A Methodology for Fast FPGA Floorplanning *

John M. Emmert and Dinesh Bhatia†
Design Automation Laboratory
ECECS Department
P.O. Box 210030
University of Cincinnati
Cincinnati, OH 45221–0030

{jemmert,dinesh} @ececs.uc.edu

Abstract

Floorplanning is an important problem in FPGA circuit mapping. As FPGA capacity grows, new innovative approaches will be required for efficiently mapping circuits to FPGAs. In this paper we present a macro based floorplanning methodology suitable for mapping large circuits to large, high density FPGAs. Our method uses clustering techniques to combine macros into clusters, and then uses a tabu search based approach to place clusters while enhancing both circuit routability and performance. Our method is capable of handling both hard (fixed size and shape) macros and soft (fixed size and variable shape) macros. We demonstrate our methodology on several macro based circuit designs and compare the execution speed and quality of results with commercially available CAE tools. Our approach shows a dramatic speedup in execution time without any negative impact on quality.
Key Words: Floorplanning, Placement, FPGA, Clustering, Tabu Search

1 Introduction

Placement and floorplanning are extensively studied topics. However, the importance of placement and floorplanning cannot ever be ignored due to changing design complexities and requirements. One technology that is evolving very rapidly is field programmable gate array (FPGA). Currently, commercially available devices can map up to one million gate equivalent designs[19] (and some of the newly announced products like Altera's APEX series will map over two million gate equivalent designs[1]). Such complex design densities also demand tools that can efficiently and quickly make use of available gates.

Improvements in CAD tools for FPGAs have not kept pace with hardware improvements. The available tools typically require from minutes to hours to map[1] designs (or circuits) with just a few thousand gates, and as design sizes increase the execution time will increase. One way to address the problem of long mapping times is create designs that use premapped macros[2] to create larger designs (macro based circuits). Then we floorplan and route these macro based circuits. In general, floorplanning is an NP-hard problem [14]. For FPGAs, it is more difficult due to fixed logic resources.

To address the problem of mapping large designs to large FPGA circuits, we have taken a macro based approach [18]. We floorplan interconnected macro based circuits. At the lowest level a macro is composed of one or more interconnected and relatively placed logic blocks. In this paper we present a method (based on clustering and tabu search (TS) optimization) to quickly floorplan macro based circuits while attempting to minimize throughput delay and meet area and routability constraints.

The basic flow of our method is summarized as follows. We start with a set of macros (M) interconnected by a set of signals (S). We then group (cluster) macros together to form clusters. Each cluster in the set of clusters (B) is smaller in area than some predefined limit[3]. We then use TS optimization to perform two-dimensional placement on the set of clusters B. Then, for each cluster that is composed of more than one macro, we perform intracluster placement[4]. Finally, for any macro whose shape was changed during the intracluster placement process, we perform intramacro placement[5].

2 Floorplanning Problem

Given a set of macros $M = \{m_1, m_2, ..., m_n\}$ and a set of signals $S = \{s_1, s_2, ..., s_q\}$, we associate with each macro $m_i \in M$, a size a_i (number of logic blocks in m_i); a width w_i (maximum width of m_i in number of logic blocks); a height h_i (maximum height of m_i in number of logic blocks); a flexibility f_i (0 for hard/fixed macros or 1 for soft/flexible macros); and a set of interconnecting signals S_{m_i} ($S_{m_i} \subseteq S$). For hard macros (macros with fixed size, shape, and internal placement), w_i and h_i are both fixed and $f_i = 0$. For soft macros (macros with fixed size and variable shape), w_i and h_i are considered flexible (both w_i and h_i can take on a range of values typically between 1 and a_i) and $f_i = 1$. Additionally, with each signal $s_i \in S$ we associate a set of macros M_{s_i} where $M_{s_i} = \{m_j \mid s_i \in S_{m_j}\}$. M_{s_i} is said to be a ***signal net***. We can divide M into two distinct sets, MS and MH (subset of soft macros and subset of hard macros), where $M = \{ MH \cup MS \mid$

*this work is partially supported by Air Force Research Laboratory of the US Air Force under contract number F33615-97-C-1043

†corresponding author

[1]typical mapping steps include technology mapping, placement, and routing

[2]macros are predefined circuit components like adders, shifters, decoders, multipliers, signal processors, CPUs, etc.

[3]predefined limit implies the total area of each cluster (sum of the areas of the macros within the cluster) is less than some maximum

[4]intracluster placement is the task of assigning the macros that make up the cluster a physical location and reshaping any macro whose shape must be altered to meet area constraints

[5]intramacro placement is the process of relatively placing the logic blocks that make up a macro component

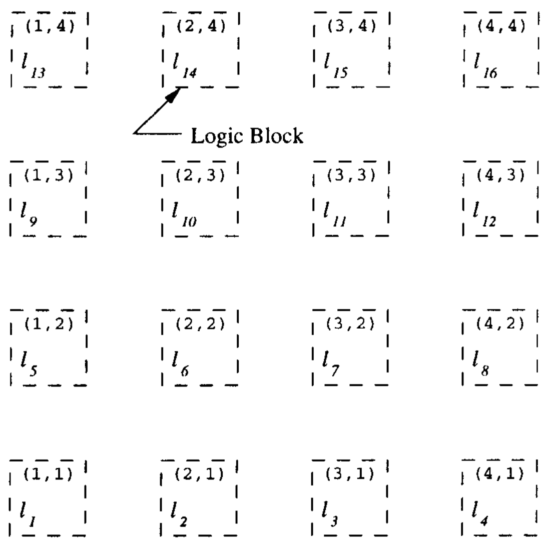

Figure 1: *Example two-dimensional array $L = \{l_1, l_2, ..., l_{16}\}$ of physical logic block locations ($W_L = 4$ and $H_L = 4$). One logic block can be assigned to each physical location $l_i \in L$.*

$MH \cap MS = \phi$, $f_i = 0 \; \forall \; m_i \in MH$, and $f_i = 1 \; \forall \; m_i \in MS$ }. We are also given a target set $L = \{l_1, l_2, ..., l_p\}$ of locations where $| \; L \; | \geq \sum_{i=1}^{|M|} a_i$. For the case of mapping $m_i \in M$ to a regular two-dimensional array, each $l_j \in L$ is represented by a unique (x_j, y_j) location[6] on the surface of the two-dimensional array where x_j and y_j are integers. Additionally, we define the two-dimensional array L by the width of physical logic block locations, W_L, and the height of physical logic block locations, H_L. The floorplanning problem then becomes how to assign each soft macro $m_i \in MS$ a shape and each macro $m_j \in M = MH \cup MS$ a unique location in L such that an objective function is optimized. Here uniqueness implies no macro overlaps. Figure 1 shows the 16 element set L for an example 4×4 two-dimensional array ($W_L = 4$ and $H_L = 4$). Our goal is to optimize the floorplanned circuit's performance while meeting area and routing constraints.

3 Related Work

Many recent papers have addressed placement and floorplanning for regular arrays. Rose et. al. use simulated annealing as the basis of their placement tool[2]. Saucier et. al. developed a floorplanner that matches the hierarchy of the circuit to the hierarchy of the target FPGA [9]. Mathur et. al. studied the placement problem and presented methods for re-engineering of regular architectures[13]. Togawa et. al. combined technology mapping, placement, and global routing[17]. Yamanouchi et. al. used partial clustering for macro based floorplanning of standard cells [18]. Callahan et. al. developed a module placement tool for mapping of data paths to FPGA devices [3]. Shi and Bhatia developed a force directed optimization based floorplanning tool for fast, high-performance floorplanning of FPGA mapped designs [15]. Krupnova et. al. combined the mapping and floorplanning stages to create a new method for mapping large, hierarchal designs to FPGAs [9]. In addition, over the past several years many non-deterministic, *random move* based solutions have also been considered [7][8]. These random move based methods typically achieve high quality results at the cost of long execution times. As circuit size increases, the time required for executing such random move methods becomes exorbitant. In most search based methods, there is a tradeoff between the execution time and the quality of the results.

Song and Vannelli developed a TS based placement algorithm for minimizing total wire length [16]. Their cost function is based on total wire length using the half-perimeter net model, and therefore, designed to enhance routability and not necessarily performance. They sum the total estimated length of all nets. Their cost function is based on allowing moves within a predefined window to define local neighborhoods. Their tabu list is composed of the most recently executed moves. Their method uses no aspiration criteria and no long term search strategy; and therefore, does not fully exploit the advantages of a TS based approach. They use their method to generate an initial placement for further refinement by other algorithms. Lim, Chee, and Wu have developed a placement with global routing strategy for placement of standard cells [11][12]. Their algorithm uses a hierarchical, divide and conquer, quad-partitioning approach. They use TS in their quad-partitioning routine. Their algorithm uses the concept of proximity of regions to approximate interconnection delays during the placement process.

4 FPGA Floorplanning

In this section we give an overview of our method, and in following subsections we describe each step in detail. First, some preliminary definitions are required. As stated earlier, a macro is a set of one or more interconnected and relatively placed logic blocks. We are given a set M of interconnected macros in our circuit or design netlist. When necessary we group macros in M together to form clusters. Therefore, we define a cluster as a set of one or more macros, and $B = \{b_1, b_2, ..., b_p\}$ as the set of all clusters. (For initialization, there is a one to one mapping of the elements of the set M to the elements of the set B, and therefore, initially $| \; B \; | = | \; M \; |$.) As stated earlier we are floorplanning the set of macros M on the two-dimensional array L of physical logic block locations. Once macros are grouped to form clusters, our approach is to perform two-dimensional placement of clusters on L. To perform this placement we divide our target two-dimensional array L into a two-dimensional array of buckets where each bucket (of physical logic block locations) has the same size and shape. (We define the bucket size by a width of W_B logic blocks and a height of H_B logic blocks.) We define the set of buckets as the set $\{l'_1, l'_2, ..., l'_m\} = L'$, where the number of buckets m equals $| \; L' \; |$. (The two-dimensional array L' is defined by a width of $W_{L'}$ buckets and a height of $H_{L'}$ buckets.) Then, instead of performing two-dimensional placement of clusters directly on L, we perform two-dimensional placement of clusters on the smaller set L'. Figure 2 shows the example L divided into four equally sized buckets of physical logic block locations where each bucket is 2 logic blocks $\times$ 2 logic blocks.

Figure 3 shows a flow chart of our floorplanning methodology. In figure 3, we read in the sets M, S, and L. Next, we initialize the set of clusters B. Initially, each element of B contains one element from M, so there is a one to one mapping of the elements of M to the elements of B. After initialization of B, we initialize the bucket width, W_B, and bucket height, H_B, using the Procedure Create_Buckets(M). Details for Create_Buckets(M) are found in subsection 4.1. After bucket size initialization, we create the set of buckets, L', as outlined in subsection 4.2. Next, we check the fit of B on L'. (It should be noted that we create and maintain the bucket width W_B and bucket height H_B so any single macro in M will fit in any bucket in L' [7]. This allows us to skip the clustering step if $| \; B \; |$ is less than or equal to $| \; L' \; |$. This usually occurs when low device utilization is sufficient and allows for very fast

[6]for our application, the location represents a physical logic block location on the FPGA

[7]The initial bucket size is based on the dimensions of the largest elements of M.

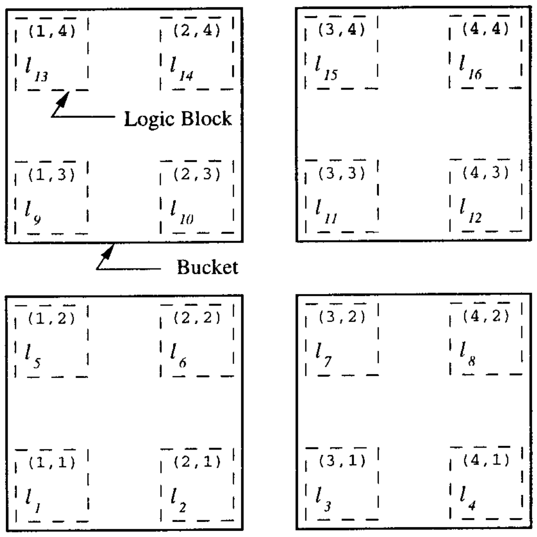

Figure 2: *Example L divided into a set L' of 4 buckets. The dimensions of L' are $W_{L'} = 2$ buckets and $H_{L'} = 2$ buckets. The dimensions of the example bucket are $W_B = 2$ logic blocks and $H_B = 2$ logic blocks.*

floorplanning.) If there is a fit, we proceed to the placement phase if there is not a fit we proceed to the clustering phase.

In figure 3, if $| B |$ is not less than or equal to $| L' |$, we proceed to the clustering phase. To ensure fit, our methodology requires $| B |$ is less than or equal to $| L' |$, and therefore, the goal of the clustering phase is to group smaller macros together thereby reducing $| B |$ until $| B |$ is less than or equal to $| L' |$. Additionally, it is required that each cluster $b_i \in B$ has size less than or equal to the bucket size. This ensures each cluster will fit in any bucket. The details of clustering are found in subsection 4.3. After clustering if $| B |$ is less than or equal to $| L' |$ then we proceed to the placement phase else we iteratively increase the bucket size (as described in subsection 4.4) and continue clustering until $| B |$ is less than or equal to $| L' |$ or the bucket size exceeds the dimensions of L.

In figure 3, if $| B |$ is less than or equal to $| L' |$, we proceed to the placement phase. In the placement phase we use TS based placement to assign each $b_i \in B$ to a bucket (see subsection 4.5). Then in intracluster placement, we assign each macro within each cluster a physical location and shape (see subsection 4.6). Finally in intramacro placement, we place the logic blocks within any soft macro whose shape has been altered during intracluster placement (see subsection 4.7). After this phase, every logic block making up the circuit or design netlist will have a physical location on the two-dimensional array L. The floorplanning process is summarized in Algorithm TS_FP(M,S,L).

In figure 3, if the circuit or design netlist will not fit we have four options. We can reduce the size of the macro set M by partitioning the design spatially or temporally. We can increase the size of the target two-dimensional array L. This assumes a larger FPGA part is available. We can flatten the netlist and attempt to use standard placement techniques. This will become more difficult as design sizes get larger. Finally, if possible we can soften some of the hard macros to allow better space utilization.

4.1 Initializing Bucket Size

In this subsection, we describe the method for determining the initial bucket size which subsequently defines $| L' |$. The main goal of our floorplanning method was fast execution time. Therefore, we quickly initialize the width of the bucket, W_B, to the width of

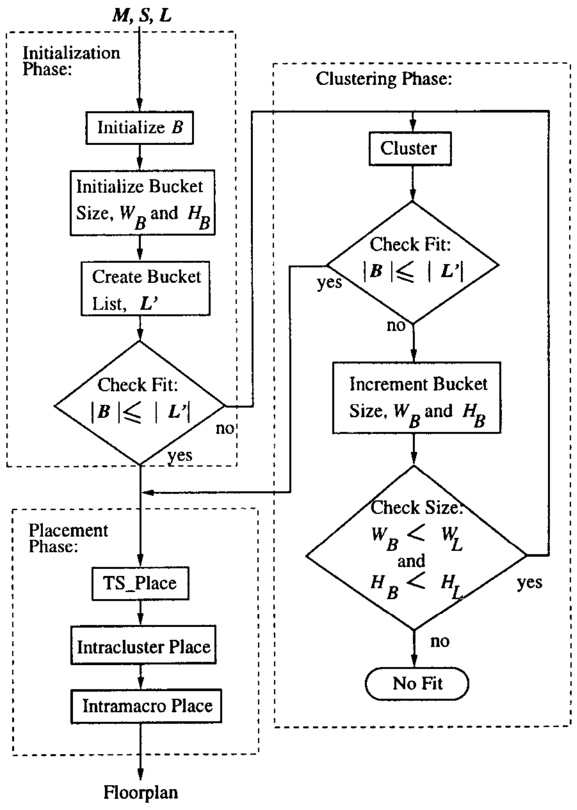

Figure 3: *Floorplanner execution flow.*

```
Algorithm TS_FP($M$,$S$,$L$)
  begin
    (* initialize the buckets *)
    $\forall\, m_i \in M$ let $b_i = \{m_i\}$;
    (* determine initial bucket size, $H_B$ and $W_B$ *)
    Create_Buckets($M$);
    create $L'$ where $W_{L'} = \lfloor \frac{W_L}{W_B} \rfloor$ and $H_{L'} = \lfloor \frac{H_L}{H_B} \rfloor$;
    success = checkfit($B$,$L'$);
    while(NOT success AND $W_B < W_L$ AND $H_B < H_L$)
      $B$ = cluster($M$,$S$,$H_B$,$W_B$);
      success = checkfit($B$,$L'$);
      if NOT success then
        increment bucket size ($H_B$ and/or $W_B$);
        update $L'$ so $W_{L'} = \lfloor \frac{W_L}{W_B} \rfloor$ and $H_{L'} = \lfloor \frac{H_L}{H_B} \rfloor$;
      end if;
    end while;
    if success then
      TS_place($B$,$S$,$L'$);
      $\forall\, b_i \in B${
        intracluster_place($b_i$,$H_B$,$W_B$);
        $\forall\, m_j \in b_i$ intramacro_place($m_j$,$b_i$,$H_B$,$W_B$);
      }
    else;
      return "ERROR: circuit not floorplanned";
    end if;
  end;
```

```
Procedure Create_Buckets($M$):
  begin
    initialize $W_B = H_B = 0$;
    for $i = 1$ to $| M |$
      if $W_B < W(m_i)$ then
        $W_B = W(m_i)$;
      end if;
      if $H_B < H(m_i)$ then
        $H_B = H(m_i)$;
      end if;
    end for;
    return $W_B$ and $H_B$;
  end;
```

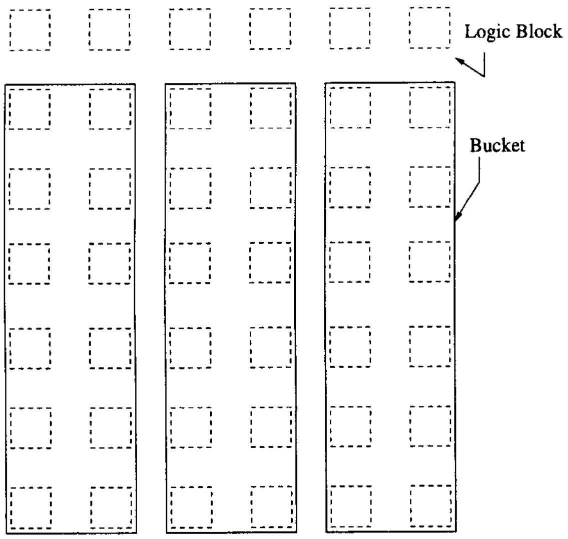

Figure 4: *Example L' made up of three* 6×2 *buckets.*

the widest macro cell (hard or soft[8]). Similarly we initialize the height of the bucket, H_B, to the height of the tallest macro cell (hard or soft). This guarantees that any macro $m_i \in M$ will fit in any bucket. The procedure used to determine the initial W_B and H_B is shown in Procedure Create_Buckets(M). In procedure Create_Buckets(M), $H(m_i)$ returns the height of macro m_i and $W(m_i)$ returns the width of macro m_i.

4.2 Bucket List, L'

The set of buckets[9], L', is created by dividing the set L into rectangles of equal size. The width of L' (in number of buckets) is defined as $W_{L'} = \lfloor \frac{W_L}{W_B} \rfloor$ and the height of L' (in number of buckets) is defined as $H_{L'} = \lfloor \frac{H_L}{H_B} \rfloor$. (Note, W_L and H_L define the width and the height (in number of logic blocks) respectively of the two-dimensional array L.) Therefore $| L' | = H_{L'} \times W_{L'}$. Figure 4 shows an example L' for a 7 logic block $\times$ 6 logic block L (W_L = 6 and H_L = 7) and a 6 logic block $\times$ 2 logic block bucket (W_B = 2 and H_B = 6).

4.3 Clustering

As stated earlier, the set B is created or initialized by assigning each $m_i \in M$ to $b_i \in B$, and initially, $| B | = | M |$. When necessary, the size of set B is reduced by clustering elements of M

[8]This assumes soft macros are supplied with some initial shape. Effort is made to maintain the shape of soft macros. The shape of soft macros is only changed if required to make the circuit fit the given area.

[9]a bucket is a set or group of physical logic block locations from L such that each bucket has the same size and shape ($H_B \times W_B$)

```
Procedure Cluster($M$,$S$,$H_B$,$W_B$,$L'$):
  begin
    $\forall\, m_i \in M$ let $b_i = \{m_i\}$;
    calculate $c_{ij}$ $\forall\, b_i$ and $b_j \in B$;
    while $| B | > | L' |$ AND $\exists c_{ij} > 0$
      choose $m_i$ and $m_j$ with highest connectivity, $c_{ij}$;
      let $b_i = m_i \cup m_j$;
      let $b_j = \phi$;
      update connectivity between clusters;
    end while;
    return $B$;
  end;
```

so more than one element of M is in some $b_i \in B$. There is no limit placed on the maximum number of macros in each b_i as long as size constraints are satisfied. Size restrictions (described below) limit the macros used to form each cluster, $b_i \in B$.

Each cluster $b_i \in B$ is divided into two parts, a hard macro part and a soft macro part. The size restriction on b_i requires the total area of the hard macro part plus the total area of the soft macro part be less than or equal to the size of the bucket ($H_B \times W_B$). We define the width of the hard macro part of each cluster b_i as the sum of the width of the hard macros in b_i,

$$HMW(b_i) = \sum_{\forall m_j \in b_i | m_j \ is \ hard} W(m_j) ,$$

where $W(m_j)$ is the width of macro m_j in cluster b_i. We define the area for the hard macro part for each cluster b_i as the width of the hard macro part times the height of the bucket

$$HMA(b_i) = HMW(b_i) \times H_B \ .$$

The size for the soft macro part for b_i is defined as the width of the bucket minus the width of the hard macro part times the height of the bucket

$$SMA(b_i) = (W_B - HMW(b_i)) \times H_B \ .$$

The sum of the areas of all soft macros in b_i must be less than or equal to $SMA(b_i)$.

With these area constraints in mind, the set M is clustered to form the set B. The clustering method is derived from the connectivity work done in [18]. The connectivity cost function includes area constraints. Our connectivity cost function is summarized below.

$$c_{ij} = feas(i,j) \cdot \sum_{s_k \in S_{m_i} \cap S_{m_j}} \frac{1}{(| s_k | - 1)} \cdot \frac{A_{tot}}{a_i + a_j} \cdot \frac{min(a_i, a_j)}{max(a_i, a_j)}$$

where a_i and a_j are areas of macro m_i and m_j respectively, A_{tot} is the total area of all macros, $| s_k |$ is the number of pins on signal s_k which connects macros m_i and m_j, $S_{m_i} \cap S_{m_j}$ is the set of all signals that connect macros m_i and m_j, and $feas(i,j)$ returns the feasibility of clustering m_i and m_j under size constraints described above. $feas(i,j)$ returns a 1 if it is possible to combine m_i with m_j else it returns a 0.

The clustering algorithm combines clusters with the highest connectivity to form larger clusters. In order to enhance routability, once area constraints have been met (i.e. $| B | \leq | L' |$) the algorithm stops and returns the set B. The clustering algorithm is summarized in Procedure Cluster(M,S,H_B,W_B,L').

After clustering is complete, it returns the set B. The empty elements of B are removed, and each $b_i \in B$ consists of a unique list of elements from M. Here uniqueness implies $b_i \cap b_j = \phi \ \forall\, b_i \wedge b_j \in B \mid i \neq j$.

Macro Statistics			
m_i	w_i	h_i	f_i
m_1	3	3	0
m_2	3	3	0
m_3	2	3	1
m_4	2	3	1
m_5	2	3	1

Table 1: *Macro statistics for example floorplan.*

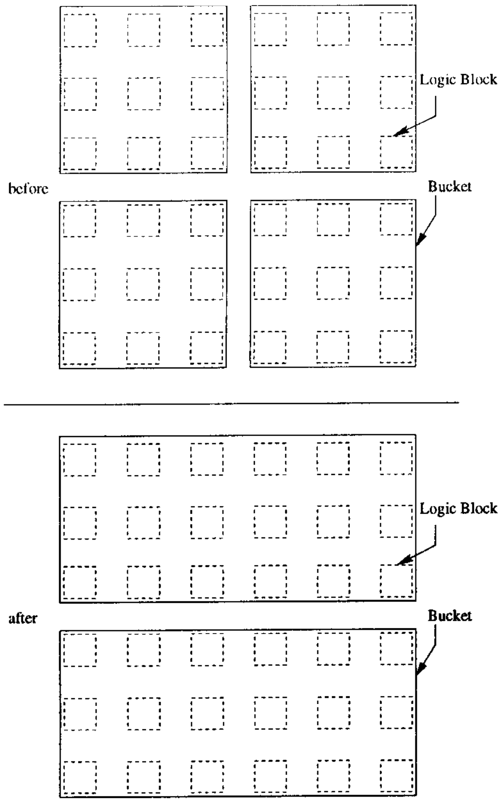

Figure 5: *Example L' made up of four* 3×3 *buckets converted to two* 3×6 *buckets.*

4.4 Increment Bucket Size

In the event that the first pass of clustering does not lead to a valid solution, the bucket size is increased to allow more flexibility during clustering. This increases the complexity of intracluster placement but allows more macros to fit in the same area. For example, consider floorplanning the 5 macros described in table 1 so they fit on an L with $W_L = H_L = 6$ and $|L| = 36$. For the set M, both W_B and H_B will be set to 3 since these values reflect the largest macro width and height respectively. Figure 5 shows the buckets on L. Therefore L' will initially have 4 buckets and M will not fit since $|B| > |L'|$. However, by doubling the width of the bucket, we can cluster m_1 and m_2 into one cluster and m_3, m_4, and m_5 into a second cluster that will fit in L.

4.5 Cluster Placement

Once the circuit is guaranteed to fit ($|B| \leq |L'|$) then the clusters $b_i \in B$ are placed using a two-step tabu search[10] (TS) based two-dimensional placement algorithm [5]. The first step of the placement strategy minimizes the circuit's total wire length (see TS_TWL in section 4.5.1) thereby enhancing the routability of the circuit. The second step attempts to average the circuit's edge lengths by weighting graph edges and minimizing the maximum weighted edge lengths (see TS_EDGE in section 4.5.2).

For our TS approach, we convert each multi-terminal net to a set of edges where each edge consists of the driving terminal and one driven terminal. We use this model to keep net sources and sinks in close proximity thereby enhancing circuit performance. We create the set of edges by converting the hyper-graph input circuit model described earlier to a graph $G = (V, E)$ where $V = \{v_1, v_2, ...v_n\}$, $|V| = n, E = \{e_1, e_2, ...e_m\}$, and $|E| = m$. Each vertex $v_i \in V$ corresponds to a cluster $b_i \in B$ (if pad IO locations are available, we also include preplaced pseudo-elements of V representing the pad locations to help guide the placement). Each edge $e_i \in E$ connects a pair of vertices $(v_j, v_k) \mid v_j, v_k \in V$. The elements of E are created by considering each signal, $s_i \in S$. If we let m_{source} (where $m_{source} \in M_{s_i}$ and $m_{source} \in b_j$) be the source macro for signal s_i then an edge (v_j, v_k) is added to E for each sink on s_i such that $m_{sink} \in M_{s_i}$, $m_{sink} \in b_k$, and $j \neq k$. (In other words, an edge is added for each source/sink combination that are not in the same cluster.) At any given time, each element of V is mapped to a unique element of L', and the minimum requirement for mapping is $|V| \leq |L'|$.

The two-dimensional placement stage basically assigns each cluster to a unique bucket. After placement of each $b_i \in B$, each $b_i \in B$ will have associated with it a unique bucket $l'_j \in L'$. The physical location (on L) of each $b_i \in B$ in bucket l'_j can be found from the following equations:

$$X(b_i) = X(l'_j) \times W_B$$

and

$$Y(b_i) = Y(l'_j) \times H_B.$$

where $X(l'_j)$ returns the X-axis coordinate of l'_j on L' and $Y(l'_j)$ returns the Y-axis coordinate of l'_j on L'. After each cluster $b_i \in B$ is assigned a unique location on L, intracluster placement takes place to assign each $m_j \in b_i \in B$ a physical location on L. Intracluster placement also reshapes soft elements $m_k \in MS \in B$ that require further modification.

4.5.1 TS_TWL

For the first step of our TS based placement strategy, TS_TWL, we seek to enhance routability by minimizing total wire length (*TWL*). We conservatively estimate *TWL* using the Manhattan length of each edge $e_i \in E$, and we seek to minimize the following function:

$$TWL = \sum_{\forall e_i \in E} MLength(e_i)$$

where $MLength(e_i)$ is the Manhattan length of edge e_i.

[10]tabu search is a meta-heuristic approach to solving optimization problems that (when used properly) approaches near optimal solutions in a relatively short amount of time compared to non-deterministic *random move* based methods [6]. Unlike approaches like simulated annealing or genetic algorithms that rely on a good random choice, TS exploits both good and bad strategic choices to guide the search process. As a meta-heuristic, TS guides local heuristic search procedures beyond local optima. In TS, a list of possible moves is created. In the short term, as moves in the list are executed, *tabu*, or restrictions, are placed on the executed moves in order to avoid local optima. This *tabu* is typically in the form of a time limit, and unless certain conditions are met (e.g. *aspiration criteria*), the move will not be performed again until the time limit has expired.

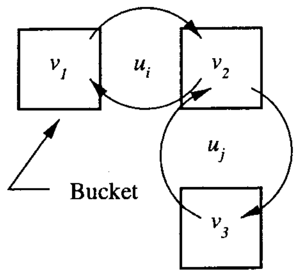

Figure 6: *Example Horizontal and Vertical Moves.*

Key to the development of a TS is a search list. For TS_TWL our search list U consists of all possible swaps of vertices occupying adjacent locations in L'. This implies two basic swap moves: horizontal (swap of adjacent vertices with the same y coordinate) and vertical (swap of adjacent vertices with the same x coordinate). Given a two-dimensional array L' of width $W_{L'}$ units and height $H_{L'}$ units, there are $| U | \approx 2(H_{L'} \times W_{L'})$ possible swaps or moves in U. Figure 6 shows an example horizontal swap move u_i and vertical swap move u_j. In figure 6, move u_i represents the horizontal swap of vertices v_1 and v_2, or moving v_1 to v_2's bucket and v_2 to v_1's bucket. For TS_TWL, given a random initial placement in L' (by selecting an appropriate sequence of moves from U), we seek to optimize our objective function, minimization of TWL.

In TS_TWL, each $u_i \in U$ has an associated attractiveness, AF_i, or sum of the adjacent forces pulling on the vertices v_j and v_k that make up u_i. U is ordered so the most attractive moves are first. For vertical moves

$$AF_i = M(v_j) \times PE(v_j) + M(v_k) \times PW(v_k) \quad ,$$

and for horizontal moves

$$AF_i = M(v_j) \times PN(v_j) + M(v_k) \times PS(v_k) \quad .$$

Each vertex $v_i \in V$ has one multiplication factor $M(v_i)$ (discussed later) and four associated pulls or forces: $PN(v_i)$, $PE(v_i)$, $PS(v_i)$, and $PW(v_i)$. The pulls are determined by summing the Manhattan lengths of the edges connecting v_i to vertices in the direction of the pulls. If we used U in a typical greedy search strategy (i.e. given an initial placement, find a move that would improve the minimum TWL) we would quickly reach a local optima. However, by applying the concepts of TS (i.e. accepting strategic moves that may not improve the current minimum TWL), we climb out of local optima. After executing move $u_i \in U$ we set a *tabu tenure* for u_i. Move u_i will not be executed again until the *tabu tenure* has expired or our *aspiration criteria* is satisfied. Initially, $\forall\ v_i \in V$, $M(v_i)$ is set to 1. For diversification, we penalize moves that are executed with high frequency in order to take the search into unexplored areas. We do this by increasing $M(v_i)$ for low frequency moves, thereby making them more attractive.

4.5.2 TS_EDGE

The second step of our TS based placement strategy, TS_EDGE, seeks to enhance circuit performance by minimizing the length of critical circuit edges. To accomplish this, we traverse G and apply a weight w_i to each edge $e_i \in E$. Edges in critical paths receive a higher weight. For TS_EDGE, we use a two part optimization function. First we minimize the weighted length of the longest edge. Second, since some configurations may have the same longest weighted edge length, we add together N of the longest edges (NLE) and minimize NLE.

$$NLE = \sum_{i=1}^{N} MLength(e_i) \times w_i \quad .$$

For TS_EDGE, we use the edge list E as our search list. We order E in descending order by weighted Manhattan length. Then, we search E looking at each of the two vertices attached to each edge as possible candidates for a move. The vertices attached to the edges with the longest weighted Manhattan lengths are the most attractive candidates for moving closer together. By moving these vertices closer together, the longest edges are shortened thereby enhancing circuit performance and reducing the longest paths. Once an edge is selected from the search list, we look at only one of the edge's two vertices as a possible move candidate. For simplicity we pick one of two possible moves for the vertex selected: vertical swap or horizontal swap (discussed earlier relative to TS_TWL). In TS_EDGE, given an initial placement in L' (by selecting an appropriate sequence of moves from E), we seek to optimize our objective function, minimization of the longest weighted edge length and minimization of NLE.

After executing a move for a vertex on edge $e_i \in E$, we set a *tabu tenure* (number of iterations a vertex' position is locked) for the moved vertex. This vertex on edge e_i will not be moved again until the *tabu tenure* has expired or our *aspiration criteria* is satisfied. In this way we climb out of local minima and accept the current *best* move even if it does not improve the current best solution.

4.6 Intracluster Placement

Once each cluster is assigned a location on L, the macros making up each cluster must be placed. Each macro $m_j \in M$ has associated with it a reference coordinate used to describe its physical location on the FPGA. Each logic block within each m_j also has a reference coordinate that describes its physical location relative to the reference coordinate for m_j. Intracluster placement is the task of assigning a reference coordinate from the set L to each macro $m_j \in b_i, \forall b_i \in B$, and, for any soft macro in M whose shape has changed, the task of assigning a set of reference coordinates for the logic blocks within the soft macro[11].

Intracluster or intrabucket placement for each $b_i \in B$ takes place in three steps. First, we place all hard macros by assigning each one an X,Y reference coordinate corresponding to some $l_j \in L$. Second, we place all soft macros by assigning each one an X,Y reference coordinate from L. Third, we change the shape of any soft macro that requires modification by assigning it a set of logic block coordinates relative to the reference coordinate of the soft macro. Figure 7 shows an example set of macros to be placed in the 9×12 Bucket 6 located at coordinates $X = 12$ and $Y = 18$. In figure 7 each hard macro is labeled with $f = 0$ and each soft macro is labeled with $f = 1$. In this subsection we will describe each of the steps for intracluster placement.

Our feasibility check during clustering guarantees the hard macros in each b_i will fit by ordering them in the horizontal direction. Therefore for each $b_i \in B$, we place hard macros in a row, each with the same Y-axis coordinate. The Y-axis coordinate of each hard $m_j \in b_i$ is found from the following equation:

$$Y(m_j) = Y(b_i)$$

where $Y(b_i)$ returns the Y-axis coordinate (from the set L) of the bucket where cluster b_i was placed. To compute the X-axis coordinate of each hard $m_j \in b_i$ a sort key is computed for each hard $m_j \in b_i$ by averaging the X-axis coordinates of all $b_k \in B$ connected to m_j (this includes IO position information). Then the hard macros in b_i are reverse ordered according to the sort key and stored in an ordered list $\{q_1, q_2, ..., q_n\} = Q$. After ordering each hard macro in b_i, the X-axis coordinate of each hard macro in b_i is

[11]Note: here only a set of reference coordinates is assigned for the set of logic blocks in the soft macro. The specific coordinates for each logic block in an altered soft macro are found during intramacro placement.

```
Procedure Find_Soft_X($b_i$,$r_k$):
  begin
    if $X(b_i) + X(r_{k-1})$ is even then
      if $lastY(r_{k-1}) \neq Y(b_i) + H_B - 1$ then
        $X(r_k) = X(r_{k-1})$;
      else
        $X(r_k) = X(r_{k-1}) + 1$;
      end else if;
    else
      if $lastY(r_{k-1}) \neq 0$ then
        $X(r_k) = X(r_{k-1})$;
      else
        $X(r_k) = X(r_{k-1}) + 1$;
      end else if;
    end else if
  end;
```

determined by the following. If we let q_k denote the kth element in the reverse ordered list of hard macros in b_i, then

$$X(q_k) = X(q_{k-1}) - W(q_k) \;\forall\; k > 1$$

where $W(q_k)$ is the width of macro q_k, $X(q_{k-1})$ is the X-axis coordinate of macro q_{k-1}, and $X(q_1) = X(b_i) + B_W - W(q_1)$. For our example macros in figure 7, since the Y-axis coordinate of the bucket is 18, the Y-axis coordinate for each hard macro(m_{16}, m_{19}, m_{27}, and m_{41}) is 18. If we assume the key for m_{16} is 3, m_{19} is 14, m_{27} is 13, and m_{41} is 43 then the X-axis coordinate for each hard macro is $X(m_{16}) = 16$, $X(m_{19}) = 20$, $X(m_{27}) = 18$, and $X(m_{41}) = 22$. Figure 8 shows the hard macros from figure 7 placed in example Bucket 6.

We now describe the method for determining the X,Y reference coordinates for each soft macro. Similar to the method of ordering the list of hard macros for b_i, a sorting key is determined for each soft $m_j \in b_i$ by averaging the X-axis coordinates of all clusters connected to soft macro m_j (this includes IO position information). Then the soft macros in b_i are ordered according to the sort key and stored in an ordered list $\{r_1, r_2, ..., r_n\} = R$. After ordering each soft macro in b_i the X,Y reference coordinate of each soft macro in b_i is determined. If we let r_k denote the kth element in the ordered list of soft macros in b_i then the X-axis reference location of r_k is found from the procedure Find_Soft_X(). In Find_Soft_X(), $lastY(r_k)$ returns the Y-axis coordinate of the last element in macro r_k and $X(r_0) = X(b_i)$. If it is required that the soft macro r_k's shape be adjusted, then its Y-axis reference location is $Y(b_i)$, but if the soft macro's shape does not require adjustment, then r_k's Y-axis reference location is set relative to the Y-axis location of the last logic block in r_{k-1} ($lastY(r_{k-1})$). If we assume $r_1 = m_{21}$, $r_2 = m_7$, $r_3 = m_6$, and $r_4 = m_{13}$ for the soft macros in the example shown in figure 7, then using the above methodology figure 9 shows the final placement and shape for the macros assigned to example Bucket 6.

4.7 Intramacro Placement

After assigning the reference coordinates for hard and soft macros in each cluster, the logic blocks that make up any reshaped soft macro are placed using intramacro_place(). Currently we use two methods for intramacro_place(), and both are described below. Instead of actually performing full placement on the logic blocks within the soft macro, we incrementally reconfigure the placement of the logic blocks using a transform that matches the X and Y coordinates of the soft macro to the X and Y coordinates of the available space on L.

The first method for incrementally reconfiguring the placement is of $O(n)$ complexity, where n is the number of logic blocks within the reshaped soft macro. Starting from the leftmost-lowest coordinate of the soft macro, the logic blocks within the soft macro

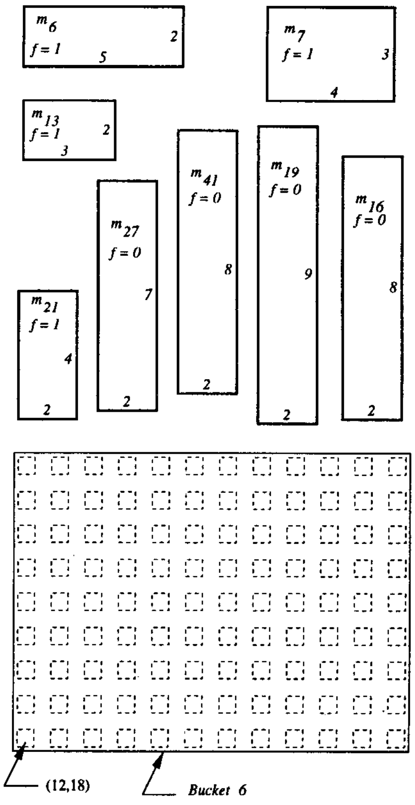

Figure 7: *Example set of hard and soft macros to be placed in Bucket 6 located at coordinate* (12,18).

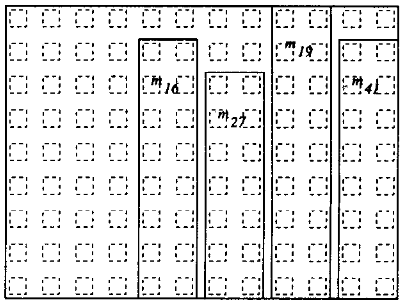

Figure 8: *Example hard macro placement for macros shown in previous figure.*

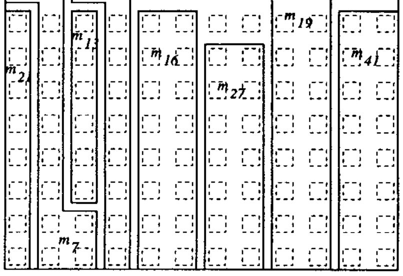

Figure 9: *Example placement of hard and soft macros.*

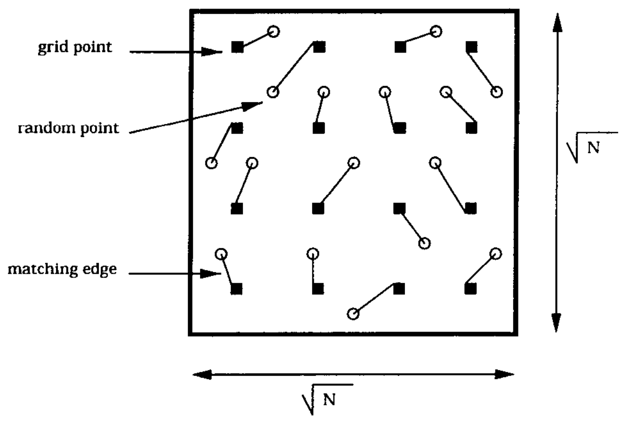

Figure 10: An instance of grid matching.

are matched to the leftmost-lowest coordinate available in the area of the bucket set aside for the soft macro. This methodology, though fast in execution, can substantially increase the length of nets connecting logic blocks; however, since the delay of the logic block is currently much greater than the interconnect delay, no substantial degradation to performance was noted.

The second method (designed to counter any performance degradation due to increased interconnect length) uses a minimax matching strategy to match locations of the logic blocks within the soft macro to coordinates available in the area of the bucket set aside for the soft macro. We use general minimax grid matching to accomplish this match. The problem of grid matching is stated below:

Instance[10]: A square with area N in the plane that contains N *grid points* arranged in a regularly spaced $\sqrt{N} \times \sqrt{N}$ array and N *random points* located independently and randomly according to a uniform distribution on the square as shown in figure 10.

Problem: Find the minimum length D such that there exists a perfect matching between the N grid points and N random points where the distance between matched points is at most D. D is also called the *minimax* matching length.

The problem of finding D for a given distribution of random points is solvable in polynomial time since the length D has an upper bound of $O(\sqrt{N})$. An algorithm that solves this problem constructs a bipartite graph between the grid points and random points. Let SG be the set of grid points and SR be the set of random points. The edges of the bipartite graph will be $< i, j >$ such that $i \in SG$ and $j \in SR$, and i, j are at most distance D apart. The algorithm starts with the construction of a bipartite graph for some initial D. It repetitively updates D, adding more edges, until a perfect matching is found in the bipartite graph. The D found by such an algorithm is also the *minimax* length. Leighton and Shor have proved a bound on the expected length of D for a random distribution of points which, with very high probability [12] is shown to be $\Theta(log^{3/4} N)$[10]. We use this tight and small bound to attain a reconfiguration that results in minimal impact on circuit performance.

Minimax matching attempts to minimize the maximum distance any logic block within the reshaped macro is displaced. More details can be found in [4].

5 Test Methodology

We empirically tested the floorplanning methodology described above using several macro based circuits (the circuits included both hard and soft macros). The top level macro based circuits were described using the `Xilinx Netlist Format` (XNF). The macros were also described using XNF files; however, they also included logic block placement information in the form of `RLOC`s so that all hard and soft macros were preplaced. The designs were mapped to the Xilinx XC4000E or XC4000XL family of FPGAs. Statistics for the macro based circuits are shown in table 2.

Macro Based Circuit Statistics						
Circuit Name	Part	$\mid M \mid$	Total Area	$\mid B \mid$	$\mid S \mid$	Num IOs
BOOTH	4013	64	264	72	473	33
CLA	4025	128	736	100	1024	133
CPU	4020	183	654	16	1051	38
MEDIAN	4013	39	295	15	392	80
MATMULT	4085	45	1998	35	891	306
BTCOMP	4036	97	403	81	768	264
XP-RI8	4025	31	723	12	417	170
XP-RI16	4085	18	2709	3	736	320
DCT	4085	122	3095	77	1089	113

Table 2: *Circuit statistics.*

For each circuit we obtained data for comparison in three ways. The first way we obtained data was to place and route flattened designs. We flattened each circuit netlist and removed all RLOC information. Then we used the Xilinx tools in the standard mapping approach (placement of logic blocks then routing of logic blocks) to map the circuit netlist. In following tables, the results of this method are shown in columns labeled **Xilinx Flat**. The second way we obtained data for comparison was to floorplan and route the macro based circuits using the Xilinx tools. In following tables, the results of this method are shown in columns labeled **Xilinx Macro**. The third way we obtained data was to floorplan the circuit with our TS_FP tool and route the circuits using the Xilinx tools. In following tables, the results of this method are shown in columns labeled **TS_FP Macro**.

We used statistics available for the Xilinx tools to compare the three mapping methods. Specifically, we used static timing analysis available from Xilinx tools to compare the quality of the mapped circuits and report data from Xilinx tools to determine placement and routing times for Xilinx tools. Table 3 shows the tool used to place (flat designs only) or floorplan (macro based designs) each of the circuits as well as the Xilinx tool suite used for routing and static timing analysis. We used the `unix` *time* function to determine system floorplanning times for TS_FP.

6 Results and Analysis

Table 4 shows the execution times required to floorplan (or place in the case of the flattened netlists) the circuits. Column **TS_FP Macro** shows the execution times required by our methodology. Columns **Xilinx Flat**[13] and **Xilinx Macro**[14] show the execution times required by the `Xilinx` tools. Column **TS_FP Macro** shows a 45X improvement in execution time for our methodology over that of the commercial `Xilinx` tools. Table 4 also demonstrates execution speedup for working with macro based circuits versus flattened netlists. (It should be noted that the DCT design was not floorplanned using the `Xilinx` tools. On our `Sun Ultra 2`, we experienced memory faults during the circuit mapping process using the `M1` tool. For the same reason, we could not route or perform static timing analysis on the DCT design after floorplanning with our methodology; however, floorplanning execution time using our

[12]very high probability means probability exceeding $1 - 1/N^{\alpha}$ where $\alpha = \Omega(\sqrt{logN})$.

[13]flattened netlist placed and routed by the `Xilinx` tools

[14]macro based netlist placed and routed by the `Xilinx` tools

tool is shown.) All circuits (that did not cause memory faults) were 100% routable.

Table 5 shows the results of static timing analysis performed on the floorplanned circuits (Note: this data is taken from completely routed circuits). The values shown indicate the worst case pad to pad delay (in the case of combinational circuits) or the minimum allowable clock period (in the case of sequential circuits). From table 5 we see the circuits floorplanned with our floorplanning methodology are similar in quality to those floorplanned by the commercial tools. Table 5 also shows there is not a substantial difference between delays encountered for our circuits with flat versus macro based netlists. This is probably due to the fact that logic block delay (for short distances or routes with few pips) is substantially greater than interconnect delay.

Table 6 gives the time taken for the Xilinx tools to route the circuits. This table shows the time taken to route our floorplanned designs is similar to that of the Xilinx placed and routed designs. It should be noted that this time could be significantly reduced by using not just preplaced macros, but preplaced and prerouted macros.

Figures 11, 12, 13, and 14 show example floorplans (from TS_FP) for the CLA, CPU, MATMULT, and DCT circuits respectively.

7 Conclusions

We have presented a performance driven fast floorplanning methodology for floorplanning macro based circuits. The methodology includes a clustering algorithm, placement algorithms, and a transform algorithm to quickly floorplan large macro based circuits. While flattening the netlist should provide better (relative to performance) results during the placement phase of the circuit, ever increasing circuit densities require an alternative method to handle large circuits in a timely (relative to execution time) fashion. Our approach shows dramatic improvement in the execution time without significant impact on quality of the mapped design.

References

[1] Altera Inc. *http://www.altera.com.*

[2] V. Betz and J. Rose. VPR: A New Packing, Placement, and Routing Tool for FPGA Research. In *Lecture Notes in Computer Science*, volume 1304, pages 213–222. Springer-Verlag, 1997.

[3] T. J. Callahan, P. Chong, A. DeHon, and J. Wawrzynek. Fast Module Mapping and Placement for Datapaths in FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 123–132, Feburary 1998.

[4] J. M. Emmert and D. K. Bhatia. Reconfiguring FPGA Mapped Designs with Applications to Fault Tolerance and Reconfigurable Computing. In *Lecture Notes in Computer Science*, volume 1304, pages 141–150. Springer-Verlag, 1997.

[5] J. M. Emmert and D. K. Bhatia. University of Cincinnati Technical Report Number: TR219/09/98/ECECS, 1998.

[6] F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publishers, 1997.

Placement or Floorplanning Tools			
Circuit Name	**Xilinx Flat**	**Xilinx Macro**	**TS_FP Macro**
BOOTH	PPR	PPR	TS_FP
CLA	PPR	PPR	TS_FP
CPU	PPR	PPR	TS_FP
MEDIAN	PPR	PPR	TS_FP
MATMULT	M1	M1	TS_FP
BTCOMP	M1	M1	TS_FP
XP-RI8	PPR	PPR	TS_FP
XP-RI16	M1	M1	TS_FP
DCT	M1	M1	TS_FP

Table 3: *Tools used for placing (flat netlist) or floorplanning (macro based netlist) test circuits. All circuit were routed using the corresponding Xilinx Router. All timing static timing analysis was performed on routed circuits.*

Execution times (cpu secs)			
Circuit Name	**Xilinx Flat**	**Xilinx Macro**	**TS_FP Macro**
BOOTH	131	36	3.1
CLA	87	61	5.5
CPU	210	101	6.9
MEDIAN	70	34	1.3
MATMULT	876	634	3.2
BTCOMP	107	87	1.2
XP-RI8	315	83	1.2
XP-RI16	698	92	2.6
DCT	–	–	3.2

Table 4: *Floorplanning or placement execution times.*

Static Timing Analysis (ns)			
Circuit Name	**Xilinx Flat**	**Xilinx Macro**	**TS_FP Macro**
BOOTH	49.5	46.8	50.0
CLA	97.2	105.1	124.4
CPU	95.6	106.9	103.3
MEDIAN	267.0	287.2	265.6
MATMULT	285.33	160.72	117.62
BTCOMP	124.09	150.74	127.58
XP-RI8	90.5	101.4	103.9
XP-RI16	296.86	283.70	289.21
DCT	–	–	–

Table 5: *Floorplanned/placed circuit (post route) static timing analysis results.*

Routing Times (cpu secs)			
Circuit Name	**Xilinx Flat**	**Xilinx Macro**	**TS_FP Macro**
BOOTH	38	31	53
CLA	307	386	374
CPU	410	376	332
MEDIAN	30	116	44
MATMULT	358	271	295
BTCOMP	25	32	33
XP-RI8	552	776	1106
XP-RI16	192	507	596
DCT	–	–	–

Table 6: *Floorplanned/placed circuit routing times.*

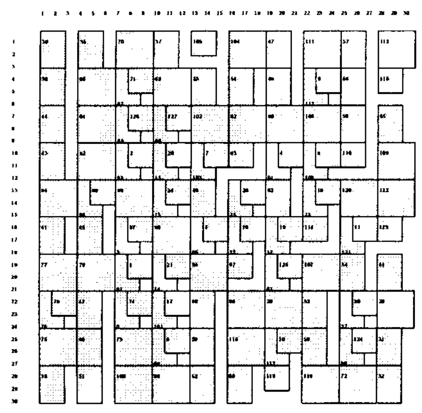
Figure 11: Floorplan for CLA circuit.

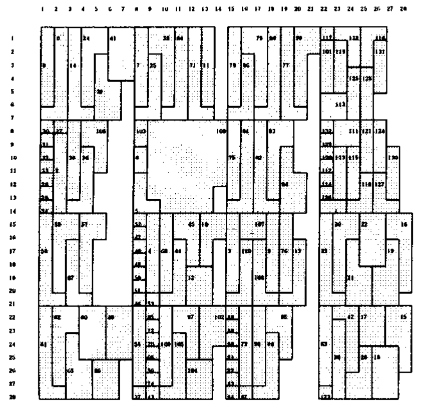
Figure 12: Floorplan for CPU circuit.

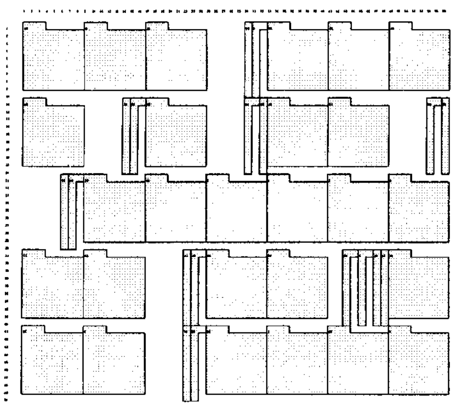
Figure 13: Floorplan for MATMULT circuit.

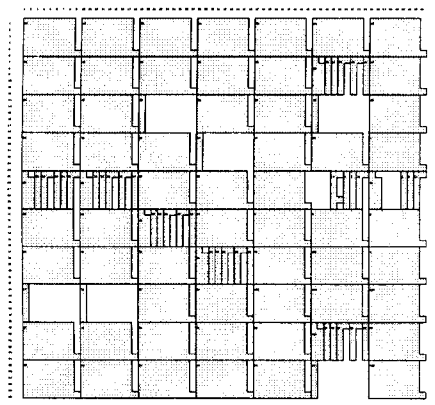
Figure 14: Floorplan for DCT circuit.

[7] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, 1989.

[8] S. Kirkpatrick, D. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, May 1983.

[9] H. Krupnova, C. Rabedaoro, and G. Saucier. Synthesis and Floorplanning for Large Hierarchical FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 105–111, Feburary 1997.

[10] F. T. Leighton and P. W. Shor. Tight Bounds for Minimax Grid Matching with Applications to Average Case Analysis of Algorithms. In *Proceedings of the Symposium on Theory of Computing*, pages 91–103, May 1986.

[11] A. Lim. Performance Driven Placement Using Tabu Search. *Informatica*, 7(1), 1996.

[12] A. Lim, Y. M. Chee, and C. T. Wu. Performance Driven Placement with Global Routing for Macro Cells. In *Proceedings of Second Great Lakes Symposium on VLSI*, pages 35–41, 1991.

[13] A. Mathur, K. C. Chen, and C. L. Liu. Re-engineering of Timing Constrained Placements for Regular Architectures. In *IEEE/ACM International Conference on Computer Aided Design*, pages 485–490, November 1995.

[14] S. M. Sait and H. Youssef. *VLSI Physical Design Automation*. IEEE Press, 1995.

[15] J. Shi and D. Bhatia. Performance Driven Floorplanning for FPGA Based Designs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 112–118, Feburary 1997.

[16] L. Song and A. Vannelli. A VLSI Placement Method Using Tabu Search. In *Microelectronics Journal*, number 3, pages 167–172, May 1992.

[17] N. Togawa, M. Yanagisawa, and T. Ohtsuki. Maple-opt: A Performance-Oriented Simultaneous Technology Mapping, Placement, and Global Routing Algorithm for FPGA's. *IEEE Transactions on Compter-Aided Design of Integrated Circuits and Systems*, 17:803–823, September 1998.

[18] T. Yamanouchi, K. Tamakashi, and T. Kambe. Hybrid Floorplanning Based on Partial Clustering and Module Restructuring. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 478–483, 1996.

[19] Xilinx Inc. *http://www.xilinx.com.*