Loop Based Scheduling for High Level Synthesis

Hassan F. Al-Sukhni Habib Youssef Sadiq M. Sait Muhammad S. T. Benten

Department of Computer Engineering
King Fahd University of Petroleum and Minerals
Dhahran-31261, Saudi Arabia
e-mail: facy009@saupm00.bitnet

Abstract

This paper describes a new loop based scheduling algorithm. The algorithm aims at reducing the runtime processing complexity of path based scheduling techniques. It partitions the control flow graph of the input specification into subgraphs before scheduling the different paths of each subgraph. Benchmark tests as well as simulation results on the scheduling algorithm indicate that the proposed algorithm results in sizeable reduction in runtime.

1 Introduction

High-Level Synthesis (HLS) refers to the process of translating a high-level specification of the behavior of a circuit into a structural design, in terms of an interconnected set of RTL components, such as ALUs, registers and multiplexers.

Scheduling is defined in the context of HLS of synchronous digital systems as, the task of assigning operations to control states so as to minimize an objective function while meeting certain constraints [4]. Scheduling is an NP-hard problem [3]. Several heuristic algorithms have been developed to find a good solution rather than an optimal one [3, 7, 8]. The input to the scheduling algorithm is usually a control flow graph (CFG) or a control data flow graph (CDFG). A possible classification of scheduling algorithms is based on the level at which the algorithm handles the control flow graph (CFG). In this context scheduling algorithms are classified into operation based and path based. Operation based scheduling (OBS) algorithms visualize the CFG as a set of operations and utilize the available parallelism to assign operations to control states. Most of the known scheduling algorithms fall into this class, such as the force directed scheduling used in HAL [9], and the scheduling algorithm used in the YSC [7]. Path based scheduling (PBS) algorithms visualize the CFG as a set of paths and exploit the mutual exclusion of operations in different paths while assigning operations to control states. Algorithms that fall in this class include AFAP [1] and DLS [2].

This paper describes a new PBS algorithm that aims at reducing the processing complexity of the AFAP scheduling algorithm. In the following section, we present the necessary terminology and formulate the problem. In Section 3, the AFAP scheduling algorithm is briefly described. Section 4 presents our new scheduling algorithm called Loop Based Scheduling (LBS). Benchmark tests and experimental results are presented in Section 5. Finally, in Section 6 advantages and limitations of the approach are discussed and conclusions are drawn.

2 Problem Formulation

In this section we present some definitions that will be used in the formulation of the scheduling problem. These definitions are taken from [1] and presented here for the sake of completeness. Other definitions are introduced for the sake of comparison.

The input to the scheduling problem is a behavioral description expressed as a directed control-flow graph (CFG) $G = (V, E)$. The nodes $v \in V$ represent operations to be scheduled, and the edges give the precedence relation, i.e. $(v_i, v_j) \in E$ iff v_i is an immediate predecessor of v_j. The node v_j is called an immediate successor of v_i. Figure 1 is a behavioral specification of the greatest common divisor (GCD) benchmark test [6]. Figure 2(a) shows the CFG of this circuit. The nodes are numbered according to the statement numbers of the input specification given after the "– –" in Figure 1.

The interpretation of G is: an operation is executed if one of its predecessors is executed. If a node v has more than one successor, v is said to be a conditional branch (for example node 2 in Figure 2(a)). Only one of the successors will be executed. The decision of which successor is chosen is taken according to a condition predicate $cond(v_i, v_j)$ attached to the corresponding edge ($cond(v_2, v_1)$ in the CFG of the GCD is $EQ1$). If $cond(v_i, v_j)$ is true, then v_j is executed after v_i. The conditions on outgoing edges from conditional branches must be all mutually exclusive. Conditions are arbitrary boolean functions that are derived from conditional constructs in the behavioral description language like IF, CASE, WHILE, etc. The control-flow graph has a unique first operation, v_1 at which execution starts. It should be possible to reach all other operations from v_1.

A longest path through the control-flow graph is a path starting at node v_1 and ending at an operation with no successor. The set of all longest paths is

0-7803-2492-7/95 $4.00 © 1995 IEEE

denoted as P_l. It represents all possible operation sequences, excluding repetition of cycles, that the specified behavior allows. The set P_l for the GCD benchmark is shown in Figure 3(a).

A loop entrance node is the first node in a loop body. Let these nodes be identified as v_l^i where i is a running index starting at 1. The index i is incremented each time a loop entrance node is encountered in the graph. Hence v_l^1 is the first node in the first loop of the CFG. Similarly, v_l^2 is the first node in the second loop of the CFG. For example node 1 in the CFG of the GCD is v_l^1, while node 5 is v_l^2. Let P_e be the set of all paths starting at all loop entrances (except for the paths starting at v_1, since they are already included in the set P_l. The set P_e for the case of the GCD benchmark is shown in Figure 3(b).

```
gcd:
        EQ1 = (rst == 0)        --1
        If EQ1 goto gcd         --2
        x = xi                  --3
        y = yi                  --4
L10:
        EQ2 = (y == x)          --5
        If EQ2 goto L6          --6
        LT1 = (y < x)           --7
        If !LT1 goto L7         --8
        y = y - x               --9
        goto L10
L7:
        x = x - y               --10
        goto L10
L6:
        out = x                 --11
        goto gcd
```

Figure 1: Behavioral description of the GCD benchmark.

Operations that can be executed in parallel may be clustered in one node or ordered arbitrarily. If they are clustered in one node, they will be always scheduled in one control state (they will be treated as one large operation). If they are ordered, they may be scheduled in one or more control states.

3 As Fast As Possible Scheduling Algorithm

The AFAP scheduling problem was formulated in [1] as: "Given $G = (V, E)$ and a set of constraints, schedule all operations $v \in V$ such that all possible longest paths P_l execute in the minimum number of control states and all constraints are met."

The AFAP algorithm starts by converting the CFG into a directed acyclic graph (DAG) by eliminating the loops and keeping lists of such eliminations, along with the conditions of transfer either back to the loop or out of it. The second step of AFAP is to schedule

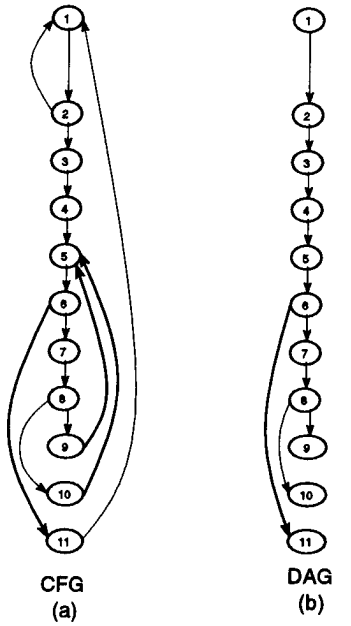

Figure 2: CFG and DAG of the GCD benchmark.

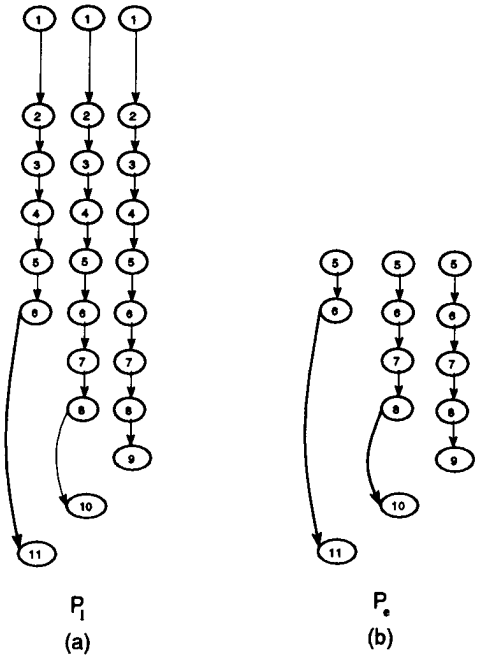

Figure 3: Paths of the GCD benchmark.

all paths of the DAG individually. Let the set of all paths processed by AFAP be denoted as P_{AFAP}, then

$$P_{AFAP} = P_l \bigcup P_e$$

Constraints are found for each of the paths in the P_{AFAP}. Constraints are either user specified or inherent to the specification. User specified constraints could be on area, timing or a combination. Inherent constraints are:

1. Variables can only be assigned once in a control state (or step).
2. I/O ports can be read or written only once in a control state.
3. Functional units can be used only once in a control state. This constraint is only relevant if the amount of hardware is constrained.
4. The maximal delay within one control state limits the number of operations that can be chained (i.e., operations that feed data to each other and are executed in the same control state).

For example, in Figure 2 a constraint of type 1 exists between nodes 4 and 9 because the variable y is assigned twice along this path in these two nodes. These two nodes have to be scheduled in different states in the resulting schedule. Hence the path has to be cut between nodes 4 and 9.

For one path, the nodes are totally ordered. Thus to each constraint corresponds an ordered sequence of nodes (a sub-path). A constraint is interpreted as an interval (from the first node to the last node of the sequence). Once all constraints are found, an interval graph is formed. The nodes of this interval graph represent constraints, and the edges join overlapping constraints. The problem of finding the minimum number of cuts along a given path is formulated as a minimum clique covering problem. The result of this step is a set of intervals. Intervals are called cuts because the path has to be cut at a node of the interval to satisfy all the constraints. An additional cut is added for each path at the first node of the path (i.e., an additional cut is added for each loop entrance node $v \in v_l^i$ and for $v1$).

To overlap the schedules of individual paths, another interval graph is generated. The nodes of this graph correspond to the cuts found in the previous step (remember that a cut is a set of nodes). Edges join nodes corresponding to overlapping cuts. A minimum clique covering of this graph will generate the minimum set of cuts that fulfills the fastest schedule for all paths.

Due to this overlapping of cuts, and since each loop entrance node is associated with a cut, the following lemma can be stated.

Lemma 1 *The number of loop entrance nodes in the CFG is the lower bound for the number of states of the schedule resulting from AFAP scheduling technique.*

The proof of this lemma is obvious from the previous discussion and is left here (it is assumed that $v1$ is a loop entrance node given that control is transferred to $v1$ after a node with no successor).

An FSM controller is then synthesized. Conditions to control the transitions between states and to control which operations are executed within each state are derived from the path execution conditions. For more details, interested readers are referred to [1] for a detailed description of the algorithm. To reduce the run-time complexity of using the clique partitioning technique, several heuristics are presented in [1] . However, since the number of paths explodes for realistic examples, the technique is limited in practice to small sized designs.

4 Loop Based Scheduling

As the name suggests, the AFAP scheduling technique results in a schedule that will execute all sequences of the input specification in the fastest way. However it is very expensive for large realistic circuits due to two reasons:

1. The use of the clique partitioning technique twice.
2. The algorithm processes all possible execution paths of the CFG. The number of paths explodes for large realistic examples.

In this work we present a new path based scheduling algorithm. The problem to be solved by this new algorithm is formulated as follows:

"Given a CFG $G = (V, E)$ and a set of constraints, schedule all operations $v \in V$ such that all constraints are met, while trying to minimize the number of control states."

The algorithm aims at scheduling the operations of the input specification utilizing the mutual exclusion among operations in different paths, while maintaining the run-time practically manageable. This is achieved by considerably reducing the number of processed paths by the use of a partitioning technique. In this technique the CFG is partitioned into *subgraphs*. Paths within each subgraph are used to generate its corresponding schedule. Then the schedules of the individual subgraphs are combined to generate the schedule of the specified design. The algorithm is called loop based scheduling (LBS) and consists of the following steps:

1. Partition the CFG into subgraphs.
2. Schedule each subgraph while meeting constraints.
3. Combine individual subgraphs schedules.

4.1 Partitioning of the CFG

The first challenge is to decide where to partition the CFG. Our choice was to partition the CFG at loop entrance nodes (LENs). This choice was guided by the result stated in Lemma 1 and the following argument.

- Partitioning the CFG results in the negative effect of preventing possible chaining, and hence may result in a slower schedule. This is true whether the graph is partitioned at LENs or at any other node in the CFG.

- LENs always force new states, since two executions of the loop can not be scheduled in the same control state. This implies that losses in speed of the resulting schedule due to the partitioning process will be limited to a single execution of the loop in the worst case. If loops are expected to execute for a relatively large number of times, then the resulting schedule will be close to As-Fast-As-Possible. This argument requires further study and experimentation.

The CFG is partitioned such that each subgraph contains exactly one loop entrance node. Figure 4 shows the subgraphs of the CFG of the GCD example. Subgraphs are generated from G as follows:

1. *Partition the CFG.* Starting at the first node of G, $v1$, subgraph 1 (denoted as sg_1) is constructed by adding nodes until v_l^2 is reached. Subgraph sg_2 is then constructed the same way starting at v_l^2 and adding nodes until another loop entrance node is reached. This process is repeated until all nodes of the CFG are consumed.

2. *Collect inter-subgraph branching information.* This is done by keeping a list of all branches outside the subgraph boundaries, their conditions and their corresponding destinations. An additional entry is added to this list from the last node of the subgraph to the first node in the following subgraph. This step is illustrated in Figure 4(b). Note the addition of the entry in sg_1 from node #4 to node #5 of sg_2, and the branch from node #11 in sg_2 to node #1 of sg_1.

3. *Check branching consistency.* Any branch outside a subgraph should be to the first node of another subgraph. If a branch does not satisfy this condition, the destination subgraph is broken into two subgraphs satisfying this condition. To illustrate this, assume that node #3 of sg_1 is a conditional branch to node #7 of sg_2 (not shown in the figure.) This will lead to cutting sg_2 just before node #7 producing three subgraphs instead of two.

4.2 Scheduling Individual Subgraphs

Each subgraph sg_i is scheduled separately as follows:

1. Convert each subgraph into a DAG.

2. Generate paths of each DAG.

3. Schedule individual paths.

4. Combine schedules of individual paths.

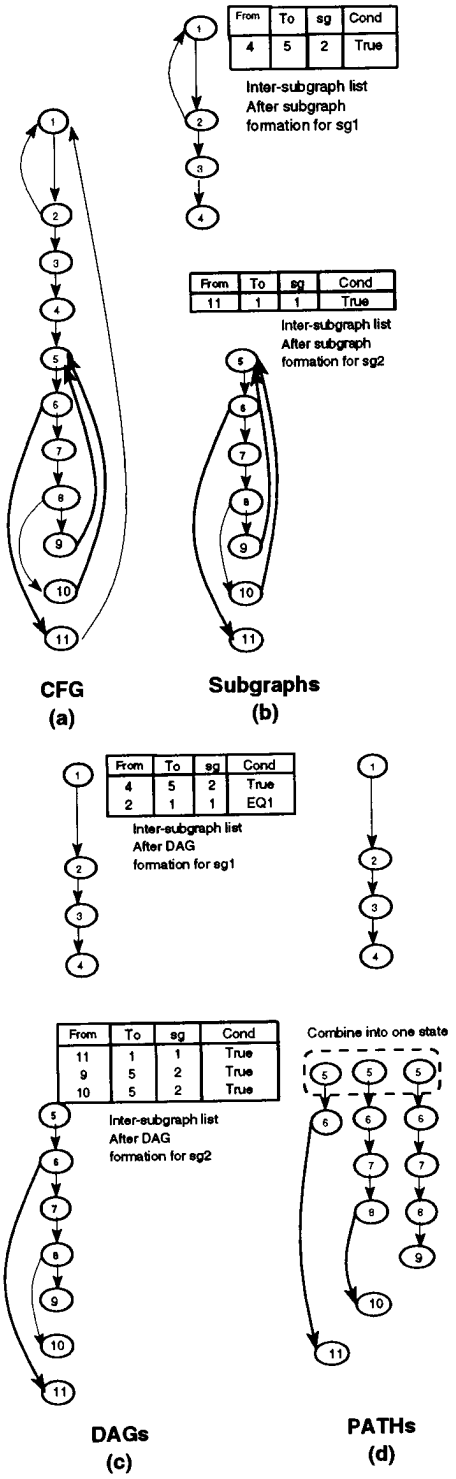

Figure 4: Subgraphs and paths for GCD in LBS.

Converting subgraphs into DAGs:

Each subgraph is converted into a directed acyclic graph (DAG). This conversion is done by eliminating loops back to the first node of each subgraph and adding an entry for it with the associated conditions in the inter-subgraph list. These lists are to be used later for transitions among subgraphs. In the GCD example, the branch from node #2 back to node #1 is eliminated in sg_1. The same applies to the branches from nodes #9 and #10 back to node #5 of sg_2, as depicted in Figure 4(c).

Generating paths of each subgraph:

All possible execution paths of each subgraph are traced starting from the first node. Each node in every path is associated with an execution condition. Execution conditions are derived from the branching nodes in the subgraph. The paths of both subgraphs of the GCD example are shown in Figure 4(d).

Scheduling individual paths:

Paths are scheduled individually in an as soon as possible (ASAP) fashion. Each path is traced from its starting node and constraints are checked (possible constraints are described in Section 3). Whenever a constraint is violated, the path is *cut*. All the nodes traced up to the cutting node (i.e., the last node before the cut) will be scheduled in the same state in a later step. Associated with each subgraph is a *cut-list*. This list includes an entry for each cut. The entry consists of the cutting node, the successor node and an associated transfer condition between the two. This condition is equal to the execution condition of the cutting node. Tracing is resumed after the cutting node until another cut is required or the path is consumed. This process will result in cutting each path into a set of intervals, each consisting of a set of nodes. Note that each node might exist in more than one interval associated with different execution conditions in different intervals. In the GCD example no paths require cutting.

Combining schedules of individual paths:

Cutting individual paths as in the previous step results in a set of intervals for each path. Intervals that start with the same node are collected together to form a control state, thus scheduling the operations of the subgraph into control states. Each node is associated within each interval with a different execution condition. These execution conditions are ORed to produce the execution condition for the instruction (or node) within the state. This results in scheduling each subgraph into a set of states. Transitions among states within each subgraph are taken from the collected information at each cutting point in the *cut-list*.

4.3 Combining schedules of subgraphs

To combine the schedules of the individual subgraphs, the inter-subgraph lists are used to produce the transitions from the associated subgraph. Note that each of these transitions is associated with a condition. This condition together with the execution condition of the node from which the branch is taking place are ANDed to form the state transition condition to the subgraph being branched to. By finishing this step the scheduling is complete, and the FSM controller is produced. Figure 5 shows the FSM for the GCD.

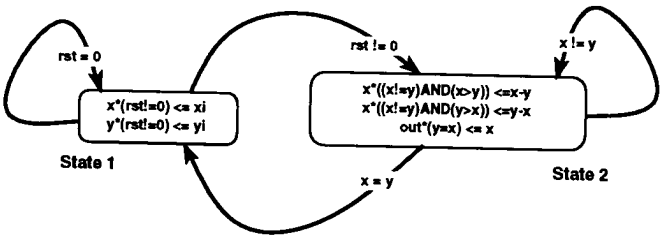

Figure 5: Finite State Machine of the GCD.

5 Experimental results

Design	Method	States	Paths	Transitions
Prefetch[1]	AFAP	2	3	3
	DLS	3	3	–
	LBS	2	2	3
GCD	AFAP	2	6	4
	DLS	2	6	4
	LBS	2	4	4
TLC	AFAP	8	19	18
	DLS	7	19	14
	LBS	5	19	14
DiffEq	AFAP	4	3	–
	DLS	–	–	–
	LBS	3	3	3

Table 1: WSHLS92 Benchmark Results.

In order to compare LBS with previous PBS techniques, we used benchmark[1] tests from the WSHLS92 [6]. The results are summarized in Table 1. We used the number of states and the number of transitions among the states as a measure of the solution quality, and the number of processed paths as a measure of heuristic time complexity (columns labeled States, Transitions and Paths, respectively). LBS outperformed previous PBS techniques in all tests. For example for the traffic light controller (TLC) circuit, the number of states for LBS is 5, compared to 8 for AFAP and 7 for DLS. The column labeled Transitions gives the number of transitions between the states of the resulting schedule. This is an indication of the controller complexity.

[1]Prefetch is taken from [1].

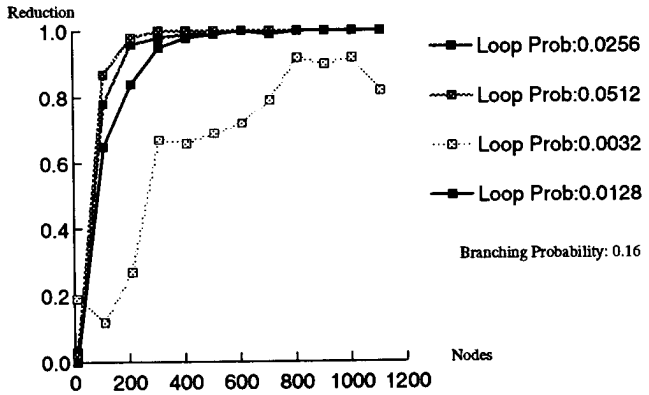

Figure 6: Simulation results.

The CFG partitioning used in LBS algorithm was introduced for the main purpose of maintaining the number of paths practically small. However, this was not manifested from the small benchmark tests. In order to demonstrate the effectiveness of this partitioning idea, CFGs with various sizes and structures were randomly generated. The topology of the generated CFGs was determined by three factors.

- The number of nodes in the CFG.
- The *branching probability* which is the probability that a certain node is a branch (fork) node.
- The *loop probability* which is the probability that a certain node is a branch (fork) and that this branch generates a loop in the CFG.

For each generated CFG, the number of paths that would have been processed by LBS and other PBS algorithms was counted. The results indicate a sizeable decrease in the number of paths to process (close to 100% reduction for graphs with more than 200 nodes.) The simulation results for a branching probability of 0.16 and various loop probabilities are depicted in Figure 6.

6 Conclusions

AFAP scheduling is a PBS algorithm that uses a clique partitioning technique twice to solve the scheduling problem. The outcome is the fastest executing schedule for all the paths of the CFG. However, the processing complexity of the algorithm is too expensive for realistic large examples. By partitioning the CFG into subgraphs, our new scheduling algorithm, LBS, reduces the run time of the algorithm considerably due to the sizeable reduction in the number of paths processed.

We do not expect partitioning at loop entrances to have a noticeable negative effect on schedule quality. However, a side effect of this technique which requires that branching always takes place to the first node in another subgraph may result in the prevention of possible chaining, thus leading to a slower schedule. This point requires more extensive experimentation.

The algorithm uses an ASAP scheduling technique while scheduling individual paths. This may be further enhanced by applying the AFAP algorithm at the subgraph level resulting in faster schedules.

Acknowledgment

Authors acknowledge King Fahd University of Petroleum and Minerals for all support. This research is supported by KFUPM Project # COE/DESIGN/145.

References

[1] R. Camposano, "Path-Based Scheduling for Synthesis," IEEE Trans. on CAD, Vol. 10, No. 1 January 1991.

[2] K. O'Brien, M. Rahmouni and A. A. Jerraya, "A VHDL-Based Scheduling Algorithm For Control-Flow Dominated Circuits," Technical report, Institute IMAG, Gernoble, France, 1992.

[3] P. G. Paulin and J. P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," IEEE Trans. on CAD, pp. 661-679, June 1989.

[4] M. C. McFarland, A. C. Parker and R. Camposano "The High-Level Synthesis of Digital Systems," in proc. of the IEEE, Vol. 78, No. 2, Feb. 1990

[5] R. A. Walker and R. Camposano "A Survey of High-Level Synthesis Systems," Kluwer Academic Publishers, 1991.

[6] N. Dutt and C. Ramachandran, "Benchmarks for the 1992 High Level Synthesis Workshop," Technical Report #92-107, University of California, Irvine, Oct. 30, 1992.

[7] R. K. Brayton, R. Camposano, G. DeMichelo, R. Otten and J. vanEijindhoven, "The Yorktown Silicon Compiler," in *Silicon Compilation*, D. D. Gajski, Ed. Reading, MA: Addison-Wesley, 1988, pp. 204-311.

[8] A. C. Parker, J. Pizarro and M. Mlinar, "MAHA: A Program for Datapath Synthesis," in Proc. of the 23rd Design Automation Conference, New York, NY, ACM/IEEE, June 1988, pp. 461-466.

[9] P. G. Paulin, J. P. Knight and E. F. Girczyc, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis," Proc. of the 23rd DAC, pp. 263-270, June 1986.