A Tree-Based Scheduling Algorithm For Control-Dominated Circuits *

S.H. Huang† Y.L. Jeang§ C.T. Hwang‡ Y.C. Hsu‡ J.F. Wang§

‡Department of Computer Science, Univ. of California, Riverside, CA 92521
†Department of Computer Science, Taiwan Univ., Taipei, Taiwan
§Department of Electrical Engineering, Cheng – Kung Univ., Tainan, Taiwan

Abstract

Scheduling algorithms for control dominated applications have not been widely published. Path-based scheduling was the first attempt to tackle this problem. This approach works well on the benchmark examples. However, it imposes a restriction on the execution order of the operations before scheduling. We alleviate this restriction by representing all the execution paths by a tree. Tree representation not only inherits all the advantages of the path representation, but also releases the execution order constraint.

A two phase algorithm is proposed to solve the scheduling problem on the tree representation. In the first phase, a partitioning algorithm is performed on the tree in a top down manner in order to optimally execute every path. In the second phase, the corresponding state transition graph is constructed in a bottom-up manner in order to minimize the total number of states as well as the control logic. By utilizing node unification, the complexity of the algorithm can be restricted to $O(pbn^2)$, where p is the number of paths, b is the number of blocks and n is the number of operations. We tested the algorithm on a set of benchmarks and achieved reductions on the number of states as compared with previous algorithms.

1 Introduction

Scheduling algorithms for control dominated applications have not been widely published[1, 2]. Path-Based Scheduling [1] is one that is well suited for the scheduling of operations that occurs in control-flow dominated machines. The basic principle of the path based scheduling is to minimize the total execution time of the algorithm, measured in control steps, by taking into account the different possible paths that may occur in the algorithm. The drawbacks of the system are a predefined order of the execution for the operations must be enforced before scheduling and the factor of control unit does not take into account.

*This work has been supported in part by UC-MICRO under project No. 92-057 and SMOS systems

In this paper, we propose a new representation and an algorithm for the synthesis of control dominated machines. We are concerning about the following properties.

- *Schedule every path as fast as possible.*
  In order to minimize every execution path, sometimes even the same operation has to be treated differently in different paths. This is the idea used in the path-based scheduling. However, their approach imposes an execution order for the operations before scheduling, which is not necessary in a behavior description. This limitation is alleviated in our approach by keeping all the paths on a tree. Being kept in a tree structure, the operations are allowed to be moved globally on the tree so that each path can be executed as fast as possible.
- *Minimize control logic.*
  The controller generates signals to supervise the execution of data path operations. The size of a controller is affected by the number of states, the number of transitions and the output logic generated in each state. A bottom-up construction algorithm is proposed to reduce the control logic by merging mutually exclusive states that contains operations with the largest similarity.

The remainder of the paper is organized as follows. Section 2 introduces the tree representation and the path optimization on the tree. The algorithm is presented in section 3. The experimental results and comparison to other approaches are made in section 4. Finally, concluding remarks are given in section 5.

2 Tree Representation

2.1 VHDL Subset and Flow Graph

The input is a sequential specification of the algorithm described in VHDL subset. Fig. 1 shows the statements acceptable by our scheduler. In addition to the VHDL description, the designer can specify the resource constraints such as the number and/or type of hardware modules to be used in the data path.

Fig 2 gives a behavior description which we will use to illustrate our algorithm in this paper. The behavior description is first compiled into a *flow graph* consisting

30th ACM/IEEE Design Automation Conference®

Signal Assignment Statement
Variable Assignment Statement
If Statement
Case Statement
Loop Statement
Exit Statement
Next Statement
Procedure Call Statement
Return Statement
Wait Statement

Figure 1: The VHDL Subset.

```
ENTITY example IS
    PORT(x1,x2,x3,x4,x5,x6: IN BIT16;
         rst: IN BIT;
         out1: OUT BIT16);
END example;
ARCHITECTURE behavior OF example IS
    BEGIN
        PROCESS
            variable t1,t2,t3,t4,t5,t6,t7: BIT16;
            BEGIN
                wait until (rst='1');                          -o1
                t1 := x1-x2;                                   -o2
                t2 := x2+x3;                                   -o3
                t3 := x1+x2;                                   -o4
                t4 := x3+x4;                                   -o5
                IF (x5/=0) THEN                                -o6
                    WHILE (t2<0) LOOP                          -o7
                        t5 := t3+1;                            -o8
                        IF (t5>0)                              -o9
                            THEN t6 := x5-x3;                  -o10
                            ELSE t6 := x5-1;                   -o11
                        END IF;
                        t3 := t3+x4;                           -o12
                        t2 := t2+t6;                           -o13
                    END LOOP;
                    t2 := t2-x5;                               -o14
                ELSE
                    t7 := t1+1;                                -o15
                    IF (t1>0) THEN                             -o16
                            t3 := t7+x6;                       -o17
                        ELSE
                            IF (x2>0) THEN                     -o18
                                    t3 := x2+x4,               -o19
                                    t1 := t7-x4;               -o20
                                ELSE t3 := t7-t4;              -o21
                            END IF;
                            t2 := t1+t7,                       -o22
                    END IF;
                END IF;
                out1 <= t2+t3,                                 -o23
        END PROCESS,
END BEHAVIOR;
```

Figure 2: A VHDL Example Program.

of a set of basic blocks linked by flow-of-controls. A flow-of-control can be forward or backward. A forward edge represents a move from a basic block to a successor block, while a backward edge represents a loop-construction.

Fig. 3(a) shows the flow graph of this example program. The *pre-test* loop (e.g, a while-loop) is translated into a *post-test* form (e.g., "repeat .. until") which starts with an "if-statement". If the control flows through the true part, the loop is executed at least once, else the loop will not be executed.

2.2 Converting a Flow Graph to a Tree

Instead of treating each path individually as in [1], we keep all the paths in a tree. Fig. 3(b) shows the tree representation for this example program.

A binary branch block is denoted as B_{if}. For a binary branch, B_{true} (B_{false}) is the successor block which will be executed immediately after B_{if} if the test o_{if} is true (false).

For each loop, there is a unique entry, called *loop header*. For instance, in Fig. 3 (b), B_4 is a loop header. In addition, a *preheader* is created. The preheader has the loop header as its successor. Initially, the preheader is empty. But, after tree optimization, some operations may be moved to the preheader.

2.3 Tree Optimization

Quite often, an operation is not redundant in a program, but it is redundant for some of its containing paths. For example, in Fig. 3(b), the operation $o3: \ t2 := x2 + x3$ is redundant for the paths through block B_{12}. Therefore, a tree can be optimized by removing the redundant operations of a path. Our idea is to propagate each operation to the latest block where it is needed to be executed.

Our tree optimization algorithm starts with a procedure call $Tree\_Optimization(B_{root})$, where B_{root} is the root of this tree. The procedure tries to propagate the operations in the root block to its successor blocks. Then, the procedure is recursively applied to its subtrees. For a branch block B_{if}, an operation o_i in B_{if} can be propagated to its successor blocks if there is no operation $o_j \in B_{if}$ which depends on o_i.

Let $d(o_i)$ be the variable defined by o_i and $in[B]$ be the set of variables which are live entering block B. An operation o_i in B_{if} is a redundant operation for the paths through B_{true} if $d(o_i) \notin in[B_{true}]$. The redundant operation of the paths can be removed by moving the operation from B_{if} to B_{false}. For instance, in Fig. 3(b), the operation $o2: \ t1 := x1 - x2$ is redundant for the paths through block B_3 for $t1 \notin in[B_3]$. Thus, the operation $o2$ will only be copied to block B_{10} (because $t1 \in in[B_{10}]$) and removed from block B_2. The result of tree optimization for the example program is shown in Fig. 4.

3 The Proposed Algorithm

The algorithm is divided into two phases. In the first phase, a partitioning algorithm is performed on the optimized tree in a top down manner in order to optimally execute every path. In the second phase, states are merged with the objective to minimize the total number of states and the control logic.

3.1 Top-Down Scheduling

Our top-down scheduling algorithm differs from those of other approaches in the following:

- It dynamically unifies the identical operations;
- It chains several operations and/or schedules several branches in a state; and
- It is capable of scheduling operations across blocks.

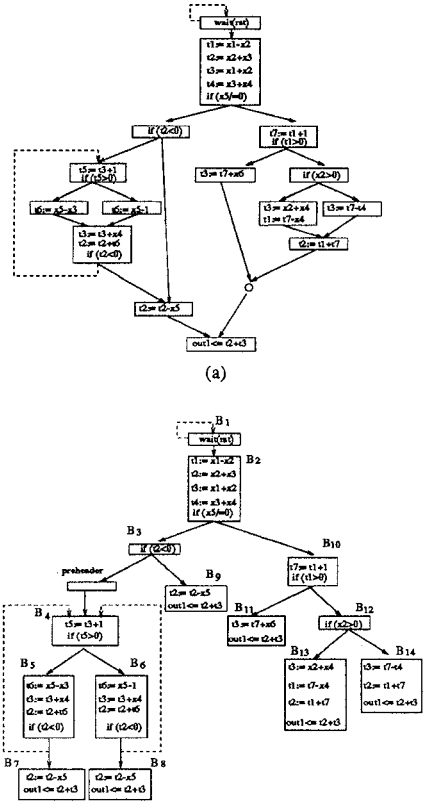

Figure 3: (a)Flow Graph (b)Tree.

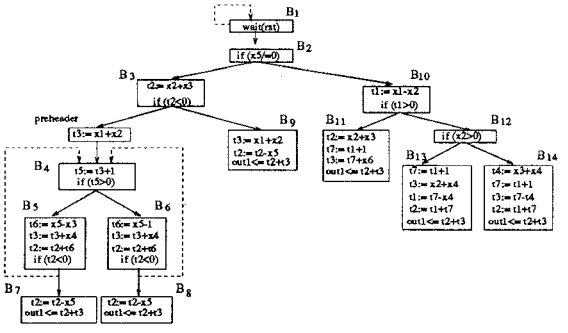

Figure 4: Optimized Tree

```
Procedure Top_Down_Scheduling()
  Procedure State_Partition(B)
    Begin
      create a new state S;
      call Schedule_a_State(S, B); (in Fig. 6)
      For each leaf blocks B_k of state S
        call State_Partition(B_k);
    End;
  Begin
    call State_Partition(B_root);
  End.
```

Figure 5: Top-Down Scheduling Algorithm

```
Procedure Schedule_a_State(S, B)
 Procedure Schedule_a_Block(B, r)
  Begin
   For the ready queue associated with block B
    Find a highest priority o_i that can be scheduled;
    if found
    then
     Begin
       schedule operation o_i into this state S;
       update corresponding ready queues;
       update the available resources as r';
       if o_i is a branch operation
         for each forward successor block B_k induced by o_i
           if B_k is not a loop header
             call Schedule_a_Block(B_k, r');
     End
    else return;
  End;
 Begin
  call Schedule_a_Block(B, r);
 End.
```

Figure 6: Schedule a State

3.1.1 State Partition on the Tree

The top-down scheduling algorithm is described in Fig. 5. The notation B_{root} denotes the root block of the optimized tree. It starts from the root of the optimized tree until all the leaves are scheduled. The procedure State_Partition is recursively called to perform the state partitioning on the optimized tree.

The procedure *Schedule_a_State* will try to schedule as many operations as possible into a state. This procedure is discussed in Section 3.1.2. When no operation can be scheduled into a state, it follows the leaf blocks of this state to generate next states. Those leaf blocks of this state will be the roots of next states.

3.1.2 Schedule a State

The algorithm of scheduling a state is described in Fig. 6. Given the root block B and the resource constraint r, the algorithm schedules a state S as a subtree which is induced by the branch operations scheduled into this state S. Therefore, a state may consist of several blocks. The details of this algorithm are discussed as follows.

- **Ready Queue for Each Block:** In order to efficiently implement the algorithm, each block is associated with a ready queue. When an operation is ready, it is put in the ready queues of the blocks in its containing path. For example, in Fig. 4, the operation $o3 : t2 := x2 + x3$ is ready, and is put in the ready queues of B_1, B_2 and B_3. The execution possibilities of $o3$ kept in ready queues of B_1, B_2 and B_3 are respectively 0.5, 0.5 and 1. When an operation is scheduled, it must be removed from all ready queues containing it.
- **Unification of Identical Operations:** In a tree representation, an operation which has been duplicated to different paths can be ready at the same time. They should be unified when they are scheduled to the same block. Therefore, when an operation o_i becomes ready, we check the ready queue to see if there is any copy of o_i. If so, operation o_i is unified with the copy. The execution

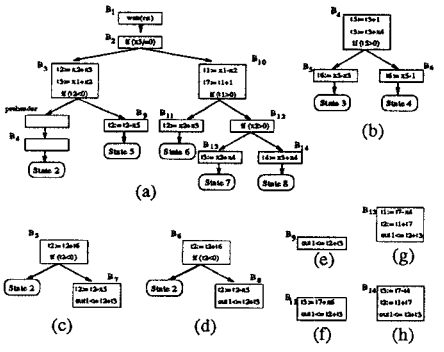

Figure 7: The Result of Top-Down Scheduling. (a) State 1 (b) State 2 (c) State 3 (d) State 4 (e) State 5 (f) State 6 (g) State 7 (h) State 8.

possibility of o_i will be the sum of possibilities of these unified operations. As more duplicated copies are unified, the operation will have a larger priority to be scheduled. For example, in Fig. 4, the operation $o4 : t3 := x1 + x2$ is duplicated to preheader and B_9. The two copies of $o4$ are ready. They are unified in the ready queues of blocks B_1, B_2 and B_3. Thus, the execution possibilities of $o4$ which are kept in ready queues of B_1, B_2, B_3, preheader and B_9, are 0.5, 0.5, 1, 1, and 1, respectively.

• **Resource Management:** Since the operations in a state are maintained in a subtree (induced by the branch operations), it can be very efficient to manage the resource utilization. The successor blocks induced by a branch can share the same resources because the blocks are mutually exclusive. Therefore, when a branch operation o_i is scheduled, the procedure $Schedule\_a\_Block(B_k, r')$ is called for each successor block B_k induced by o_i, where r' is the remaining resources available after o_i is scheduled. Those successor blocks induced by o_i can share the same resources r'.

Fig. 7 shows the result after top-down scheduling. Suppose we are given 2 adders and 1 subtractor. For state 1, the leaf blocks B_4, B_9, B_{11}, B_{13} and B_{14} are respectively the roots of state 2, state 5, state 6, state 7 and state 8. Similarly in state 2, the leaf blocks B_5 and B_6 are the roots of state 3 and state 4, respectively.

3.1.3 Time Complexity

Let n be the number of operations in the behavior description and p be the number of paths. The number of operations in the tree is $O(pn)$. Because of the unification procedure, the length of a ready queue is limited by $O(n)$. Suppose b is the number of blocks in the tree. Since we set a ready queue for each block, the time complexity for updating ready queues is $O(bn)$. Therefore, the time complexity of the algorithm is $O(pbn^2)$.

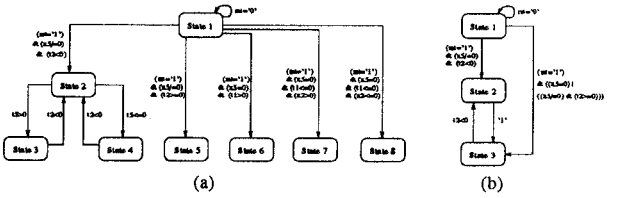

Figure 8: (a) The State Transition Graph after Top-Down Scheduling. (b) The State Transition Graph after Bottom-Up Rescheduling.

3.2 Bottom-Up Rescheduling

After top-down scheduling, the number of states of each path is minimum. The state transition graph of this example is shown in Fig. 8 (a). We next merge the mutually exclusive states in a bottom-up manner so that the cost of a FSM is minimized.

The bottom-up rescheduling algorithm achieves the following properties:

- It retains the as-fast-as-possible execution of every path.
- It uses a minimal number of states and transitions.

The details of this algorithm is discussed as the below.

3.2.1 State Merging

After top-down scheduling, a state transition graph is constructed. For a state S_i, we define its level $L(S_i)$ as followings:

1. if S_i is a leaf state, $L(S_i) = 1$;
2. $L(S_i) = MAX_{S_j \in S} L(S_j) + 1$, where S is the set of forward successor states of S_i.

From this definition, we can see that the states on the same level are mutually exclusive. We can merge the states on the same level to minimize the number of states and transitions. In Fig. 8 (a), state 3, state 4, state 5, state 6, state 7 and state 8 are the leaves of this graph. Those states are on level 1. Besides, state 2 is on level 2 and state 1 is on level 3. After state merging, there are only three states needed. The state transition graph after state merging is in Fig. 8 (b).

The duplicated operations can be unified during state merging. For example, in Fig. 7, the operation $o23 : out1 \Leftarrow t2 + t3$ are duplicated to state 3, state 4, state 5, state 6, state 7 and state 8. After state merging, the copies of operation $o23$ are unified in the same state.

3.2.2 Schedule Operations Across States

Suppose there is a state transition $S_i \rightarrow S_j$. Then, there must be a block B, which is the leaf of S_i and also the root of S_j. After we merge the states on level $L(S_j)$, some operations in the leaf block B of state S_i become ready. If resource is enough, those ready operations in S_i can be moved downward to S_j, that is to mean they can

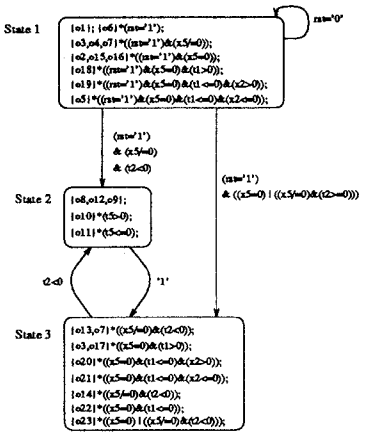

Figure 9: Final State Transition Graph

	#ALU	#add	#sub	cn	#C_STEPS states	#1	#2	#3
TS	0	1	1	1	7	7	4	4
	0	1	1	2	7	7	4	3
	2	0	0	2	6	6	4	3
[4]	0	1	1	1	7	7	4	4
	0	1	1	2	7	7	4	3
	2	0	0	2	6	6	5	3
path [1]	0	1	1	2	8	7	6	3
	2	0	0	2	6	6	5	3
Cyber [3]	0	1	1	1	7	7	6	5
	0	1	1	1	7	7	5	4

Table 1: Results of Wakabayashi's Example.

be scheduled in the merged state which includes S_j. Moving an operation downward tends to make the mutually exclusive states having the largest similarity.

The final state transition graph for our example program is shown in Fig. 9.

4 Experiments

The algorithm, Tree-based Scheduling (TS), has been implemented. This secetion reports the experiments on the benchmark examples. In the first example [3], we compare our results with those produced by Wakabayashi's scheduling algorithm [3], the path-based scheduling algorithm [1], and the conditional resource sharing algorithm in [4]. The second example was adopted from [5]. We compare our results with those by MAHA (critical-path-first scheduling) [5], path based approach [1], and [4].

4.1 Wakabayashi's Example

This example has 16 operations including two branch nodes which result in three paths. We assume an equal probability for a conditional branch. Table 1 shows the results ($\#C\_STEP$) including (i) the total number of states in the FSM (*states*), (ii) the number of steps of paths #1, #2 and #3. Our results are similar to those by [4] except in the case of two ALUs where the #2 of our result is 4 while it is 5 in both [4] and the path based approach [1].

	#add	#sub	cn	#C_STEP states	long	short	avg
TS	1	1	1	5	5	2	$3\frac{5}{16}$
	1	1	2	5	5	2	$3\frac{1}{16}$
	2	3	3	3	3	1	$1\frac{9}{16}$
Liu	1	1	1	8	8	3	-
	1	1	2	6	5	2	-
	2	3	3	3	3	2	-
path [1]	1	1	2	9	5	2	-
	2	3	5	4	3	1	-
MAHA	1	1	2	8	8	-	-
	2	3	3	4	4	-	-

Table 2: Results of the MAHA's Example

4.2 MAHA's Example

There are 22 operations, including 6 branches and 12 paths in this example [5]. The longest paths contain 9 operations (this number does not count the branch operations [4, 5]). It is reduced to 8 after tree optimization. Given constraints on *#adder*, *#sub* and *cn*, Table 2 shows the results ($\#C\_STEP$) in terms of (i) *states*, (ii) the number of control steps in the longest execution instance (*long*), (iii) the number of control steps in the shortest execution instance (*short*), and (iv) the average number by taking the $CDFG$ in [5] as input (*avg*).

5 CONCLUSION

A scheduling algorithm for the synthesis of control dominated circuit has been presented. Our primary objective is to schedule every execution path as fast as possible. To achieve this goal, we remove redundant operations from each path, merge branches into a multi-branch, and allow operations to be scheduled across blocks. Future work includes data path synthesis with control minimization and behavioral synthesis for testability.

References

[1] R. Campssano and R. Bergamaschi. Synthesis using path-based scheduling: Algorithms and exercises. In *Proc. 27th Design Automation Conference*, pages 450–455, June 1990.

[2] Wayne Wolf, A. Takach, C.Y. Hwang, R. Manno, and E. Wu. The princeton university behavioral synthesis system. In *Proc. of the 29th Design Automation Conference*, June 1992.

[3] K. Wakabayashi and T. Yoshimura. A resource sharing and control synthesis method for conditional branches. In *Proc. ICCAD'89*, pages 62–65, November 1989.

[4] T. Kim, J. Liu, and C.L. Liu. A scheduling algorithm for conditional resource sharing. In *Proc. ICCAD'91*, pages 84–87, November 1991.

[5] A.C. Parker, J. Pizarro, and M.J. Mlinarr. Maha: A program for data path synthesis. In *Proc. 23rd Design Automation Conference*, July 1986.