

# MODULAR PARTITIONING FOR INCREMENTAL COMPILATION

Mehrdad Eslami Dehkordi, Stephen D. Brown

Dept. of Electrical and Computer Engineering  
University of Toronto, Toronto, Canada  
email: {eslami,brown}@eecg.utoronto.ca

Terry Borer

Altera Toronto Technology Center  
151 Bloor St. E., Toronto, Canada  
email: tborer@altera.com

## ABSTRACT

This paper presents an automated partitioning strategy to divide a design into a set of partitions based on design hierarchy information. While the primary objective is to use these partitions in an Incremental Design flow for compile time reduction, the performance of the partitioned design should not be degraded after partitioning. Experimental results using the incremental design feature of Quartus CAD tool from Altera show that our algorithm can generate partitioning solutions comparable with a set of manually partitioned real industrial circuits and results in more than 50% compile time reduction.

## 1. INTRODUCTION

With today's design sizes of more than a couple of million gates, reducing the complexity of the physical design process has become a necessity. Different approaches have been made in different stages of physical design CAD flows to reduce this complexity. Design partitioning is one of these approaches, in which the design is partitioned into smaller pieces while a certain objective is achieved, which could be reducing the number of connections among partitions, or minimizing the delay of the final partitioned design [1]. Another major problem with traditional physical design flows is dealing with huge design compile times even after applying minor changes to the design. The problem of reducing compilation time has been already addressed using traditional incremental compilation techniques, where any change in the design netlist is detected and passed to an updated physical design flow that is composed of incremental synthesis, incremental placement, and/or incremental routing [2]. These incremental phases will then perform the compilation process as fast as possible. However, since the initial compilation has been done without any knowledge of design information, a change in the design may need many changes in the final placed and routed circuit since the components participating in the design change may have been placed in different places of the target chip.

Recent state-of-the art CAD tools have added *incremental design* features to their CAD flows. A simple definition for incremental design is the ability to recompile a design using the information from a previous compilation. Conventionally, these tools by default does not differentiate between a full and an incremental compilation. As a result, the entire design is always processed from scratch when the compiler is invoked. Nevertheless, there are situations in which a more incremental compilation flow is desirable.

For example, a designer may reach a later point in the design cycle where she is uninterested in improving timing further. In this case, it is desirable to save compile time by reusing previous results for portions of the design that are unmodified.

In order to use such a feature the design must be properly partitioned. A good partitioning solution can be defined as one in which the performance of the partitioned circuit is not degraded and significant compile time savings is achieved after applying design changes. In this paper we present an algorithm that can be used to create partitions for this purpose. We will show that our partitioning creates partitions comparable with a set of manually partitioned industrial circuits with compile time savings of up to 50%.

Before continuing on to the next section we emphasize that the term "partitioning" used here is different from the traditional partitioning used in a custom physical design flow. While in the traditional flow any part of a design can be assigned to any partition, here we limit the partitioning process to design modules only. Also, as discussed in the next section, certain constraints based on design hierarchy information are applied to the partitioning process to comply with the requirements of the incremental design feature in Quartus.

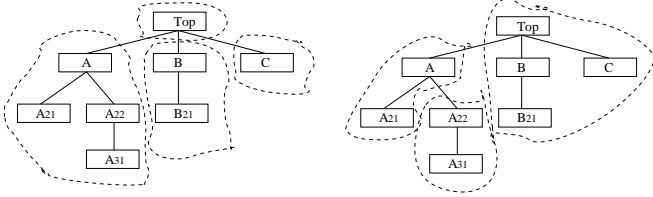
The rest of the paper is organized as follows: section 2 presents partitioning and incremental compilation definitions. Our modular partitioning algorithm is described in detail in section 3. Section 4 discusses experimental results, and finally, section 5 concludes the paper.

## 2. PARTITIONING AND INCREMENTAL COMPILATION

By definition a partition is a user-designated portion of the design where optimizations between it and the rest of the design are disallowed. The smallest design element that can be assigned to a partition is a design entity/module. Different modules can be assigned to a single partition only when they all belong to a single hierarchy. Fig. 1 shows a few valid partitioning solutions for a simple design hierarchy. The partition boundaries are specified with the dotted lines around the modules. As illustrated, it is possible to have sub-partitions, in which case they will be processed separately from the parent partition.

### 2.1. Partitioning Requirements

Besides compile time savings a partitioning solution should satisfy the following requirements: should not degrade performance;



**Fig. 1.** Valid partitioning solutions for a simple module tree

should not increase area significantly; and should have a reasonable number of partitions. If critical parts of a design are assigned to different partitions the performance of the final partitioned design may degrade significantly. This can happen due to absence of cross-boundary optimizations between partitions. Therefore, partitioning should not create partitions with too many critical paths crossing the partitions.

As for area, due to absence of cross boundary optimizations between partitions, the total area of the design may increase after partitioning. A good partitioning should not create a large amount of area increase. In this work we indirectly control the amount of area increase by reducing the number of partitions. Our experiments show that the area increase due to our partitioning is below 4%.

As the number of partitions increases the compile time saving will increase as well, but the performance may degrade due to preventing optimization among too many partitions. Therefore we should ensure that the number of partitions is reasonable.

### 3. MPART: MODULAR PARTITIONING

In this section we present our modular partitioning (mPart) strategy to create partitioning solutions based on the requirements explained in section 2.1. We use design hierarchy information to guide the partitioning process. mPart uses a simulated annealing methodology to improve the quality of the final partitioning solution based on a cost function. Our cost function contains information about criticality, connectivity, and size of the partitions in a partitioning solution. After reading the design netlist, a module tree is formed based on design hierarchy information. The module tree is actually the same as design hierarchy tree without its leaf nodes (design elements). The flow is then continued and after a preprocessing on the initial solution, the final partitioning solution is built using netlist information and the modules on the module tree. Finally, the partitioning solution is refined in a postprocessing stage. Details of the major steps of mPart flow including our simulated annealing methodology will be presented in subsequent sections.

#### 3.1. Partitioning Preprocessing

The preprocessing step starts with removing all the modules on the module tree that belong to the Library of Parameterized Modules as Quartus does not allow such modules to be assigned to partitions. We may also remove very small modules from the module

```

PS = initial_partitioning_solution();
T = initial_temperature();
while (outer_loop_criterion() == false) {
  while (inner_loop_criterion() == false) {
    prev_crit = crit(PS);
    prev_conn = conn(PS);
    prev_part_size_stdev = part_size_stdev(PS);
    PS_new = create_new_solution();
    Δcrit = crit(PS_new) - prev_crit;
    Δconn = conn(PS_new) - prev_conn;
    Δpart_size_stdev = part_size_stdev(PS_new) - prev_part_size;
    ΔC = α ·  $\frac{\Delta_{crit}}{prev\_crit}$  + β ·  $\frac{\Delta_{conn}}{prev\_conn}$  + γ ·  $\frac{\Delta_{part\_size\_stdev}}{prev\_part\_size\_stdev}$ ;
    if (ΔC < 0)
      PS = PS_new;
    else {
      if (random(0, 1) < e-ΔC/T)
        PS = PS_new;
    }
  }
  T = update_temperature();
}

```

**Fig. 2.** Pseudo-code for the annealing partitioner in mPart

tree (this means merging a small module into its parent and modifying the module tree). This seems like a valid decision as there is no point in assigning small modules to partitions as this cannot have any benefit in terms of compile time savings. But removing even such small modules may prevent us from finding better partitioning solutions. Therefore, we do not remove any module from the module tree based on size constraints in the preprocessing step. We later explain that this can be done in the postprocessing stage. Finally, each module on the module tree is assigned to a single partition to create an initial partitioning solution.

#### 3.2. Using Simulated Annealing for Partitioning

We use simulated annealing [3] to minimize a cost function associated with our partitioning strategy. Our simulated annealing partitioner initially starts with a partitioning solution based on the set of all initial partitions after preprocessing step. The solution is then iteratively improved by randomly changing the partitioning solution and evaluating the “goodness” of each change with a cost function. If the change results in a reduction in the partitioning cost, then the change is accepted. If the change would cause an increase in the partitioning cost, then the change still has some chance of being accepted even if it makes the partitioning worse. The purpose of accepting some “bad” changes is to prevent the simulated annealing based partitioner from becoming trapped in a local minimum. Fig. 2 shows the pseudo-code of the annealing. Details of the algorithm including our cost function and other simulated annealing parameters will be discussed in subsequent sections.

##### 3.2.1. Cost function

As mentioned in section 2.1 a good partitioning solution should have minimum number of inter-partition connections. Also, the number of critical paths crossing partitions should be minimal. In order to achieve a partitioning solution with such requirements we use a cost function that contains a criticality component and a connectivity component. The idea behind the criticality component is to minimize the number of critical paths crossing partitions. The

connectivity component tries to minimize total number of inter-partition connections and at the same time maximize internal connectivity of the partitions.

We use the well-know Rent formulation [4] for the connectivity component of our cost function. The rule relates the number of design elements,  $S$ , in a partition to the number of external connections,  $E$ , on a partition and is given by  $E = \bar{p} \cdot S^r$ , where  $\bar{p}$  denotes the average number of interconnections for a design element in the partition and  $r$  is Rent's exponent and is in the range of  $[0,1]$ . A value close to 1 indicates that most of the connections in the partition are external and a value near 0 indicates that almost all connections are internal. For each partition  $P$  we define connectivity as  $conn(P) = r(P)$ , where  $r(P)$  is the rent value associated with partition  $P$ .

Our criticality component is based on timing information using Quartus Timing Analyzer. We first give a brief overview of the timing analysis needed by our algorithm to get these timing information. The circuit netlist is represented as a graph where nodes in the graph represent input and output pins of circuit elements and I/O pads. Connections between these nodes are modelled with edges in the graph. These edges are assigned with delay and slack info based on the timing information derived from Quartus Timing Analyzer. We define the criticality of each edge  $e$  as

$$crit(e) = 1 - \frac{slack(e)}{\max_{\forall e_1} slack(e_1)} \quad (1)$$

The criticality provides an indication of the relative importance of each edge and is used in defining timing cost, which is the main term in our criticality cost component. The timing cost for each edge  $e$  is defined as  $t\_cost(e) = delay(e) \cdot crit(e)$ . For each partition  $P$  in the partitioning solution we define an internal criticality  $crit_{int}(P)$  as

$$crit_{int}(P) = \frac{\sum_{\forall e \in I(P)} t\_cost(e)}{|I(P)|} \quad (2)$$

where,  $I(P)$  is the set of internal edges in partition  $P$ . Similarly, we define an external criticality  $crit_{ext}(P)$  for each partition  $P$  as  $crit_{ext}(P) = \frac{\sum_{\forall e \in E(P)} t\_cost(e)}{|E(P)|}$ . The criticality cost associated with each partition  $P$  is then defined as

$$crit(P) = \frac{crit_{ext}(P)}{crit_{int}(P)} \quad (3)$$

The criticality and connectivity for a partition solution,  $PS$ , is then defined as the average criticality and connectivity of all the partitions in the partitioning solution, respectively:

$$crit(PS) = \frac{\sum_{\forall partition P \in PS} crit(P)}{|PS|} \quad (4)$$

$$conn(PS) = \frac{\sum_{\forall partition P \in PS} conn(P)}{|PS|} \quad (5)$$

We now present our cost function based on the criticality and connectivity components. As in [5], to properly balance the trade-off between the two components in the cost function, we use an auto-normalized cost function defined as

$$\Delta C_1 = \alpha \cdot \frac{\Delta crit}{prev\_crit} + \beta \cdot \frac{\Delta conn}{prev\_conn} \quad (6)$$

The auto-normalization cost function depends on the change in criticality and connectivity. Two normalization variables ( $prev\_crit$  and  $prev\_conn$ ) are used to normalize the weight of these two components. The effect of these variables is to make the function weight the two components only with the  $\alpha$  and  $\beta$  variables, independent of their actual values.

As can be seen, a partitioning solution containing a single partition (top-level module and all its sub-modules) will result in a low cost as all connections are put inside the partition. To resolve this problem we set a size constraint for the partitions. We control partition size by forcing partitions to be of equal size. In this way, we not only prevent partitions from growing into big parts, but also ensure that recompiling any partition will result in almost the same compile time for any partition.

We define a new cost component for our cost function to control partition size as

$$\Delta C_2 = \gamma \cdot \frac{\Delta part\_size\_stdev}{prev\_part\_size\_stdev} \quad (7)$$

where  $part\_size\_stdev$  is the standard deviation of the size of partitions. Finally, we present our cost function for the partitioning problem:

$$\Delta C = \Delta C_1 + \Delta C_2 \quad (8)$$

We have  $0 \leq \alpha, \beta, \gamma \leq 1$  and  $\alpha + \beta + \gamma = 1$ .

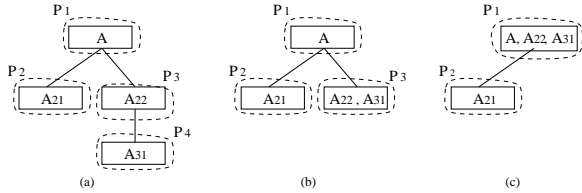
### 3.2.2. Creating new partitioning solutions

At each iteration of the annealing process a change in the partitioning process is made. We define two types of changes:

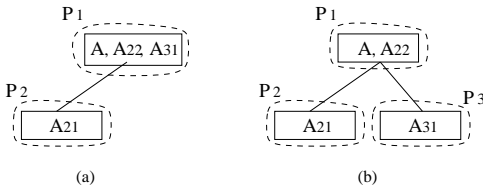
- *merge*: a partition is removed from the partitioning solution and all the modules inside the removed partition is merged with the parent partition. Parent partition is the partition containing the immediate parent module of the removed module in the module tree. Fig. 3(a) shows a simple module tree where each module on the tree is assigned to a single partition. A *merge* operation for partition  $P_4$  will remove that partition and merge its module  $A_{31}$  with the parent partition  $P_3$ , as shown in Fig. 3(b). If another *merge* operation occurs for  $P_3$  then the new module tree and the resulting partitioning solution of Fig. 3(c) will be resulted.
- *split*: a module that is already assigned to a partition is removed from that partition and a new partition is created for the module. Fig. 4(a) shows the partitioning solution resulted in Fig. 3(c). A *split* operation for module  $A_{31}$  will result in the partitioning of Fig. 4(b).

### 3.2.3. Annealing parameters

In this section we discuss different parameters that control the annealing process. The starting temperature for the annealing is obtained using a method similar to that described in [5]. A set of  $n$  moves is randomly generated. Each move is then evaluated and the change in cost is observed. The initial temperature is computed to be 20 times the standard deviation of the set of cost changes. At each temperature in the anneal,  $n$  moves are generated and evaluated. The value of  $n$  is equal to the number of partitions in the



**Fig. 3.** Creating new partitioning solution: *merge* operation (a) module tree showing 4 modules each assigned to a single partition, (b) merging  $P_4$  with  $P_3$ , (c) merging  $P_3$  with  $P_1$



**Fig. 4.** Creating new partitioning solution: *split* operation (a) partitioning solution and the updated hierarchy of Fig. 3(c), (b) after splitting  $P_1$  based on module  $A_{31}$

current solution. As the partitioning process advances this number is reduced and few moves will be made as we reach the end of annealing.

At each temperature a change in the partitioning solution will be made using a *merge* or *split* operation. A move is randomly selected and is applied to randomly selected partitions in the current solution. Once the move at a particular temperature has been generated and evaluated, the temperature is reduced for the next iteration in the anneal. The new temperature,  $T_{new}$ , is given by  $T_{new} = \tau \cdot T_{old}$ , where the value of  $\tau$  depends on the fraction of attempted moves that were accepted ( $R_{accept}$ ) at  $T_{old}$  and is determined using an approach similar to [6] and [7].

Finally, the outer loop exit criterion stops the annealing process if the temperature is less than a small fraction ( $\epsilon$ ) of the average criticality cost per external connection. The moves in the annealing process will always affect external connections in the partitioning solution. If the temperature drops below a fraction of the average criticality cost of an external connection, it is unlikely that any move that results in a cost increase will be accepted, and the annealing can be terminated. The value of  $\epsilon$  is set to 0.05 during the experiments.

### 3.3. Postprocessing

In the postprocessing stage we remove small partitions based on a proposed minimum partition size. It should be noted that keeping small partitions may help when a change occurs in the modules corresponding to those partitions. Also, if a design contains very small critical modules, then assigning these modules to partitions will preserve performance for incremental compilation.

## 4. EXPERIMENTAL RESULTS

In this section we present experimental results for the modular partitioning approach. We first introduce the incremental design flow used in all of our experiments and show how it is used to investigate the effect of the partitioning process. We evaluate our mPart algorithm to set different parameters of the algorithm. To measure the effectiveness of our approach we present partitioning results for a set of manually partitioned industrial circuits and will compare the results for the evaluated algorithm with the manual partitioning results and a few heuristical partitioning strategies. All of the experiments have been done using Quartus II software from Altera on a Pentium 4 - 866MHz with 1GB of RAM.

### 4.1. Experimental Methodology

Our experiments are performed on a set of real industrial circuits<sup>1</sup> with an average size of 20,000 LEs. Experiments start with performing full compilation for circuits without any partitioning information. We call this a setup compile for the flat circuits. For any design, we then make a specific modification and recompile the whole design from scratch. This is called an incremental compile for the flat netlist. Note that since no partitioning information is used the incremental compile for the flat netlist is a full recompilation. We then use partitioning and floorplanning<sup>2</sup> settings and do a full compilation using this information. This is also called a setup compile, but it is for the partitioned circuits. Similar to the experiments for the flat circuits, we then apply the same design modifications used for incremental compilation for the flat circuits and recompile the design with the incremental design feature turned on. In this case only the partitions corresponding to the modified portions of the designs will be recompiled. We then compare the results for setup compile and incremental compile for partitioned circuits with those of the flat circuits.

In order to have the same incremental change for all of the experiments we use a compiler setting change for all the circuits. We pick a module in a design and change a compiler optimization option in Quartus.

### 4.2. Algorithm Evaluation

As mentioned in section 3.2 our simulated annealing based partitioning uses three components in the cost function: criticality, connectivity, and partition size. In this section we evaluate the parameters controlling the effect of each of these components.

<sup>1</sup>Note that while a total of 22 circuits are used in the experiments, due to the nature of the incremental change in the designs some of the circuits may not pass the incremental compilation. This happens when an incremental change results in an area increase in one or more partitions where there is not enough room for extra logic cells. The other problem that may occur is failure to place the floorplanned partitions by the placer. Nevertheless, we always use the circuits that pass all flows when any comparison is made.

<sup>2</sup>In order to floorplan a design each partition is assigned to a physical location on the chip using LogicLock feature in Quartus. LogicLock regions are flexible, reusable floorplanning constraints that increase the ability to guide placement. Quartus then determines the optimal size and location for each LogicLock region.

**Table 1.** Incremental compilation results for different values for  $\gamma$ 

	Flat		$\gamma = 0.1$ vs. Flat		$\gamma = 0.2$ vs. Flat		$\gamma = 0.3$ vs. Flat		$\gamma = 0.4$ vs. Flat	
	setup	incr.	setup	incr.	setup	incr.	setup	incr.	setup	incr.
$f_{max}$ (MHz)	100.07	100.90	1.41%	0.74%	0.46%	-0.22%	0.79%	-1.20%	-1.02%	-2.98%
Area	19045	19262	2.37%	2.11%	3.21%	2.92%	3.83%	3.49%	4.61%	4.25%
Total time (s)	4181	4163	8.03%	-37.51%	3.77%	-46.10%	2.50%	-48.19%	5.83%	-49.22%
# of partitions	-	-	6.1		7.2		10.0		11.2	
Size of partitions	-	-	3164		2685		1934		1730	

**Table 2.** Incremental compilation results for different values for  $\alpha$  when  $\gamma = 0.30$ 

	Flat		$\alpha = 0.0$ vs. Flat		$\alpha = 0.15$ vs. Flat		$\alpha = 0.35$ vs. Flat		$\alpha = 0.55$ vs. Flat		$\alpha = 0.70$ vs. Flat	
	setup	incr.	setup	incr.	setup	incr.	setup	incr.	setup	incr.	setup	incr.
$f_{max}$ (MHz)	88.51	88.70	-1.05%	-1.28%	-1.55%	-3.25%	1.15%	-0.75%	1.02%	0.04%	1.88%	0.35%
Area	19384	19626	4.82%	4.21%	4.13%	3.74%	3.97%	3.59%	3.81%	3.50%	3.18%	2.88%
Total time (s)	4579	4571	3.90%	-49.61%	3.73%	-48.55%	6.06%	-46.56%	6.54%	-46.74%	7.50%	-38.15%
# of partitions	-	-	10.8		10.8		9.6		8.7		7.3	
Size of partitions	-	-	1820		1826		2056		2264		2687	

#### 4.2.1. Size and number of partitions

We first determine a value for  $\gamma$ , which indirectly controls the size of partitions and the number of partitions in the final partitioning solution. The value of  $\alpha$  and  $\beta$  is determined using  $\alpha = \beta = \frac{1-\gamma}{2}$ . Table 1 shows the effect of sweeping the value of  $\gamma$  on circuit speed ( $f_{max}$ ), area, total compile time (the sum of synthesis, placement, routing, and timing analysis times), number of partitions, and partition size (all results are geometric mean values for all circuits passing the experiments; also note that detailed results for all the experiments have been removed due to page limit). Columns 2-3 show the results for the flat compilation in which no partitioning is performed. Each subsequent pair of columns show the results when compilation is done using partitioning and floorplanning settings for different values of  $\gamma$ . The “setup” columns show the results when the original circuits are used and the “incr.” columns show the results after an incremental change has been made in the design. In order to compare the results before and after partitioning we compare the setup and incremental results for partitioning case with the setup and incremental results for flat compilation, respectively. The value of  $\gamma$  is increased incrementally from 0.1 to 0.4. Note that a value of 0 for  $\gamma$  will create a few big partitions for most of the designs as there is no constraint on the partition size. Such partitioning is definitely not of any interest.

As expected, by increasing the value of  $\gamma$  size of partitions gets smaller and the number of partitions is increased. As for compile time savings, there is no compile time benefits for the setup compile for partitioned circuits as a full compilation is performed. In fact, due to floorplanning, total compile time is slightly increased for all cases. However, for the incremental compilation, we can get compile time savings in a range of 37% to 49%. As shown, lower values of  $\gamma$  results in less compilation time benefits as most of the partitions are large and an incremental change usually results in recompilation of big partitions. The best compilation time benefit is for the highest value of  $\gamma$ , but higher values of  $\gamma$  will result in higher area increase and higher performance degradation. As shown, by increasing the value of  $\gamma$  we have higher number of partitions, and therefore fewer cross-boundary optimizations occur in the designs. The area increase varies from 2% for the lowest  $\gamma$  value to around 4.5% for the highest. The effect of less cross boundary optimization can also be seen from circuit speed, where it gets degraded more significantly, specially for the incremental compilation, as we increase  $\gamma$ . It should be mentioned that as we

increase  $\gamma$  the values of  $\alpha$  and  $\beta$  get smaller and the algorithm spends more time to get a solution with minimum partition size. Based on these results and to have a reasonable number of partitions, we choose a value of  $\gamma = 0.3$  to continue our algorithm evaluation.

#### 4.2.2. Criticality and Connectivity

We now evaluate our mPart algorithm for criticality and connectivity parameters ( $\alpha$  and  $\beta$ ). The value of  $\gamma$  is set to 0.3 and we have  $\beta = 1 - \alpha - \gamma$ . Table 2 shows the effect of sweeping the value of  $\alpha$ , which is increased incrementally from 0 to 0.7 (in other words, the value of  $\beta$  is decrease from 0.7 to 0). As shown, by increasing the criticality factor ( $\alpha$ ) we get better circuit speed for both setup and incremental compilation. For small values of  $\alpha$  (the first two values) we see large performance degradations both for setup and incremental modes. Experiments show that the change in connectivity component of the cost function (rent value for the partitions) is much lower than the change in criticality component. Therefore, for small values of  $\alpha$  change in the partitioning solution results in a small change in the connectivity component of the cost function and many moves do not get accepted in the annealing. Therefore, the annealing process stops very soon and final partitioning solutions contain too many partitions, many of which are very small. As the small partitions are removed from the final partitioning solution this leads to a random selection of the partitions and we can not get good performance preservation results. As the value of  $\alpha$  is increased the annealer results in more acceptable partitioning solutions and we can see the effect of  $\alpha$  for higher criticality factor values.

As for area, by increasing  $\alpha$  the amount of area overhead is reduced. As mentioned above, our experiments show that after a move in the partitioning solution, the change in criticality is higher than the change in connectivity. So, as  $\alpha$  increases, more moves get accepted and size of partitions in the final partitioning solution is increased. This will results in less number of partitions, which in turn reduces the amount of area increase. As for the total compile time, the compile time benefit is the least for the case when  $\alpha$  is 0.7 as we have the lowest number of partitions and highest partition size. Based on these experiments we choose a value of 0.55 for  $\alpha$  (0.15 for  $\beta$ ). We also tested several values for the minimum allowable partition size and set this value to 250. The detail of these experiments is omitted due to page limit.

**Table 3.** Incremental compilation results for manual, random, and mPart partitioning

	Flat		Manual vs. Flat		mPart vs. Flat		Random1 vs. mPart		Random2 vs. mPart	
	setup	incr.	setup	incr	setup	incr	setup	incr	setup	incr
$f_{max}$ (MHz)	98.04	99.98	1.64%	0.17%	2.04%	0.07%	-1.18%	-1.88%	-1.37%	-2.35%
Area	18375	18559	3.49%	3.21%	3.98%	3.75%	0.22%	0.23%	0.88%	0.70%
Total time (s)	4362	4250	0.62%	-51.26%	0.91%	-51.21%	1.22%	17.63%	4.56%	9.02%
Number of partitions	-		8.5		9.8		8.8		10.5	
Size of partitions	-		2384		1912		2136		1803	

The run time for the algorithm is in the range of tens of seconds to a couple of minutes depending on the initial size of the partitioning solution. If we invest more time in the annealing, the results tend to improve in terms of performance preservation, however, as more partitions get merged, compile time saving may decrease. Details of effect of annealing parameters on run time are omitted due to page limit.

### 4.3. Manual and Random Partitioning

In order to verify the effectiveness of the automatic partitioning tool we first compare our results for the mPart algorithm with a set of manually partitioned industrial circuits. We use the manually created partitions for these circuits and perform the incremental design flow by applying the same design changes done for previous experiments. For manual partitioning, each circuit is repeatedly partitioned until a partitioning with no performance degradation and good compile time saving is achieved. To have a fair comparison, size and number of partitions are chosen to be close to those created by mPart.

Table 3 shows the results. Columns 4-5 and 6-7 show the results when compilation is done using partitioning and floorplanning settings for manual partitioning and mPart, respectively. As shown, mPart produces good results in terms of performance and compile time savings. Proper partitioning has resulted a slight improvement in circuit performance for mPart. Also mPart results in more than 51% compile time savings and is comparable with manual results. As for area, mPart has slightly higher area increase due to a higher number of partitions.

We now compare mPart with the partitioning solutions created by random partitioning. The random partitioning scheme works in two different modes. First we modules are randomly assigned to different partitions without any consideration for size of partitions (Random1). The other random partitioning (Random2) is done while trying to balance the size of partitions. Columns 8-11 in Table 3 show the results of Random1 and Random2 compared with mPart. As expected, random partitioning results in partitioning solutions that degrades circuit performance and also does not results in good compile time savings. The compile time savings is much worse for the case where no constraint is set for the partition size (Random1). It should be noted that comparison with other

state-of-the-art partitioning algorithms, like hMetis[8] is not possible as these algorithms do not keep the hierarchy intact and tend to use elements from different hierarchies in the final partitioning solution.

## 5. SUMMARY

We have presented a partitioning strategy to divide a design into a set of partitions based on design hierarchy modules. Experimental results using the Quartus software from Altera showed that our algorithm can generate partitioning solutions comparable with a set of manually partitioned industrial circuits with compile time savings of more than 50% without any performance degradation.

## 6. REFERENCES

- [1] C. Ababei, S. Navaratnasothie, K. Bazargan, and G. Karypis, "Multi-objective circuit partitioning for cutsize and path-based delay minimization," in *ICCAD*, 2002, pp. 181–185.
- [2] W. Choi and K. Bazargan, "Incremental placement for timing optimization," in *ICCAD*, 2003, pp. 463–466.
- [3] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simulated annealing," *Science*, pp. 671–680, May 1983.
- [4] B. S. Landman and R. L. Russo, "On a pin versus block relationship for partitions of logic graphs," *IEEE Trans. on Computers*, vol. C-20(12), pp. 1469–1479, Dec. 1971.
- [5] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.
- [6] V. Betz and J. Rose, "VPR: A new packing, placement, and routing tool for fpga research," in *FPL*, 1997, pp. 213–222.
- [7] V. Manohararajah, T. Borer, S. D. Brown, and Z. Vranesic, "Automatic partitioning for improved placement and routing in complex programmable logic devices," in *FPL*, 2002, pp. 232–241.
- [8] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multi-level hypergraph partitioning: Applications in VLSI domain," *IEEE Trans. on VLSI Systems*, vol. 7, no. 1, pp. 69–79, 1999.