Architecture-Aware FPGA Placement using Metric Embedding

Padmini Gopalakrishnan, Xin Li, Lawrence Pileggi
Electrical & Computer Engineering, Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213, USA
padmini@cmu.edu, xinli@ece.cmu.edu, pileggi@ece.cmu.edu

ABSTRACT

Since performance on FPGAs is dominated by the routing architecture rather than wirelength, we propose a new architecture-aware approach to initial FPGA placement that models the relationship between performance and the routing grid, using concepts from graph embedding and metric geometry. Our approach, CAPRI, can be viewed as an embedding of a graph representing the netlist into a metric space that is representative of the FPGA. First, we develop an analytic metric of distance that models delays along the FPGA routing grid. We then embed a netlist into the defined metric space using matrix projections and online bipartite matching. Experimental comparisons with the popular FPGA tool, VPR, show that with CAPRI's initial solution, the resulting placements show median improvements of 10% in critical path delays for the larger MCNC benchmarks. Total placement runtime is also improved by 2x on average.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids

General Terms

Algorithms, Design, Performance

Keywords

FPGAs, Placement, Metric Embedding

1. INTRODUCTION

Delays on modern Field Programmable Gate Arrays (FPGAs) are heavily influenced by their routing architectures, which typically are complex and heterogeneous [23,25]. Studies [10,17,21,26] have shown that the delay of a route on an FPGA is dominated by the number of switches required to program it. Given the rapidly improving performance of these ICs, this dependence of delays on the routing architecture must be modeled during physical design [22].

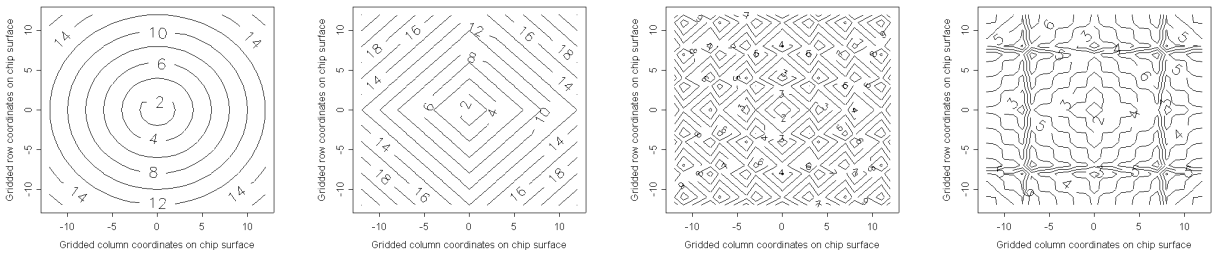
We present a new, analytical, and *architecture-aware* approach to FPGA placement called *CAPRI (Convex Assigned Placement for Regular ICs)* that captures this relationship. Underlying our approach is the idea that any placement algorithm, whether for ASICs or FPGAs, can be viewed as an embedding of a graph representing the netlist into a chosen metric space. For FPGA placement in particular, we define an analytic metric of "distance" in terms of the total delay through switches on the FPGA routing architecture, and use it to construct a metric space that captures FPGA performance accurately. We then embed the netlist graph into this metric space with a heuristic technique based on matrix projections and online bipartite graph matching. The resulting solution is a legal initial placement, which tries to minimize delays on driver-sink connections and is thus "good" from a global timing perspective. Subsequently we apply local optimization using any existing move-based placement technique to improve specific critical paths and routability.

CAPRI is analogous to analytical wirelength minimization in ASICs, which produces a good initial placement by reducing the geometric length (and therefore the delay) of connections. The crucial difference in our work is the use of a metric space that accurately captures delays on the FPGA, rather than a Euclidean or Manhattan space. Fig. 1 illustrates the motivation for our approach. The contour lines in Fig. 1 join points on the chip surface that are equidistant from the origin using geometric metrics such as (a) Euclidean and (b) Manhattan distances, as well as delays measured in terms of the number of routing switches on two different FPGA routing architectures ((c) and (d)). These plots demonstrate that to model delays accurately in FPGA placement, we need a metric that captures the delay contours of the FPGA routing architecture, rather than the Euclidean or Manhattan metrics used in ASIC placement.

Prior work on timing-driven FPGA placement models the dependence of delay on the routing architecture using either look-up tables [11], detailed timing models [13, 18], or empirical models [15,21,26]. These models are used in move-based algorithms such as simulated annealing [11] or partitioning [14,15]. The novelty of CAPRI lies in the analytical framework for modeling the impact of routing architectures on delay during placement. Importantly, this framework enables *global*, rather than local, optimization. Furthermore, our placement strategy is computationally efficient, and can be used to quickly find an initial solution.

We evaluate our approach via a placement methodology that uses CAPRI to produce an initial legal placement, followed by local optimization using low-temperature simulated annealing in the popular FPGA tool VPR [11]. When compared with running VPR alone, the placements we obtain show an improvement of 10.08% (median) and 11.13% (mean) in the post-routing delay of top critical paths. Total placement runtime is improved by 2.05x; CAPRI itself takes just 4.8% of this total runtime.

DAC 2006, July 24–28, 2006, San Francisco, California, USA.



(a) (b) (c) (d)

Figure 1: Contour plots showing points of equal distance (i.e., delay) from the origin using the following metrics: **(a)** Euclidean Distance, **(b)** Manhattan Distance, **(c)** Delays along an FPGA routing grid with two different kinds of route segments, and **(d)** Delays along an FPGA routing grid that mimics the commercial Xilinx Virtex FPGA.

2. BACKGROUND AND RELATED WORK

A *distance metric* on a set of points Γ, is a map $\phi : \Gamma \times \Gamma \rightarrow \Re^+$, such that the following hold for all $x, y, z \in \Gamma$: (1) $\phi(x, x) = 0$, (2) $\phi(x, y) = \phi(y, x)$, and (3) $\phi(x, z) \leq \phi(x, y) + \phi(y, z)$ i.e., the triangle inequality.

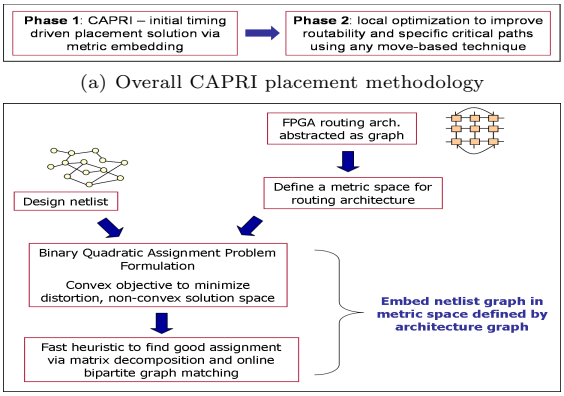
A *metric space* is defined by the pair (Γ, ϕ). When Γ is a finite set of points, (Γ, ϕ) is called a finite metric space. A *metric embedding* of (Γ, ϕ) into a space (Γ', ϕ') is given by a map $f : \Gamma \rightarrow \Gamma'$. If distances are preserved, i.e., if $\phi(x, y) = \phi'(f(x), f(y))$ for all $(x, y) \in \Gamma \times \Gamma$, then the embedding is said to be *isometric*. In general, however, embeddings incur *distortion*, meaning that distances are expanded or contracted in the new metric space. A detailed discussion of metric embeddings is available in [12].

Any undirected graph G has a natural distance metric on its vertices, with the distance between any pair of vertices given by the shortest path between them in G. Graph embedding maps the vertices of G into a chosen metric space, typically to preserve certain properties such as edge lengths (i.e., distances) by minimizing distortion. Thus, IC placement has a natural correspondence with graph embedding into a chosen two-dimensional metric space representing the chip surface. For ASICs, Euclidean or Manhattan metrics correlate well with delays of connections and average congestion. However, as Fig. 1 illustrates, FPGA placement requires a very different metric to model delays accurately.

FPGAs typically consist of a heterogeneous array of programmable logic blocks (PLBs), with routing segments of different lengths that can be used to make fast direct connections between specific, but not necessarily adjacent, elements in the array [23, 25]. It is advantageous to use these direct connections where possible, to minimize use of the relatively slow SRAM-based switches that program the routes.

Timing-driven FPGA placement tools typically use move-based algorithms such as recursive partitioning [13,15], clustering [6], simulated annealing [5, 6, 11, 18], or graph-based search [26]. Efficient look-up tables [11], empirical models [15, 21], or detailed delay models [18, 26] capture the dependence of delay on the routing architecture. Critical paths are targetted using timing analysis and net-weighting [6, 11, 13], or incremental or adaptive delay estimation schemes [13, 18]. Some commercial tools use analytical wirelength minimization methods(e.g., [3]) for global placement along with net-weights from timing analysis; however delays on the routing grid are not modeled directly in the objective.

CAPRI complements these existing approaches by producing an initial legal placement via an *analytical* framework that models the delay contours of the FPGA routing grid during *global* optimization. Furthermore, use of graph distance matrices and their largest singular vectors (rather than Laplacian matrices and their smallest eigenvectors) makes CAPRI fundamentally different from earlier spectral (eigenvalue-based) discrete placement methods such as [2, 4], as well as computationally more efficient.



(a) Overall CAPRI placement methodology

(b) Conceptual Overview of CAPRI (Phase 1)

Figure 2: Overview of CAPRI

3. PLACEMENT STRATEGY OVERVIEW

Our overall placement methodology is a two-phase approach, as shown in Fig. 2(a). CAPRI is used in *Phase 1* with the goal of producing a good initial placement from a timing perspective. Intuitively, our objective is to reduce delays on driver-sink connections, in a global sense. *Phase 2* is a local optimization step to improve specific critical paths and routability, performed using any existing move-based technique, such as low-temperature simulated annealing. A conceptual overview of CAPRI (i.e., Phase 1) is shown in Fig. 2(b). We abstract both the design netlist and the routing architecture as graphs, build a metric space corresponding to the architecture, and then embed the netlist graph into that metric space to minimize distortion.

Our problem formulation is in the form of an *assignment problem*; a crucial component is a graph-drawing technique [9] using distance matrices, which is used to build the appropriate metric space to model delays on the FPGA grid. While this formulation has a convex quadratic objective function, the underlying solution space is discrete (i.e., non-convex), and convex optimization techniques cannot be directly applied. Therefore, a key component of CAPRI is a fast and effective heuristic to minimize the above objective using two steps: (1) a fast analytical step to minimize distortion using low-rank approximations of the distance matrices, producing a concurrent illegal placement of all nodes in the netlist, and (2) a legalization step using online bipartite matching to find a legal assignment. This format of a fast analytical step followed by legalization is typical of most analytical placers.

In CAPRI, we do not presently model routing congestion explicitly in the objective function. Instead congestion is implicitly controlled via a hybrid net-model which allows segments of the same net to share routing switches and improve routability. Doing so also helps performance, since meandering nets could incur delay penalties by requiring a large number of switches on the FPGA.

The analytical framework in CAPRI is easy to generalize. It accommodates a wide variety of architectures with heterogeneity in both the routing grid and the logic array, models physical constraints that occur in practice, and can be applied hierarchically to scale to large problem sizes. Furthermore, since CAPRI is architecture-aware, it could also be used to evaluate early architectural decisions for an FPGA.

4. PLACEMENT BY ASSIGNMENT

We construct an undirected graph called the *Architecture Graph* to represent the FPGA (e.g., Fig. 3). Vertices in this graph (called *locations* for short) correspond to the fixed locations of logic elements on the FPGA. Edges represent fast direct routing connections between locations. Edges representing the fastest type of connection (typically between adjacent locations) are given unit weights; edges representing longer connections are assigned appropriate positive weights to account for their delays relative to edges of unit weight. We also transform the design netlist into an undirected graph called the *Design Graph*; each vertex is called a *logic-node*. Edges in the Design Graph connnect the driver of each net with its sinks, and are given unit weights.

We assume, for ease of exposition, that (a) a logic-node in the Design Graph can be assigned to any location in the Architecture Graph, and (b) the number of logic-nodes, n, is equal to the number of locations. Our formulation easily accommodates a more general scenario, as shown later in Sections 4.2 and 5.4. A unique ID between 1 and n is assigned to each location in the Architecture Graph, and to each logic-node in the Design Graph. In our placement formulation, we seek to embed the Design Graph into the metric space of the Architecture Graph, i.e., to find a good assignment of a location ID to each logic-node ID. We measure the quality of the assignment by the total distortion of the embedding.

Our rationale for this choice of objective is as follows. Suppose that logic-nodes could be assigned to locations such that each edge in the Design Graph (with weight of 1) could be connected with a best-case delay (i.e., total weight) of 1 along the edges in the Architecture Graph. Essentially such a solution would have no distortion of the edges in the Design Graph, and would produce the best possible performance on the FPGA. Note that such an assignment is feasible only in the special case when the Design Graph and the Architecture Graph are isomorphic, which is almost never the case in practice. Thus, some distortion of edges is inevitable in realistic settings. By minimizing the extent by which edges in the Design Graph are stretched or distorted when embedded into the Architecture Graph, we tend to reduce the delay of driver-sink connections (or edges) and obtain a better global placement from a timing perspective. Just like wirelength reduction for ASICs minimizes delays of connections by minimizing their geometric length, our approach minimizes delays of connections by minimizing distortion in a metric space that captures FPGA delays.

4.1 Formulation of Objective

We represent the metric-space of a graph using its shortest-path *distance matrix*, which is defined as follows. Let G be a graph with n vertices. Let each vertex be assigned a unique ID between 1 and n. Let d_{ij} represent the length of the shortest path between vertices with IDs i and j in G. The distance matrix D_G, is an $n \times n$ matrix, where $D_G(i,j) = d_{ij}$. Note that d_{ij} satisfies the definition of a distance metric given in Section 2, and thus D_G is a metric space that represents G. We compute d_{ij} efficiently using breadth-first search for unweighted graphs, or Dijkstra's algorithm for weighted graphs, respectively [20].

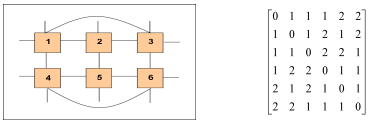


Figure 3: An Arch. Graph (edge weights 1) and distance matrix

We construct the $n \times n$ distance matrix D_A corresponding to the Architecture Graph (e.g., Fig. 3). The underlying geometric intuition is that each row of D_A represents the delays with respect to a single location on the FPGA, similar to the contours in Fig. 1. Thus D_A encapsulates delay contours for the entire Architecture Graph, representing the metric space of the FPGA. Similarly, we construct the $n \times n$ distance matrix D_D, corresponding to the Design Graph.

As noted earlier, the placement problem is equivalent to determining the optimal assignment of a location ID to each logic-node ID. This assignment is represented mathematically by an $n \times n$ permutation matrix P [7]. The column indices in P represent the node IDs and the row indices the location IDs. If $P(k,i) = 1$, then logic-node i is assigned to location k. Thus only one element in each row and each column of P can be 1; all others must be 0. The action of P on the Design Graph is represented by $P^T D_D P$.

To minimize distortion, P is chosen to minimize the difference between (a) the permuted distance matrix $P^T D_D P$ due to the assignment, and (b) the distance matrix D_A representing the metric space of the Architecture Graph. We express this mathematically by the objective $f_{obj} = \|P^T D_D P - D_A\|_F^2$. The operator $\|X\|_F = \sqrt{\sum_{i,j} X(i,j)^2}$, i.e., the Frobenius norm of the matrix X. P is orthogonal ($PP^T = I$), so we simplify f_{obj} to a convex quadratic function [1]: $f_{obj} = \|P \cdot (P^T D_D P - D_A)\|_F^2 = \|D_D P - P D_A\|_F^2$.

Our objective is to find P to minimize f_{obj}, subject to linear and integer constraints, as shown in Eqn. 1.

$$\begin{aligned} \min f_{obj} &= \|D_D P - P D_A\|_F^2 \\ \sum_{i=1}^{n} P(i,j) &= 1, \forall j = 1,2,\ldots,n \\ \sum_{j=1}^{n} P(i,j) &= 1, \forall i = 1,2,\ldots,n \\ P(i,j) &\in \{0,1\} \end{aligned} \tag{1}$$

These contraints express the fact that P is a permutation matrix, where exactly one element in each row and each column is 1, and all other elements are 0. Note that despite the convex objective f_{obj}, the $\{0,1\}$ constraints on the elements of P restrict the solution space to a non-convex set. Thus, convex optimization techniques such as gradient descent cannot be directly applied to solve this problem [1]. In fact, a formulation of the type shown in Eqn. 1, termed a *Binary Quadratic Assignment Problem*, is known to be NP-hard [19]. The approach we take in CAPRI is to approximate f_{obj} such that assignments for the n logic-nodes can be found without first solving an optimization problem in n^2 variables. Furthermore, these approximations also en-

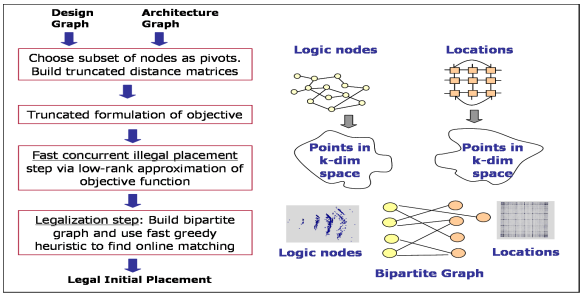


Figure 4: Overview of CAPRI Implementation

able us to effectively overcome the difficulties associated with the large, dense matrices present in our formulation. We describe our approach in detail in Section 5.

4.2 Features and Practical Considerations

Our placement formulation has several features of practical importance. In particular, linear constraints on the permutation matrix P are used to accommodate restrictions that might fix or confine certain logic blocks in the design to certain locations on the FPGA. For example, if k is an illegal location for logic-node i, we set variable $P(k,i) = 0$. Similarly $P(k,i) = 1$ if logic-node i is fixed at location k. Thus, only *legal* locations for *mobile* logic-nodes are unknown variables in the optimization. These constraints ensure that the assignment is *resource aware* when the FPGA has slices of different resources (e.g., control and datapath), large macros, and pre-placed or constrained I/Os.

In CAPRI, I/Os are placed simultaneously with other logic-nodes in the design, thereby providing additional flexibility during optimization. *No fixed objects are required to seed the placement.* At the same time, we accommodate I/Os that are constrained or fixed in a top-down flow. Furthermore, a logic-node in the Design Graph and a location in the Architecture Graph can represent either the most fine-grained logic elements on the fabric or larger clusters of these elements. Thus CAPRI can be applied at any level of granularity, enabling a hierarchical methodology.

5. IMPLEMENTATION ISSUES

We now present key insights and approximations that are used to find a good solution to the NP-hard assignment problem formulated in Section 4.1. First we describe a truncated formulation of Eqn. 1, which can be constructed efficiently. Next we present our placement heuristic, which comprises two steps — a fast analytical illegal placement step to minimize distortion, followed by legalization. An overview of our implementation of CAPRI is shown in Fig. 4.

5.1 Truncated Formulation

We use the truncated formulation described below to compute the distance matrices from Eqn. 1 efficiently in log-linear time in the size of the graph, rather than log-quadratic time. Note that we only need to compute D_D. D_A can be computed offline and stored for a given routing architecture.

Using Dijkstra's shortest path algorithm with Fibonacci heaps, we can construct a distance matrix in $O(n(n\log n + |E|))$ time, where n is the number of vertices in the graph, and $|E|$ the number of edges [20]. We simplify this expression to $O(n^2 \log n)$ since the graphs corresponding to typical designs and FPGA architectures are sparse. For more efficient matrix construction we use a small subset of vertices, called *pivots*, to approximate a graph's distance matrix. Thus, this *truncated* distance matrix would now just include distances between all vertex-pivot pairs, rather than those between all pairs of vertices. This approximation was proposed in [9], where it is shown that a small and *practically constant* number of pivots can effectively capture the topology of very large graphs, especially if these pivots are chosen to be well spread out through the graph, such as with Gonzalez's k-center selection algorithm [8]. Mathematically, pivot selection amounts to picking a small number of *dominant rows* (say m) to approximate the complete $n \times n$ distance matrix. The complexity of building the truncated $m \times n$ distance matrix reduces to $O(m(n\log n + |E|))$, or $O(n \log n)$ for sparse graphs and constant m. Constant values of $50 < m < 100$ are used for graphs with millions of vertices in [9]; we found that $m = 50$ works well.

Assume that we pick m pivot-nodes in the Design Graph, and m pivot-locations in the Architecture Graph[1] Let the corresponding $m \times n$ truncated distance matrices be represented by $\hat{D_D}$ and $\hat{D_A}$. The assignment is now represented by two permutation matrices P_1 (mapping pivot-nodes to pivot-locations) and P_2 (mapping all nodes to locations). Eqn. 2 gives the corresponding objective function $\hat{f_{obj}}$. The derivation is similar to that of Eqn. 1 from Section 4; the constraints on P_1 and P_2 are similar to those on P.

$$\hat{f_{obj}} = \|P_1(P_1^T \hat{D_D} P_2 - \hat{D_A})\|_F^2 = \|\hat{D_D} P_2 - P_1 \hat{D_A}\|_F^2 \quad (2)$$

In practice, the choice of pivots has significant impact on the quality of the solution. In CAPRI, we use Gonzalez's k-center algorithm [8] to pick m vertices that are well spread out through the graph. During the course of this selection, we also give priority to choosing those vertices which lie on critical paths, are I/Os, or have large degree.

5.2 Fast Analytical Illegal Placement Step

The first step of our placement heuristic is a low-rank approximation [7] of the truncated distance matrices. As explained below, this approximation expresses the dense rectangular matrices $\hat{D_A}$ and $\hat{D_D}$ as the product of significantly smaller (and therefore more tractable) matrices; it can also be computed efficiently.

A rank-k approximation of an $m \times n$ matrix D is given by $\bar{D} = U\Sigma V^T$; Σ is a diagonal $k \times k$ matrix containing the k dominant (i.e., largest) singular values of D, U is an orthonormal $m \times k$ matrix containing the corresponding k dominant left singular vectors of D, and V is an orthonormal $n \times k$ matrix containing the corresponding k dominant right singular vectors of D. When $k \ll rank(D)$, $\bar{D}$ is called a low-rank approximation of D. Essentially, it is a least-squared approximation of D as a product of the significantly smaller matrices U, Σ, and V. For a rank-k approximation to be effective, the k largest singular values must *dominate*, i.e., the remaining $(rank(D) - k)$ singular values must be significantly smaller and decay quickly to 0. A detailed discussion of low-rank approximations is given in [7].

Geometrically, a rank-k approximation of a distance matrix D for a set of points Γ, represents a projection of the points in Γ onto that k-dimensional hyperplane which will best preserve the distances in D in a least-squared sense in k dimensions. Eqn. 3 and Eqn. 4 describe the low-rank approximations of $\hat{D_D}$ and $\hat{D_A}$, respectively, using the notation from the previous paragraph. The resulting approximation of the objective $\hat{f_{obj}}$ to $\bar{f_{obj}}$ is given in Eqn. 5.

$$\hat{D_D} \approx \bar{D_D} = U_D \Sigma_D V_D^T = U_D \sqrt{\Sigma_D} \sqrt{\Sigma_D} V_D^T \quad (3)$$

[1]For ease of exposition, we assume here that the number of pivot-nodes, m, is equal to the number of pivot-locations. However this constraint is not required in our minimization strategy (see Section 5.4).

$$\hat{D_A} \approx \bar{D_A} = U_A \Sigma_A V_A^T = U_A \sqrt{\Sigma_A}\sqrt{\Sigma_A} V_A^T \quad (4)$$

$$\bar{f_{obj}} = \|U_D \sqrt{\Sigma_D}\sqrt{\Sigma_D} V_D^T P_2 - P_1 U_A \sqrt{\Sigma_A}\sqrt{\Sigma_A} V_A^T\|_F^2 \quad (5)$$

The columns of the matrix $\sqrt{\Sigma_A} V_A^T$ correspond to points in k-dimensional space — essentially a projection of the metric space of the Architecture Graph into k dimensions, as shown by the examples in Fig. 5. Likewise, the columns of $\sqrt{\Sigma_D} V_D^T$ essentially represent our k-dimensional *concurrent illegal placement* of the logic-nodes in the Design Graph (e.g., Fig. 6(a)). This placement is *illegal* in the sense that it is on a k-dimensional hyperplane rather than the two-dimensional chip surface, and also contains overlaps between logic-nodes. Similarly the rows of $U_D \sqrt{\Sigma_D}$ correspond to a k-dimensional placement of the pivot-nodes alone, and the rows of $U_A \sqrt{\Sigma_A}$ to a k-dimensional projection of the metric space corresponding to the pivot-locations alone.

The dominant singular values of the matrices $\hat{D_D}$ and $\hat{D_A}$ for typical designs decay very sharply (e.g., Fig. 6(b)). Thus, small values of $2 \leq k \leq 6$ yield good approximations. In addition, given the spread in singular values for these typical matrices, we use Orthogonal Iterations to efficiently compute the rank-k approximation [7], making this placement step fast. We find that the Orthogonal Iterations method converges in a small number of iterations (typically less than 30), each iteration taking $O(kn)$ time [7].

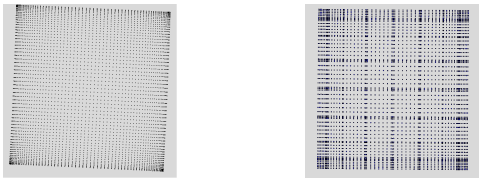
In our current version of CAPRI, we find that $k = 2$ works well, therefore, our fast analytical placement step is equivalent to projecting the Design and Architecture Graphs onto planes. To understand why this placement step is *architecture-aware*, recall that the plane chosen via the rank-2 approximation is one that best preserves distances in the distance matrix, and that these distances in turn correspond to delays of connections along the FPGA routing grid.

5.3 Legalization Step

Having obtained an abstract k-dimensional coordinate for each logic-node in the Design Graph and each location in the Architecture Graph, the next step is *legalization*, where each logic-node is assigned to a legal location corresponding to a logic element on the two-dimensional FPGA surface. Essentially we *map* the points in k-dimensional space representing the logic-nodes, to the points in k-dimensional space representing the metric space of the Architecture Graph. (Recall that the latter points correspond to legal locations on the FPGA.) As shown in Fig. 4, we construct a bipartite graph $\mathcal{B}$ with the two partite sets representing the logic-nodes and locations, respectively. Edges in $\mathcal{B}$ connect each logic-node to potential legal locations; each edge is given a positive weight (referred to as *cost*) to reflect the cost of assigning the logic-node to the corresponding location.

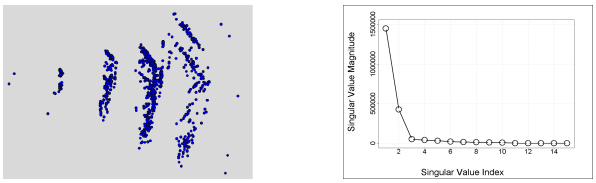
From the structure of the decomposition in Eqn. 5, a heuristic for minimizing $\bar{f_{obj}}$ is to assign logic-nodes to locations to minimize the k-dimensional distance between them in the projection from Section 5.2. Thus, one component of the cost on the edges of $\mathcal{B}$ is the k-dimensional distance between the corresponding logic-node and location; this component minimizes distortion in the embedding. A second component reflects the bounding-box wirelength of the nets connecting a logic-node to its previously placed neighbors. It models the fact that sharing routing segments improves routability, thereby *controlling congestion*. Costs are weighted by each logic-node's timing criticality.

We determine a good assignment via a minimum weighted matching on the bipartite graph $\mathcal{B}$ [20]; however the following critical issues must be considered. First, $\mathcal{B}$ could potentially have a quadratic number of edges (e.g., in the case when every logic-node could be assigned to every location), making it expensive to build. Second, the cost assignments to edges in $\mathcal{B}$ are not static — for example, if a given node η has been assigned to some location, we would like this assignment to be reflected in the costs of assigning any neighbors of η to their potential locations. We overcome these difficulties by using a *dynamic* rather than a static bipartite graph, and an *online* (i.e., dynamic) heuristic to find a matching. In this online matching scheme, we maintain the set of locations in a quad-tree [16], and consider the logic-nodes in order, from most critical to least critical. Rather than construct all edges upfront, as a given logic-node η is considered, we *dynamically* construct the edges corresponding to potential locations for η, and then greedily determine the location corresponding to the edge with best cost. Importantly, we prune the set of potential edges to first consider the legal available locations which are close to η's k-dimensional coordinate in the quad-tree, and expand the search outwards only when necessary. Pruning edges results in substantial runtime savings. Note also that by considering logic-nodes in order of their criticality, we ensure that logic-nodes on critical paths are placed in locations that are best for timing, while less critical logic-nodes "negotiate" for the remaining locations.



(a) 2-D grid. (b) Virtex-like grid.

Figure 5: Rank-2 approximation ($k = 2$) of metric space corresponding to two architecture graphs. Note the clusters of locations in the metric space of a Virtex-like architecture, due to its complex hierarchy of direct routes.



(a) Illegal placement of design *des* with $k = 2$. (b) Largest 15 singular values for a typical matrix.

Figure 6: Low-rank approximation

5.4 General Comments

Note that our implementation strategy does not impose strict constraints on the relative sizes of $\hat{D_A}$ and $\hat{D_D}$, since we find the low rank approximation of these matrices separately, and the corresponding k-dimensional points form the individual partite sets in the bipartite graph $\mathcal{B}$. Thus, we do accommodate different sizes of the Design and Architecture Graphs, and different number of pivot-nodes and locations.

Recall that we use a star-model for nets (edges connect the driver with each sink) when building distance matrices; this captures best-case timing on a net and is used to compute the k-dimensional coordinates. We also use a bounding-box model during the legalization step to model sharing of route segments, implicitly controlling congestion. Currently, we do not have a global analytical model for FPGA congestion (a challenging problem in its own right). However this is one focus of our ongoing work on modeling routing resources via the adjacency matrix of the graph.

The local optimization step (Phase 2 in Fig. 2(a)) can be performed using any existing move-based FPGA placement

technique. For simplicity, and to enable comparison with VPR, we currently use low-temperature simulated annealing in the VPR framework. Alternatively, one could use faster methods such as [6, 15] in this phase.

6. EXPERIMENTAL HIGHLIGHTS

We present highlights of experiments performed on 12 of the largest MCNC FPGA benchmarks [24]; each benchmark is a netlist of 4-input lookup tables (4-LUTs). The FPGA array in each case consists of a homogeneous array of PLBs, each containing one 4-LUT. All routing switches are buffered. For each benchmark, the placement area chosen is the smallest square array that accomodates all logic-nodes and I/O pads. All experiments were conducted on a 1GHz Sun Ultra-sparc processor. We compare the following methodologies to place both I/Os and PLBs concurrently:

(a) **CAPRI:** We use a C++ implementation of CAPRI to produce an initial solution, followed by a cool anneal in VPR for local optimization (c.f., Fig. 2(a)). The initial anneal temperature for the cool anneal is set to approximately 10^{-4} times the normal initial temperature in VPR, and the *timing-weight* parameter in VPR is set to 0.8. For a fair evaluation of CAPRI's initial placement, we made no other changes to tune the annealer or annealing schedule in VPR.

(b) **VPR:** This is the standard timing driven placement flow in VPR by simulated annealing with the *timing-weight* parameter set to 0.8.

We use VPR to route the placements in both flows. Fig. 7 shows the improvement with CAPRI in the delay of the worst 20 critical paths after detailed routing for two architectures: (a) a heterogeneous architecture with routes spanning 1 PLB and 4 PLBs, and (b) a complex heterogeneous architecture which mimics the Xilinx Virtex FPGA [25]. Improvements in the single-most critical path are similar, but are not shown here due to lack of space.

These comparisons are obtained at routing channel widths close to the minimum width, at which placements from both the CAPRI and VPR flows can be routed. The CAPRI flow does better in most designs with an improvement in critical path delays of 10.08% in the median, and 11.13% in the mean. The CAPRI flow does worse on the design `elliptic` at these small routing widths primarily because the design is heavily congested. By adding a few more routing tracks to alleviate congestion for this particular design, we obtained improvements of 5.9% and 6.7% with CAPRI's initial placement on the architectures in (a) and (b) above, respectively. We believe that our ongoing work of incorporating routing congestion more directly in CAPRI's objective function will improve results for such designs.

When CAPRI is used to produce an initial solution, we also obtain an average speedup of 2.05x in total placement runtime as shown in Fig. 8. The time to run CAPRI alone is only 4.8% of this total runtime on average.

7. CONCLUSION

In summary, we present CAPRI, a new analytical framework that uses graph-embedding concepts to accurately model routing architectures during FPGA placement. CAPRI is used as a global optimization step to produce a good initial placement, and preliminary experimental results look very promising. In extensions to this work, we are currently looking at similar analytical models for routing congestion, a hierarchical methodology, and different options for the local optimization phase. We also plan to look into using CAPRI for FPGA architecture exploration.

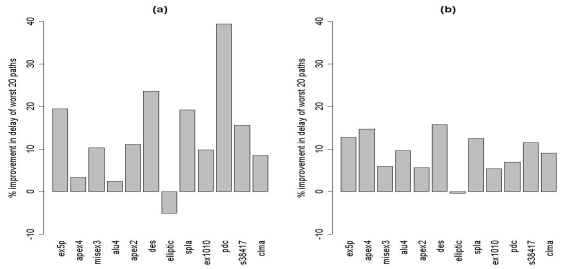


Figure 7: CAPRI vs. VPR : Percentage improvement in the delay after detailed routing of top 20 critical paths using CAPRI's initial placement on two FPGA routing architectures : (a) With routes spanning 1 PLB and 4 PLBs, and (b) Virtex-like.

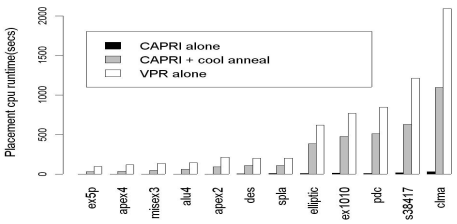


Figure 8: CAPRI vs. VPR : Runtime comparison

Acknowledgements: We thank Prof. R. Rutenbar, A. Ranjan, B. Taylor, and the reviewers for their feedback.

8. REFERENCES

[1] D. P. Bertsekas. *Nonlinear Programming*. Athena, 1999.
[2] J. P. Blanks. Near-optimal placement using a quadratic objective function. In *DAC*, 1985.
[3] H. Eisenmann and F. M. Johannes. Generic global placement and floorplanning. In *DAC*, 1998.
[4] J. Frankle and R. M. Karp. Circuit placements and cost bounds by eigenvector decomposition. In *ICCAD*, 1986.
[5] G. Parthasarathy, et al. Interconnect complexity-aware FPGA placement using Rent's rule. In *SLIP*, 2001.
[6] G.Chen and J.Cong. Simultaneous Timing Driven Clustering and Placement for FPGAs. In *FPL*, 2004.
[7] G.H.Golub and C. Loan. *Matrix Computations*. JHU, 1983.
[8] T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Comp. Sci.*, 38, 1985.
[9] D. Harel and Y. Koren. Graph Drawing by High Dimensional Embedding. In *Graph Drawing*, 2002.
[10] M. Khellah, S.Brown, and Z.Vranesic. Modeling routing delays in SRAM-based FPGAs. In *Canadian Conf. on VLSI*, 1993.
[11] A. Marquardt, V. Betz, and J. Rose. Timing-Driven Placement for FPGAs. In *FPGA*, 2000.
[12] J. Matousek. *Lectures in Discrete Geometry*. Springer, 2002.
[13] M.Hutton, et al. Timing Driven Placement for Hierarchical Programmable Logic Devices. In *FPGA*, 2001.
[14] N.Selvakkumaran, et al. Partitioning Algorithms for FPGAs with Heterogeneous Resources. In *FPGA*, 2004.
[15] P.Maidee, et al. Fast Timing-driven Partitioning-based Placement for Island-style FPGAs. In *DAC*, 2003.
[16] R.A.Finkel and J.L.Bentley. Quad Trees: A Data Structure for Retrieval of Composite Keys. *Acta Informatica*, 4(1), 1974.
[17] R.Jayaraman. Physical Design for FPGAs. In *ISPD*, 2001.
[18] S.K.Nag and R.Rutenbar. Performance-driven Simultaneous Placement and Routing for FPGAs. *IEEE TCAD*, 17(6), 1998.
[19] S.Sahni and T.Gonzalez. P-complete approximation problems. *Journal of ACM*, 23, 1976.
[20] T.H.Cormen, C.E.Leiserson, and R.L.Rivest. *Introduction to Algorithms*. MIT Press, 1995.
[21] T.Karnik and S-M.Kang. An Empirical Model for Estimation of Routing Delays in FPGAs. In *ICCAD*, 1995.
[22] T.Taghavi, S.Ghiasi, A.Ranjan, S.Raje, and M.Sarrafzadeh. Innovate or Perish: FPGA Physical Design. In *ISPD*, 2004.
[23] www.altera.com.
[24] www.eecg.toronto.edu/~vaughn/challenge/challenge.html.
[25] www.xilinx.com.
[26] Y.Chang, et al. An architecture-driven metric for simultaneous placement and global routing for FPGAs. In *DAC*, 2000.