A connection-oriented binding model for binding algorithms

En-Shou Chang
Daniel D. Gajski

Technical Report #96–49
October 9, 1996

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
(714) 824-8059

echang@ics.uci.edu

Abstract

A new binding model which can formulate any architectures is presented in this paper. Several simple algorithms which employ this model are proposed to demonstrate the performance on this model. Experimental results show that these algorithms though are simple yet can obtain better results than previous complex algorithms. In addition, exchanging sources of functional units and sharing functional units, which can improve synthesized results by way of reducing MUX inputs required, are also discussed in this paper.

1 Introduction

In recent years, **behavioral (high-level) synthesis**[1, 2] has been recognized as one of the major design methodologies. It allows us to specify a design in a purely behavioral form, devoid of any implementation details. For example, we can describe a design using Boolean equations, finite-state machines and other conceptual models. Then, an implementation for the design can be generated by automatic synthesis tools, instead of tedious manual design.

Behavioral synthesis involves the transformation of a design specification into a set of interconnected **RT-components**[2] which satisfy the behavior and some specified constraints, such as the number of functional units(FU), performance etc. Three major synthesis tasks are applied during the transformation[1]: allocation, scheduling, and binding. The purpose of **allocation** is to determine the number of resources, such as registers, buses, and FUs, that will be used in the implementation. The task of **scheduling** is intended to partition the behavioral description into time intervals, called **control steps**. During each control step, which is usually one clock-cycle long, data will be fetched from a register, transformed in a FU, and written back to a register. All register transfers in any given control step will be executed concurrently. The **binding** task assigns variables to storage units, assigns operations to FUs, and as well as makes sure that there is a distinct communication path or bus assigned for each transfer of data between the storage and FUs.

1.1 Previous works

The binding task not only has to find an assignment of each of the operations, variables, and data transfers for the hardware components such that the hardware can carry out the behavior of the design correctly, but also makes the assignments coming with best value toward a certain goal, for example, the hardware cost, as far as possible. To achieve good binding, several methods have been proposed. We can categorize these methods into five groups as explained in the following subsections:

1.1.1 Constructive methods

The operations are bound to RT-components in a step-by-step fashion[3, 4, 5]. A constructive algorithm starts with an empty datapath and builds the datapath gradually by adding FUs, storages, or interconnections as necessary.

For each operation, the algorithm tries to find an FU on the partially designed datapath that is capable of executing the operation and also idle during the control step in which the operation must be executed. On the other hand, if none of the FUs on the partially designed datapath meet the conditions, the algorithm adds a new FU from the component library that is capable of carrying out the operation. Similarly, the algorithm assigns a variable to an available register only if its lifetime interval does not overlap with those of variables already assigned to that register. A new register is allocated only when no allocated register meets the above condition. Finally, as soon as both the source and sink of a data transfer have been bound, the algorithm assigns an available interconnection for it, or allocates a new interconnection when none of those allocated interconnection is available.

Sometimes an expensive former binding can make several very low-cost latter bindings possible. Since the algorithm can't foresee the best final result, solutions derived by constructive approaches often far away from optimal. Thus, **iterative improvement**[2], which tries to optimize the result by inter-changing bound operations, data transfers, and storage locations, is usually required[6].

1.1.2 Integer linear programming model

Some researchers[7, 8, 9] have proposed methods to transform the binding problem into an Integer Linear Programming(ILP) problem. They formulate the assignments of operations, variables, and data transfers by ILP constrains and transform the goal of the binding problem into the cost function for the ILP problem. Once the binding problem is transformed into an ILP problem, we can use any ILP solvers to find the point with minimum cost, then transform the coordinate of the minimum point back into assignments of operations, variables, and data transfers for hardware components. Since the cost of the point is minimum, the final design is optimal.

The ILP modeling is the only way to find the optimal solution for the binding problem. However, since a ILP problem is NP-complete[10], it would be very time-consuming to solve a large description by way of ILP modeling.

1.1.3 Polynomial-time optimal algorithms

In addition to ILP model, some researchers[11, 12, 13] have proposed polynomial-time algorithms which can find optimal or near optimal solutions for some subtasks of the binding task, for example, variable-to-register assignments or data-transfer-to-interconnection assignments. However, since there are interdependencies among these subtasks, no optimal solution is guaranteed even if all of the subtasks are solved optimally. For example, a design using five registers may need ten more multiplexor inputs than another design using six registers. Therefore, the optimal register allocation may cause a worse final design. Again, to obtain better solution than those local optimals, iterative improvement is usually required for this approach[13].

1.1.4 2-D placement model

Since the ILP methods are too time-consuming, heuristic methods which can find good enough solutions in reasonable time are required. The most popular binding model for heuristic approaches is 2-D placement[14]. This model starts with *determined resource allocation*, using a 2-D table which employs the resources as column index, one column for each resource, and the control steps as row index, one row per control step, then formulate the binding task as placing all of the operations, variables, and data transfers on the table under following conditions:

1. Each field can be occupied by only one operation, variable, or data transfer.
2. Each operation has to be placed on a field with row index (a control step) at which the operation should be executed and column index which is an FU and able to execute the operation.
3. Each variable has to be placed on several fields which is continuous on a column with column index is a register, start at row index the same with the control step at which the variable is born, and end at row index the same with the control step at which the variable dies.

Many 2-D placement heuristics[14, 15, 16, 13] can be applied on this model. Once the placement is completed, those operations, variables, or data transfer on each column will be bound to the same component which is the same type with the column index. As soon as the operations, variables, and data transfers which go through buses are bound, the interconnections can also be determined.

Since resource allocation has to be determined beforehand, the 2-D placement model is often used for iterative improvement[14, 13, 15, 16].

1.1.5 Clustering model

In addition to 2-D placement model, some researchers[17, 18, 19] developed their heuristic algorithms based on another model, clustering model. This model starts with *undetermined resource allocation*, dealing the binding task as a clustering problem where those operations, variables, or data transfers which are clustered together will be bound to the same hardware component. In HAL[18], they use weight-directed clique partitioning to reduce total cost for MUXes during merging variables into registers, then merge MUXes into buses to reduce more interconnection cost. In Elf[19], they use both partition and grouping algorithms to cluster data transfers into buses.

1.2 Our approach

We follow the clustering concept in our work. The major contribution in this paper is the generic binding model we proposed in Section 2. Our model can provide following excellent features:

1. Our model allows to bind operations, variables, and data transfers concurrently to globally search a best design.

2. Our model can work for arbitrary architectures, accept arbitrary component libraries, and be employed by most algorithms.

3. Our model can express complete information for the binding algorithm to work on detail improvements, for example, *source exchange* and *FU sharing*, and to be fine tuned to obtain excellent design.

4. Our model can not only accept partial design but also has partial designs always available during the whole binding process. This feature is very important for interactive high-level synthesis environments such as *iSE*[20], since design engineers would like to monitor the automatic synthesis process, add some partial design, and/or change part of the automatic synthesized design. This feature also allows people to incorporate estimation tools, for example, wiring estimation tools or layout estimation tools, to obtain faster or more accurate synthesis results.

1.3 Paper organization

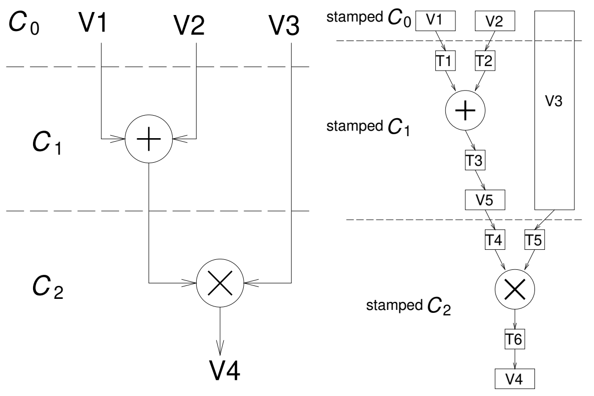
Our binding model is proposed in Section 2. Several simple binding algorithms based on our binding model are proposed in Section 3 and a source exchange algorithm is proposed in Section 4. Our model is so powerful that even simple algorithms can produce competitive or better results than previous complex binding algorithms. We have done many experiments to confide our analysis. Statistics on the experimental results and some typical samples of our experiments are shown in Section 5. Finally, we conclude our contributions and drawbacks in Section 6.

2 The binding model

In this section, we first define our binding model, then illustrate how popular design rules are formulated in our binding model.

2.1 Definition

The binding task assigns hardware to a scheduled **control/data flow graph (CDFG)** to implement a specified behavior while meeting performance and timing constraints and minimizing implementation cost. To perform this task, we first transform the scheduled CDFG into a register-transfer flow-graph, which is defined follow, then group those data structures which



(a) DFG **(b)** RTFG

Figure 1: A scheduled DFG and its corresponding RTFG

will be bound to one hardware component into one cluster.

Definition 1 *A* **register-transfer flow-graph (RTFG)** *is a directed graph $\langle V, E\rangle$ where*

1. *A vertex $v \in V$ represents an operation, a variable, or a data transfer.*
2. *An edge $e \in E$ indicates a data movement from one vertex to another.*
3. *Each vertex is associated with clock stamps indicating its life time.*

Figure 1 shows an example of the transformation from a scheduled CDFG to its corresponding RTFG: Figure 1(a) is a CDFG which has two operations, $+$ and $\times$, and four input/output variables, V1, V2, ... V4, and is scheduled into two control steps, C_1 and C_2. Figure 1(a) will be transformed into a 13-vertex RTFG as Figure 1(b), where vertex $+$ and vertex $\times$ represent operations, vertexes V1, V2, ... V5 represent variables, and vertexes T1, T2, ... T6 represent data transfers. Each operation or variable in Figure 1(a) will be transformed into a corresponding vertex associated with the corresponding clock stamps in Figure 1(b). For examples: the operation $+$ in Figure 1(a) will be transformed into the vertex $+$ associated with a clock stamp C_1 in Figure 1(b), and the variable V3 in Figure 1(a) will be transformed into the vertex V3 associated with clock stamps C_0 and C_1 in Figure 1(b). In addition, vertexes represent temporary variables, for example V5, will be added into the RTFG, and corresponding clock stamps, for example, C_1 for V5, will be stamped on them. Finally, each data transfers between variables and operations will be transformed into a corresponding vertex. For example, the data transfer from V3 to operation $\times$ in Figure 1(a) will be transformed into the vertex T5 associated with a clock stamp C_2 in Figure 1(b).

Once the scheduled CDFG is transformed into a RTFG, we can formulate the binding task as a **clustering problem** where we cluster all of the vertexes in the RTFG into clusters such that those vertexes which are clustered together are assigned to the same hardware component. In most cases, this formulation implies:

1. Vertexes associated with same clock stamps will not be clustered together, since a hardware component can't perform more than one jobs at the same time.
2. Operations will not be clustered together if there is not a FU which can execute all of those operation in the component library.

On the other hand, there *may be some exceptions*. For example, the component library has a super-scalar FU which can execute two additions parallelly, then two addition-vertexes associated with the same clock stamp can be clustered into a cluster.

2.2 Some paradigms

Our binding model can work for arbitrary architectures, accept arbitrary component libraries, and be employed by arbitrary algorithms. Any design rules which are applied on the architecture style can be translated into certain restrictions on clustering. In this section, we are going to illustrate how popular design rules are formulated in our binding model.

2.2.1 MUX-based design and bus-based design

In case the design rule allows using only single-level MUX interconnections, we can first cluster every vertex which represents a data transfer with its predecessor. On the other hand, when the design rule allows using both MUX and bus interconnections, any data transfer vertexes whose time stamps don't conflict with each other can be clustered together to form an interconnection. In this case, a cluster with only one predecessor cluster will be assigned to a MUX, and so will one with single successor cluster. A cluster with both more than one predecessor clusters and more than one successor clusters will be assigned to a bus.

2.2.2 Pipeline FU

We use Figure 2 to show how pipeline components are interpreted in our model. In case there is no pipeline multiplier in component library, two vertexes which both represent multiplies and have some coincident time stamps can not be grouped into a cluster. For example, three operation vertexes for the scheduled DFG in Figure 2(a) can only be clustered as in Figure 2(b). On the other hand, in case there are pipeline multipliers in component library, two vertexes which both represent multiplies and have some coincident time stamps can be grouped into a cluster as long as their first time stamps are different. For example, three operation vertexes for the scheduled DFG in Figure 2(a) can be clustered as in Figure 2(c) if there are pipeline multipliers available.

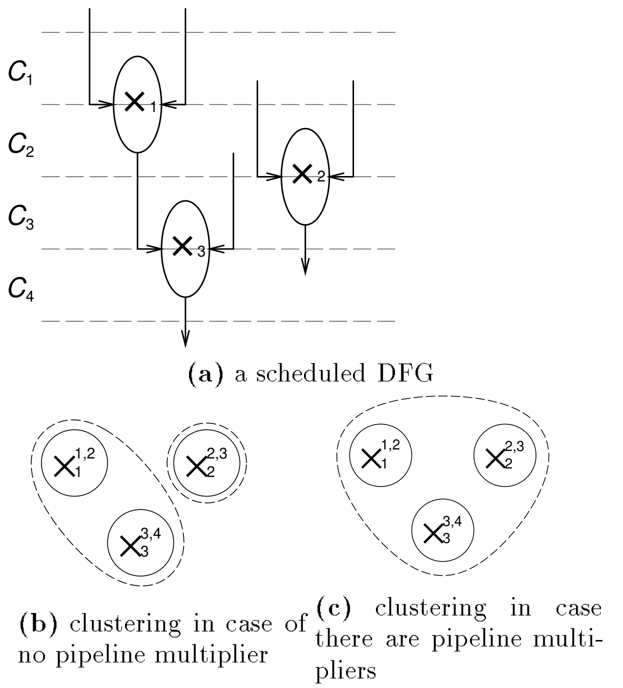
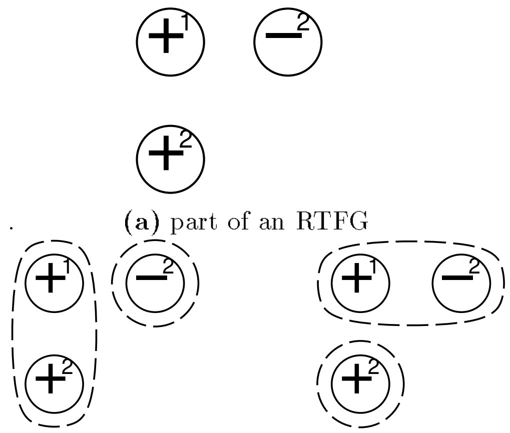


Figure 2: Clusterings with and without pipeline FU

2.2.3 Multi-function unit

Our model allows using multi-function units, for example, implementing additions by using adders, add-subtractors, and ALUs. Figure 3 shows an example of using multi-function units. In case there is no multi-function unit in component library, an RTFG as shown in Figure 3(a) will be clustered as shown in Figure 3(b). On the other hand, in case there are add-subtractor available, the RTFG in Figure 3(a) can be clustered either as in Figure 3(b) or as in Figure 3(c), depending on which cost is smaller.



(a) part of an RTFG

(b) clustering 1 (c) clustering 2

Figure 3: Clusterings

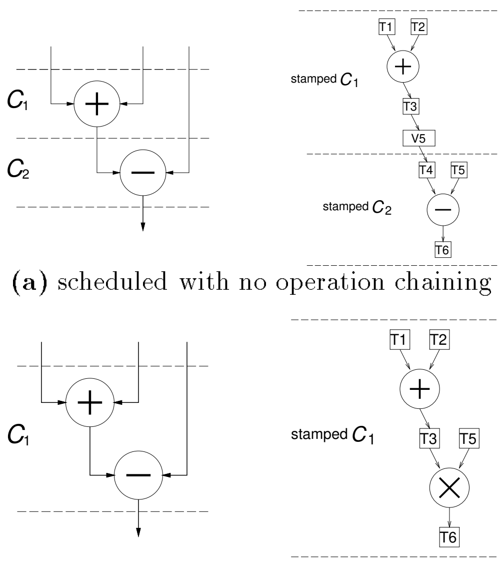
2.2.4 Multiclock FU and FU chaining

Our model can also express the idea of multiclock FU and FU chaining. Once an operation is scheduled to be executed in more than one clock, more than one time stamps would be attached to the operation when the scheduled CDFG is transformed into RTFG. On the other hand, once two operations are scheduled to be chained together, the same time stamp would be attached to both operations and those variable vertexes and data transfer vertexes between two operations could be removed. Figure 4 shows an example of how to express operation with or without chaining in our binding model.

2.2.5 Customer components and binding algorithms

Special custom components can be expressed by certain rules on clustering, too. For example, given a component library with a register design which can perform shifting, we will allow vertexes which represent variables and vertexes which represent shift operations to be clustered together.

Basicly, a binding algorithm is assigning data structures, such as operations, variables, or data transfers



(a) scheduled with no operation chaining

(b) scheduled with operation chaining

Figure 4: Operation chaining

to hardware components. Thus, we can directly translate "assign data structure e to hardware component C" in any algorithms into "put data structure e to cluster C" in our binding model, using the same process, cost functions, priorities, etc. Therefore, our binding model can work with any algorithms and express any customer components.

3 Simple algorithms

In the previous section, we have formulated the binding task as a clustering problem. In this section, we are going to propose some simple algorithms based on our binding model and discuss their performance.

3.1 Preference functions

In general, people measure the quality of binding with an cost function. This cost function is usually also be used as the directive metric for the binding process. Intuitively, the real cost[13, 14] or a real-cost-related function, for examples, number of selectors, number of registers, number of FUs, or a weighted combination of them[6], was used as the cost function for the binding process. Actually, it is not necessary to use the real-cost-related cost function as the only directive metric for the binding process. As long as we can obtain better binding, any metrics could be used to direct the binding process. In addition, we may use or switch around several metrics to direct the binding process when necessary.

In our experiments, we first followed the intuitive thought using the real cost and real-cost-related functions to direct the binding process. But, we found out later that using the number of common sources and sinks as the primary key and the real cost as the secondary key can obtain even better results than that of using the real-cost-related functions. To distinguish them from the conventional real-cost-related functions, we call these directive metrics **preference functions**. Experimental results about the preference functions will be given in Section 5.1.

The cost function(directive metric) used in Section 3.2 and Section 3.3 could be based on the conventional real-cost-related functions or based on the preference functions.

3.2 Deterministic algorithms

3.2.1 Greedy method

Our first solution is to cluster vertexes in the RTFG using a greedy approach. Started with an initial clustering in which each cluster contains only one vertex (when no partial design is input), we iteratively merge two clusters with the best gain. At each step, we compute the gains from merging every two clusters, then merge the best pair of clusters. We keep merging clusters step-by-step until no more gain can be obtained. In case partial design was input, each of the clusters in the initial clustering might contain several vertexes to correspond the partial design.

Algorithm Greedy

```
begin
  clusters = initial_clustering;
  repeat
    max_gain = −∞;
    for i , j ∈ clusters do
      begin
        gain = merge_gain( i, j );
        if gain > max_gain then
          begin
            ki = i;
            kj = j;
            max_gain = gain;
          end;
      end;  /* for */
    if max_gain > 0 then
      merge_clusters( kj, ki );
  until max_gain < 0;
  return( clusters );
end
```

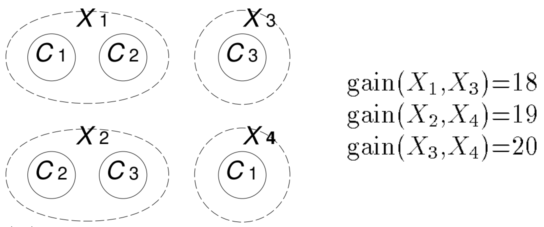
Figure 5: Algorithm Greedy

Figure 5 shows the algorithm of our greedy method. In Algorithm Greedy, the function `merge_gain( i, j )` returns the gain from merging two clusters `i` and `j`. When cluster `i` and cluster `j` can not be merged, for example, there are clock stamps which conflict between two clusters, `merge_gain( i, j )` returns $-\infty$ to prohibit they from being merged. The complexity of Algorithm Greedy is $O(n^3)$, where n is number of vertexes.

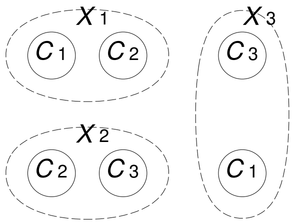
Experimental results in Section 5.2 show that Algorithm Greedy though is very simple yet can obtain excellent results, especially when it deals with small designs.

3.2.2 Finish one cluster first

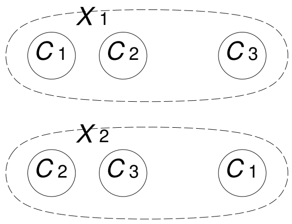
Some mishaps might happen on Algorithm Greedy in which we merge the best pair in every step. Figure 6 shows a typical example of the mishaps. In the figure, solid circles represent vertexes in a RTFG. C_1, C_2, and



(a) four clusters and the prospected gains from merging every two of them



(b) total gain = 20



(c) total gain = 19 + 18 = 37

Figure 6: A typical example of the mishaps

C_3 represent different time stamps on these vertexes. In Figure 6(a), six vertexes are clustered to 4 clusters, $X_1, X_2, ...X_4$. The prospected gain from merging X_1 and X_3 is 18, from merging X_2 and X_4 is 19, and from merging X_3 and X_4 is 20. Following Algorithm Greedy, X_3 and X_4 which have the best gain will be merge next and finally become the clustering shown in Figure 6(b) with total gain 20. But, if we merge X_2 and X_4 first, we can merge X_1 and X_3 further more. Thus, we can obtain total gain 37, which is much better than that of Algorithm Greedy, and obtain the final clustering with only two clusters as shown in Figure 6(c).

To overcome this mishap, another step-by-step approach is proposed. In this approach, we grow one cluster at a time. We first compute the gains from merging every two clusters, as that in Algorithm Greedy, and merge the best pair of clusters into one cluster. This cluster is then used as a seed. We compute the gains from merging the seed with its neighbor clusters and merge it with the best neighbor from which we can obtain the best gain. We keep growing the seed by iteratively incorporating the best neighbor cluster into it until no more gains can be obtained. Once a seed is grown to the maximum, we select the next seed by computing the gains from merging every two clusters in current clustering again and merging the best pair into one, which will be used as the new seed. We keep doing this process, growing a cluster to the maximum then select another seed, until every cluster is grown into maximum.

Algorithm One-Cluster

```
begin
  clusters = initial_clustering;
  i = find_a_seed( clusters );
  while i ≠ φ do
    begin
      repeat
        max_gain = −∞;
        for j ∈ clusters do
          begin
            gain = merge_gain( i, j );
            if gain > max_gain then
              begin
                k = j;
                max_gain = gain;
              end;
          end;  /* for */
        if max_gain > 0 then
          i = merge_cluster( i, k );
      until max_gain ≤ 0;
      i = find_a_seed( clusters );
    end  /* while */
  return( clusters );
end
```

Figure 7: Algorithm One-Cluster

Figure 7 shows the algorithm of this approach. In

Algorithm One-Cluster, the function `find_a_seed` computes the gains from merging any two clusters in the given clustering, then return the cluster formed by merging the best pair, as what Algorithm Greedy do within one iteration. The function `merge_cluster` merges two clusters which are given to it then return the merged cluster. And, as in Algorithm Greedy, the function `merge_gain( i, j )` returns the gain from merging two clusters `i` and `j`. The complexity of Algorithm One-Cluster is $O(n^3)$, where n is number of vertexes.

It is easy to guess that there would be more chance for the mishap mentioned in Figure 6 when dealing with larger designs. As expected, experimental results in Section 5.2 show that Algorithm One-Cluster can obtain better results than that of Algorithm Greedy when dealing with larger designs.

3.2.3 Spanning

The clusters which are grown by Algorithm Greedy or Algorithm One-Cluster may not be adjacent to each other. Some clusters which are later grown and surrounded by large existed clusters would have less freedom to be grown well, since most of their chances to merge with other clusters were blocked by those large clusters surrounding them. Thus, another algorithm, Spanning, is proposed to overcome this problem.

To avoid a later-grown cluster might be surrounded by other existed large clusters, Algorithm Spanning works in a best-first spanning-tree[21] style. Similar to Algorithm One-Cluster, we first select the cluster created by merging the best pair of clusters as the seed then iteratively incorporate the best neighbor of the seed into it until no more gains can be obtained. Once there are no neighbors that can be incorporated into the seed, we select a new seed from the neighbors of the old seeds following the best-first criteria, then

Algorithm Spanning

```
begin
  clusters = initial_clustering;
  kernel = φ;
  i = find_a_seed( clusters );
  while i ≠ φ do
    begin
      repeat
        max_gain = −∞;
        for j ∈ clusters do
          begin
            gain = merge_gain( i, j );
            if gain > max_gain then
              begin
                k = j;
                max_gain = gain;
              end;
          end;  /* for */
        if max_gain > 0 then
          i = merge_cluster( i, k );
      until max_gain ≤ 0;
      kernel = kernel ∪ { i };
      i = find_a_seed( neighbor( kernel ) );
      if i = φ then
        i = find_a_seed( clusters );
    end  /* while */
  return( clusters );
end
```

Figure 8: Algorithm Spanning

grow the new seed to maximum in the same way. This process will be continue as far as we can obtain further gains.

Figure 8 shows the Algorithm Spanning. Similar to Algorithm One-Cluster, the function `find_a_seed` merges the best pair of clusters given and return it as the seed, the function `merge_cluster` merges two clusters which are given to it then return the merged cluster, and the function `merge_gain( i, j )` returns the gain from merging two clusters `i` and `j`. The variable `kernel` is the set of the old seeds and the function `neighbor( kernel )` returns all of the neighbors of the elements in `kernel`. The complexity of Algorithm Spanning is $O(n^3)$, where n is number of vertexes.

Unfortunately, experimental results, which would be given in Section 5.2, show that Algorithm Spanning did not obtain better results than that of Algorithm One-Cluster. It seems that clusters which are grown independently in RTFG usually have no problem to fit each other later. Moreover, since Algorithm Spanning puts more restrictions on selecting new seeds, the results may be worse than that of Algorithm One-Cluster, and even worse than that of Algorithm Greedy.

3.3 Random approaches

There is a partial ordering[22] among all of the valid clustering for an RTFG. This partial ordering looks very good for simulated-annealing-like exploding. We can define the partial ordering as following:

Definition 2 Given *an RTFG, let* Ψ *be the set of all the valid clusterings for the RTFG.* Define

$$X_1 \leq X_2 \;\equiv\; x_2 \text{ is equal to or contained in } x_1$$
$$X_1, X_2 \in \Psi,\ \exists x_1 \in X_1,\ \forall x_2 \in X_2$$

Then, $(\Psi, \leq)$ *is a partial order.*

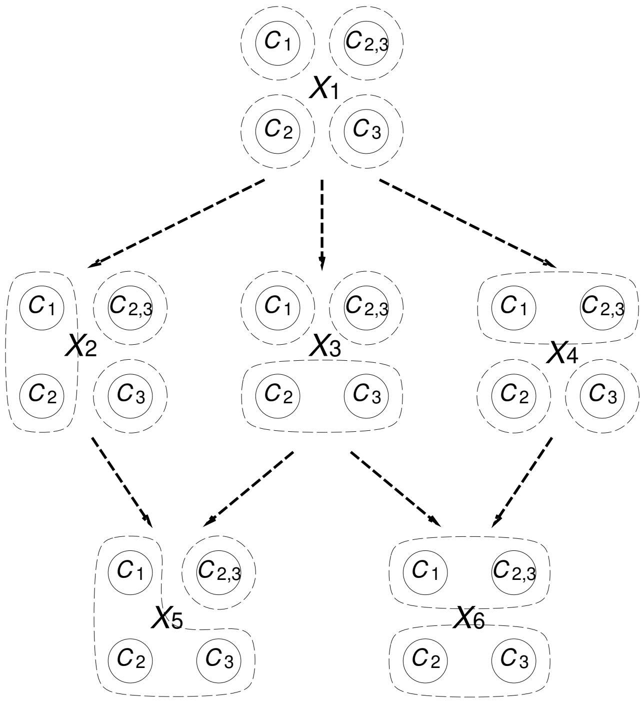


Figure 9: An example of the partial ordering among clusterings

Figure 9 shows an example of this partial ordering. In the figure, solid circles represent four vertexes in an RTFG and X_1, X_2, ... X_6 are six valid clusterings for these vertexes. C_1, C_2, and C_3 represent three different time stamps. There are two time stamps, C_2 and C_3, in the up-right vertex. X_1 is the initial clustering. Thick arrows show possible ways to merge clusters. Two clusterings connected by a thick arrow are immediate predecessor and successor in the partial order, for example, $X_2 \leq X_1$. As transitivity applied on partial ordering, a predecessor of one's predecessor is also its predecessor, for example, $X_5 \leq X_1$.

Two observations on the above partial ordering can be made as following:

Observation 1 Let Ψ *denote the set of all the valid clusterings for an RTFG.* Given $A, B \in \Psi$, $A \leq B \Longrightarrow cost(A) \leq cost(B)$, where $cost()$ *is a hardware cost function, cost-related function, or preference*

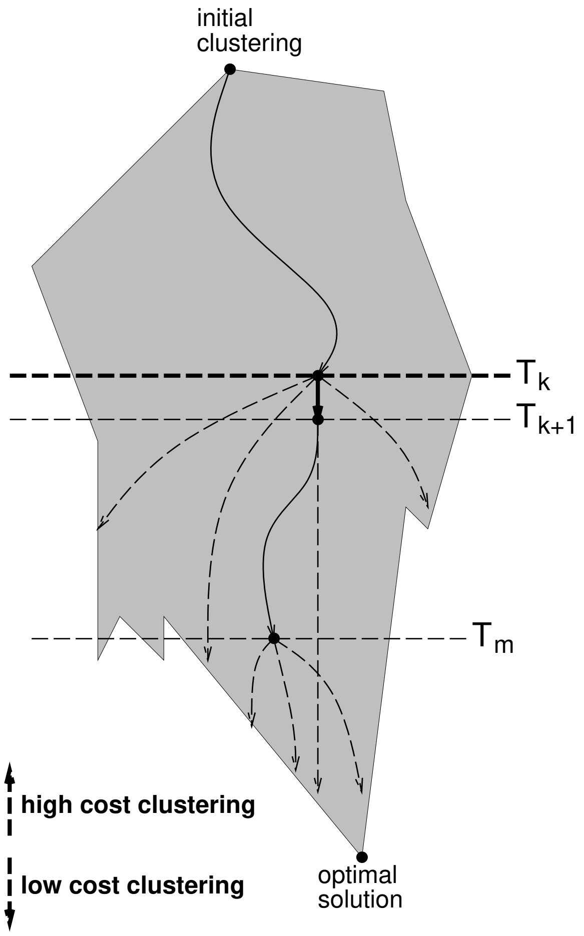


Figure 10: The structure of all the valid clusterings for an RTFG and the random approach

function.

Observation 2 *The cost(X) for an entity X which is in the neighborhoods of an entity Y which is associated with a local or global minimal cost(Y) would also be small,* where $X, Y \in \Psi$.

Based on these observations, two simulated-annealing-like algorithms, Best-path and Average-path, are proposed in this section. Figure 10 shows the idea of these algorithms: If we sort higher cost clusterings upward and lower cost clusterings downward, the shape of the partial ordering structure of all the valid clusterings for an RTFG will look like the shadow area in Figure 10. The path comprised by solid arrows shows

Algorithm Best-path

```
begin
  clusters = initial_clustering;
  repeat
    kc = ∞;
    repeat
      i = pick_a_cluster( clusters );
      j = pick_a_cluster( clusters );
      c = cost_explore( i, j );
      if c < kc then
        begin
          ki = i;
          kj = j;
          kc = c;
        end
    until stop_criterion_satisfied;
    merge_cluster( kj, ki );
  until no_merging_available( clusters );
  return( best_clustering_ever_found );
end
```

Figure 11: Algorithm Best_path

the trace of which we search downward the optimal solution. We start at the initial clustering, which is always associated with the highest cost, then iteratively search downward. At an iteration with temperature(cost) T_k (thick horizontal dashed line), we randomly sample a number of leaf descendants of the current partial design, as those thin dashed arrows coming out from the dot(the current partial design) on the thick dashed line, then pick the best branch(the dot on the head of the thick solid arrow) as the next partial design and lower the temperature to T_{k+1} (the upper thin dashed line). Finally, if we can reach a partial design at the mouth of the optimal solution valley, for example, the dot on the lower thin dashed line, we will eventually reach the optimal solution.

Figure 11, Algorithm Best-path, shows an implementation of this idea. In the algorithm, the function `pick_a_cluster` randomly selects a cluster. The function `cost_explore` randomly finds a leaf descendant in the branch where we merge clusters `i` and `j` of the cur-

Algorithm Average-path

```
begin
  clusters = initial_clustering;
  repeat
    kc = ∞;
    repeat
      i = pick_a_cluster( clusters );
      j = pick_a_cluster( clusters );
      c = 0 ;
      for count = 1 to a_number do
        c = c + cost_explore( i, j );
      if c < kc then
        begin
          ki = i;
          kj = j;
          kc = c;
        end
    until stop_criterion_satisfied;
    merge_cluster( kj, ki );
  until no_merging_available( clusters );
  return( best_clustering_ever_found );
end
```

Figure 12: Algorithm Average_path

rent partial design. If a better clustering is found when `cost_explore` is executed, the clustering will be stored in `best_clustering_ever_found`. When there are no more clusters which can be merged in `clusters`, the function `no_merging_available` returns true, otherwise it returns false. Several criteria may be used as the terminating condition for the inner loop, for examples, number of trying, or number of continuous non-progressing trying. In our experiments, we number of trying as the terminating condition.

Figure 12, Algorithm Average-path, shows another implementation of the idea. To increase the confidence of cost exploration in each branch, this algorithm samples *a_number* of leaf descendants for each branch and use the average cost of the leaf descendants of each branch to decide which branch to go, in place of sampling just one leaf descendant as we do in Algorithm Best-path. On the other hand, within a limited time,

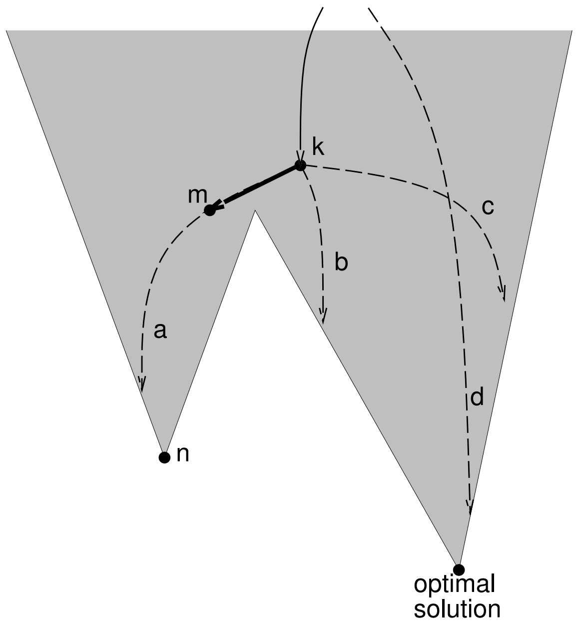


Figure 13: An example of going into a wrong valley

Algorithm Average-path can try less branches than Algorithm Best-path can do.

In stead of returning the final clustering `clusters`, both Algorithm Best-path and Algorithm Average-path returns `best_clustering_ever_found`, which is the best clustering that has ever been found during cost exploring. The final clustering would be different from `best_clustering_ever_found` if we unfortunately went into a wrong valley. As illustrated in Figure 13, we sample three instances(dashed arrow `a`, `b`, and `c`) among leaf descendants of the current partial design(dot `k`). The best sampled instance `a` is unfortunately located in a valley other than the optimal valley. Thus, the current design will move from `k` to `m` and we would have no chance to reach the optimal solution any more. In this case, the final clustering `n` could be a worse design than a cost exploration `d`. However, if only we sample effectively and enough, we can have just little probability to go into a wrong valley.

(a) Operations bound to the adder:

add_1(bus 1 , bus 2)
add_2(bus 2 , bus 3)
add_3(bus 3 , bus 1)

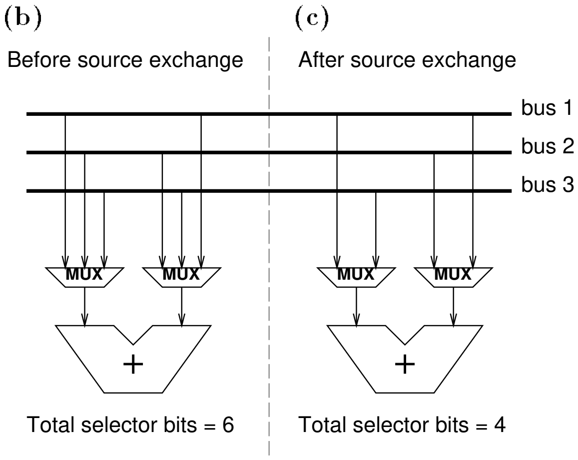


Figure 14: An example of saving selectors by exchanging sources

Unfortunately, our research in random approaches did not get better results than that of deterministic algorithms. However, experimental results in Section 5.3 show that our methodology is correct but better exploring strategy is required.

4 Source exchange

There are a number of two-input operations whose sources are **commutative**[22], for examples, additions, multiplications, etc. Properly exchanging the sources of these operations would save a considerable number of selectors.

Figure 14 shows an example of how exchanging the sources can save a number of selectors. In the example, three additions, as shown in Figure 14(a), are bound to an adder. They source three buses. Under straightforward implementation as shown in Figure 14(b), six selector bits are required. However, we could exchange the sources of add_2, where the functionality of the design won't change, and only four selector bits are required, as shown in Figure 14(c). Two selector bits are saved after the source exchange.

Ly et. al.[19] have noticed this fact and exchange sources of some commutative operations according some heuristic before running their binding algorithms. On the other hand, Choi and Levitan[13] randomly exchange sources of commutative operations to get better results. However, there are no proposed algorithms which are able to find the optimal solution in polynomial time. Different from their works, we propose a post-processing algorithm which has linear time complexity.

In our work, the sources are exchanged into an optimal or a nearly optimal combination after the operations, variables, and data transfers are bound to hardware components. We first describe the relation among the sources of the operations which are bound to the same FU with an incompatible graph, which is defined in Definition 3, then use a red-black coloring algorithm to decide which input-port a source should be bound to.

Definition 3 *An* **incompatible graph** $\langle V, E\rangle$ *for an FU is an undirected graph where*

1. *A vertex* $v \in V$ *represents a source component.*
2. *An edge* $e \in E$ *represents an operation which sources two components connected by the edge* e*. Two vertexes(sources) are said to be* **incompatible** *if they are connected by an edge.*

Since two sources of an operation can't be bound to the same input of an FU, two incompatible vertexes should be bound to different inputs.

Figure 15 shows an example of the incompatible graph. As shown in Figure 15(a), there are four operations $Op_1 \ldots Op_4$ bound to an FU, which sources

(a) Four operations which are bound to an FU and their sources:

$Op_1(S_1, S_2)$
$Op_2(S_1, S_3)$
$Op_3(S_2, S_4)$
$Op_4(S_2, S_3)$

(b) The corresponding incompatible graph

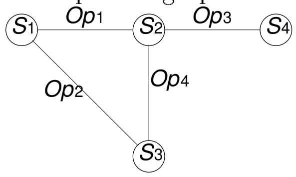


Figure 15: An incompatible graph example

four components $S_1 \dots S_4$. Figure 15(b) shows the corresponding incompatible graph. Each edge in Figure 15(b) represents an operation in Figure 15(a). For example, the edge Op_1, which connects S_1 and S_2, represent the operation Op_1, which sources S_1 and S_2.

Once we have the incompatible graph, we can color each vertex with red or black, where two incompatible vertexes should have different colors, to decide which input of the FU the vertex(source) will be bound to. Those which are colored with red will be bound to one input and those which are colored with black will be bound to the other. However, there might be some sources which inevitably have to be bound to both inputs of the FU. These vertexes will be colored with both colors. For example, one of the sources S_1, S_2, and S_3 in Figure 15(a) has to be bound to both inputs of the FU, and one of the vertexes S_1, S_2, and S_3 in Figure 15(b) will has both colors, too. Finally, if we can find a coloring for the incompatible graph with least number of bi-colored vertexes, we can obtain a design with least number of selector bits.

There are no polynomial time algorithms which can find out such a coloring so far. In our work, we use a quick and effective red-black coloring algorithm which can find a near optimal solution in linear time complexity. Figure 16 shows our coloring algorithm, Algorithm Red-black-coloring. The algorithm works in a breadth-first spanning-tree[21] style. We first select a seed `v` from the vertexes which are not colored and

Algorithm Red-black-coloring

```
begin
  repeat  /* spanning tree */
    v = select_seed( incompatible_graph );
    v.color = red ;
    repeat  /* coloring neighbors */
      if v.color = red then
        neb_color = black
      else
        neb_color = red;
      for i ∈ neighbor( v ) do
        if i.color = no_color then
          if j.color = v.color
            or bi_color or no_color ,
            ∀ j ∈ neighbor( i ) then
            begin
              i.color = neb_color;
              add_queue( i );
            end
          else
            i.color = bi_color;
      v = pop_queue();
    until v = φ;
  until all_vertexes_are_colored;
end
```

Figure 16: Algorithm Average_path

color it with one color, for example, red, then color all of v's neighbors with the other color `neb_color` or bi-color. When coloring a neighbor `i` of `v` with `neb_color` will cause no color conflict with all of `i`'s neighbors, we color `i` with `neb_color`. Otherwise, we color `i` with bi-color. Once we have colored all of the neighbors of the seed `v`, we follow the breadth-first criterion to select a newly uni-colored(black or red) vertex as the next seed and repeat the coloring-neighbor process. When there are no seed candidates among newly colored vertexes, we again select a vertex with no color as the next seed and start another breadth-first spanning-tree process. This process will not stop until all of the vertexes are colored.

In Algorithm Red-black-coloring, the function `select_seed` selects a vertex with no color as the seed and the function `neighbor` returns all of the neighbors of the vertex given. We use a queue to maintain the breadth-first order. New uni-colored vertexes are stored into the queue and will be popped out one-by-one to be used as the next seeds. Two subroutines `add_queue` and `pop_queue` are used to access the queue. When the queue is empty, `pop_queue` returns ϕ.

Our approach can find a near optimal solution within linear time. Actually, we obtained optimal solutions in almost all of our experiments, which are shown in Section 5.4.

In the rest of this section, we are going to introduce a series of theorems which help evaluate the performance of Algorithm Red-black-coloring in Section 5.4.

Lemma 1 *There are only two colorings for a connected 2-colorable graph. These colorings are isomorphic.*

Lemma 2 *A graph is 2-colorable if and only if all of its connected subgraphs are 2-colorable.*

Since Algorithm Red-black-coloring can color any connected 2-colorable graph, we have:

Theorem 1 *Given a 2-colorable graph, Algorithm Red-black-coloring can find a coloring for the graph.*

Theorem 1 implies that a graph which is colored by Algorithm Red-black-coloring with bi-colored vertexes is not 2-colorable. Moreover, since one is the smallest positive integer, we have:

Theorem 2 *A coloring obtained by Algorithm Red-black-coloring with only one bi-colored vertex is optimal.*

Furthermore, with Lemma 2, we have:

Theorem 3 *A coloring obtained by Algorithm Red-black-coloring with only one or no bi-colored vertex in each connected subgraph is optimal.*

5 Experimental results

We have done many experiments to confide our analysis, since some algorithms might be good for only one or two cases. In the rest of this section, we are going to show statistics on our experimental results and couples of typical examples. In the cases we referred real cost, we use VLSI Technology Inc. VCC4DP3 Datapath Library[23].

5.1 Preference function vs. real area cost

To prove that preference function is a better directive metric than real area cost, we compared results which are synthesized under same conditions except using different directive metrics. Some typical examples are shown in Table 1. Statistic results on performance of preference function and real area cost are shown in Table 2.

Elliptic Filter[24] : use registers											
algorithm	dir. metric	bus	reg	sel	drv	muxin	total	+	*	area	imp.
Greedy	real cost	8	10	12	26	38	48	2	2	78.0	
	preference	11	9	8	18	26	35	3	2	66.4	15%
One-Cluster	real cost	10	8	17	24	41	49	2	2	76.8	
	preference	8	8	9	22	31	39	2	2	66.8	13%
Spanning	real cost	10	8	17	24	41	49	2	2	76.8	
	preference	8	7	13	25	38	45	2	2	71.7	7%

Elliptic Filter[24] : use register files												
algorithm	dir. metric	bus	RF	sel	drv	muxin	total	mem	+	*	area	imp.
Greedy	real cost	11	6	21	12	33	39	12	2	2	84.4	
	preference	10	6	2	15	17	23	11	4	2	72.8	14%
One-Cluster	real cost	8	5	16	17	33	38	11	2	2	79.6	
	preference	8	5	16	2	18	23	10	2	2	64.0	20%
Spanning	real cost	9	5	18	16	34	39	10	2	2	80.0	
	preference	9	5	12	15	27	32	10	2	2	73.0	9%

Table 1: Some samples of results obtained by using preference function and real area cost as the directive metric

algorithm	reg	muxin	total	area
Greedy	0.979	0.455	0.798	0.908
One-Cluster	0.984	0.409	0.613	0.873
Spanning	0.984	0.444	0.651	0.906
overall	0.983	0.434	0.687	0.895

Table 2: Average reduction ratio of results obtained by using preference function as the directive metric over results obtained by using real area cost

Table 1 shows some synthesized results of an Elliptic Filter[24]. The first and second columns are the algorithms and the directive functions used. The columns entitled by 'sel' and 'muxin' are the numbers of selector bits required and the summations of selector bits and bus drivers. The column entitled by 'total' are the total numbers of selector bits, bus drivers, and registers/register files. The column entitled by 'mem' are total memory in register files. The last column are improvements(reduction in area) obtained by using preference function as the directive metric in place of using real area cost.

Table 2 shows the statistics of improvements in numbers of registers/register files, MUX inputs, MUX inputs plus registers/register files, and area. The number in each field is the average reduction ratio of results obtained by using preference function over results obtained by using real area cost. 16 cases are tested on each algorithm for the comparison. The last row are the overall averages. From this statistics, we can see *using preference function as the directive metric can reduce more than 50% MUX inputs.* The area reduction for whole design is approximately 10% in average, since MUXes occupy approximately 20% area in a design and there is only little reduction in either registers/register files or FUs.

5.2 Deterministic algorithms

To show our algorithms though are simple yet can obtain results as well as that of complex algorithms in previous works, we compare our results with Schalloc[25], STAR[6], and M&M[13], which have their scheduling available in their articles. In Table 3 and Table 4, to make comparison with their works[25, 6], we restricted our algorithms to use the same numbers of FUs with

algorithm	bus	reg	muxin
Schalloc	11	6	11
Greedy	9	5	4
One-Cluster	7	4	6
Spanning	7	4	6

Table 3: Results for the HAL

algorithm	bus	RF	mem	sel	drv	muxin
STAR	†5	‡3	11	16	13	29
M&M	†4	5	9	12	13	25
Greedy	11	6	11	21	12	33
One-Cluster	8	5	10	16	2	18
Spanning	9	5	10	12	15	27

Table 4: Results for the Elliptic Filter

† Because M&M and STAR use different bus counting rule from ours, the number of buses in our work looks much larger than that of theirs. In fact, the numbers of buses required in their works are similar to that of ours.

‡ In addition to register files, STAR uses a ROM.

what they used, where our algorithms might be degraded a little since our algorithms are designed to target the best total cost. However, our algorithms still obtained better results than theirs. On the other hand, since different scheduling certainly results in different binding, we are not able to make fair comparison with other previous works.

Table 3 shows experimental results of Schalloc and our work. The design tested comes from HAL[26] which is scheduled into 6 control-steps as in[25]. Table 4 shows experimental results of M&M, STAR, and our work. The design tested is an Elliptic Filter[24] which is scheduled into 17 control-steps as in[6]. The numbers of MUX inputs and select bits required for our algorithms in both tables are the amounts required after performing source exchange. Similar to our work, M&M performs source exchange, too.

Table 5 shows the statistics on results among our algorithms. We use Algorithm Greedy as the basis for

algorithm	†reg	muxin	total	mem	area
Greedy	1.000	1.000	1.000	1.000	1.000
One-Cluster	0.883	0.786	0.814	0.900	0.960
Spanning	0.883	0.914	0.925	0.950	1.018

Table 5: Average ratio of results obtained by each algorithm over results obtained by Algorithm Greedy

† number of registers or register files

comparison. The number in each field is the average ratio of results obtained by each algorithm over results obtained by Algorithm Greedy. 18 cases are tested on all of the algorithms for this comparison. When calculating MUX inputs, we ignore samples which result in different numbers of FUs from majority, since a design using one extra FU may reduce tremendous number of MUX inputs required. However, the area ratios have reflected this impact. The column entitled by 'mem' are ratios for total memory in register files.

From Table 5, we can see Algorithm One-Cluster performs the best. Actually, when dealing with small design, for example, the HAL in Table 3, three algorithms perform similar. One algorithm may be better in some cases but worse in other cases. However, when dealing with larger design, for example, the Elliptic Filter in Table 4, Algorithm One-Cluster becomes significantly better than others.

5.3 Random approaches

Our research in random approaches did not get better results than that of deterministic algorithms. However, experimental results strongly support our model for random approaches.

Table 6 shows some typical results of our random approaches. The second column are numbers of RTFG vertexes in the designs. The last column indicates whether or not the final clustering is the same with the best clustering which is found during cost exploring. In the third column, those algorithms indicated by '*5'

	vertex	algorithm	bus	reg	sel	drv	muxin	total	area	same
case#1	58	Greedy	7	9	19	17	36	45	76.0	–
		Best-path	8	9	23	22	45	54	85.0	N
		Best-path*5	7	8	25	20	45	53	82.9	N
		Average-path	7	8	19	22	41	48	78.9	Y
		Average-path*5	7	8	17	19	36	43	73.9	Y
case#2	179	Greedy	12	14	27	30	57	71	984	–
		Best-path	12	13	56	69	125	138	1643	N
		Average-path	11	13	50	65	115	128	1543	N

Table 6: Some experimental results of our random approach, compared with results of Algorithm Greedy

explore 5 times of branches while making each decision. We can see that more branch exploring can obtained better results. As we understand in Section 3.3, more branch exploring, less chance to go into a wrong valley.

Table 7 shows the statistics on experimental results of our random approach. The number in each field is the average increasing percentage of the resource required for the occasions over the resource required for results obtained by Algorithm Greedy. From this table, we can see that the random approaches so far didn't do better than the deterministic approaches in general, except using less registers/register files.

It is a necessary condition (but not sufficient) for going into the optimal valley that the final clustering is the same with the best clustering which is ever found during cost exploration. In our experiment, there are 10 out of 16 occasions where the final clustering is different from the best clustering. It implies that the number of instances we sampled in both of our random algorithms is not capable of providing enough stochastic confidence. Moreover, in case the final clustering is the same with the best clustering, the results are better in average, as shown in Table 7. Actually, only 1 out of 16 occasions obtained better result than the deterministic algorithms. That is to say that the probability where our random algorithms obtain optimal designs is less than $\frac{1}{16}$ so far. Although there is only one case successful, yet it shows that our random approach has potential to find optimal designs. Our methodology is correct but better cost exploring strategy is required.

number of instances tested	2496
average selector bits reduced	34.00%
non-optimal results	less than 54
	less than 2.16%

Table 8: Summary of experiments on source exchange

5.4 Source exchange

To show the improvement obtained by performing source exchange, we computed the numbers of selector bits saved. It is amazing that properly exchanging sources of commutative operations can improve results as much as very time-consuming iterative improvement[6].

Table 8 shows the summary of our experiments on source exchange. We can see that a considerable amount of selector bits are saved by performing source exchanges. Moreover, Theorem 3 provides a necessary condition for non-optimal instances. According to the theorem, we can determine the upper bound on the number of non-optimal results by those designs where each has more than one bi-colored vertexes within one connected subgraph. As a result, we also can see that our red-black-coloring algorithm can

	bus	reg	sel	drv	muxin	total	area
overall	13.0%	-6.9%	20.0%	26.6%	23.2%	17.3%	8.7%
same clustering†	6.5%	-7.7%	16.7%	20.2%	18.4%	13.3%	5.3%
different clustering‡	17.4%	-6.4%	22.2%	30.8%	26.4%	20.0%	10.8%

Table 7: Average increasing ratio of results obtained by our random approach over results obtained by Algorithm Greedy

† Occasions where the final clustering is the same with the best clustering
‡ Occasions where the final clustering is different from the best clustering

	HAL			Elliptic Filter		
algorithm	before	after	imp	before	after	imp
Greedy	6	2	67%	24	21	13%
One-Cluster	6	4	33%	23	16	30%
Spanning	6	4	33%	22	12	45%

Table 9: Numbers of selector bits required before and after source exchange

obtain optimal solutions in more than 97% cases from this table. Some examples of improvement obtained by source exchange are shown in Table 9.

5.5 Functional unit sharing

To show the improvement obtained by FU sharing, we compared results which are synthesized under same conditions except allowing using different FUs. Table 10 shows average improvement obtained by FU sharing, where 8 designs are tested by every algorithm. The second column are average improvements obtained by allowing using add-subtractors, which can perform both additions and subtractions, from that of allowing using only single-function components, which can perform only one type of operations. The third column are improvement obtained by allowing using both add-subtractors and ALUs from that of allowing using only single-function components. The ALU we used can performs comparison and logic operations, in addition to addition and subtraction.

In practical works, there used to be several types of operations in a design. There is trade-off between using multi-function FUs and using mono-function FUs to implement these operations. Using multi-function FUs can reduce the number of FUs required, since the utilization of FUs is increased. However, a multi-function FU is more expensive than a mone-function FU. Total FU cost for using less number of multi-function FUs may be higher than total FU cost for using more number of mono-function FUs.

	add-sub	add-sub & ALU
Greedy	4.90%	7.03%
One-Cluster	5.19%	7.98%
Spanning	5.19%	7.31%
overall	5.12%	7.41%

Table 10: Average improvement obtained by FU sharing

In addition to the cost for FUs, there is another even bigger impact when using multi-function FUs. Since there are more selections for each operation to be bound to, better interconnection can be obtained. Thus, the number of MUX inputs required is reduced. Actually, after we inspected the final designs, we understand that the improvement for using multi-function FUs sometimes is not obtained by reduction in the cost for FUs but is obtained by using less MUX inputs.

6 Concluding remarks

A clear binding model which can formulate any architectures is presented in this paper. This model which provides complete binding information can help people working on detail implementations and fine tune their binding algorithms. In addition, this model can also be used in interactive synthesis environments. Several simple algorithms which employ this model are proposed to demonstrate the performance on this model.

It is not necessary to use a real-cost-related function as the only directive metric for a binding algorithm. Some other metrics may work even better. In addition, sharing FUs can reduce the cost of a synthesized result by increasing utilization of FUs and more important, reducing a considerable number of MUX inputs required. Moreover, it is amazing that properly exchanging sources of FUs can save more than 30% selector bits. A linear-time post-processing source exchange algorithm, which can find an optimal solution in most cases(more than 97%), is also presented in this paper.

Experimental results show that our binding model is very useful in performing the binding task. Using our binding model, even simple algorithms can obtain better or competitive results than previous complex algorithms did. However, better sampling strategies are needed for our random approaches.

References

[1] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and design of embedded systems.* New Jersey: Prentice Hall, 1994.

[2] D. Gajski, N. Dutt, C. Wu, and Y. Lin, *High-Level Synthesis: Introduction to Chip and System Design.* Boston, Massachusetts: Kluwer Academic Publishers, 1991.

[3] K. Kucukcakar and A. Parker, "Datapath tradeoffs using MABAL," in *Proceedings of the Design Automation Conference*, 1990.

[4] B. Pangrle and D. Gajski, "Design tools for intelligent silicon compilation," *IEEE Transactions on Computer-Aided Design*, pp. 1098–1112, November 1987.

[5] B. Haroun and M. Elmasry, "Architectural synthesis for dsp silicon compilers," *IEEE Transactions on Computer-Aided Design*, pp. 431–447, April 1989.

[6] F. Tsai and Y. Hsu, "Star: An automated data path allocator," *IEEE Transactions on Computer-Aided Design*, pp. 1053–1064, September 1992.

[7] M. Rim, R. Jain, and R. D. Leone, "Optimal allocation and binding in high-level synthesis," in *Proceedings of the Design Automation Conference*, pp. 120–123, 1992.

[8] C. Gebotys and M. Elmasry, "Global optimization approach for architectural synthesis," *IEEE Transactions on Computer-Aided Design*, pp. 1266–1278, September 1993.

[9] B. Landwehr, P. Marwedel, and R. Dömer, "Oscar: Optimum simultaneous scheduling, allocation and resource binding based on integer programming," in *Proceedings of the European Design Automation Conference (EuroDAC)*, pp. 90–95, 1994.

[10] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco, CA: Freeman, 1979.

[11] F. Kurdahi and A. Parker, "Real: A program for register allocation," in *Proceedings of the Design Automation Conference*, 1987.

[12] C. Huang, Y. Chen, Y. Lin, and Y. Hsu, "Datapath allocation based on bipartite weighted matching," in *Proceedings of the Design Automation Conference*, 1990.

[13] K. Choi and S. Levitan, "A robust datapath allocation method for realistic system design," in *Proceedings of the International Conference on VLSI and CAD*, 1995.

[14] S. Devadas and A. Newton, "Algorithms for hardware allocation in data path synthesis," *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 7, pp. 768–781, 1989.

[15] G. Krishnamoorthy and J. Nestor, "Data path allocation using an extended binding model," in *Proceedings of the Design Automation Conference*, pp. 279–284, 1992.

[16] D. Mallon and P. Denyer, "A new approach to pipeline optimisation," in *Proceedings of the European Design Automation Conference (EuroDAC)*, pp. 83–88, 1990.

[17] D. Thomas, E. Dirkes, R. Walker, J. Rajan, J. Nestor, and R. Blackburn, "The system architect's workbench," in *Proceedings of the Design Automation Conference*, 1988.

[18] P. Paulin and J. Knight, "Scheduling and binding algorithms for high-level synthesis," in *Proceedings of the Design Automation Conference*, pp. 1–6, 1989.

[19] T. Ly, W. Elwood, and E. Girczyc, "A generalized interconnect model for data path synthesis," in *Proceedings of the Design Automation Conference*, pp. 168–173, 1990.

[20] D. D. Gajski, T. Ishii, V. Chaiyakul, H.-P. Juan, and T. Hadley, "A design methodology and environment for interactive behavioral synthesis." UC Irvine, Dept. of ICS, Technical Report 96-29,1996.

[21] E. Horowitz and S. Sahni, *Fundamemtals of Data Structures.* Computer Science Press, Inc., 1982.

[22] C. Liu, *Elements of discrete mathematics.* New York: McGraw-Hill, 1977.

[23] VLSI Technologies Inc., *0.8-Micron Datapath Library (VCC4DP3)*, 1992.

[24] S. Kung, H. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing.* Prentice-Hall, 1985.

[25] N. Berry and B. Pangrle, "Schalloc: An algorithm for simultaneous scheduling and connectivity binding in a datapath synthesis system," in *Proceedings of the European Design Automation Conference (EuroDAC)*, pp. 78–82, 1990.

[26] P. Paulin, J. Knight, and E. Girzyc, "HAL: A multi-paradigm approach to datapath synthesis," in *Proceedings of the Design Automation Conference*, 1986.