Implicit Computation of Minimum-Cost Feedback-Vertex Sets for Partial Scan and Other Applications

Pranav Ashar
C&C Research Labs, NEC USA
Princeton, NJ

Sharad Malik
Dept. of EE, Princeton Univ.
Princeton, NJ

Abstract

The contribution of this paper is an implicit method for computing the minimum cost feedback vertex set for a graph. For an arbitrary graph, we efficiently derive a Boolean function whose satisfying assignments directly correspond to feedback vertex sets of the graph. Importantly, cycles in the graph are never explicitly enumerated, but rather, are captured implicitly in this Boolean function. This function is then used to determine the minimum cost feedback vertex set. Even though computing the minimum cost satisfying assignment for a Boolean function remains an NP-hard problem, we can exploit the advances made in the area of Boolean function representation in logic synthesis to tackle this problem efficiently in practice for even reasonably large sized graphs. The algorithm has obvious application in flip-flop selection for partial scan. Our algorithm was the first to obtain the MFVS solutions for many benchmark circuits.

1 Introduction

The approach of making only a subset of flip-flops scannable is called partial scan [10]. With an intelligent choice of flip-flops to scan, partial scan can result in mitigated area and performance penalties and a reduced test application time compared to complete scan without sacrificing testability. A number of flip-flop designs and test application strategies have been proposed in the past [10, 1, 4] to demonstrate the feasibility of partial scan. Since the circuit design aspects of partial scan are not the focus of this paper, we will assume the feasibility of partial scan and not review the design aspects in detail.

The topic of interest in this paper is the algorithm used for flip-flop selection. An algorithm for flip-flop selection based on the knowledge that it is the cyclical structure of sequential circuits that leads to long test sequences was proposed in [4]. In this algorithm, each flip-flop is characterized by the number of cycles containing the flip-flop. Making a flip-flop scannable breaks all the cycles containing that flip-flop. Flip-flops are selected so that a minimal amount of feedback remains in the resulting circuit. Variations on this basic approach have been proposed by other researchers. This simple heuristic has received much practical success and to date consistently out performs other competing scan-insertion algorithms.

We present an implicit method for computing exactly the minimum cost feedback vertex set for a graph. Given a graph, a Boolean function is derived such that satisfying assignments for the function correspond to feedback vertex sets for the graph. The method is implicit in that cycles in the graph are never explicitly enumerated during the construction of the Boolean function. This function is then used for determining the minimum cost feedback vertex set. Even though computing the minimum cost satisfying assignment for a Boolean function remains an NP-hard problem, we can exploit the advances made in this area in logic synthesis to tackle this problem efficiently in practice for even reasonably large sized graphs. We believe that our algorithm was the first to compute the MFVS for a number of circuits from the ISCAS89 benchmark set [2].

Other exact methods have been proposed recently and in the past. An interesting algorithm for pruning the search space for the minimum FVS selection problem was proposed by Smith and Walford [9]. Their contribution was the proposal of a simple sufficient condition for one or more vertices to be a part of some minimum FVS. In their algorithm, they tested one vertex at a time, two vertices at a time and so on for satisfaction of this condition. If some set of vertices was found to satisfy it, the graph was modified by removing the selected set of vertices from the graph. The process was then repeated on the resulting graph. The proposed condition was the following: Consider a set of vertices A. Let B be the set of vertices in all the loops containing A. A is a part of some minimum FVS if removing A from the graph also breaks all loops formed from vertices contained in B. In other words, A is a part of some minimum FVS selection if there are no chords containing vertices other than those in A in all the cycles containing A. There are two problems with this algorithm: (1) Given that the test is only a sufficient condition, the algorithm would end up enumerating the exponential number of all possible vertex combinations in the worst case. (2) The algorithm is not useful if one is interested in enumerating all or a large subset of minimum FVS solutions. Another interesting experiment was reported recently in [3]. They reported results on a simple branch-and-bound algorithm for vertex selection with known graph reductions/partitioning applied after each selection. Interestingly, their implementation was able to compute the MFVS for all the ISCAS89 benchmark circuits in very short amounts of time pointing to the fact that the graphs corresponding to these circuits are highly partitionable.

Among the various approximate algorithms proposed for computing the MFVS, the algorithms of Lee and Reddy [6], and Park and Akers [8] are notable. Both first pre-process the graph by applying known reduction techniques. The contribution of Lee and Reddy was to try a number of heuristics for minimum FVS selection on the reduced graph. The heuristics select vertices with large in and out degrees, and vertices with a large number of loops through them for insertion into the FVS. They also showed how the path lengths in the resulting acyclic graph could be reduced by transforming the path length reduction problem into a minimum FVS selection problem. To their credit, their heuristics result in FVS selections that are very close to the minimum for the benchmark examples. The algorithm proposed by Park and Akers is very similar to the Smith-Walford algorithm. In fact, their algorithm works by identifying essential cycles in the graph where their definition of essential cycles is in direct correspondence with the sufficiency condition of Smith and Walford. A cycle is defined as an essential cycle if no proper subset of its vertices forms a cycle on its own. In their algorithm, one essential cycle is identified for each vertex, and the vertex which occurs in the maximum number of these essential cycles is inserted in the FVS. This heuristic also results in FVS selections that are very close to the minimum for the benchmark examples.

2 Minimum Cost Feedback Vertex Sets

Let $G(V, E)$ be a directed graph, with V being the set of vertices, and E being the set of directed edges. Each edge is an ordered pair, (v_i, v_j),$v_i, v_j \in V$. This edge is termed an *in-edge* for vertex v_j and an *out-edge* for vertex v_i. A path in the graph is an alternating

```
dfs_initialize(G)
{
    foreach vertex u  ∈  V {
        status[u] = unvisited;
        function[u] = 0;
    }
    foreach vertex u  ∈  V {
        if(status[u] == unvisited)
    dfs_initialize_visit(u);
    }
}

dfs_initialize_visit(u)
{
    status[u] = visiting;
    foreach edge(u,v){
        if(status[u] == unvisited)
    dfs_initialize_visit(v);
        if(status[u] == visiting){
    /* back edge found */
            add_to_set(B, u);
            function[u] = 1;
    }
    status[u] = visited;
}
```

Figure 1: Algorithm: Initialization Phase

```
compute(G, B)
{
    for i = 1 to |B| do{
        foreach vertex v ∈ V {
            status = unvisited;
        }
        foreach vertex v ∈ B {
            compute_step(v);
        }
    }
    f = 0;
    foreach vertex v ∈ B {
        f = f ∪ f[v];
    }
}

compute_step(v)
{
    status = visiting;
    g = 0;
    foreach edge(u, v){
        if(status[u] == unvisited)
            compute_step(u);
        g = g ∪ f[u];
    }
    f[v] = v ∩ g;
    status = visited;
}
```

Figure 2: Algorithm: Iterative Computation

sequence of edges and vertices. If the first and last vertex in the path are the same, the path is said to be cyclic. Let us assume that we have a cost function that maps a set of vertices to a real number. The minimum cost feedback vertex set (FVS) problem is defined as follows: find a minimum cost set of vertices V_{min}, such that removal of these vertices from G eliminates all cyclic paths in G. This problem has been shown to be NP-hard [5].

This problem is interesting for supporters of partial-scan since it forms the basis of the popular Cheng-Agrawal heuristic for flip-flop selection. This heuristic is as follows:

Given the circuit under consideration, construct a graph G with one vertex for each flip-flop, and an edge between two vertices, v_i and v_j, if and only if there exists a path through some gates from the flip-flop corresponding to v_i, to the one corresponding to v_j. Since there is a one-to-one correspondence between flip-flops and vertices in G, we will refer to the vertices themselves as flip-flops without ambiguity.

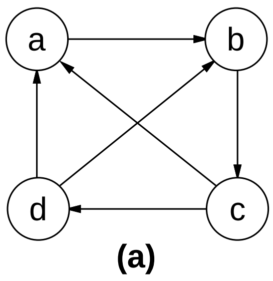
Cheng and Agrawal used the fact that it is easy to test circuits which have an underlying graph that is acyclic. In this case the testing problem is similar to that for combinational circuits, which is what full-scan provides. Thus, for non-acyclic circuits, if a set of flip-flops is selected that makes the graph acyclic, these flip-flops are ideal candidates for use as scan flip-flops in partial scan. This set is nothing but a FVS for G. In computing G, Cheng and Agrawal ignored self loops, i.e., edges of the form (v_i, v_i), since these loops did not cause much problems during test generation. Since the use of partial scan was motivated by the need to reduce the area and delay penalties associated with full scan, it is important that we keep these measures in mind during the selection of the FVS. Thus, the real problem that we need to solve is the minimum cost FVS problem. The cost of the FVS now reflects the delay/area constraints.

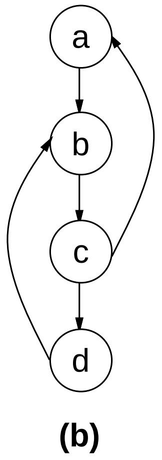
Since the problem is NP-hard, they proposed a heuristic algorithm for this purpose [4]. This algorithm selects a vertex that lies on the most number of cycles, adds it to the FVS being constructed, and repeats this step till the graph is acyclic. Computing the number of cycles through each vertex is obviously expensive. To overcome this, only a fixed number of cycles are enumerated, potentially compromising the quality of the final solution. In addition, the algorithm gives no indication about the optimality of the final solution it computes.

In this paper, we present an algorithm for computing the minimum-cost FVS that is exact but is still practical for large graphs. The key idea behind this algorithm is to avoid the explicit enumeration of cycles. This is accomplished by constructing in polynomial time a Boolean formula, $f(G)$, for a graph G, whose satisfying assignments are in one to one correspondence with feedback vertex sets of G. The construction of this formula implicitly accounts for all cycles in the graph. Of course, we cannot guarantee that the size of the Boolean representation will be polynomial in the size of G. However, here we exploit the efficient Boolean representation/manipulation techniques recently developed for use in logic synthesis, to make this problem tractable. In fact, with the representation we use for $f(G)$, the minimum-cost FVS can be obtained in time linear in the size of this representation.

We will illustrate the algorithm on a simple example. Consider the example shown in Figure 3 (a). We associate a logic variable with each vertex in the graph, for convenience we use the same identifier for this as the vertex name. In addition with each vertex, u, we store a logic function, $f(u)$, which we will compute iteratively. We start by first doing a depth first search of G, as described in algorithm, `dfs_initialize(G)` which is described in Figure 1. This reveals two back edges, i.e. edges that would result in cycles, in the graph. The graph is redrawn in Figure 3 (b), to reflect the back edges and the topological ordering imposed by the depth first search. Vertices c and d are added to the set B which contains all edges from which we have back edges. Clearly, removing all vertices in B will remove all cycles in the graph, however this is sub-optimal in general, otherwise this problem would not be NP-hard. As part of this initialization we assign $f(c) = 1$ and $f(d) = 1$, i.e. the logic functions for all vertices in the set B are the constant 1 function. With this initialization done, are ready for an iterative pass of the graph as shown in algorithm `compute(G, B)` in Figure 2. The new values for the logic functions for the vertices in B are computed using the old values. This computation is done in a reverse depth first order starting from the vertices in B as shown in the recursive algorithm `compute_step`. Thus the logic function for a vertex, v, is computed only after all the logic functions for all vertices u, $(u, v) \in E$, have been computed. The logic function at a vertex v is computed as:

$$f(v) = v \cap (\cup_{(u,v)\in E} f(u))$$





First Iteration:
f(a) = 1.a = a
f(b) = (1 + a).b = b
f(c) = b.c
f(d) = b.c.d

Second Iteration:
f(a) = b.c.a.
f(b) = (b.c.d + b.c.a).b = b.c.d + b.c.a
f(c) = (b.c.d + b.c.a) c = b.c.d + b.c.a
f(d) = (b.c.d + b.c.a).d = b.c.d

(c)

Figure 3: Example Graph

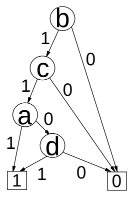


Figure 4: Example BDD

```
min_cost(f)
{
    if(f == 1){
        f.cost = 1;
        f.assignment = { };
    }
    if(f == 0){
        f.cost = ∞;
        f.assignment = { };
    }
    if(f->1branch.cost < f->0branch.cost){
        f.cost = f->1branch.cost;
        f.assignment = f->1branch.assignment;
    } else {
        f.cost = f->0branch.cost +
            cost(f.variable);
        f.assignment =
            f->0branch.assignment ∪ f.variable;
    }
}
```

Figure 5: Minimum Cost Satisfying Assignment with BDDs

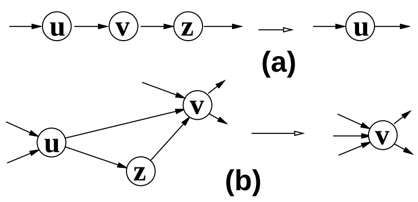


Figure 6: Example of Chain Collapsing

$f(v)$ implicitly stores all paths that pass though v. The equation above naturally expresses the fact that the set of paths through a vertex are those that pass through some in-edge of this vertex followed by this vertex. Figure 3 (c) shows the values of the logic functions after the first iteration. Since the longest simple path can possibly go through all of the vertices in B, we need k iterations to catch all such paths, where k is the number of elements in B. This is shown in algorithm `compute(G,B)` in Figure 2. In practice, not all k iterations may be needed and the algorithm terminates as soon as a fixed point is reached. After at most k iterations, the functions at the k vertices of B capture information about all cycles in the circuit. Let f be the union of these functions. Assuming no simplification of any formulas is done, f is computed in time proportional to $k.|E|$, where k is the number of elements of V and $|E|$ is the number of edges in the graph. Since is k is bounded by $|V|$, the complexity of generating f is $O(|V|.|E|)$. In practice it is closer to $O(|E|)$. In our example of Figure 3, $f = bcd + bca$. This directly provides bcd and bca, the two fundamental cycles (those not contained in any other cycle). Of course, if we were to actually write out f in a sum-of-products notation, we would be enumerating all the fundamental cycles. Even though that is more manageable than the set of all cycles, it is something what we would like to avoid.

Luckily, Binary Decision Diagrams (BDDs) provide an alternative representation for logic functions that in practice is more compact than the sum-of-product representation. This is the representation that we have used in our implementation.

The BDD for the function f for our example is shown in Figure 4. A BDD has two terminal vertices denoting the constant functions 1 and 0. Each non-terminal vertex has is labeled with a variable, and has two branches. The 1 (0) branch points to the function to be evaluated when the value of the variable is 1 (0). Evaluation of the function for any given assignment of inputs proceeds by walking down the BDD and taking the branch corresponding to the value of the variable at each vertex. The terminal vertex reached is the value of the function for that particular assignment.

The complement of f, denoted by $\bar{f}$ is the function that interests us most. The complemented variables in each satisfying assignment to $\bar{f}$ correspond to a feedback vertex set for the graph. For example, for the example in case, $\bar{b}$ is a satisfying assignment for $\bar{f}$, indicating that removal of b will break all cycles in the graph. In general there will be several satisfying assignments and hence several feedback vertex sets, we are interested in directly determining the one with the least cost. This is easily accomplished for most cost functions with the BDD representation. A cost of ∞ is assigned to the 0 terminal vertex, there is no satisfying assignment for it. A cost of 1 is assigned to the 1 terminal vertex. With this initial assignment Figure 5 shows how the minimum cost assignment can be obtained for an additive cost function. Here $f.variable$ indicated the variable at the root of f, $f.cost$ indicates the minimum cost satisfying assignment for f, and $f.assignment$ provides the minimum cost satisfying assignment. $cost(variable)$ is the cost of including this variable in the satisfying assignment. The logic function in consideration is a monotonically decreasing (negatively unate) function in all its variables. Thus, none of the variables need to be assigned to 1. This algorithm is linear in the size of the BDD since each edge is traversed exactly once.

2.1 Graph Reduction Prior to MFVS Selection

Significant performance enhancements can be obtained by using existing graph partitioning and graph compression techniques for computing the MFVS [9, 7]. Firstly, the MFVS for a graph is the union of the minimum cost feedback sets for its strongly connected components. Thus, there is no loss of optimality in considering each strongly connected component individually. This can potentially lead to significant savings since a large graph may be broken up into

CKT	#I	#O	#G	#L
s13207	31	121	7875	669
s1423	17	5	635	74
s35932	35	320	15998	1728
s38417	28	106	22263	1636
s5378	35	49	2195	164
s838	35	2	390	32
s9234	19	22	5556	228
s953	16	23	395	29

Table 1: **Statistics of Examples**

several smaller strongly connected components that are individually much easier to handle. Secondly, a vertex with a single fanin/fanout vertex can be merged into its fanin/fanout vertex and the MFVS algorithm can be applied on the compressed graph without loss of optimality. This process can be applied repeteadly as shown in Figure 6. A side effect of this is that if a self loop is introduced during compression, the vertex in the self loop must occur in any solution and can be deleted from the graph right away.

2.2 Handling Large Graphs

If the original graph is too large to handle, flip-flops should be selected, applying graph partitioning and compression after each selection, until the circuit has been partitioned enough that largest strongly connected component is of manageable size. One could either do this exactly in a branch-and-bound algorithm, or by heuristically picking the initial flip-flips. In the branch-and-bound approach, our algorithm is essentially used to bound the search.

Another heuristic is the following: The Boolean OR of the expressions at the back edge nodes always encapsulates enough information to find a (possibly suboptimal) solution to the feedback vertex problem. If memory limits are reached at any time, one could either pick the minimum cardinality complete solution from the expressions computed thus far, or one could pick a part of the solution and proceed from there on. Both these heuristics introduce suboptimality in the final solution. In the second option, one could possibly pick a partial solution of a certain cardinality which occurs in the largest number of solutions from among the solutions currently possible.

3 Experimental Results

In our experiments, we applied the algorithm to the circuits in the ISCAS89 benchmark set [2]. It should be noted that the largest examples in this set have more than 1500 flip-flops, making them a good test for our approach. Some of the circuits in the ISCAS89 benchmark set are small enough that our algorithm handles them trivially. Results for these examples are not reported here. From among the larger examples, we only report the results for examples for which the current implementation is able to find the MFVS. In particular, results are not reported for `s38584` and `s15850` since we were not able to obtain the optimum solutions for these two circuits. The statistics of the examples are provided in Table 1. **#I** is the number of primary inputs, **#O** the number of primary outputs, **#G** the number of gates, and **#L** the number of flip-flops in the circuit.

In Table 2, **Heuristic Loop Cutting** corresponds to the application of the heuristic loop breaking algorithm proposed by Lee and Reddy [6], and **Implicit Approach** corresponds to the application of our approach. Also in able 2, **# FF** corresponds to the cardinality of the solution obtained, **CPU Time** to the time required to compute the solution. The CPU time required by the method of [6] for these examples is generally very small.

An interesting aspect of our approach is that it enables the enumeration of all minimum cardinality solutions to the loop-breaking problem. This is exciting for the following reason: It is well known

CKT	# FF for Heur. Loop Cutting	Implicit Appr.	
		# FF	CPU Time (s)
s13207	59	59	31
s1423	22	21	3469
s35932	306	306	91
s38417	374	374	306
s5378	30	30	8
s838	0	0	1
s9234	53	53	1770
s953	5	5	1.5

Table 2: **Results of the Application of the Implicit Loop-Breaking Algorithm**

that the relative cost of scanning a flip-flop depends on various factors. For example, since scanning a flip-flop adds delay to paths going through it, if there was a choice between selecting two flip-flops, one would always prefer to pick the flip-flop with shorter paths through it. Given a circuit with relative costs associated with the flip-flops, our algorithm can automatically pick for the user the minimum weighted-cost solution from the millions of solutions possible. The data provided in Table 2 is for the special case of all flip-flops being assigned equal costs. The assignment of unequal costs does not add any amount of complexity to the selection problem.

References

[1] V. Agrawal and K. Cheng. A complete solution to the partial scan problem. In *The Proceedings of the International Test Conference*, pages 44–51, September 1987.

[2] F. Brglez, D. Bryan, and K. Kozminski. Combinational Profiles of Sequential Benchmark Circuits. In *Proceedings of the International Symposium on Circuits and Systems*, Portland, Oregon, May 1989.

[3] S. T. Chakradhar, A. Balakrishnan, and V. D. Agrawal. An exact algorithm for selecting partial scan flip-flops. In *The Proceedings of the Design Automation Conference*, June 1994.

[4] K-T. Cheng and V. D. Agrawal. An economical scan design for sequential logic test generation. In *The Proceedings of the Fault Tolerant Comupting Symposium*, pages 28–35, June 1989.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-complete ness.* W. H. Freeman and Company, 1979.

[6] D. Lee and M. Reddy. On determining scan flip-flops in partial-scan designs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 322–325, November 1990.

[7] E. Lloyd and M. Soffa. On locating minimum feedback vertex sets. In *Journal of Computer and System Science*, number 37, pages 292–311, 1988.

[8] S. Park and S. Akers. A graph theoretic approach to partial scan design by k-cycle elimination. In *The Proceedings of the International Test Conference*, pages 303–311, October 1992.

[9] G. Smith and R. Walford. The identification of a minimal feedback vertex set of a directed graph. In *IEEE Transactions on Circuits and Systems*, volume CAS-22, 1, January 1975.

[10] E. Trischler. Incomplete scan design with an automatic test generation methodology. In *The Proceedings of the International Test Conference*, pages 153–162, November 1980.