| - i - 1}},$$

where D *is the diameter of* K*.*

B. Independent Edge Set

In a relation graph, two edges are called *dependent* if these edges share an endpoint. Also, an edge set is called

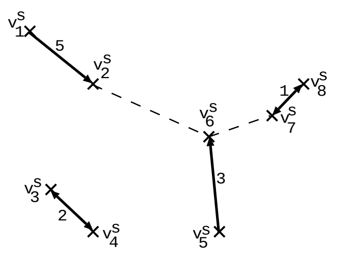

Figure 3: Independent Edge Set for Fig. 2 (solid lines).

independent if no two edges in the set are dependent. An example of the independent edge set is shown in Fig. 3.

Since zero-skew merges associated with dependent edges cannot be implemented at the same time, we need to select an independent edge set from a relation graph to minimize the total wire length or the total delay. We describe the selection algorithm in the next section.

IV. Algorithm

In this section, we propose a zero-skew routing algorithm with linear time complexity on the average. In our algorithm, we use a parameter $k > 1$. Typically, $2 \leq k \leq 5$. Also, we use a function $s(a, b, c)$ where $a \leq c$ defined by

$$s(a,b,c) = \begin{cases} a; & a \geq b, \\ b; & a < b < c, \\ c; & b \geq c. \end{cases}$$

First, we outline our algorithm. Let K be a subset of nodes in the zero-skew routing we are constructing. Initially, $K \equiv S$. At each step of our algorithm, after constructing a relation graph for K, an independent edge set on the relation graph is selected. Then, zero-skew merges between v_i and v_j are applied for all edges (v_i, v_j) in the set, and K is updated by deleting v_i and v_j and inserting v that is generated by the zero-skew merge associated with (v_i, v_j). Repeating this procedure several times, the position of the center v_c is determined.

After that, a clock routing is generated by embedding wires in a top-down fashion using positions calculated above.

It is important to note that, as discussed in [4], applying zero-skew merges associated with all edges in an independent edge set could cause longer total wire length and/or total delay time. They pointed out that it is a better idea to limit only $|K|/k$ smallest-weight edges in the relation graph. This technique is also implemented in our algorithm.

Let us take a simple example in Fig. 4. We assume $k = 2$. For a set of leaves S (Fig. 4 (a)), the relation graph is depicted in Fig. 4 (b), in which weight $w(e)$ is attatched on each edge e. An independent edge set is shown in Fig. 4 (c). Zero-skew merges are implemented for two edges in the independent set because of the limit $k = 2$. The new segments are v_9 from (v_3^s, v_4^s) and v_{10} from (v_7^s, v_8^s) (Fig. 4 (d)). K is updated to $\{v_1^s, v_2^s, v_5^s, v_6^s, v_9, v_{10}\}$, and v_9 (v_{10}) becomes a father of v_3^s and v_4^s (v_7^s and v_8^s) in the zero-skew routing we are constructing (Fig. 4 (e)). Then, a relation graph is reconstructed for the updated K (Fig. 4 (f)). After repeating this operation, the segment for v_c is obtained. Figure 4 (g) shows all calculated segments, and their tree structure is depicted in Fig. 4 (h).

Then, we first draw a wire from v_r to the nearest position on the segment for v_c. Drawing all wires from father to children on the clock tree in a top-down fashion, a zero-skew routing is generated (Fig. 4 (i)).

In the following sections, we describe our algorithm in detail.

A. Relation Graph Construction

In the algorithm proposed in [4], the nearest neighbor graph is constructed from a Delaunay triangulation [9]. However, calculating Delaunay triangulations seems expensive in practice. In this section, we propose a new construction method using the bucketing technique, which accomplishes linear time complexity on the average.

(1) Bucket Decomposition

Buckets are spatial partitioning by meshes. In our algorithm, all segments in a node set K are distributed in $\Theta(|K|)$ buckets. Since we need to store diagonal segments in buckets, we use buckets partitioned by diagonal lines in our implementation. An example of the bucket decomposition for Fig. 4 (a) is shown in Fig. 5.

The bucket size is determined by the following way. Let D be the diameter of K, that is, the distance between the farthest pair of elements in K. Also, let $d = \sqrt{s(2, (1 - 1/k)|K| + 1, |K|)} - 1$. Then, the size of a bucket is calculated by $(D/d) \times (D/d)$.

There are two significant characteristics in this partitioning method. First, on this partition, we have only to check nine buckets for a portion of each segment v_i inside a bucket to find an edge (v_i, v_j) that is possible to belong to $|K|/k$ smallest-weight edges in the nearest neighbor graph. This is simply proven. By Property 1, for the $(|K|/k)$-th smallest-weight edge e in the nearest neighbor graph, the upper bound of $w(e)$ is D/d. This means that we do not need to check any segment whose distance from v_i is more than D/d. Since the bucket size is $(D/d) \times (D/d)$, for any portion of segment inside a bucket, we have only to check nine buckets.

The second characteristic is that the number of buckets is $\Theta(|K|)$. This is also proven easily. From the definition of buckets, the number of buckets is d^2, which is $\Theta(|K|)$.

Now, we assume that segments in K are uniformly distributed, and that length of any segment in K is not very long so that each segment intersects with $O(1)$ buckets. Note that these assumptions are satisfied for benchmark data [7, 11]. Under these assumptions, it is easy to see that each bucket contains $O(1)$ segments in K on the average, and it requires only $O(1)$ time for a segment v_i in K to find an edge (v_i, v_j) that is possible to belong to $|K|/k$ smallest-weight edges in the nearest neighbor graph.

(2) Relation Graph Construction

Next, a relation graph is constructed using the bucket decomposition described above. In our algorithm, however, we do not construct an entire relation graph, but generate its subgraph that includes at least $|K|/k$ smallest-weight edges in the nearest neighbor graph. This

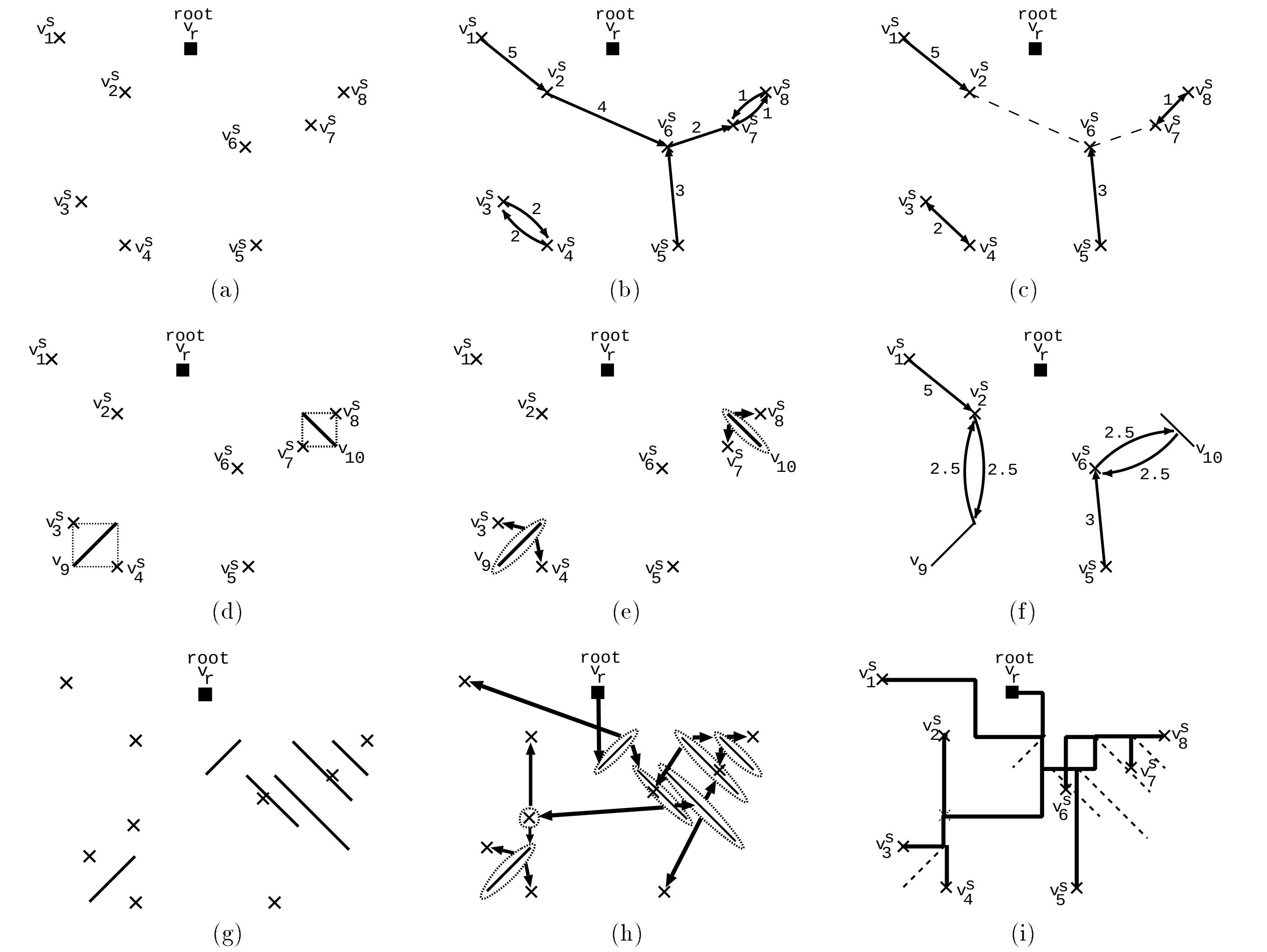

Figure 4: Example of Our Algorithm.

subgraph is sufficient for our purpose because only $|K|/k$ smallest-weight edges are necessary in our algorithm as we have discussed above.

In order to calculate the subgraph, we first construct the bucket decomposition. Then, for each segment $v_i \in K$, we find a segment v_j, which minimizes the relation function $f(v_i, v_j)$ among all segments in K contained in nine buckets around v_i. From the above discussion, the following property is clear.

Property 2 *Our bucket algorithm requires $O(|K|)$ time on the average to construct a subgraph of the nearest neighbor graph for K, which includes at least $|K|/k$ smallest-weight edges in the nearest neighbor graph.*

B. Independent Edge Set Selection

In this section, we describe a linear-time algorithm to select an independent edge set from a relation graph. It is desired that edge weights in the independent set are as small as possible. First, we describe a simple algorithm whose complexity is not linear.

(1) Basic Algorithm

In this algorithm, we first sort all edges in the relation graph for K in non-decreasing order. Also, let I be a set of edges and initially $I = \phi$. Then, edges are taken one by one in the sorted order. If an edge is independent from all edges in I, add it to I; and otherwise, the edge is just discarded. After checking $|K|/k$ smallest-weight edges, an independent edge set has been stored in I. We call the independent edge set by this algorithm the *canonical independent set*. Fig. 4 (c) is the canonical independent set on the relation graph in Fig. 4 (b) for K.

Note that the time complexity for this algorithm is $O(|K| \log |K|)$ for sorting.

(2) Linear Time Algorithm

Next, we propose a linear time algorithm to find the canonical independent set. Before explaining our algorithm, we state the condition for the canonical independent set.

Property 3 *Edge $e = (v_i, v_j)$ is in the canonical independent set if no edge e' of $w(e') \leq w(e)$ incident to v_i or v_j is in the canonical independent set.*

Now, we propose our algorithm to find the canonical independent set. Our algorithm uses a depth first search. In our algorithm, each edge e has a flag $flag(e)$ initialized by $UNSEEN$. If an edge e is to be in the canonical independent set, $flag(e)$ is set to YES, and if not, $flag(e)$ is set to NO.

```
Algorithm Find_Independent_Set
Step 1: For all $v \in K$, $check\_dependency(v, NULL)$.
Step 2: Canonical Independent Set := {All edges with $YES$
    flag}.
```

```
procedure check_dependency(v, e0)
Step 1: If $e0 \neq NULL$ AND $flag(e0) \neq UNSEEN$, return.
Step 2: Take a smallest-weight edge $e$ with $flag(e) \neq NO$
    among all edges incident to $v$.
Step 3: If $e = e0$, $flag(e0) = YES$ and return.
Step 4: $check\_dependency(v', e)$, where $v'$ is the other end-
    point of $e$.
Step 5: If $flag(e) = YES$, $flag(e0) = NO$ and return.
Step 6: Go to Step 2.
```

As we discussed above, we have only to check $|K|/k$ smallest-weight edges. In order to implement it, we need to calculate the weight of the $(|K|/k)$-th smallest edge. There is a theoretically linear-time algorithm [1] for this calculation, but another algorithm shown in [10] is practically more efficient. Once the weight is known, our algorithm is easily modified for the weight.

Next, we analyze the time complexity of this algorithm. Since the algorithm traverses the relation graph in a depth-first fashion, it is clear that the time complexity is linear if sorting in Step 2 of *check_dependency* can be performed in a constant time. In order to prove it, the following property is useful [4].

Property 4 *At most c_e edges are in-coming to any node in the nearest neighbor graph, where c_e is a constant.*

By this property, the number of edges to be sorted in Step 2 of *check_dependency* is constant. As a result, it is proven that our algorithm has linear time complexity.

Property 5 *Our algorithm requires $O(|K|)$ time to find a canonical independent set on the nearest neighbor graph for K.*

C. Zero-Skew Routing Algorithm

Now, we present our linear-time zero-skew routing algorithm. Let K be a set of segments. Initially, $K = S$. Our algorithm has two phases, *Find_Center* and *Embedding*. In *Find_Center*, segments for all internal nodes are calculated in a bottom-up fashion, and then, in *Embedding*, the best position of each node is determined in a top-down fashion.

The algorithm is stated as follows:

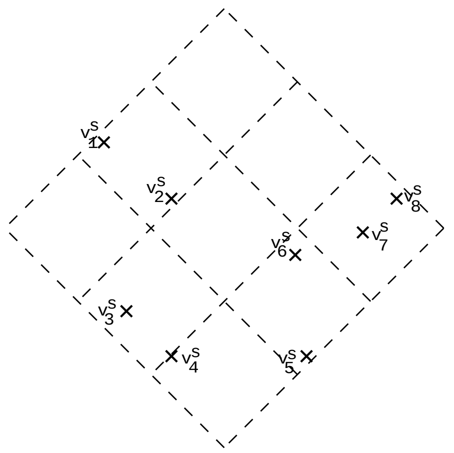

Figure 5: Bucket Decomposition for $K = \{v_1^s, \ldots, v_8^s\}$ in Fig. 4 (a).

```
Algorithm Find_Center
Step 1: $K := S$.
Step 2: If $|K| = 1$, stop.
    (The element in $K$ is the segment for the center $v_c$.)
Step 3: Let $D$ be the diameter of $K$, and let
    $d = \sqrt{s(2,(1-1/k)|K|+1,|K|)} - 1$.
    Construct buckets of size $(D/d) \times (D/d)$ for $K$.
Step 4: Construct a relation graph $G(K, E)$ on $K$.
Step 5: Calculate $i$-th minimum value $v^*$ of $w(e)$ among
    $\forall e \in E$, where $i = s(1, |K|/k, |K|-1)$.
Step 6: Select the canonical independent set from $\{e \in$
    $E | w(e) \leq v^*\}$.
Step 7: Apply zero-skew merges associated with all edges in
    the canonical independent set.
Step 8: Go to Step 2.
```

```
Algorithm Embedding
Step 1: Determine the center $v_c$ by selecting the nearest point
    to the root $v_r$ on the segment for $v_c$. Route from $v_r$ to
    $v_c$.
Step 2: $local\_embedding(v_c)$
```

```
procedure local_embedding(v)
Step 1: If $v$ has no child, return.
Step 2: Let $v_1$, $v_2$ be the children of $v$. For $i = \{1, 2\}$,
    determine a point $v_i$ on the segment for $v_i$ so as to satisfy
    the zero-skew merge equations. Route from $v$ to $v_i$.
Step 3: $local\_embedding(v_1)$, $local\_embedding(v_2)$.
```

It is clear that the time complexity of Algorithm *Embedding* is linear. Also, from the analysis in the previous sections, Steps 3-7 in Algorithm *Find_Center* are performed in $O(|K|)$ time on the average for K. Since the degree of nodes in relation graphs is constant (Property 4), the size of the canonical independent set is $\Theta(|K|)$

Table 1: Total Delay Time [nsec] for three algorithms.

	#pins	part. [2]	clst. [4, 5]	proposed
prim1	269	6.56	5.60	5.62
prim2	603	16.99	12.77	12.84
r1	267	2.49	1.91	1.95
r2	598	5.61	4.06	4.30
r3	862	7.71	5.40	5.57
r4	1903	18.58	11.58	11.69
r5	3101	31.48	17.60	18.10

Table 2: CPU Time [sec] for three algorithms. (Clustering-based algorithm [4, 5] has $O(n^2)$ time complexity because we did not implement Delaunay triangulation algorithm.)

	#pins	part. [2]	clst. [4, 5]	proposed
prim1	269	0.42	0.90	0.25
prim2	603	1.92	5.03	0.70
r1	267	0.18	0.87	0.27
r2	598	0.52	4.63	0.68
r3	862	0.82	9.72	1.05
r4	1903	2.20	48.1	2.55
r5	3101	3.77	127.6	4.08

for K. Therefore, after zero-skew merges associated with a canonical independent set for a set K, the size of the resultant set is $O(c|K|)$ where c is a constant less than 1. As a result, we obtain the linearity of our algorithm.

Lemma 1 *Our zero-skew routing algorithm requires $O(n)$ time on the average.*

V. Experimental Results

Now, we show experimental results with benchmark data prim1-prim2 [7] and r1-r5 [11]. In this experiment, we compared our algorithm with the partitioning-based [2] and the clustering-based algorithms [4, 5]. The clustering-based algorithm [4, 5] is currently the best algorithm in the total delay time and theoretically the most efficient in the time complexity. However, this algorithm might not be efficient in practice because the algorithm needs to construct Delaunay triangulations for segments on Manhattan distance, for which no practical algorithm has been proposed. On the other hand, the partitioning-based algorithm [2] is the most practical in the sense of execution time, though the total wire length is much longer compared with the clustering-based algorithm [4, 5].

In our algorithm, we used the minimum-delay relation function and the parameter $k = 4$. The wire-width optimization proposed in [5] was used in the clustering-based algorithm [4, 5] and our algorithm.

Tables 1 and 2 show the total delay time and CPU time for the experiment. The total delay time is estimated by the formula in [5] that was justified using SPICE simulation. CPU time was measured on a 90MIPS RISC workstation. Note that our implementation for the clustering-based algorithm [4, 5] has $O(n^2)$ time complexity because we did not implement the Delaunay triangulation algorithm, though the total delay time on zero-skew routings generated by our program for the algorithm [4, 5] is equal to that reported in [5].

It is observed in Table 1 that our algorithm is 10%-40% better than the partitioning-based algorithm [2] for the total delay time. Also, it is important to note that Table 2 indicates that the time complexity of our algorithm is linear, while that of the partitioning-based algorithm [2] is a little more than $O(n)$ (theoretically, $O(n \log^2 n)$). Although CPU time for the clustering-based algorithm [4, 5] shows $O(n^2)$ time complexity, we believe that our algorithm is still much more efficient than $O(n \log n)$ algorithm because the algorithm needs to construct Delaunay triangulations several times. Tables show that our algorithm generates a zero-skew routing for r5 (3101 pins) within 5 seconds with only 3% increase of the total delay time compared with the best known algorithm [4, 5].

VI. Conclusions

We have proposed a zero-skew routing algorithm with linear time complexity on the average. In order to achieve linearity, relation graphs are constructed using bucketing technique, and independent edge sets are found by a depth first search. This algorithm is much simpler and more efficient than the best known algorithm which uses Delaunay triangulations for segments on Manhattan distance. Experimental results show that our algorithm generates a zero-skew routing for 3000-pin benchmark data within 5 seconds on a 90MIPS RISC workstation.

Acknowledgment

The author would like to thank A. B. Kahng and K. D. Boese of UCLA and R. S. Tsay of ArcSys Inc. for providing benchmark data, and R. J. Lipton of Princeton University for valuable discussion.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1976.

[2] T. H. Chao, Y. C. Hsu, J. M. Ho, K. D. Boese, and A. B. Kahng, "Zero skew clock routing with minimum wirelength," *IEEE Trans. on CAS II*, Vol. 39, pp.799-814, 1992.

[3] J. C. Cong, A. B. Kahng, and G. Robins, "Matching-based methods for high-performance clock routing," *IEEE Trans. on CAD*, Vol. 12, pp.1157-1169, 1993.

[4] M. Edahiro, "A clustering-based optimization algorithm in zero-skew routings," *Proc. of 30th DAC*, pp.612-616, 1993.

[5] M. Edahiro, "Delay minimization for zero-skew routing," *Proc. of 1993 ICCAD*, pp.563-566, 1993.

[6] M. Edahiro, "Equi-spreading tree in Manhattan distance," unpublished.

[7] M. A. B. Jackson, A. Srinivasan, and E. S. Kuh, "Clock routing for high-performance ICs," *Proc. of 27th DAC*, pp.573-579, 1990.

[8] Y. M. Li and M. A. Jabri, "A zero-skew clock routing scheme for VLSI circuits," *Proc. of 1992 ICCAD*, pp.458-463, 1992.

[9] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[10] R. Sedgewick, *Algorithms in C.* Addison-Wesley, 1990.

[11] R. S. Tsay, "An exact zero-skew clock routing algorithm," *IEEE Trans. on CAD*, Vol. 12, pp.242-249, 1993.