ELSEVIER Microprocessing and Microprogramming 41 (1995) 501–519

Microprocessing and Microprogramming

A new scheduling algorithm for synthesizing the control blocks of control-dominated circuits

Shih-Hsu Huang [a], Yu-Chin Hsu [b], Yen-Jen Oyang [a,*]

[a] *Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan*
[b] *Department of Computer Science, University of California, Riverside, CA 92521, USA*

Received 12 September 1994; revised 24 November 1994, 16 March 1995; accepted 12 May 1995

Abstract

This paper describes a new scheduling algorithm for automatic synthesis of the control blocks of control-dominated circuits. The proposed scheduling algorithm is distinctive in its approach to partition a control/data flow graph (CDFG) into an equivalent state transition graph. It works on the CDFG to exploit operation relocation, chaining, duplication, and unification. The optimization goal is to schedule each execution path as fast as possible. Benchmark data shows that this approach achieved better results over the previous ones in terms of the speedup of the circuit and the number of states and transitions.

Keywords: Scheduling; High-level synthesis; Control-dominated circuit synthesis; Operation chaining; Finite state machines

1. Introduction

Research on behavior synthesis has been very active and has made a lot of progress during the past few years [1–4]. The issue of behavior synthesis is to design a circuit from a hardware behavior description. Since the designs for different application domains differ substantially in their characteristics, it is believed that a successful synthesis system should be domain specific and architecture specific. For example, traditional Mealy or Moore type finite state machines are good for control-dominated applications [5], while superscalar or VLIW architectures are good for computation intensive applications.

A control-dominated application typically consists of very few arithmetic operations. Most operations in a control-dominated application are short-delay operations such as data transfer, logic, decision making, and I/O operations. Since the execution delay for such operations is usually short, it is very common to chain several operations which have control or data

* Corresponding author. Email: yjoyang@csie.ntu.edu.tw

dependencies into a single control step as long as their total propagation delay is less than the clock cycle time. In other words, chaining consecutive control-dependent and/or data-dependent operations into one state is an important attribute which a good scheduling algorithm for control-dominated circuit synthesis must have.

1.1. Previous work

Most previous scheduling algorithms [6–8] use control/data flow graph (CDFG) to represent the dependencies of operations. Based on CDFG representation, they try to exploit the potential concurrency of operations and take advantage of the freedom left to move operations across control steps. List scheduling [6] is a widely used heuristic. Global transformation techniques such as trace scheduling [7] and percolation scheduling [8] rely on code motion across basic blocks. The problem of trace scheduling is that the patching codes generated during the bookkeeping phase can be exponential to the size of the input program. Many approaches [9–11] have been proposed to solve this problem.

The feature of chaining consecutive data-dependent operations into a single control step has been implemented in some behavior synthesis systems. In the Slicer state synthesizer [12], if the cumulative delay through a computation path is short enough to allow the operations to perform sequentially, the operations are chained into a single state. In force-directed scheduling [13], the range of possible control steps for each operation is used to form a so-called distribution graph. Operations are then selected and placed so as to balance the distribution as much as possible. The feature of chaining consecutive data-dependent operations is implemented by extending the time frames of fast combinatorial operations into the previous and/or next control steps (when the total propagation delay in those control steps is less than the clock cycle time). Both approaches [12,13] are restricted to a basic block.

Path-based scheduling algorithm [14] was the first attempt to tackle scheduling for control-dominated circuits. Its basic principle is to minimize the total execution time of the design, measured in number of control steps, by taking into account all the possible execution paths due to the presence of loops and branches. In order to chain consecutive control-dependent operations into a single state, path-based approach expands all the possible execution paths before carrying out scheduling. One major limitation of path-based approach is that a predefined order of the operations must be chosen before scheduling.

1.2. Our approach

In this paper, we present a new scheduling algorithm for automatic synthesis of the control blocks of control-dominated circuits. It works on the CDFG to exploit operation relocation, chaining, duplication, and unification. The main distinctions of the proposed scheduling algorithm are elaborated in the following:

- *It partitions a CDFG into an equivalent state transition graph.* Different from path-based approach, the proposed scheduling algorithm works on the CDFG to exploit chaining of consecutive control-dependent and/or data-dependent operations. The basic idea behind the proposed algorithm is to schedule as many operations as possible, which may have control or data dependencies, into a single state. Control-dependent operations (in consecutive basic blocks) are chained into the same state as long as their total propagation delay is less than the clock cycle time. As a result, we can schedule several basic blocks linked by control constructs into a single state. After the scheduling process is finished, the CDFG is partitioned into an equivalent state transition graph.
- *It schedules every execution path as fast as possible.* Due to the presence of loops and branches, the execution path may be different as the input

Signal Assignment Statement
Variable Assignment Statement
While Statement
Wait Statement
Next Statement
Exit Statement
If Statement
Case Statement
Procedure Call Statement (recursion is not allowed)
Function Call Statement (recursion is not allowed)
Return Statement
Null Statement

Fig. 1. The subset of VHDL statements (within a process) acceptable by our scheduler.

changes. In order to optimize every execution path, sometimes an operation has to be treated differently in different paths. This is the idea used in path-based approach. However, because path-based scheduling must choose an execution order of operations before scheduling, it does not use the additional freedom obtained by reordering operations which leads to more improvements. This limitation is alleviated in our approach by keeping all execution paths on a CDFG representation. The proposed scheduling algorithm not only preserves the advantage of path-based approach (i.e. the capability of scheduling an operation into multiple states), but also preserves the flexibility of operation reordering.

The rest of the paper is organized as follows. The next section discusses the optimization of execution paths through control constructs such as loops and branches. Section 3 presents an effective scheduling algorithm to partition a CDFG into an equivalent state transition graph. In Section 4, we report the experimental results and the comparison to other approaches. Finally, concluding remarks are made in Section 5.

2. Optimization of execution paths

The input is a behavior-level description written in the VHDL hardware description language. A VHDL hardware description may contain several processes. In our control-dominated circuit synthesis system [15], each process is synthesized independently; i.e. each process results in a data path and controller with exactly one single-phase clock. For each process, the designer can specify the resource constraints such as the number and/or type of hardware modules to be used in the data path. The designer is responsible for the inter-process communication. Fig. 1 shows the subset of VHDL statements (within a process) acceptable by our scheduler. Note that all *wait* statements (which wait on clock) in a single process must use the same edge triggering of the same clock. Either a rising-edge clock or a falling-edge clock can be the active edge.

The behavior-level hardware description (within a process) is first compiled into a CDFG representa-

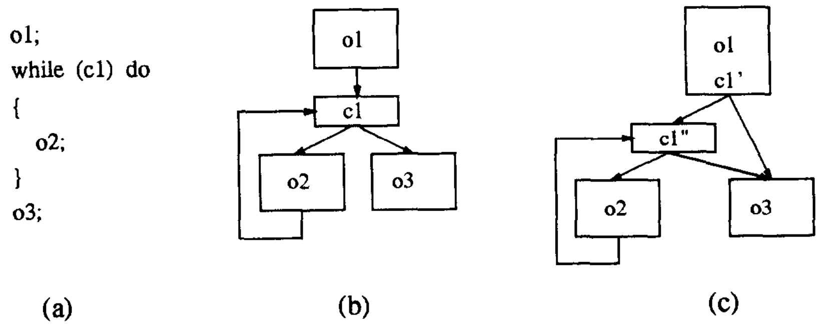

```
o1;
while (c1) do
{
  o2;
}
o3;
```

Fig. 2. (a) A program consisting of a while-loop. (b) The CDFG using the pre-test construction. (c) The CDFG using the new loop construction.

tion which consists of a set of basic blocks linked by flow-of-controls. A flow-of-control can be forward or backward. A forward edge represents an execution order from a basic block to a successor block, while a backward edge represents a loop-construction. The basic block without a predecessor is called the *entry block*. Each VHDL process has only one entry block.

Due to the presence of loops and branches, there are several execution paths on a CDFG representation. This section discusses the optimization of execution paths through control constructs such as loops and branches. Section 2.1 describes a loop construction to overcome the limitation of pre-test construction on operation chaining. Section 2.2 presents a preprocessing algorithm to remove the ineffective operations in each execution path.

2.1 A loop construction for operation chaining

The loop structure supported in VHDL is the while-loop. Fig. 2(a) describes a program which consists of a while-loop. Fig. 2(b) shows the corresponding CDFG using the pre-test construction. To preserve the semantics of a program, operations $o1$ and $c1$ cannot be scheduled in the same state if operation $o1$ is not a loop invariant. This property limits the possible chaining of operations $o1 \rightarrow c1 \rightarrow o3$ into a single state.

We use a different loop construction to overcome the limitation of pre-test construction. Fig. 2(c) shows the corresponding CDFG using the new loop construction. By duplicating loop comparison operation $c1$, the while-loop is translated into an *if* construction (i.e. 'loop-test' operation) whose true part is the original loop. Since operation $c1$ is duplicated, let's distinguish the two copies of $c1$ as $c1'$ ($if-c1$) and $c1''$ ($while$-$c1$). If the control flows through the true part of 'loop-test' operation $c1'$, the loop is executed at least once. Otherwise, the loop will not be executed. The advantages of the new loop construction on operation chaining are discussed as follows:

- **For the path in which the loop-test condition is false**. In Fig. 2(c), the path is $o1 \rightarrow c1' \rightarrow o3$. If these three operations can be chained together, we can schedule them in the same state. Thus, this is the best we can do for this path.
- **For the path in which the loop-test condition is false**. In Fig. 2(c), the path is $o1 \rightarrow c1' \rightarrow c1'' \rightarrow o2 \rightarrow \ldots \rightarrow c1'' \rightarrow o3$. If operations $c1''$ and $o2$ can be chained together, we can schedule them in the same state. The advantage of the pre-test construction is preserved.

The idea of the new loop construction is to bypass the loop in case the entry condition is not satisfied. In control-dominated applications, there are usually many *wait* statements. The *wait* statements are used to wait on clock and/or conditions. In our control-dominated circuit synthesis system, a *wait* statement which only waits on conditions is translated into an equivalent while-loop. The improvement of the new loop construction may be significant if the waiting conditions seldom occur.

The new loop construction, however, may unnecessarily duplicate some operations. For example, in Fig. 2(c), if operation $o1$ is a loop invariant and operations $o1$, $c1''$ and $o2$ can be chained together, then we can schedule them into the same state. It is not necessary to do comparison operation $c1$ twice (i.e. $c1'$ and $c1''$) in the same state. Thus, in this case, we can remove operation $c1'$ and then reduce the while-loop to a pre-test construction.

In our control-dominated circuit synthesis system, a behavior description is first compiled into a CDFG using the new loop construction to represent each loop. In Section 3.4, we will present an algorithm to determine the right loop construction of each loop during the scheduling process.

2.2 Preprocessing

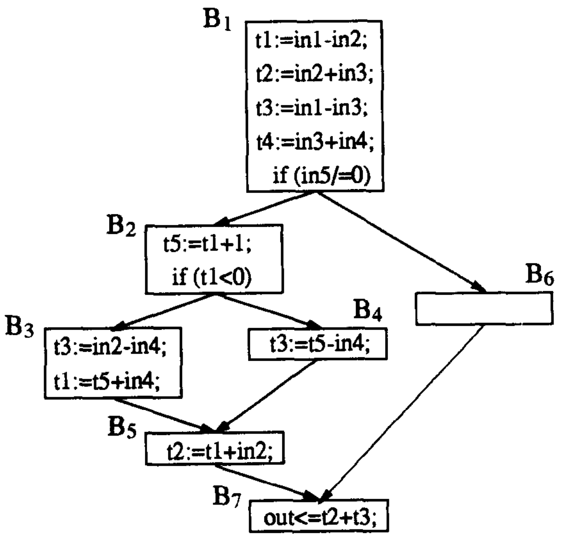
An *if* construct spreads a *true* part and a *false* part. For an *if* block B_{if}, there are two immediate successors: (1) the *true* block B_{true}, which will be immediately executed after B_{if} if the comparison is true; (2) the *false* block B_{false}, which will be immediately executed after B_{if} if the comparison is false. A *case* construct can be translated into nested ifs and treated accordingly.

Because the delay time of an operation in a control-dominated application is usually short, it is possible to chain several conditional branches into a single state. The mutually exclusive operations with the same functionality can share the same resource. For example, an addition operation in B_{true} and another addition operation in B_{false} can share the same adder, even though they are scheduled in the same state. In order to improve resource sharing, it is desired to move an operation from *if* block to branch parts. Let $d(o_i)$ be the variable defined by operation o_i and $in[B]$ be the set of variables which are live at the entry of block B. The following Lemmas state the conditions of propagating an operation from an *if* block to its adjacent blocks B_{true} and B_{false}.

Lemma 1. *If an operation o_i has no dependency successor in B_{if}, it can be duplicated and moved to both blocks B_{true} and B_{false}.*

Lemma 2. *An operation o_i in B_{if} can be moved downward to $B_{true}(B_{false})$, if*
(1) *it has no dependency successor in B_{if}; and*
(2) $d(o_i) \notin in[B_{false}]$ $(d(o_i) \notin in[B_{true}])$.

Lemma 2 can be applied to relocate the operations that are ineffective in some of the containing paths of a conditional branch. For example, in Fig. 3(a), operation $t1 := in1 - in2$ is ineffective for the path $B_1 \Rightarrow B_6 \Rightarrow B_7$ because $t1 \notin in[B_6]$. Therefore, according to Lemma 2, we can move this operation from block B_1 to block B_2. Note that even if an operation is not ineffective for both the paths through B_{true} and B_{false} (i.e. $d(o_i) \in in[B_{true}]$ and $d(o_i) \in in[B_{false}]$), it may still be ineffective for some of the subpaths. In order to discover all indirect-ineffective



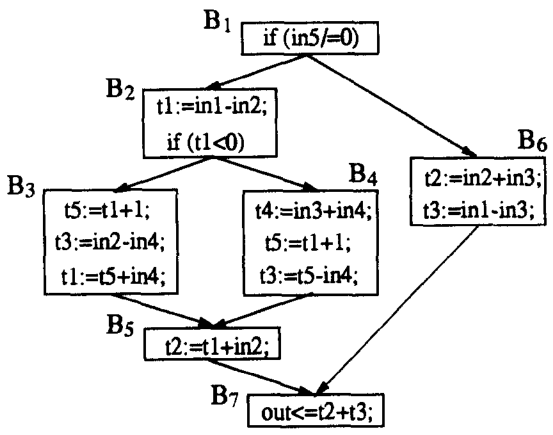


Fig. 3. Example 1. (a) Original CDFG. (b) The CDFG after preprocessing is carried out.

```
Procedure Preprocessing()
 Begin
  For each basic block B do (in the sequence of topological ordering)
   if block B is an if block then
    For each operation $o_i$ in $B$ from last to first do
     Begin
      if $o_i$ can be propagated then /*apply Lemma 1*/
       Begin
        For each successor block $B_k$ of $B$ do
          if ($d(o_i) \in in[B_k]$)  /*apply Lemma 2*/
            then copy $o_i$ to the head of $B_k$;
        Remove operation $o_i$ from $B$;
       End;
     End;
 End;
```

Fig. 4. The preprocessing algorithm.

operations, we apply Lemma 1 to propagate each operation as downward as possible.

Fig. 4 shows the preprocessing algorithm which removes the ineffective operations in each execution path on a CDFG representation. The basic blocks are processed in the sequence of their topological ordering (ignore backward edges). The operations in a block are processed from the last to the first (ignore the comparison operation). When an operation is propagated, we place it at the head of the target block. By repetitively applying operation movement/duplication, the algorithm optimizes each execution path induced by conditional branches on a CDFG representation.

Example. Let us use the CDFG in Fig. 3(a) as an example to illustrate the preprocessing algorithm. The blocks are processed according to the following sequence B_1, B_2, B_3, B_4, B_5, B_6 and B_7. By applying the preprocessing algorithm, we can remove the ineffective operations in each execution path.

The algorithm starts from block B_1. Because block B_1 is an *if* block, we try to propagate the operations to its successor blocks. First, operation $t4 := in3 + in4$ is moved to block B_2. Operations $t3 := in1 - in2$ and $t2 := in2 + in3$ are moved to block B_6. Then, operation $t1 := in1 - in2$ is moved to block B_2.

Next, block B_2 is processed. At this point, block B_2 contains operations $t1 := in1 - in2$, $t4 := in3 + in4$, and $t5 := t1 + 1$. Operation $t5 := t1 + 1$ is duplicated and moved to blocks B_3 and B_4. Then, operation $t4 := in3 + in4$ is moved to block B_4. Operation $t1 := in1 - in2$ cannot be propagated because variable $t1$ is used by the comparison operation $if(t1 < 0)$.

Similarly, we examine blocks B_3, B_4, B_5, B_6 and B_7, and find none of them is an *if* block. The final result is shown in Fig. 3(b).

The CDFG after preprocessing is carried out has the following properties:

- Each block B_{if} contains only the dependency predecessors of branch operation o_{if}. Any operation which does not produce a value for the decision of o_{if} is propagated to the successor blocks. Besides, the ineffective operations in each execution path are removed.
- The successor blocks induced by a conditional branch can share the same resources because those blocks are mutually exclusive. Therefore, through preprocessing, we can improve resource sharing.
- Assume an operation o_i is originally in *if* block B_{if}. After preprocessing is carried out, operation o_i may be duplicated and moved to the successor blocks of B_{if}. During the scheduling phase, we say an operation is *ready* if its control and data dependencies have been satisfied. Note that all the copies of operation o_i will be ready before we schedule the conditional branch o_{if} in block B_{if}. Thus, those copies may be moved upward and unified during the scheduling of block B_{if}.

3. The scheduling algorithm

In our control-dominated circuit synthesis system, a behavior description is first compiled into a CDFG using the new loop construction to represent each loop. Then, preprocessing is performed to relocate ineffective operations in each execution path and improve resource sharing. *With the output of preprocessing, we propose an effective scheduling algorithm to partition a CDFG into an equivalent state transition graph.*

The proposed scheduling algorithm has the following features:

- It is capable of moving operations across basic blocks;
- It unifies the identical copies of an operation into a single state;
- It carries out operation chaining for the conditional branches;
- It allows resource sharing among mutually exclusive operations;
- It duplicates an operation into different states so that each execution path can be scheduled as fast as possible; and
- It determines the right loop construction for each loop.

3.1. Basic ideas

A pseudo-code of the proposed scheduling algorithm is presented in Fig. 5. The algorithm starts from the entry block B_{entry} and processes basic blocks in the sequence of their *topological ordering* (ignore backward edges). Procedure *Schedule_a_Block* is invoked iteratively to schedule each basic block. Whenever the scheduling of a block is done, chaining information is passed to its successor blocks through *forward control flows*. Consequently, we can chain several basic blocks linked by control constructs into a single state. Because a state may consists of several control constructs, we also use the CDFG structure to represent each state.

```
Procedure Scheduling_Algorithm()
Begin
  For each basic block $B$ do (in the sequence of topological ordering)
    Begin
      call procedure Schedule_a_Block($B$) (in Fig. 6);
      For each forward successor block $B_k$ of $B$ do
        Begin
          $S(B_k) = S(B_k) \cup S(B)$;
          $r(B_k) = r(B_k) \cap r(B)$;
        End;
    End;
End.
```

Fig. 5. The scheduling algorithm.

The idea behind the proposed scheduling algorithm is to schedule as many operations as possible, which may have control or data dependencies, into one state. Two or more dependent operations in an execution path can be chained into a single state as long as their total propagation delay is less than the clock cycle time. (Note that the total propagation delay may be less than the lumped execution time of chained operations. For example, some adder output bits are available early with bit-level timing. Hence, in this case, we can consider bit-level timing when determining the possibility of chaining operations in the same clock cycle [16].) As a result, all execution paths can be scheduled as fast as possible. To achieve the goal, the algorithm stresses the following two concerns:

(1) **It chains consecutive control-dependent operations**. Operations in consecutive basic blocks can be scheduled into the same state as long as their total propagation delay is less than the clock cycle time. Consequently, we can schedule several basic blocks linked by control constructs into a single state. For each basic block B, let $S(B)$ represents the set of states into which the ready operations in block B must be scheduled. Initially, $S(B_{entry}) = \{S0\}$, where S0 is the starting state of the finite state machine, and $S(B) = \emptyset$ for other blocks. The set $S(B)$ for a block B will be updated during scheduling. After a block B is processed, the current $S(B)$ must be passed to its forward successor blocks for possible chaining. As a result, the operations in consecutive blocks can be chained together. In order to

schedule each path as fast as possible, sometimes an operation has to be duplicated and placed into different states. For example, the chaining information of a joint block B_{joint} comes from its two predecessors; i.e. blocks B_{true} and B_{false}. After blocks B_{true} and B_{false} are processed, set $S(B_{joint})$ is updated to the *union* of the two predecessors $S(B_{true})$ and $S(B_{false})$.

(2) **It allows resource sharing among conditional branches**. Note that the operations in a state only execute under certain combinations of conditional values. The mutually exclusive operations with the same functionality can share the same resource. For example, if two addition operations lie in separate branches of an *if* or *case* statement, they are not on the same path and can share the same adder. Let R denote the set of given resources and $r(B)$ denote the currently available resources for each block B during scheduling. Before carrying out scheduling, $r(B)$ is initialized to R for every block B. The set of available resources $r(B)$ for a block B will be updated during scheduling. After a block B is processed, $r(B)$ is passed to constraint its forward successor blocks. As a result, the operations in an execution path will not use the same resource more than one times within a state. Furthermore, resource sharing among mutually exclusive operations is allowed because there is no control flow between any two of them. Note that the available resources of a joint block B_{joint} are constrainted by its two predecessors; i.e. blocks B_{true} and B_{false}. After blocks B_{true} and B_{false} are processed, set $r(B_{joint})$ is updated to the *intersection* of the two predecessors $r(B_{true})$ and $r(B_{false})$.

```
Procedure Schedule_a_Block($B$)
Begin
  if block $B$ is a loop header of the new loop construction then
    Begin
      create a new state $s'$;
      let $S(B) = \{s'\}$ and $r(B) = R$;
    End;
  Repeat
    Find a highest priority ready operation $o_i$ that can be scheduled into each state $s \in S(B)$;
    if found then
      Begin
        if $o_i$ is a "loop-test" operation then
          call Procedure Loop_Transformation($B, o_i$) (in Fig. 10);
        else
          Begin
            schedule $o_i$ into each state $s \in S(B)$;
            put ready operations into the ready queue;
            update the available resources $r(B)$;
          End;
        if $o_i$ is a control operation then return;
      End;
    else   /* not found */
      Begin
        create a new state $s'$;
        let $S(B) = \{s'\}$ and $r(B) = R$;
      End;
  Until all "must" operations of block $B$ are scheduled;
End.
```

Fig. 6. The procedure *Schedule_a_Block*.

3.2. Scheduling a block

When the scheduler moves to a block B, the procedure *Schedule_a_Block*(B) is invoked to schedule the block. The procedure *Schedule_a_Block* is described in Fig. 6. The details of the procedure are discussed below.

● **Operation Relocation.** In the output of preprocessing, every operation is called a 'must' operation for the block it belongs to because we cannot schedule the operation in a later block. Therefore, during the scheduling of a block, the execution probability of a 'must' operation in the block is 1. For example, in Fig. 3(b), operation $if(in5/=0)$ is a 'must' operation for block B_1. The operation that can be moved *upward* to a block is called a 'may' operation for the block. During the scheduling of a block, the execution probability of a 'may' operation (in a path starting from the block) can be computed by using branch probabilities. Let us use the CDFG in Fig. 3(b) as an example. During the scheduling of block B_1, the execution probabilities of operations $if(in5/=0)$, $t1 := in1 - in2$ and $t4 := in3 + in4$ are 1, 0.5 and 0.25, respectively. Note that the 'must' operations in a block are the operations that must be accommodated in the block. Meanwhile, we try to schedule as many 'may' operations into the block as long as the number of states does not increase [17]. The scheduling of a block is finished when all its 'must' operations are scheduled.

● **Operation Unification.** After preprocessing is carried out, an operation may be duplicated and moved to different blocks. The duplicated copies must be unified when they are moved upward and scheduled into the same block. For a unified operation, its execution probability is the lumped sum of the probabilities of all the unified copies. Thus, as more duplicated copies are unified, the operation will have a higher execution probability. For example, in Fig. 3(b), operation $t5 := t1 + 1$ is duplicated to blocks B_3 and B_4. After operation $t1 := in1 - in2$ in block B_2 is scheduled, the two copies of operation $t5 := t1 + 1$ are ready at the same time. Then, during the scheduling of block B_2, the two copies can be moved upward and unified and its execution probability is 1.

● **Operation Chaining.** Two or more dependent operations in an execution path can be chained into one state, if the total propagation delay based on their dependencies is less than the clock cycle time. (Note that the total propagation delay may be less than the lumped execution time of chained operations. For example, some adder output bits are available early with bit-level timing. Hence, in this case, we can consider bit-level timing when carrying out the chaining operations.) In order to execute each path as fast as possible, we have to schedule as many operations, which may have control or data dependencies, into one state. To achieve the goal, sometimes an operation has to be duplicated and placed into multiple states. As mentioned earlier, during the scheduling of a block B, the set $S(B)$ includes the current states into which the ready operations must be scheduled. In our algorithm, we will try to find a ready operation o_i that can be scheduled in each state in set $S(B)$. Then, operation o_i is duplicated and placed into each state in set $S(B)$ to speedup all execution paths in which operation o_i is effective. If no such operation exists, a new state s' is created and set $S(B)$ becomes $\{s'\}$.

● **Priority Function.** When an operation is ready, it is put in the ready queue. During the scheduling of a block B, the ready operations come not only from block B (i.e. 'must' operations) but also from the successor blocks (i.e. 'may' operations). They will compete for the resources. In event resource conflict occurs, the following rules are applied to resolve the problem:

(1st) An operation which has a higher execution probability has a higher priority.

(2nd) If a computation operation and a branch operation have the same execution probabilities, the computation operation has a higher priority than the branch operation. The reason is explained as follows.

If we schedule the branch operation first, we must propagate the computation operation to all the successor blocks. This propagation does not improve resource sharing. Furthermore, if these propagated copies are not scheduled in the same state, we need to pay extra control costs to supervise the execution of those copies scheduled in different states. Thus, we prefer to unify all the copies of the computation operation. Note that such unification also reduces the time complexity of the scheduling algorithm.

(3rd) If two computation operations have the same execution probabilities, the one on the critical path has a higher priority.

● **Loop Transformation.** The new loop construction presented in Section 2.1 translates a loop into an *if* construction which starts with a 'loop-test' operation. While we are scheduling a 'loop-test' operation o_i, procedure *Loop_Transformation* (B, o_i) is invoked to determine the right loop construction of this loop. Note that the loop header of the new loop construction must start a new state. The detail of loop transformation is discussed in Section 3.4.

3.3. A scheduling example

Let us use the CDFG in Fig. 3(a) as an example to illustrate our scheduling algorithm. The output of preprocessing on this example is shown in Fig. 3(b).

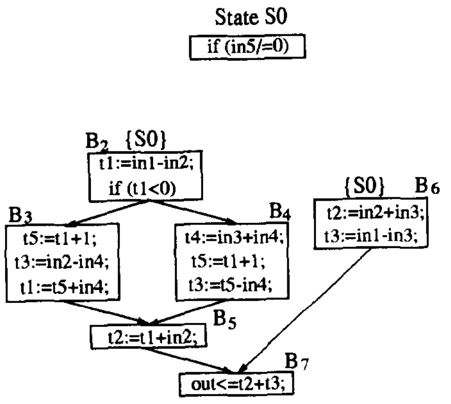
We denote the set of available resources as a 2-element vector. In the 2-element vector, the first element denotes the number of adders, and the second element the number of subtracters. Assume we are given 2 adders and 1 subtracter. Then, for each block B, the set of resources $r(B)$ is initialized to $\langle 2, 1\rangle$. Initially, $S(B_1)$ is $\{S0\}$, where $S0$ is the starting state of the finite state machine, and $S(B) = \phi$ for the blocks except B_1. With the output of preprocessing, procedure *Schedule_a_Block* is invoked iteratively to process the basic blocks according to the following sequence B_1, B_2, B_3, B_4, B_5, B_6 and B_7.

The scheduling algorithm starts from block B_1. Let us examine how procedure *Schedule_a_Block* processes block B_1. Operation $if(in5 /= 0)$ is scheduled and then the scheduling of block B_1 is finished. The information $S(B_1)$ and $r(B_1)$ are passed to blocks B_2 and B_6. As a result, sets $S(B_2)$ and $S(B_6)$ become $\{S0\}$, and sets $r(B_2)$ and $r(B_6)$ become $\langle 2, 1\rangle$. The result (except $r(B)$) is shown in Fig. 7(a).

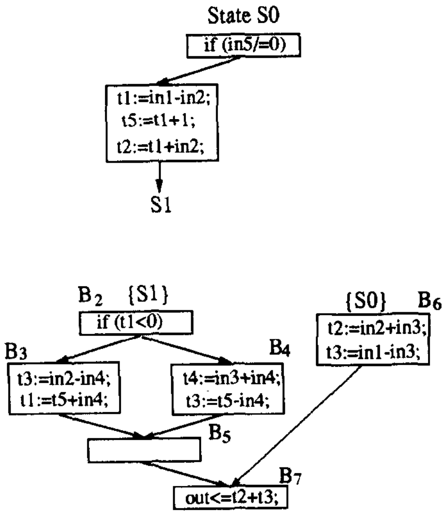
Now, *Schedule_a_Block* moves to block B_2. Operation $t1 := in1 - in2$ is scheduled. Hence, $r(B_2)$ is updated to $\langle 2, 0\rangle$. Next, the two copies of operation $t5 := t1 + 1$ are unified and scheduled in this state. Set $r(B_2)$ is updated to $\langle 1, 0\rangle$. At this point, operation $t2 := t1 + in2$ in block B_5 is ready. It is moved upward and placed in block B_2. As a result, $r(B_2)$ becomes $\langle 0, 0\rangle$. Note that operation $if(t1 < 0)$ cannot be scheduled in state $S0$, since the input variable (i.e. variable $t1$) for the finite state machine must be ready at the beginning of a state. Now, because no resource is available, a new state $S1$ is created. $S(B_2)$ becomes $\{S1\}$ and $r(B_2)$ becomes $\langle 2, 1\rangle$. The result (except $r(B)$) is shown in Fig. 7(b). Next, operation $if(t1 < 0)$ is scheduled into state $S1$. At this point, *Schedule_a_Block* has finished processing block B_2. The information $S(B_2)$ and $r(B_2)$ are passed to blocks B_3 and B_4. Consequently, sets $S(B_3)$ and $S(B_4)$ become $\{S1\}$, and sets $r(B_3)$ and $r(B_4)$ become $\langle 2, 1\rangle$. The result (except $r(B)$) is shown in Fig. 7(c).

Similarly, our algorithm goes through blocks B_3, B_4 and B_5. Operations in these blocks are scheduled into state $S1$. After the scheduling of block B_5 is finished, the information $S(B_5)$ and $r(B_5)$ are passed

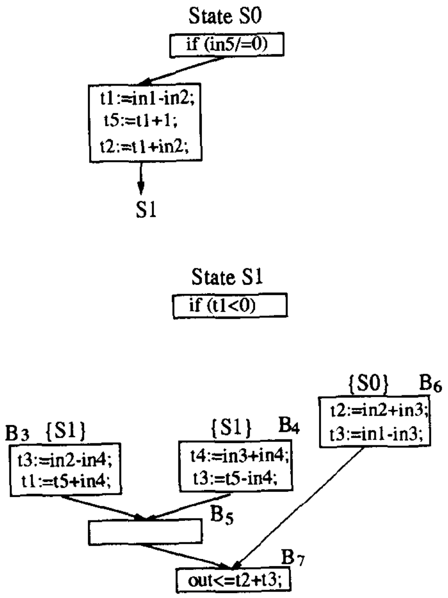
Fig. 7. Snapshots of our scheduling algorithm on Example 1.


State S0
if (in5/=0)
B2 {S0}
t1:=in1-in2;
if (t1<0)
B3
t5:=t1+1;
t3:=in2-in4;
t1:=t5+in4;
B4
t4:=in3+in4;
t5:=t1+1;
t3:=t5-in4;
B5
t2:=t1+in2;
{S0} B6
t2:=in2+in3;
t3:=in1-in3;
B7
out<=t2+t3;

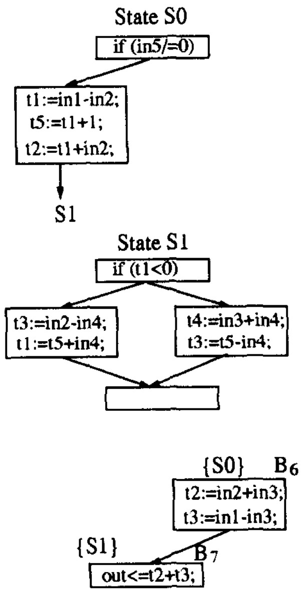
(a)


State S0
if (in5/=0)
t1:=in1-in2;
t5:=t1+1;
t2:=t1+in2;
S1
B2 {S1}
if (t1<0)
B3
t3:=in2-in4;
t1:=t5+in4;
B4
t4:=in3+in4;
t3:=t5-in4;
B5
{S0} B6
t2:=in2+in3;
t3:=in1-in3;
B7
out<=t2+t3;

(b)


State S0
if (in5/=0)
t1:=in1-in2;
t5:=t1+1;
t2:=t1+in2;
S1
State S1
if (t1<0)
{S0} B6
t2:=in2+in3;
t3:=in1-in3;
B3 {S1}
t3:=in2-in4;
t1:=t5+in4;
{S1} B4
t4:=in3+in4;
t3:=t5-in4;
B5
B7
out<=t2+t3;

(c)


State S0
if (in5/=0)
t1:=in1-in2;
t5:=t1+1;
t2:=t1+in2;
S1
State S1
if (t1<0)
t3:=in2-in4;
t1:=t5+in4;
t4:=in3+in4;
t3:=t5-in4;
{S0} B6
t2:=in2+in3;
t3:=in1-in3;
{S1}
B7
out<=t2+t3;

(d)

to block B_7. Thus, $S(B_7)$ becomes $\{S1\}$ and $r(B_7)$ becomes $\langle 1, 0\rangle$. The result (except $r(B)$) is shown in Fig. 7(d).

Next, procedure *Schedule_a_Block* processes block B_6. Note that $S(B_6)$ is $\{S0\}$. Hence, operations in this block are scheduled into state $S0$. After procedure *Schedule_a_Block* completes block B_6, $r(B_6)$ becomes $\langle 1, 0\rangle$. The information $S(B_6)$ and $r(B_6)$ are passed to block B_7. As a result, $S(B_7)$ becomes $\{S0, S1\}$ and $r(B_7)$ becomes $\langle 1, 0\rangle$. The result (except $r(B)$) is shown in Fig. 8(a).

Finally, block B_7 is processed. Since $S(B_7)$ is $\{S0, S1\}$, operation *out* $<= t2 + t3$ is duplicated and scheduled into states $S0$ and $S1$. The result (except $r(B)$) is shown in Fig. 8(b).

The state transition graph obtained by our scheduling algorithm is shown in Fig. 9. Note that our optimization goal is to schedule every execution path as fast as possible. For instance, in this example, there are two states used to schedule the path $B_1 \Rightarrow B_2 \Rightarrow B_3 \Rightarrow B_5 \Rightarrow B_7$. Under the constraint with 2 adders and 1 subtracters, the result of this example is optimum.

3.4. Loop transformation

While we are scheduling a 'loop-test' operation o_{test} in a block B, the following possible conditions may occur:

- **Condition 1**. Suppose an operation o_i in a state $s \in S(B)$ is not a loop invariant. The new loop construction is chosen for the loop. Note that the

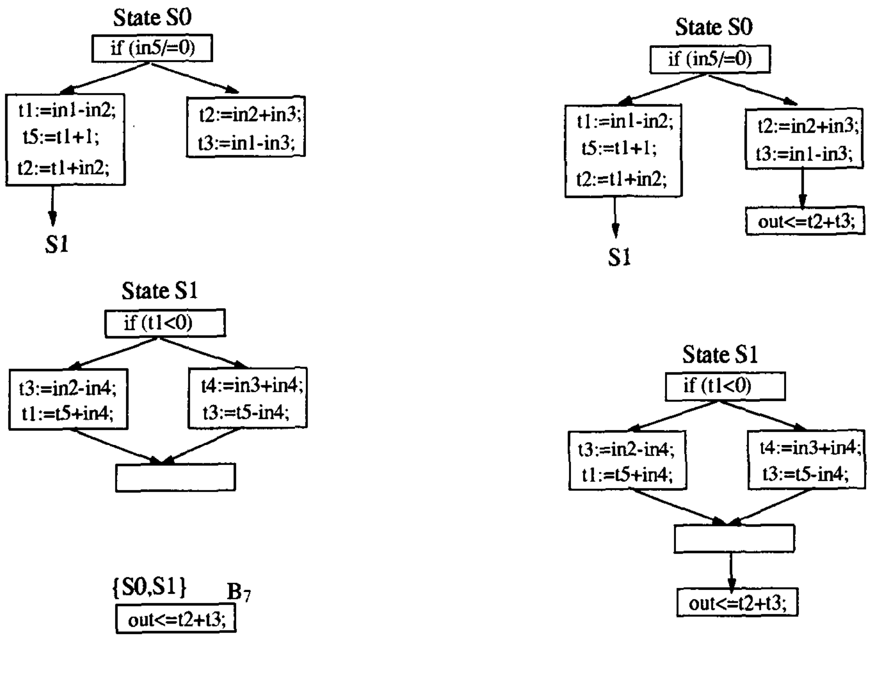


Fig. 8. Snapshots of our scheduling algorithm on Example 1. continued.

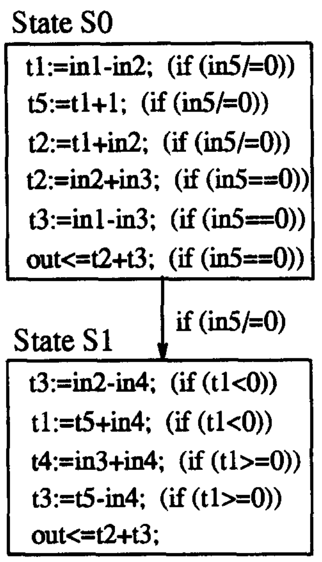

State S0
t1:=in1-in2; (if (in5/=0))
t5:=t1+1; (if (in5/=0))
t2:=t1+in2; (if (in5/=0))
t2:=in2+in3; (if (in5==0))
t3:=in1-in3; (if (in5==0))
out<=t2+t3; (if (in5==0))
if (in5/=0)
State S1
t3:=in2-in4; (if (t1<0))
t1:=t5+in4; (if (t1<0))
t4:=in3+in4; (if (t1>=0))
t3:=t5-in4; (if (t1>=0))
out<=t2+t3;

Fig. 9. The state transition graph of Example 1 obtained by our scheduling algorithm.

pre-test construction is not suitable for the loop because it will limit possible chaining for the path in which the 'loop-test' condition is false. Thus the 'loop-test' operation o_{test} is scheduled into each state $s \in S(B)$ so that each execution path will be scheduled as fast as possible. By duplicating the loop comparison operation, we can schedule each execution path as fast as possible.

- **Condition 2.** In other cases, the pre-test construction is chosen for the loop. The 'loop-test' operation o_{test} will be ignored, and hence the loop will be reduced to a pre-test construction. The pre-test

```
Procedure Loop_Transformation($B$,$o_{test}$)
Begin
  if ( Condition 2 occurs )
    then
      ignore $o_{test}$ and reduce this loop into the pre-test construction;
    else
      schedule $o_{test}$ into each state $s \in S(B)$;
End.
```

Fig. 10. The procedure *Loop_Transformation.*

```
o1: n= in1+1;
o2: sum1= in2+in3;
o3: sum2= in4+1;
c1: while (n<=0) do
    {
o4:   m= in5+in6;
o5:   sum1= sum1+n;
c2:   while (m<=0) do
      {
o6:     sum2=sum2+m;
o7:     m=m+1;
      }
o8:   n= n+1;
    }
o9: out<=sum1+sum2;
```

Fig. 11. Example 2 (An example used to illustrate the idea of loop transformation).

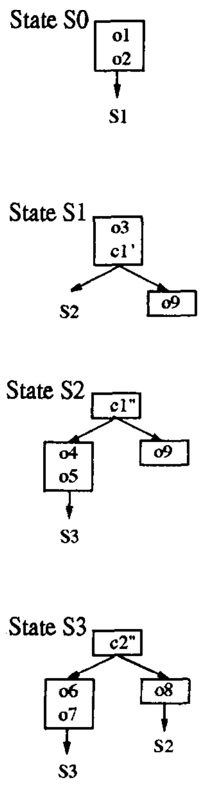
construction can be scheduled so that each path is executed as fast as possible without duplicating the loop comparison operation.

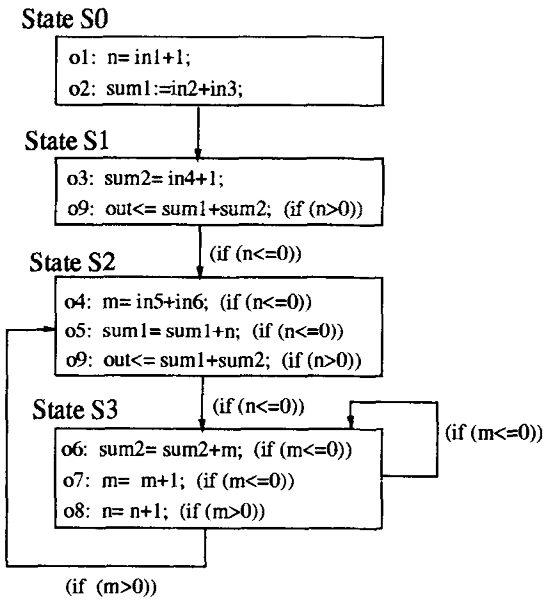
The above discussion concludes that we need to transform a loop into the pre-test construction when Condition 2 occurs. The transformation is performed by ignoring the 'loop-test' operation. The idea of loop transformation has been implemented in the procedure *Loop_Transformation* which is described in Fig. 10.

Example. Let us use the program in Fig. 11 as an example to illustrate the idea of loop transformation. There is a two-level nested loops in the program. Suppose we are given two ALUs as the resource constraint. By applying the proposed approach, we can determine the right loop construction of each loop during the scheduling process. Fig. 12 shows the snapshots that our scheduling algorithm goes through.

At the beginning, the program is transformed into a CDFG using the new loop construction. The CDFG is displayed on Fig. 12(a). Then, the preprocessing procedure is performed. No operation is relocated during preprocessing. Next, our scheduling algorithm iteratively invokes procedure *Schedule_a_Block* to schedule the basic blocks according to the following sequence B_1, B_2, B_3, B_4, B_5, B_6 and B_7.

Let us see how *Schedule_a_Block* schedules block B_1. Initially, set $S(B_1)$ is $\{S0\}$. Operations $o1$





(a) (b)

Fig. 12. Snapshots of our scheduling algorithm on Example 2.

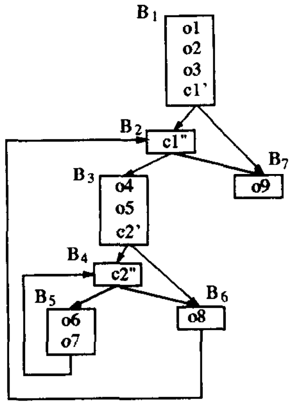
and $o2$ are scheduled into state $S0$. Then, since no resource is available, a new state $S1$ is created and set $S(B_1)$ is updated to $\{S1\}$. The result (except $r(B)$) is shown in Fig. 12(b). Operation $o3$ is scheduled into state $S1$. Next, procedure *Loop_Transformation*$(B_1, c1')$ is invoked to process 'loop-test' operation $c1'$. Note that there is an operation $o3$ scheduled in state $S1$ already. Because operation $o3$ is not a loop invariant, the new loop construction is used to represent the loop. Thus, operation $c1'$ is scheduled into state $S1$. The result (except $r(B)$) is displayed on Fig. 12(c).

After block B_1 is processed, we pass the information $S(B_1)$ to successor blocks B_2 and B_7. As a result, sets $S(B_2)$ and $S(B_7)$ become $\{S1\}$.

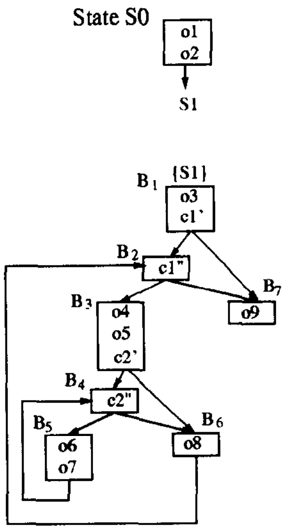
Then, we process block B_2. Since block B_2 is a loop header with the new loop construction, the block must start a new state. A new state $S2$ is created and set $S(B_2)$ is updated to $\{S2\}$. Operation $c1''$ is scheduled into state $S2$.

After block B_2 is processed, the information $S(B_2)$ is passed to successor blocks B_3 and B_7.

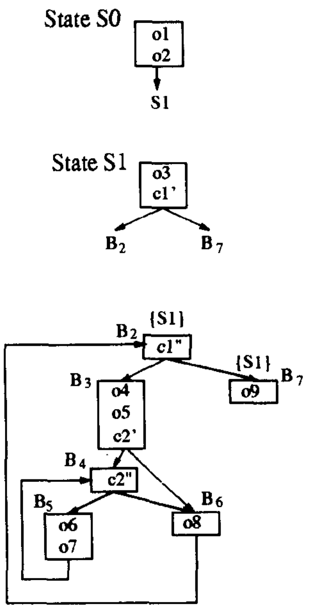
Fig. 13. Snapshots of our scheduling algorithm on Example 2, continued.


B1
o1
o2
o3
c1'
B2
c1"
B7
o9
B3
o4
o5
c2'
B4
c2"
B5
o6
o7
B6
o8

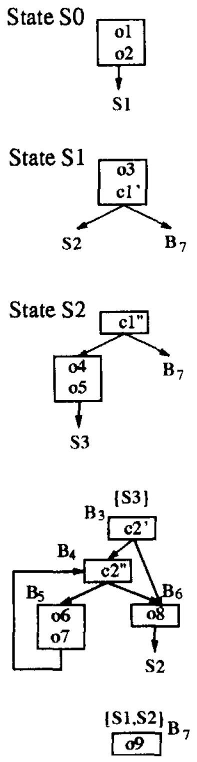
(a)


State S0
o1
o2
S1
B1 {S1}
o3
c1'
B2
c1"
B7
o9
B3
o4
o5
c2'
B4
c2"
B5
o6
o7
B6
o8

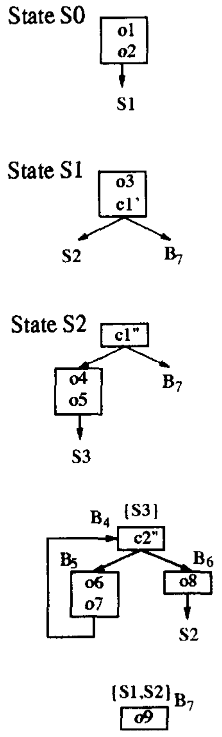
(b)


State S0
o1
o2
S1
State S1
o3
c1'
B2
B7
B2 {S1}
c1"
{S1} B7
o9
B3
o4
o5
c2'
B4
c2"
B5
o6
o7
B6
o8

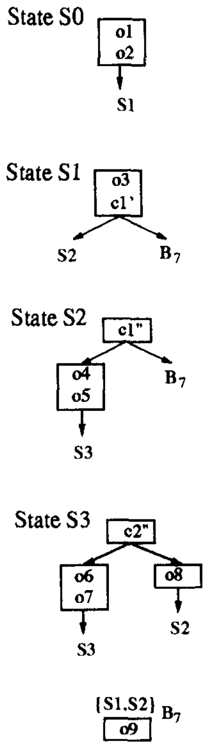
(c)


State S0
o1
o2
S1
State S1
o3
c1'
S2
B7
State S2
c1"
o4
o5
B7
S3
B3 {S3}
c2'
B4
c2"
B5
o6
o7
B6
o8
S2
{S1,S2} B7
o9

(d)


State S0
o1
o2
S1
State S1
o3
c1'
S2
B7
State S2
c1"
o4
o5
B7
S3
B4 {S3}
c2"
B5
o6
o7
B6
o8
S2
{S1,S2} B7
o9

(e)


State S0
o1
o2
S1
State S1
o3
c1'
S2
B7
State S2
c1"
o4
o5
B7
S3
State S3
c2"
o6
o7
o8
S3
S2
{S1,S2} B7
o9

(f)

Next, we process block B_3. Operations $o4$ and $o5$ are scheduled into state $S2$. Since no resource is available, a new state $S3$ is created and set $S(B_3)$ is updated to $\{S3\}$. The result (except $r(B)$) is shown in Fig. 12(d). Then, procedure *Loop_Transformation*(B_3, $c2'$) is invoked to process 'loop-test' operation $c2'$. Because there is no operation in state $S3$, we ignore operation $c2'$ and then reduce the loop into the pre-test construction. The result (except $r(B)$) is shown in Fig. 12(e).

Similarly, our algorithm goes through blocks B_4, B_5 and B_6. Operations $c2''$, $o6$, $o7$ and $o8$ are scheduled into state $S3$. The result (except $r(B)$) is shown in Fig. 12(f).

Finally, block B_7 is processed. Note that set $S(B_7)$ is $\{S1, S2\}$. Thus, operation $o9$ is duplicated to states $S1$ and $S2$. The result (except $r(B)$) is given in Fig. 13(a).

The state transition graph obtained by our scheduling algorithm is described in Fig. 13(b). The outer loop is transformed into the new loop construction, and the inner loop is transformed into the pre-test construction. Each execution path is scheduled as fast as possible without unnecessary duplication. Furthermore, we only use four states to schedule the program. Because the hardware parallelism is 2, the size of the finite state machine is minimal.

3.5. Time complexity

Let n be the number of operations in the CDFG after preprocessing is carried out. Since the length of a ready queue is limited by $O(n)$, the search for a candidate operation to be scheduled is $O(n)$. Moreover, for a 'loop-test' operation, the decision of loop transformation has the time complexity of $O(n)$. Therefore, the time complexity for scheduling an operation is $O(sn^2)$, where s the total number of states. It is easy to see that $s \leq n$. Therefore, the time complexity of scheduling an operation is $O(n^3)$. Hence, the time complexity of our scheduling algorithm is $O(n^4)$.

4. Experiments

We have implemented the proposed scheduling algorithm in a C program running on the SUN-Sparc workstation and have integrated it into a control-dominated circuit synthesis system [15]. The benchmarks from the open literature are used to test the effectiveness of the proposed scheduling algorithm. We make the following assumptions for synthesis:

- Since the inputs to the finite state machine are clocked, the input variables of the finite state machine must be ready at the beginning of a state.
- I/O operations cannot be moved across the associated 'wait' operation. Such dependency is inspected by the programmer and specified as a control dependency.
- A procedure call requires at least two states. The first state contains a jump (state transition) to the called procedure. The second state is the return state where the called procedure transfers control to the caller.
- The selected clock cycle time must be greater than the execution time for the maximum number of chained operations.

The first experiment uses a VHDL description of a telephone answering machine [18]. The second experiment is the benchmarks from the 1989 Workshop on High-Level Synthesis. All experimental results are verified by simulation.

4.1. Telephone answering machine

The characteristics of the answering machine example include real-time constraints, synchronization, hierarchy and nested-loops. The initial VHDL description contains 110 operations and 25 loops.

Table 1 tabulates the results, where DLS denotes the scheduler used in AMICAL system [18]. Given the constraints on the number of adders (*Adds*) and the maximum number of data-path operations that can be chained in one control step (*cn*), Table 1

Table 1
Results on telephone answering machine

Approach	Constraints		Results		CPU time (seconds)
	Adds	cn	States	Trans	
Ours	1	1	20	55	0.300
	2	2	19	54	0.310
	–	–	19	54	0.300
DLS	–	–	22	67	–

shows (i) the total number of states in the finite state machine (*states*) and (ii) the total number of transitions in the finite state machine (*Trans*). The CPU time (seconds) is the time the synthesis tool takes to complete the job on this benchmark.

Without any resource constraint, the DLS scheduler translates this example into 22 states and 67 transitions, while our approach translates it into 19 states and 54 transitions. With further analysis, we found that our approach has reached the lower bound of the number of control states. Since there are 18 control states because of the existence of the loop headers and 1 control state because of the control dependencies (the input variables of a finite state machine must be ready at the beginning of a state), the lower bound of the number of control states is 19.

Because there is no 'goto' statement in VHDL language, the designers have to use the structure 'loop...exit' to represent the unconditional branch in a VHDL program. The 'loop...exit' is treated as an unconditional branch. Since there are some 'loop...exit' structures in this example, we only need 19 states to schedule it even though the program contains 25 loops.

4.2. *High-level synthesis workshop benchmarks*

We applied the proposed scheduling algorithm to the benchmarks from 1989 Workshop on High-Level Synthesis, including COUNTER, GCD, PREFETCH, KALMAN, HUNT and TX8251. A brief introduction to these benchmarks, including the functionality, the number of operations and paths, is given in [14].

Table 2
Results on the benchmarks from 1989 high-level synthesis workshop

Design	Approach	Results					CPU time (seconds)
		States	Trans	Short	Long	cn	
COUNTER	Ours	1	1	1	1	1	0.040
	PBS	1	1	1	1	–	–
	DLS	2	5	–	–	–	–
GCD	Ours	2	4	1	2	1	0.020
	PBS	2	4	1	2	–	–
	DLS	2	4	–	–	–	–
	CALLAS	3	5	–	–	–	–
PREFETCH	Ours	2	4	1	2	1	0.020
	PBS	4	9	1	3	–	–
	DLS	3	4	–	–	–	–
KALMAN	Ours	17	38	1	10	4	0.450
	PBS	23	115	1	17	–	–
HUNT	Ours	6	14	1	5	2	0.080
	PBS	6	25	1	6	–	–
TX8251	Ours	16	51	1	12	1	0.330
	PBS	22	112	2	18	–	–

Most operations in these benchmarks are logic, decision making, I/O and data transfer operations. There are very few data dependencies between the operations in these benchmarks.

Without any resource constraint, Table 2 tabulates the synthesized results. Note that we also do not enforce the constraint on the maximum number of chained operations. Hence, the CDFG is partitioned into states due to control dependency (e.g. the input variable of the finite state machine must be ready at the beginning of a state) or program structure (e.g. loop, procedure call, and so on). For each benchmark, Table 2 shows the results produced by our approach and the published results. We compare our synthesized results with the path-based scheduling (PBS), the scheduler used in AMICAL system (DLS), and the scheduler used in CALLAS system (CALLAS). The first five columns present the synthesized results, including the total number of states (*States*), the number of transitions (*Trans*), the number of states in the shortest path (*Short*), the number of states in the longest path (*Long*), and the maximum number of chained data-path operations after the scheduling is finished (*cn*). The last column shows the time the synthesis tool takes to complete the scheduling on each benchmark.

The synthesized results show that our approach requires fewest states in the finite state machine and optimizes each path with fewest control states. With further analysis, we find that our scheduler achieves better results because it uses both operation reordering and the approach to schedule an operation into multiple states. It is important to note that some of the results given in PBS are for the descriptions written in V not VHDL. This might imply some discrepancies.

5. Conclusions

In this paper we described a new scheduling algorithm for automatic synthesis of the control blocks of control-dominated circuits. The main distinction of the proposed algorithm is that it partitions a CDFG into an equivalent state transition graph. It works on the CDFG to exploit operation relocation, chaining, and loop transformation to schedule each execution path as fast as possible. Benchmark data shows that this approach achieved better results over previous ones in terms of the speedup of the circuit and the number of states and transitions. Some of the results are proved to be optimum.

References

[1] D.D. Gajski (ed.), *Silicon Compilation* (Addison-Wesley, New York, 1988).

[2] M.C. McFarland, A.C. Parker and R. Camposano, Tutorial on high-level synthesis, in *Proc. 25th Design Automation Conf.* (June 1988) 330–336.

[3] G. Goossens, J. Rabaey, J. Vandewalle and H.D. Man, An efficient microcode compiler for application specific DSP processors, *IEEE Trans. Computer-Aided Design* (Sep. 1990) 925–937.

[4] M. Koster, M. Geiger and P. Duzy, ASIC design using the high-level synthesis system CALLAS: a case study, in *Proc. Int. Conf. on Computer Design* (Sep. 1990) 141–146.

[5] G. Saucier and J. Trilhe (eds.), *Synthesis for Control-Dominated Circuits: Selected Papers from the IFIP WG10.2 / WG10.5 Workshops*, Grenoble, France (April/Sep. 1992) (North-Holland, 1993).

[6] S. Davidson, D. Landskov, B.D. Shriver and P.W. Mallett, Some experiments in local microcode compaction for horizontal machines, *IEEE Trans. Comput* (July 1981) 460–477.

[7] J.A. Fisher, Trace scheduling: A technique for global microcode compaction, *IEEE Trans. Comput.* (June 1981) 478–490.

[8] A. Nicolau, Uniform parallelism exploitation in ordinary programs, *Proc. Int. Conf. on Parallel Processing* (Aug. 1985) 614–618.

[9] J. Lah and D.E. Atkins, Tree compaction of microprograms, *Proc. 16th Annual Microprogramming Workshop* (Oct. 1983) 23–33.

[10] J.L. Linn, SRDAG compaction – a generalization of trace scheduling to increase the use of global context information, in *Proc. 16th Annual Microprogramming Workshop* (Oct. 1983) 11–22.

[11] T. Makatani and K. Ebcioglu, Using a lookahead window in

a compaction-based parallelizing compiler, in *Proc. Int. Symp. on 23rd Microarchitecture*, (Nov. 1990) 57–68.

[12] B.M. Pangrle and D.D. Gajski, Design tools for intelligent silicon compilation, *IEEE Trans. Computer-Aided Design* (Nov. 1987) 1098–1112.

[13] P.G. Paulin and J.P. Knight, Force-directed scheduling for the behavioral synthesis of ASIC's, *IEEE Trans. Computer-Aided Design* (June 1989) 661–679.

[14] R. Camposano, Path-based scheduling for synthesis, *IEEE Trans. Computer-Aided Design*, (Jan. 1991) 85–93.

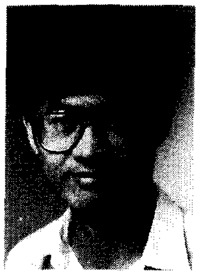
[15] Y.C. Hsu, MEBS user guide: A multiple-entry behavior synthesis system for digital system rapid prototyping, Technical Report, Department of Computer Science, University of California, Riverside, June 1994.

[16] Synopsys behavioral compiler user guide, Version 3.2a, Synopsys Inc., Oct. 1994.

[17] S.H. Huang, C.T. Hwang, Y.C. Hsu and Y.J. Oyang, A new approach to schedule operations across nested-ifs and nested-loops, in *Proc. 25th Int. Symp. on Microarchitecture*, (Dec. 1992) 268–271.

[18] K. Obrien, M. Rahmouni and A. Jerraya, A VHDL-based scheduling algorithm for control-flow dominated circuits, in *Proc. High-Level Synthesis Workshop* (Nov. 1992) 135–145.

Shih-Hsu Huang received the B.S. degree in computer science and information engineering from National Chiao Tung University, Hsinchu, Taiwan, in 1989, and the M.S. degree in computer science from National Tsing Hua University, Hsinchu, Taiwan, in 1991. He is currently a Ph.D. candidate in the Department of Computer Science and Information Engineering at National Taiwan University, Taipei, Taiwan. His research interests include instruction scheduling and VLSI synthesis. He can be reached by e-mail at huang@solar.csie.ntu.edu.tw



Yu-Chin Hsu received the B.S. degree in computer science from National Taiwan University, Taipei, Taiwan, in 1981, and the M.S. and Ph.D. degrees in computer science from the University of Illinois at Urbana-Champaign in 1986 and 1987, respectively. He is currently serving as Associate Professor of Computer Science at University of California, Riverside. Previously, Dr. Hsu served on the faculty of Tsing Hua University, Hsinchu, Taiwan. His research interests include automated synthesis of digital systems from VHDL description. He co-received an Outstanding Young Author Award from IEEE Circuits and Systems Society in 1990. He can be reached by e-mail at hsu@cs.ucr.edu

Yen-Jen Oyang received the B.S. degree in information engineering from National Taiwan University in 1982, the M.S. degree in computer science from California Institute of Technology in 1984, and the Ph.D. degree in electrical engineering from Stanford University in 1988. He is currently an Associate Professor in the Department of Computer Science and Information Engineering, National Taiwan University. His research interests include computer architecture, distributed systems, and VLSI system design. He can be reached by e-mail at yjoyang @csie.ntu.edu.tw