DESIGN SPACE EXPLORATION FOR BUILDING-BLOCK PLACEMENTS

by

Henrik Esbensen and Ernest S. Kuh

Memorandum No. UCB/ERL M95/84

24 October 1995

DESIGN SPACE EXPLORATION FOR BUILDING-BLOCK PLACEMENTS

by

Henrik Esbensen and Ernest S. Kuh

Memorandum No. UCB/ERL M95/84

24 October 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

A genetic algorithm for building-block placement of ICs and MCMs is presented. Layout area, routing congestion and an Elmore-based estimate of the maximum path delay is minimized while trying to meet a target aspect ratio. The design space is explicitly explored by using a vector-valued, 4-dimensional cost function and searching for a set of distinct solutions representing the best tradeoffs of the cost dimensions. Designers can then choose from the output solution set. In contrast to existing approaches such as simulated annealing, no weights or bounds are needed, thereby eliminating the inherent problems of specifying these quantities. Furthermore, due to the explicit minimization of routing congestion, the need for iterations between placement and global routing is reduced. Promising experimental results are obtained for various placement problems.

1 Introduction

During placement of an integrated circuit (IC) or a multichip module (MCM) the objective is to find a solution which is satisfactory with respect to a number of competing criteria. Most often specific constraints has to be met for some criteria, while for others, a good tradeoff is wanted. However, at this point in the design process, the available information as to which values are obtainable for each criteria is based on relatively rough estimates only. Consequently, the designers notion of the overall design objective is rarely clearly definable.

Virtually all existing placement tools minimizes a weighted sum of some criteria subject to constraints on others. I.e., if k criteria are considered, the objective is to minimize the single-valued cost function

$$c = \sum_{i=1}^{j} w_i c_i \quad \text{subject to} \quad \forall\, i = j+1, \ldots, k : \; c_i \leq C_i \tag{1}$$

for some j, $1 \leq j \leq k$. Here c_i measures the cost of the solution with respect to the i'th criterion and the w_i's and C_i's are user-defined weights and bounds, respectively.

However, in practice it may be very difficult for the designer to specify a set of bounds and weights which makes the placement tool find a satisfactory solution. If the bounds are too loose, perhaps a better solution could have been found, while if they are too tight, a solution may not be found at all. Furthermore, it is far from clear how to derive a suitable set of weight values from the vaguely defined design objectives. An additional complication is that, depending on the nature of the c_i functions, the relative magnitude of the $w_i c_i$ terms may change during the optimization process itself, in which case constant weights are unlikely to keep the cost function properly balanced throughout the process.

Our work is motivated by the need to overcome these fundamental problems. A building-block placement algorithm for both ICs and MCMs is presented, which supports *explicit* design space exploration in the sense that 1) a set of alternative solutions rather than a single solution is generated by a single program execution, and 2) solutions are characterized explicitly by a cost value for each criterion instead of a single, aggregated cost value. The algorithm simultaneously minimizes layout area, routing congestion, maximum path delay and the deviation from a target aspect ratio. It searches for a set of alternative, good solutions where "good" is defined by the user in a simple manner. From the output solution set, the designer chooses a specific solution representing the preferred tradeoff. The use of both the weights and the bounds of (1) are avoided and consequently the above mentioned problems concerning weight and bound specification are eliminated.

The approach has three additional significant characteristics:
1) The maximum routing congestion is minimized, thereby improving the likelihood that the placement is routable without further modification. Consequently, the traditional need for multiple iterations of the placement and global routing phases is significantly reduced.
2) Despite the fact that delay is inherently path oriented, most existing timing-driven placement approaches are net-based. While simple, these approaches usually over-constrains the problem, thereby potentially excluding good solutions from being found [13]. The few existing path-based approaches includes [14, 16, 18], all of which, however, relies on very simple net models (stars

and bounding boxes). The approach presented here obtains a more accurate path delay estimate by approximating each net by an Elmore-optimized Steiner tree.
3) The approach is based on the genetic algorithm (GA), since it is particularly well suited for design space exploration in the above sense [12]. We are only aware of three previous GA approaches to building-block placement [6, 7, 11], none of which considers delay or routing congestion or performs explicit design space exploration. In fact, for CAD problems in general, existing work on design space exploration is very limited, although approaches for scheduling and channel routing are presented in [8].

The remaining of this paper is organized as follows. Section 2 presents the problem definition used. The algorithm is described in Section 3 and in Section 4 experimental results are presented. Conclusions are given in Section 5. The work presented in this paper is based on significant extensions and improvements of our earlier placement approach described in [10].

2 Problem Definition

The placement model described in Section 2.1 is relevant for both MCMs and for IC technologies with at least two metal layers available for routing. Section 2.2 characterizes the solution set searched for by the algorithm.

2.1 Placement Model

A placement problem is specified by the following input :

- A set of rectangular building-blocks of arbitrary sizes and aspect ratios with a set of pins located anywhere within each block.
- A set of IO-pins/pads. Constraints on relative IO-pin positions are expressed using a two-dimensional array $I_{u \times v}$ as illustrated in Fig. 1. Each IO-pin can be assigned to an entry of I and the physical location corresponding to entry (i, j) will be $(id_x/(u-1), jd_y/(v-1))$, where d_x and d_y are the horizontal and vertical dimensions of the layout, respectively[1]. An IO-pin assigned to I by the user is called a *fixed* IO-pin, while the remaining IO-pins are *flexible*. Each flexible IO-pin will automatically be assigned to a vacant entry of I not specified as illegal. Since any subset of the entries of I can be specified as illegal, pins can be restricted to placement along the periphery of the layout, they can be uniformly distributed over the entire layout, etc.
- A specification of all nets, including for each net 1) the capacitance of each sink pin, and 2) a designated source pin p, its driver resistance and an associated internal delay $t(p)$ in the block $m(p)$ to which p belongs. $t(p)$ is the time it takes a signal to travel through $m(p)$ to p.[2]

[1]Relative to the building-blocks, the entire set of IO-pins can be oriented and/or reflected in eight distinct ways, while still satisfying the constraints on relative positions specified by I. The given absolute position of entry (i, j) assumes that the IO-pin set is positioned on top of the blocks without changing neither the orientation nor the reflection relative to the blocks.

[2]If p is an IO-pin, $m(p)$ is p itself and if it is also a source, $t(p) = 0$, i.e., input IO-pins have no internal delay.

- A specification of a set of paths $\mathcal{P}$. A path connects either two registers of distinct blocks or an IO-pin and a register, i.e., it is an alternating sequence of wires passing through blocks and net segments. For a sink pin p, denote by $s(p)$ the source pin of the net to which p belongs. Each path $P \in \mathcal{P}$ is then uniquely specified by an ordered set of sink pins $P = \{p_0, p_1, \ldots, p_{l-1}\}$ of distinct nets, such that $m(p_i) = m(s(p_{i+1})), i = 0, 1, \ldots, l-2$. $s(p_0)$ or p_{l-1} may be an IO-pin. Each path in an MCM will have length $l = 1$ assuming that all signals are latched at the inputs of the components.

- Technology information : Number of metal layers available for routing on top of blocks and between blocks, denoted by l_{block} and l_{space}, respectively. The routing wire resistance $\bar{r}$ and capacitance $\bar{c}$ per unit wire length, and the wire pitch w_{pitch}. For simplicity, these values are all assumed to be constants.

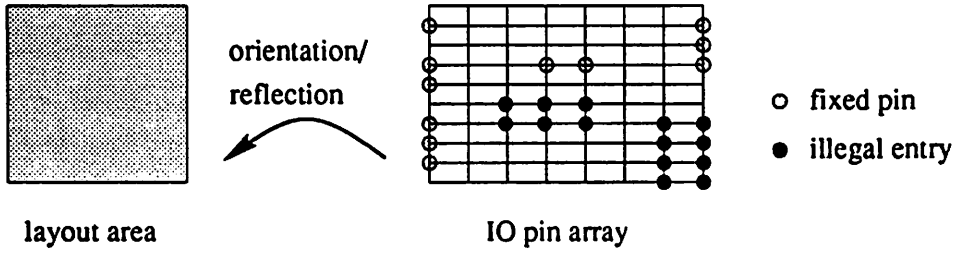

Figure 1: *Specification of constraints on placement of IO-pins. Here I has dimensions* 8×10 *and 11 fixed IO-pins are assigned to specific entries of I while 14 entries are illegal. The remaining entries are available for flexible IO-pins. I will be oriented and/or reflected and subsequently scaled so that it exactly covers the layout area of the placement.*

Each output solution is a specification of

- An absolute position of each block so that no pair of blocks or a block and an IO-pin are closer than a specified minimum distance $\lambda \geq 0$. This parameter allow physical constraints (design rules) to be met and is not intended for routing area allocation. Since multi-layer designs are considered, it is assumed that a significant part of the routing is performed on top of the blocks.

- An orientation and reflection of each block. Throughout this paper, the term *orientation* of a block refers to a possible 90 degree rotation, while *reflection* of a block refers to the possibility of mirroring the block around a horizontal and/or a vertical axis. Changing the orientation of a block generally alters its contour, while reflecting it does not. In an IC, each block can be oriented and/or reflected in a total of eight distinct ways. For MCMs, only two distinct reflections exist, since the direction of its pins is fixed, giving a total of four distinct orientations/reflections.

- An absolute position of each IO-pin, satisfying the specified constraints on their relative positions.

2.2 What is a "Good" Tradeoff ?

Let Π be the set of all placements and $\Re_+ = [0, \infty[$. The cost of a solution is defined by the vector-valued function $c : \Pi \mapsto \Re_+^4$ which will be described in Section 3.2. This Section describes how to specify what a "good" cost tradeoff is, and how to compare the cost of two solutions without resorting to a single-valued cost measure.

The user specifies a *goal vector* $g = (g_1, g_2, g_3, g_4) \in (\Re_+ \cup \{\infty\})^4$ and a *feasibility vector* $f = (f_1, f_2, f_3, f_4) \in (\Re_+ \cup \{\infty\})^4$ such that $0 \leq g_i \leq f_i \leq \infty$ for $i = 1,2,3,4$. For the i'th criterion, g_i is the maximum value wanted, if obtainable, while f_i specifies a limit beyond which the solution is unconditionally of no interest. For example, if the i'th criterion is layout area, $g_i = 20$ and $f_i = 100$ states that an area of 20 or less is wanted if it can be obtained, while an area larger than 100 is unacceptable. Areas between 20 and 100 are acceptable, although not as good as hoped for.

The vectors g and f defines a set of *satisfactory* solutions $S = \{x \in \Pi \mid \forall i : x_i \leq g_i\}$ and a set of *acceptable* solutions $A = \{x \in \Pi \mid \forall i : x_i \leq f_i\}$, where x_i is the cost of x wrt. the i'th dimension, i.e., $c(x) = (x_1, x_2, x_3, x_4)$. $S \subseteq A \subseteq \Pi$, i.e., a satisfactory solution is also acceptable. The values specified by g and f are merely used to guide the search process and in contrast to traditional, user-specified bounds, need *not* be obtainable. Therefore, they are significantly easier to specify than traditional bounds.

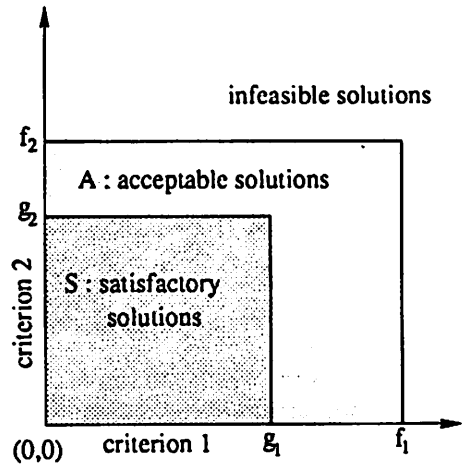

Figure 2: *The sets of satisfactory and acceptable solutions, illustrated in two dimensions.*

In order for the algorithm to compare solutions, a notion of relative solution quality is needed, which takes the goal and feasibility vectors into account. Let $x, y \in \Pi$. The relation x *dominates* y, written $x <_d y$, is defined by

$$x <_d y \quad \Leftrightarrow \quad (\forall i : x_i \leq y_i) \wedge (\exists i : x_i < y_i) \tag{2}$$

Using $<_d$, the relation x *is preferable to* y, written $x \prec_p y$, is then defined as follows, depending on how $c(x)$ compares to g : If x satisfy all goals, i.e., $x \in S$, then

$$x \prec_p y \quad \Leftrightarrow \quad (x <_d y) \vee (y \notin S) \tag{3}$$

If x satisfies none of the goals, i.e., $\forall i : x_i > g_i$ then

$$x \prec_p y \quad \Leftrightarrow \quad (x <_d y) \vee [(x \in A) \wedge (y \notin A)] \tag{4}$$

Finally, x may satisfy some but not all goals. Assuming a convenient ordering of the optimization criteria, $\exists\, k \in \{2,3,4\} : (\forall i < k : x_i \leq g_i) \wedge (\forall i \geq k : x_i > g_i)$. Then

$$\begin{aligned} x \prec_p y \quad \Leftrightarrow \quad & [(\forall i \geq k : x_i \leq y_i) \wedge (\exists i \geq k : x_i < y_i)] && (5) \\ & \vee \\ & [(x \in A) \wedge (y \notin A)] && (6) \\ & \vee \\ & [(\forall i \geq k : x_i = y_i) \wedge && (7) \\ & \{((\forall i < k : x_i \leq y_i) \wedge (\exists i < k : x_i < y_i)) \vee (\exists i < k : y_i > g_i)\}] && (8) \end{aligned}$$

Due to (3), (4) and (6) an acceptable solution is always preferable to an unacceptable solution. The right hand side of (5) states that x dominates y wrt. the dimensions for which x does not satisfy the goals. (7) and (8) states that in the special case when x equals y wrt. the non-satisfied dimensions, then x is still preferable to y if it either dominates y wrt. the satisfactory dimensions or if y does not satisfy a goal satisfied by x. Notice from (5) that when two solutions satisfy the same subset of goals, they are considered equal with respect to these goals, regardless of their specific values in these dimensions. Hence, when goals are satisfied, they are "factored out", focusing the search on the remaining, unsatisfactory dimensions.

The algorithm outputs a set of distinct solutions Φ_0 which are the best found in the sense defined by $\prec_p$. As a special case, if $g = (0,0,0,0)$ the algorithm searches for (a sample of) the Pareto-optimal set, i.e., the set of solutions, in which no solution can be improved in any dimension without being deteriorated in another. Since $\forall\, x, y \in \Phi_0 : \neg(x \prec_p y)$ the output solutions represents distinct design alternatives.

The above definition of $\prec_p$ is an extension of the definition introduced in [12], adding the feasibility vector f and the concept of acceptable solutions. The extension can be shown to preserve the transitivity of the ordering, as intuitively needed. The purpose of introducing f is to promote restriction of the search to the region of practical interest, that is, to prevent the algorithm from wasting time exploring solutions which are non-dominated but in practice infeasible, e.g., layouts yielding very good delays but having unacceptable areas.

3 Description of the Algorithm

The concept of genetic algorithms is based on natural evolution. In nature, the individuals constituting a population adapt to the environment in which they live. The fittest individuals have the highest probability of survival and tend to increase in numbers, while the less fit individuals tend to die out. This *survival-of-the-fittest* Darwinian principle is the basic idea behind the GA. The algorithm maintains a *population* of *individuals*, each of which corresponds to a specific solution to the optimization problem considered. Based on a given cost function, a measure of *fitness* defines the relative quality of individuals. An evolution process is simulated,

starting from a set of random individuals. The main components of this process are *crossover*, which mimics propagation, and *mutation*, which mimics the random changes occurring in nature. After a number of *generations*, highly fit individuals will emerge corresponding to good solutions to the optimization problem.

A *phenotype* is the physical appearance of an individual, while a *genotype* is the corresponding representation or genetic encoding of the individual. Crossover and mutation are performed in terms of genotypes, while fitness/cost is defined in terms of phenotypes. For a given genotype, the corresponding phenotype is computed by a *decoder*. A good overview of genetic algorithms is given in [2, 3]. Section 3.1 outlines our specific GA, Section 3.2 presents the genotype and its interpretation, and Sections 3.3 and 3.4 describe the selection strategy and the genetic operators, respectively.

3.1 Overview

Fig. 3 outlines our GA. Let $\Phi = \{\phi_0, \phi_1, \ldots, \phi_{N-1}\}$ denote the current population. The *rank* $r(\phi)$ of $\phi \in \Phi$ is the number of currently existing individuals which are preferable to ϕ, i.e., $r(\phi) = |\{\gamma \in \Phi \mid \gamma \prec_p \phi\}|$. Furthermore, let $\Phi_0 = \{\phi \in \Phi \mid r(\phi) = 0\} \subseteq \Phi$, i.e., Φ_0 is the current best solutions. Initially, Φ is constructed by routine *generate* (line 1) from random individuals subject only to the restriction that they all have distinct cost values, i.e., $\forall\, \phi, \psi \in \Phi : c(\phi) \neq c(\psi)$. One iteration of the repeat loop (lines 2-12) corresponds to the simulation of one generation. Throughout the optimization process $N = |\Phi|$ is kept constant and all solutions will have distinct cost values.

```
01   generate(Φ);
02   repeat :
03      select φ1, φ2 ∈ Φ;
04      defψ2 := crossover(φ1, φ2, ψ1, ψ2);
05      mutate(ψ1);
06      insert(Φ, ψ1);
07      if defψ2 :
08         mutate(ψ2);
09         insert(Φ, ψ2);
10      if converged() :
11         optimize(Φ0);
12   until goals() or converged() or cpuLimit();
13   output Φ0;
```

Figure 3: *Outline of the algorithm.*

In each generation, two parent individuals ϕ_1 and ϕ_2 are selected for crossover as described in Section 3.3 (line 3). The crossover operator, described in Section 3.4, generates offspring ψ_1 and possibly ψ_2 (line 4). defψ_2 is true if and only if ψ_2 was also generated. The algorithm is a steady-state GA, which means that only one crossover operation is performed per generation. Routine *mutate*, described in Section 3.4, subjects the generated offspring to random changes (lines 5, 8) and the resulting individuals are inserted in Φ by routine *insert* (lines 6, 9). An inserted individual ψ replaces a maximum rank solution to which it is preferable or, if ψ is not preferable to any solution, it replaces a maximum rank solution, which is not preferable to ψ. However, ψ is not inserted if $c(\psi)$ is already represented by another solution. Fig. 4 describes the detailed replacement scheme.

```
if ∀ φ ∈ Φ : c(φ) ≠ c(ψ) :
   Φ_ψ := {φ ∈ Φ | ψ ≺_p φ};
   if Φ_ψ = ∅ :
      Φ_ψ := {φ ∈ Φ | ¬(φ ≺_p ψ) ∧ r(φ) > r(ψ)};
   if Φ_ψ ≠ ∅ :
      randomly select φ ∈ Φ_ψ with r(φ) maximal;
      Φ := (Φ \ {φ}) ∪ {ψ};
```

Figure 4: *Routine insert(Φ, ψ). Φ may be unaltered.*

Routine *converged* (line 10) detects if no improvement has occurred in T consecutive generations, that is, if Φ_0 has not changed in this period. In that case routine *optimize* (line 11) attempts to optimize all rank zero individuals by simple hillclimbing. On each individual $\phi \in \Phi_0$ a sequence of mutations is tried. Each mutation yielding ϕ' from ϕ is only executed if $\phi' \prec_p \phi$ and if $c(\phi')$ is not already represented by another solution. The algorithm terminates (line 12) when either a) Φ_0 contains a solution satisfying all goals (detected by routine *goals*()), b) the process has converged, or c) a CPU-time limit has been reached (detected by *cpuLimit*()). Φ_0 is then the output set of solutions (line 13).

Routines *insert* and *optimize* assures that a solution ϕ' can never replace ϕ if $\phi \prec_p \phi'$. Hence, the number of acceptable solutions $|A|$ is a non-decreasing function of time and so is Φ_0, in the sense inferred by $\prec_p$, while $|\Phi_0|$ is not. For single-valued cost functions, GAs keeping the current best solution throughout the optimization are often referred to as *elitist* GAs. The above scheme can therefore be seen as a generalization of the elitist GA to vector-valued domains.

3.2 Solution Representation and Decoder

The representation, or genotype, of a placement consist of five components a) through e) :

a) An inverse Polish expression of length $2b-1$ over the alphabet $\{0,1,\ldots,b-1,+,*\}$, where b is the number of blocks. The operands $0,1,\ldots,b-1$ denotes block identities and $+,*$ are operators. The expression uniquely specifies a slicing-tree for the placement, as first introduced in [22], with $+$ and $*$ denoting a horizontal and a vertical slice, respectively.

b) A bitstring of length $2b$ for ICs and b for MCMs, representing the reflection of each block. For ICs, the reflection of the i'th block is specified by bits $2i$ and $2i+1$ and for MCMs, it is specified by bit i.

c) An integer in the interval $[0;7]$ selecting one of the eight possible orientations/reflections of the IO-pin array I relative to the placed blocks, cf. Fig. 1.

d) A string of n_{IO} integers, where n_{IO} is the number of nets having at least one flexible IO-pin. The string is a permutation of the numbers $0,1,\ldots,n_{IO}-1$ and specifies an ordering of these nets to be used when placing flexible IO-pins.

e) A string of n_{multi} integers, where n_{multi} is the number of multi-pin nets, i.e., nets having at least 3 pins. If the i'th multi-pin net has s_i sinks, the i'th integer k_i satisfies $0 \leq k_i \leq s_i - 1$ and specifies k_i as being the critical sink of net i, to be used when routing the net.

Let Ω denote the set of all such representations and let $d : \Omega \mapsto \Pi$ be the decoder. The search space $\tilde{\Pi}$ considered by the algorithm is by definition the image of d, i.e., $\tilde{\Pi} \equiv d(\Omega) \subset \Pi$. Given a representation the corresponding placement (phenotype) and its cost $c = (c_{area}, c_{ratio}, c_{delay}, c_{cong})$ is computed by the decoder in eight steps as follows :

1. From the slicing-tree specified by the Polish expression, the orientation of each block is determined such that layout area is minimized. The orientations are computed using an exact algorithm by Stockmeyer [19] which guarantees a minimum area layout for the given slicing-structure. The reflection of each block is as specified by component b) of the representation.

2. Absolute coordinates are determined for all blocks by a top-down traversal of the slicing-tree. At each operator node, if relative movement of the two subtrees along the slicing axis is possible, the centerpoints of the subtrees are aligned (perpendicular to the slicing axis).

3. The layout is compacted, first vertically and then horizontally, using a simplified version of the one-dimensional channel compaction algorithm presented in [23], which adapts a scan-line approach. Fig. 5 illustrates the first three steps of decoding.

4. Given the orientation/reflection of the IO array I specified by component c) each flexible IO-pin is assigned to an unused entry of I. The n_{IO} nets having flexible IO-pins are treated one at a time, in the order defined by component d). For each net, each flexible pin is assigned to the unused entry of I, which is closest to the center of gravity of the remaining

pins of the net. Then the area and aspect ratio of the layout can be computed. c_{area} is the area of the smallest rectangle enclosing all blocks and IO-pins and $c_{ratio} = |r_{actual} - r_{target}|$, where r_{actual} is the actual aspect ratio of the layout (height divided by width) and r_{target} is a user-defined target aspect ratio.

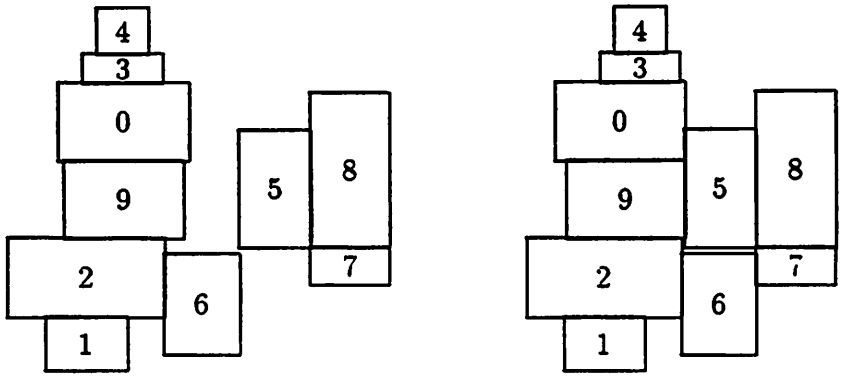

Figure 5: *Given 10 blocks and the Polish expression 1 2 + 6 * 9 0 + + 3 4 + + 5 * 7 8 + *, the placement on the left is the result of step 2 of the decoding. Subtrees are recursively centered and oriented optimally. Since the height of the layout is determined by the blocks 1,2,9,0,3,4, no blocks are moved when attempting vertical compaction. Subsequent horizontal compaction moves blocks 8,7,5,9 and 0 towards the left, so that blocks 2,6,7 now determines the width of the layout. The placement to the right is the result of compaction (step 3), i.e., the final placement.*

5. A global routing graph $G = (V, E)$ is constructed which forms a two-dimensional, uniformly spaced lattice, exactly covering the entire layout. The pitch of the lattice is determined by the user-defined parameter g_{pitch}. Each pin is then assigned to the closest vertex in V.

6. The topology of each net is approximated by a Steiner tree embedded in G. Each Steiner tree is computed independently by the SERT-C algorithm ("Steiner Elmore Routing Tree with identified Critical sink") introduced in [5]. SERT-C is a deterministic heuristic inspired by Prims algorithm, which explicitly minimizes the Elmore delay from the source to a designated critical sink, illustrated in Fig. 6. For nets with more than one sink, the critical sink is specified by component e) of the representation. Since all pins were mapped to V in the previous step all trees are automatically embedded in G.

7. The delay $D(P)$ of each path $P = \{p_0, p_1, \ldots, p_{l-1}\}$ is estimated as

$$D(P) = \sum_{i=0}^{l-1} [d_E(p_i) + t(s(p_i))]$$

where $d_E(p_i)$ is the Elmore delay [9] from $s(p_i)$ to p_i computed in the Elmore-optimized Steiner tree. The delay cost c_{delay} is the maximum path delay, i.e., $c_{delay} = \max_{P \in \mathcal{P}}\{D(P)\}$.

8. Finally, the maximum routing congestion is estimated. For each edge $e \in E$, cap(e) denotes the capacity of e and equals $(g_{pitch}/w_{pitch}) \times l_{block}$ if (a part of) e is on top of a block, and $(g_{pitch}/w_{pitch}) \times l_{space}$, otherwise. usage($e$) is the number of nets using e. The congestion cost c_{cong} is computed as

$$c_{cong} = 100 \times \max\left[\max_{e \in E}\left\{\frac{\text{usage}(e) - \text{cap}(e)}{\text{cap}(e)}\right\}, 0\right]$$

i.e., c_{cong} is the maximum percentage by which an edge capacity has been exceeded.

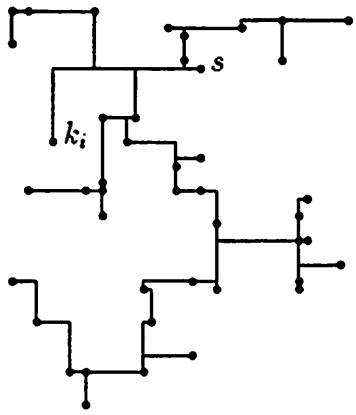

Figure 6: *The Elmore-optimized Steiner tree computed by SERT-C for a 44-pin net. s is the source and k_i the selected critical sink.*

The decoders use of Stockmeyers algorithm as well as the compactor may seem to introduce an unwanted bias towards optimizing area at the cost of other criteria. However, they only favor area for a given, fixed Polish expression, while the GA will optimize the Polish expressions themselves. Experiments have shown that the algorithms have a positive impact on the overall optimization process.

Although the nets are routed independently, the resulting Steiner trees represents a very accurate estimation of the net topologies compared to the estimations of previous approaches, cf. Section 1, which in turn indicates that c_{delay} is an accurate estimate. Furthermore, for ICs the topology dependent Elmore delay estimate has high fidelity in the sense that a solution which is near-optimal according to the estimate will also be near-optimal wrt. actual delay [4]. The smaller the congestion estimate c_{cong} is, the fewer nets needs to be rerouted to obtain 100% global routing completion and, by assumption, the easier is the global routing task.

Since thousands of representations will be decoded during a typical execution of the algorithm, the computational complexity of the decoder is crucial and has been the most important criterion when choosing the algorithms applied. Stockmeyers algorithm and our implementation of the compactor each requires time $O(b^2)$, and SERT-C is quadratic in the number of pins. The calculation of c_{cong} has been implemented in such a way that it is, in practice, independent of g_{pitch}. This is obtained by not explicitly forming G and by using upper bounds on the possible congestion of an edge such that only few edges need be considered.

3.3 Selection Strategy

The scheme for selection of parents for crossover (line 3 of Fig. 3) should reflect the principle of survival-of-the-fittest. The parents are selected independently of each other, subject only to the constraint that they are distinct. Assume that the current population $\Phi = \{\phi_0, \phi_1, \ldots, \phi_{N-1}\}$ is sorted in ascending order according to rank, i.e., $r(\phi_0) \leq r(\phi_1) \leq \ldots \leq r(\phi_{N-1})$. Each parent ϕ is selected using a scheme presented in [12, 20], which have two properties : 1) The probability that $r(\phi)$ equals $r(\phi_k)$, written $P[r(\phi) = r(\phi_k)]$, decreases linearly with k and $P[r(\phi) = r(\phi_0)] = \beta P[r(\phi) = r(\phi_{N/2})]$, where $1 < \beta \leq 2$ is a user-defined parameter controlling the selection pressure. 2) All individuals having the same rank have the same probability of being selected.

In the traditional GA, selection is based on a fitness function, which defines the relative quality of each individual by a (non-trivial) transformation of the cost values, cf. Section 3. In contrast, the rank-based selection scheme described eliminates the need for an explicit fitness function.

3.4 Genetic Operators

The crossover operator (line 4 of Fig. 3) as well as the mutation operator (lines 5, 8), which is also used by routine *optimize* (line 11), operates on each of the five components of the genotype (cf. Section 3.2) independently. For components b), c) and e), standard operators extensively studied in the GA literature are applied. Crossover of bitstrings, component b), is done using uniform crossover [3] : Scanning through the bitstrings of the given parent individuals, the value of the i'th bit in the offspring is copied from one of its parents, chosen at random and with equal probability. The term 'uniform' means that each value of the offspring string is determined independently of the rest of the string. Mutation is pointwise, i.e., each bit is independently inverted with a small user-defined probability p_{mut}. Integers and strings of independent integers, i.e., components c) and e), are handled similarly [3] : Crossover is uniform and the value of the i'th integer of the offspring is computed as the average of the corresponding parent values. Mutation is pointwise, randomly altering each integer within its feasible range with probability p_{mut}.

Polish expressions, component a), are handled by highly specialized operators introduced in [7][3]. For crossover, one of four distinct operators CO1, C02, C03 and C04 is chosen uniformly at random. C01, C02 and C03 generates a single offspring, while C04 generates two offspring. When using C01, the offspring inherits the order and positions of all operands from the same parent. Similarly when using CO2, the order and positions of all operators are inherited from the same parent i.e., C02 preserves the slicing-structure. In addition to also preserving the complete slicing-structure from one parent, C03 preserves a complete subtree. Finally, C04 generates two offspring by interchanging two subtrees of the parents. If this is not possible while preserving feasibility of the generated expressions, C04 fails and one of the other operators are applied instead. Four operations exists for mutation of a Polish expression : A pair of operands can be interchanged, a pair of operators can be interchanged, the type of an operator can be

[3]While operations on Polish expressions are as in [7], it should be noted that the *interpretation* of an expression, i.e., the decoder, is significantly different. This is a consequence of the fundamental differences of the two approaches, e.g., the distinct optimization criteria, multi-dimensional versus one-dimensional optimization, etc.

changed and an operator and an operand can be interchanged. Only the latter operation requires a check for feasibility of the produced expression.

Finally, component d), a permutation of a set of integers, is handled using the operators of [21]. The crossover operator applies a simple heuristic to preserve as many immediate predecessor-successor relationships as possible. The only mutation operation is the interchange of two values.

For further description of the genetic operators the reader is referred to the references cited, in which detailed descriptions can be found. A crucial property of both the crossover operator and the mutation operator is that they preserve feasibility, i.e., only feasible genotypes, which can be interpreted by the decoder, are ever generated. If feasibility were not preserved by the operators, either a potentially time-consuming repair algorithm would be required in the decoder, or a cost penalty method would be needed, jeopardizing a main objective of our approach, the need to eliminate weight factors.

4 Experimental Results

Evaluating performance by comparison to an existing approach is complicated by a number of factors. Firstly, the placement model assumption of routing on top of the blocks, is not compatible to earlier channel-based IC models applied by previous approaches. Secondly, there are no building-block benchmarks which includes appropriate timing information. And thirdly, and most importantly, it is inherently difficult to fairly compare the 4-dimensional optimization approach to existing 1-dimensional approaches. However, using the examples described in Section 4.1, comparisons to simulated annealing and random search have been established as described in Section 4.2. Results are presented in Sections 4.3 and 4.4.

4.1 Test Examples

The characteristics of four of the circuits used for testing are given in Table 1. xeroxT, ami33T and ami49T are constructed from the CBL/NCSU building-block benchmarks xerox, ami33 and ami49, respectively, by adding the required timing information. Paths are generated in a random fashion and internal block delays, output driver resistances and input capacities are assigned randomly assuming normal distributions and using mean values from [4, 18], representative of a 0.8 μm CMOS process.

Circuit	Type	Blocks	Pins	IO	Nets	Paths
xeroxT	IC	10	698	2	203	86
ami33T	IC	33	522	42	123	230
ami49T	IC	49	953	22	408	116
SPERT	MCM	20	1,168	36	248	574

Table 1: *Main characteristics of test examples. The columns are : type, no. of blocks, total no. of pins, no. of IO-pins, no. of nets and no. of paths.*

SPERT is an MCM consisting of a vector processor (ASIC), 16 SRAMs and 3 buffer components. It is the key component of a dedicated hardware system for speech recognition based on

neural networks, currently being developed at the International Computer Science Institute in Berkeley, California [1].

For all examples, the target aspect ratio is $r_{target} = 1.0$, the number of routing layers on blocks is $l_{block} = 2$ and between blocks $l_{space} = 3$. For the ICs all IO-pins are fixed, $w_{pitch} = 4$ μm and $g_{pitch} = 40$ μm. The routing wire resistance $\bar{r}$ is 0.03 Ω/μm and the capacitance $\bar{c}$ is 0.352 fF/μm, again typical for an 0.8 μm CMOS process [4]. The minimum block spacing is $\lambda = 0$, i.e., blocks can be abutted. For the MCM, all IO-pins are flexible but restricted to positioning along *two* parallel sides of the layout. We assume $w_{pitch} = 40$ μm, $g_{pitch} = 400$ μm, $\bar{r} = 0.008$ Ω/μm, $\bar{c} = 0.06$ fF/μm, and a minimum spacing of $\lambda = 5.0$ mm.

4.2 Method

The GA is implemented in 9,000 lines of C and runs on a DEC station 5000/125. Performance is compared to that of a simulated annealing algorithm, denoted SA, and a random walk, denoted RW. Both algorithms uses the same placement representation and decoder as the GA. The RW simply generates genotypes at random, evaluates them and stores the best (rank 0) solutions ever found. The SA generates moves using the mutation operator of the GA and the cooling schedule is implemented following [15, 17] : The initial temperature is determined so that the probability of accepting a move increasing cost by $2\hat{\sigma}$ is 0.8, where $\hat{\sigma}$ is an estimate of the standard deviation of cost obtained by initial sampling of the search space. The inner loop criterion is that $2b$ moves have been accepted or that $6b$ moves have been tried. The temperature is then decreased by the factor 0.92 and the stop criterion is that the probability of accepting a move increasing the cost by $10^{-3}\hat{\sigma}$ is less than 10^{-6} or that the average cost at a fixed temperature has not decreased for 5 consecutive temperatures.

The use of the same representation and decoder by all three algorithms means that the effects of the representation and the decoder algorithms, e.g., Stockmeyers algorithm and the compactor, does not a priori give an advantage to any of the algorithms. Furthermore, all algorithms explore exactly the same search space, namely $\tilde{\Pi}$. Consequently, any performance differences observed can be attributed to the different search strategies themselves, rather than e.g. the decoder algorithms.

Since RW does not rely on cost comparisons, it can use the same 4-dimensional cost function as the GA, allowing the two approaches to be directly compared. In contrast, the traditional SA algorithm relies on absolute quantification of change of cost when determining if a move should be accepted, and consequently, cost has to be single-valued. Using a SA cost function of the form (1), it is far from clear how to fairly compare the *single* solution output by the SA algorithm to the *set* of solutions output by the GA. Therefore, comparisons of the GA to SA has to be based on optimizing one criterion only, in which case the GA output reduces to a single solution, comparable to the single SA solution. On the other hand, such comparisons are still very informative since they reveal whether the GA is competitive in this special case.

A fixed parameter setting is used for each algorithm, disallowing problem-specific tuning. The GA parameters are : Population size $N = 40$, selection bias $\beta = 2.0$ and mutation rate $p_{mut} = 0.0005$. The hillclimber attempts 1,000 mutations on a given individual, and the search is considered converged if no improvement has been observed for $T = 10,000$ generations.

4.3 One-Dimensional Optimization

Neither aspect ratio deviation nor routing congestion are suitable dimensions for one-dimensional comparisons since optimal results can easily be obtained by distributing the blocks over a large area. Instead, one-dimensional optimization for area and delay are performed, for which the GA uses the goal vectors $g = (0, \infty, \infty, \infty)$ and $g = (\infty, \infty, 0, \infty)$, respectively.

Fig. 7 illustrates the results. For each circuit and each of the two criteria, the three algorithms was executed 10 times each and the result indicated by a bar. The center point of each bar indicates the average result obtained in the 10 runs and the height of each bar is two times the standard deviation. For each circuit and criterion, the average result of RW is normalized to 1.00.

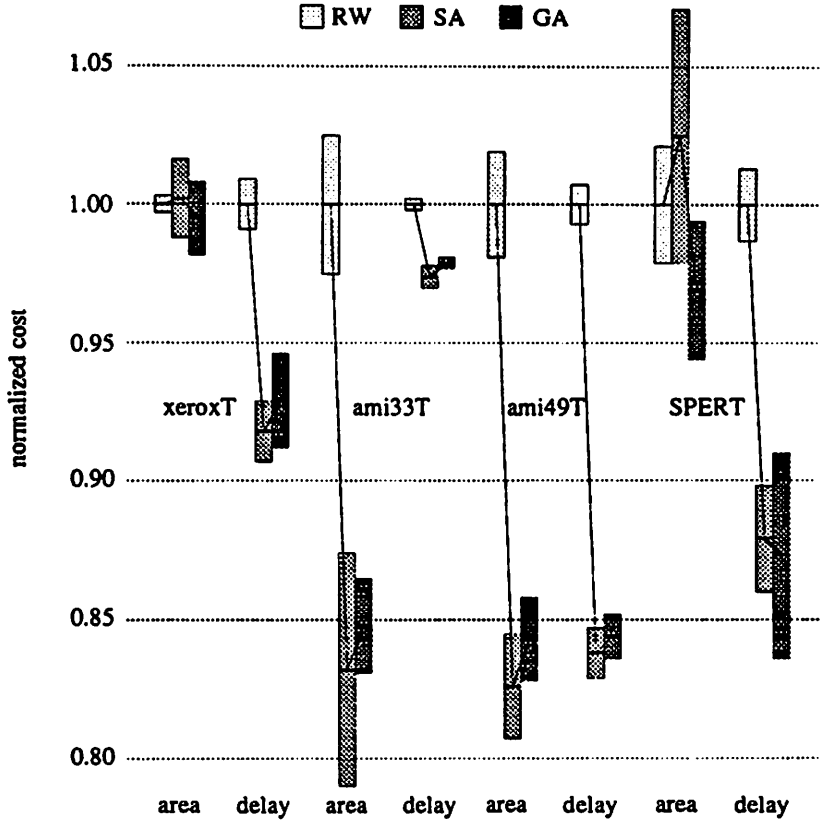

Figure 7: *Relative performance of the GA, SA and RW for one-dimensional optimization.*

The SA was executed first, and the average consumed CPU-time enforced on each of the GA and RW as a CPU-time limit, thereby obtaining the exact same average runtime for all three algorithms. However, the SA approach has an advantage of defining the CPU-time requirement. Average CPU-time per run varied from 183 seconds for area optimization of SPERT to 3,903 seconds for delay optimization of ami49T. For area optimization of xeroxT and SPERT the SA averages are slightly worse than those of RW, indicating that the SA got trapped at local minima. However, in all other cases, both the GA and SA performs significantly better than RW. Overall, the GA and SA performance is very similar, indicating that the efficiency of the GA search is comparable to that of SA in the special case of one-dimensional optimization. Fig. 8 shows the smallest layouts obtained by the GA.

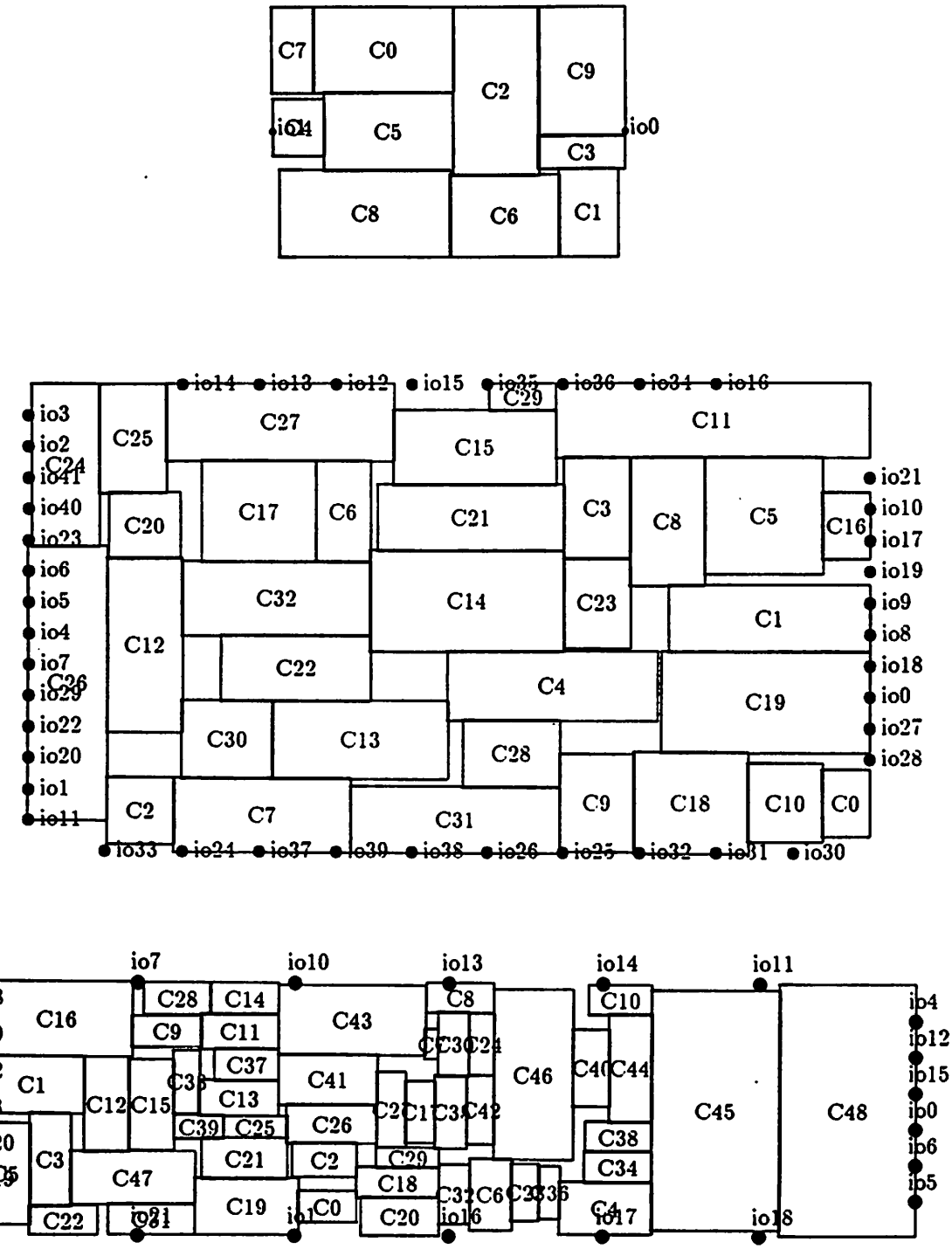

Figure 8: *The minimum area results obtained by the GA in 10 runs for xeroxT (top), ami33T (center) and ami49T (bottom). The empty space constitutes 3.1 %, 9.3 % and 7.5 % of the layout area, respectively.*

4.4 Four-Dimensional Optimization

For 4-dimensional optimization, Figures 9 and 10 compares the solution sets Φ_0 found by a sample execution of the GA using a 1 CPU-hour time limit to those found by RW using a 10 CPU-hour limit. For all examples, the GA uses the goal vector $g = (0, 0.2, 0, 30)$ and the feasibility vector $f = (1.5B, 0.5, \infty, 200)$, where B is the sum of the areas of all blocks of the circuit in question. The GA results are always significantly better than the RW results in all dimensions.

The solution for SPERT indicated by a circle in Fig. 11 is shown in Fig. 12 and illustrates the effect of minimizing routing congestion. The processor block T0 has 416 connected pins and is the cause of potential congestion problems. However, by moving T0 close to the center of the layout, cf. Fig. 12, a congestion cost c_{cong} of only 15 % is obtained. The other solutions in Φ_0 have higher congestion, but better areas and/or delays, cf. Fig. 11.

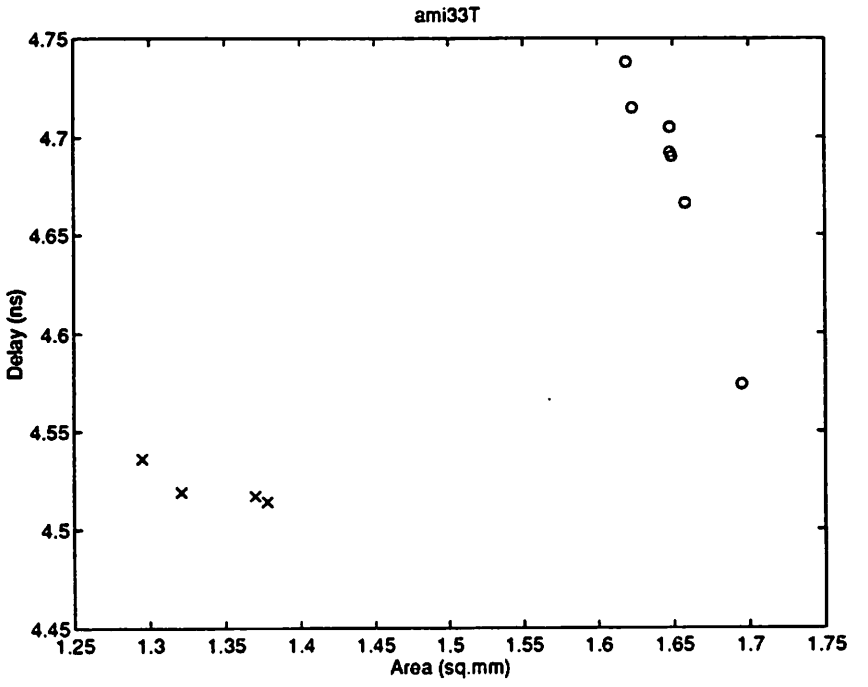

Figure 9: *Comparison of the solution sets Φ_0 found by RW and the GA for ami33T. The o's are RW solutions and the x's are GA solutions. Only solutions satisfying both the aspect ratio goal and the congestion goal are shown.*

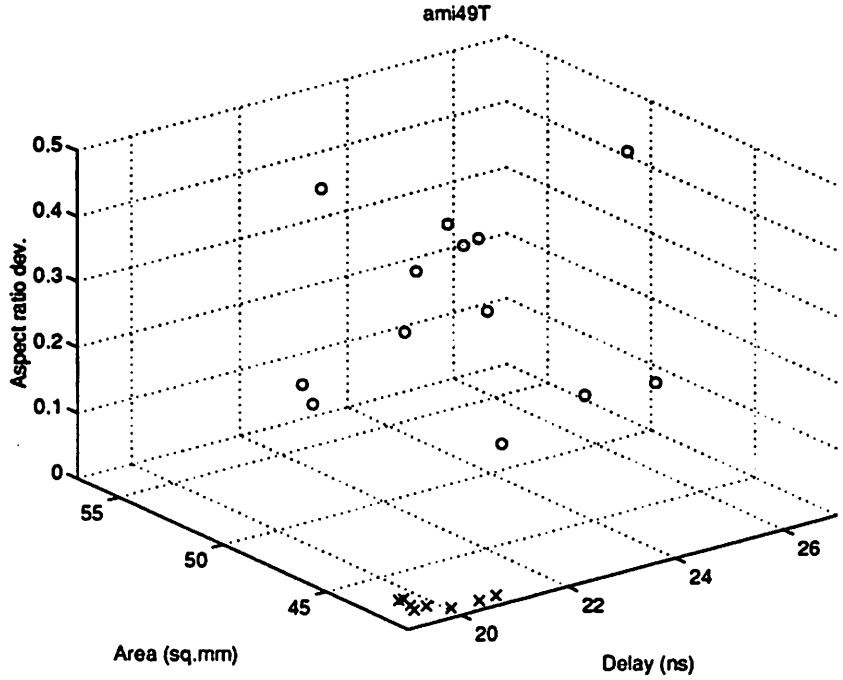

Figure 10: *Comparison of the solution sets Φ_0 found by RW and the GA for ami49T. All solutions shown satisfy the routing congestion goal, while all GA solutions also satisfy the aspect ratio goal.*

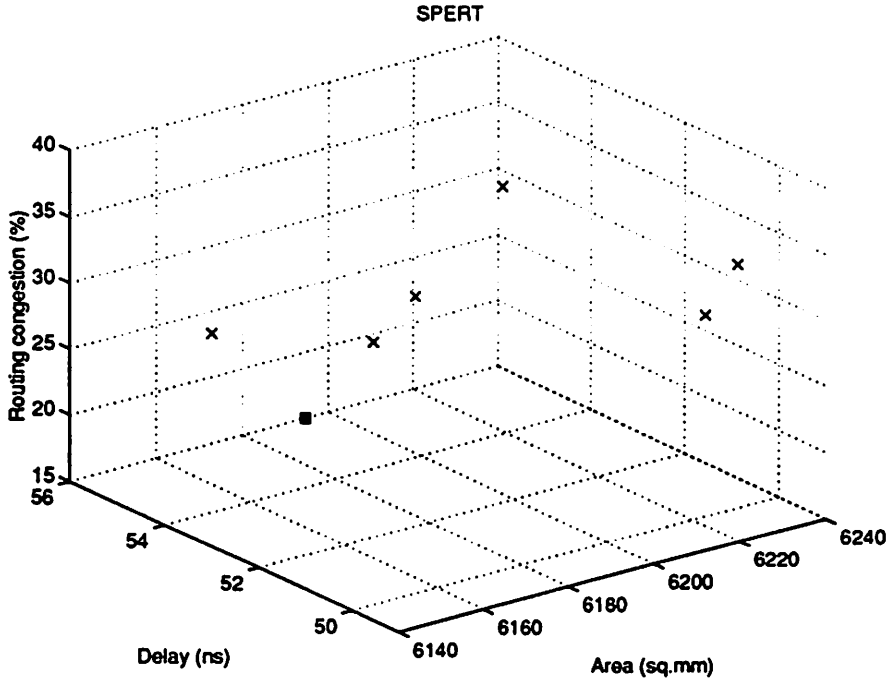

Figure 11: *The solution set Φ_0 found by the GA for SPERT. Only solutions satisfying the aspect ratio goal are shown.*

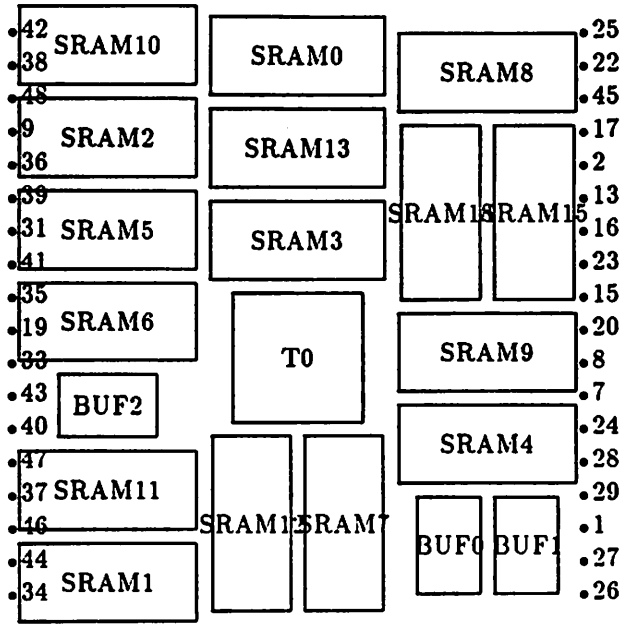

Figure 12: *A placement of SPERT with low routing congestion, obtained by moving the processor T0 towards the center of the layout, at the cost of increased area and delay.*

5 Conclusions

A genetic algorithm for building-block placement of ICs and MCMs has been presented, which minimizes area, path delay and routing congestion while attempting to meet a target aspect ratio. The key feature is the explicit design space exploration performed, which results in the generation of a solution *set* representing good, alternative cost tradeoffs.

The inherent problem of existing approaches wrt. specification of suitable weights and bounds is solved by eliminating these quantities, and another practical problem, the traditionally required placement-routing iterations, is significantly reduced by explicitly minimizing routing congestion.

The experimental work includes results for a real-world design and shows that the solution sets found represent good, balanced tradeoffs. The usefulness of minimizing routing congestion is also illustrated and it is shown that the efficiency of the search process is comparable to that of simulated annealing in the special case of one-dimensional optimization. Furthermore, the required runtime ranging from about 3 CPU-minutes to 1 CPU-hour is very reasonable from a practical point of view. It is concluded that the presented algorithm is a promising approach for building-block placement.

Acknowledgments

The authors would like to thank Dongsheng Wang at University of California, Berkeley, CA, for implementing the simulated annealing algorithm, and James Beck and Brian Kingsbury at the International Computer Science Institute, Berkeley, CA, for providing us with their data for the SPERT design. This research was supported by SRC grant no. DC-324-012 and by the Danish Technical Research Council.

References

[1] K. Asanović, J. Beck, "T0 Engineering Data, Revision 0.14," Technical Report, International Computer Science Institute, Berkeley, CA, 1994.

[2] D. Beasley, D. R. Bull, R. R. Martin, "An Overview of Genetic Algorithms : Part 1, Fundamentals," *University Computing,*, Vol. 15, No. 2, pp. 58-69, 1993.

[3] D. Beasley, D. R. Bull, R. R. Martin, "An Overview of Genetic Algorithms : Part 2, Research Topics," *University Computing,*, Vol. 15, No. 4, pp. 170-181, 1993.

[4] K. D. Boese, A. B. Kahng, B. A. McCoy, G. Robins, "Fidelity and Near-Optimality of Elmore-Based Routing Constructions," *Proc. of the Intl. Conf. on Computer Design*, pp. 81-84, 1993.

[5] K. D. Boese, A. B. Kahng, G. Robins, "High-Performance Routing Trees With Identified Critical Sinks," *Proc. of the 30th Design Automation Conference*, pp. 182-187, 1993.

[6] H. Chan, P. Mazumder, K. Shahookar, "Macro-cell and module placement by genetic adaptive search with bitmap-represented chromosome," *Integration, the VLSI Journal*, Vol. 12, No. 1, pp. 49-77, Nov. 1991.

[7] J. P. Cohoon, S. U. Hedge, W. N. Martin, D. Richards, "Distributed Genetic Algorithms for the Floorplan Design Problem," *IEEE Transactions on Computer-Aided Design*, Vol. 10, pp. 484-492, April 1991.

[8] P. Dasgupta, P. Mitra, P. P. Chakrabarti, S. C. DeSarkar, "Multiobjective Search in VLSI Design," *Proc. of The 7th International Conference on VLSI Design*, pp. 395-400, 1994.

[9] W. C. Elmore, "The Transient Response of Damped Linear Network with Particular Regard to Wideband Amplifiers," *J. Applied Physics*, Vol. 19, pp. 55-63, 1948.

[10] H. Esbensen, E. S. Kuh, "An MCM/IC Timing-Driven Placement Algorithm Featuring Explicit Design Space Exploration," *Proc. of the 1996 IEEE Multi-Chip Module Conference,* 1996 (to appear).

[11] H. Esbensen, P. Mazumder, "SAGA: A Unification of the Genetic Algorithm with Simulated Annealing and its Application to Macro-Cell Placement," *Proc. of The 7th International Conference on VLSI Design*, pp. 211-214, 1994.

[12] C. M. Fonseca, P. J. Fleming, "Multiobjective Optimization and Multiple Constraint Handling with Evolutionary Algorithms I : A Unified Formulation," *Research Report 564*, Dept. of Automatic Control and Systems Eng., University of Sheffield, U.K., January 1995.

[13] T. Gao, P. M. Vaidya, C. L. Liu, "A Performance Driven Macro-Cell Placement Algorithm," *Proc. of the 29th Design Automation Conference*, pp. 147-152, 1992.

[14] T. Hamada, C.-K. Cheng, P. M. Chau, "Prime : A Timing-Driven Placement Tool using A Piecewise Linear Resistive Network Approach," *Proc. of the 30th Design Automation Conference*, pp. 531-536, 1993.

[15] M. D. Huang, F. Romeo, A. Sangiovanni-Vincentelli, "An Efficient General Cooling Schedule for Simulated Annealing," *Proc. of the 1986 International Conference on Computer-Aided Design*, pp. 381-384, 1986.

[16] M. A. B. Jackson, E. S. Kuh, "Performance-Driven Placement of Cell Based IC's," *Proc. of the 26th Design Automation Conference*, pp. 370-375, 1989.

[17] C. Sechen, A. Sangiovanni-Vincentelli, "TimberWolf3.2 : A New Standard Cell Placement and Global Routing Package," *Proc. of the 23rd Design Automation Conference*, pp. 432-439, 1986.

[18] A. Srinivasan, K. Chaudhary, E. S. Kuh, "RITUAL : A Performance-Driven Placement Algorithm," *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, Vol. 39, pp. 825-840, Nov. 1992.

[19] L. Stockmeyer, "Optimal Orientations of Cells in Slicing Floorplan Designs," *Information and Control*, Vol. 57, pp. 91-101, 1983.

[20] D. Whitley, "The Genitor Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best," *Proc. of the Third International Conference on Genetic Algorithms*, pp. 116-121, 1989.

[21] D. Whitley, T. Starkweather, D. Fuquay, "Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator," *Proc. of the Third International Conference on Genetic Algorithms*, pp. 133-140, 1989.

[22] D. F. Wong, C. L. Liu, "A new algorithm for floorplan design," *Proc. of the 23rd Design Automation Conference*, pp. 101-107, 1986.

[23] X.-M. Xiong, E. S. Kuh, "Geometric Approach to VLSI Layout Compaction," *International Journal of Circuit Theory and Applications*, Vol. 18, pp. 411-430, 1990.