Wavesched: A Novel Scheduling Technique For Control-flow Intensive Behavioral Descriptions[*]

Ganesh Lakshminarayana, Kamal S. Khouri, and Niraj K. Jha
Department of Electrical Engineering,
Princeton University, Princeton, NJ 08544

Abstract

In this paper, we present a novel scheduling algorithm targeted towards minimizing the average execution time of control-flow intensive behavioral descriptions. Our algorithm uses a control-data flow graph (CDFG) model, which preserves the parallelism inherent in the application. It explores previously unexplored regions of the solution space through its ability to overlap the schedules of independent iterative constructs, whose bodies share resources. It also incorporates well-known optimization techniques like loop unrolling in a natural fashion. This is made possible by a general loop-handling technique, which we have devised. Application of the algorithm to several common benchmarks demonstrates up to 4.8-fold improvement in expected schedule length over existing scheduling algorithms, without paying a price in terms of the best- and worst-case schedule lengths required to execute the behavioral description (in fact, frequently, the best/worst-case schedule lengths are also better for our algorithm).

1 Introduction

The scheduling problem arises in many different forms in divergent fields of study. Scheduling of operations on functional units and variables on registers in high-level synthesis, instructions on a processor during compilation, packets on communication links in networks, and tasks on processing elements in distributed computing are a few typical examples.

In high-level synthesis, there are three different kinds of behavioral descriptions: data-dominated, control-flow intensive (CFI), and control-dominated. Data-dominated descriptions are characterized by a predominance of arithmetic operations and the absence of control-flow constructs. CFI descriptions reflect a mix of arithmetic and logical operations, and control flow structures like loops and conditional operations. Control-dominated applications are characterized predominantly by control-flow operations. Problems in the digital signal and image processing domains, such as filtering, convolution, and discrete cosine transform, translate into data-dominated descriptions. Applications related to network protocol handling, and many other network-centric applications are characterized by CFI descriptions. Sequencers, which generate sequences of signals to control events in a process, are control-dominated. The scheduling problem for each of these domains has its own unique flavor, and distinct classes of algorithms are used to handle the problems in each domain. A survey of scheduling techniques for data-dominated applications can be found in [1]. Scheduling for control-dominated applications was considered in [2] and [3].

In this paper, we deal with the scheduling problem for high-level synthesis of CFI behavioral descriptions. The algorithm accepts as input, a CDFG, a target clock period, and an allocation constraint (restrictions on the number and type of different library elements allowed) and produces a scheduled CDFG which is optimized for average execution time. It is assumed that the type of library element executing each operation is known.

Past research in scheduling of CFI behaviors has modeled the input description as a *control flow graph* (CFG), which is basically a graphical description of a sequential program based implementation of the functionality. Scheduling techniques based on this model were presented in [4]-[7]. While the CFG model is well suited to capture execution of instructions on a general-purpose uniprocessor, it has been shown to be inadequate in exploiting the parallelism inherent in typical CFI applications. CDFG models, equipped to handle control and data dependencies, without sequentializing independent operations, and algorithms for scheduling CDFGs, were presented in [8]-[10]. While these algorithms exploit parallelism within loop structures, they are incapable of optimizing across loop boundaries, and do not support loop unrolling. Methods such as those presented in [11,12] are capable of exploiting loop unrolling and pipelining to enhance performance, but are not capable of performing concurrent optimization of independent loops. A survey of various techniques targeted towards achieving better schedules for very large instruction word processors, which is closely related to the problem of scheduling for high-level synthesis, can be found in [13].

Our scheduler, called *Wavesched* for its wave-like scheduling ability, uses a CDFG model of the input behavioral description. It can parallelize the execution of independent loops whose bodies share resources. The scheduler is also capable of transcending loop boundaries in its quest for a globally optimal solution, *i.e.* it subsumes several well-known optimization techniques such as loop unrolling and functional pipelining. This is made possible by a general technique we have developed to handle inter-iteration dependencies that arise as a result of loop unrolling. Our algorithm can support multi-cycled and pipelined functional units and perform chaining to enhance cycle time utilization. A synergy of the above techniques results in schedules which are optimized for average execution time and have same or better best- and worst-case execution time characteristics as those produced by conventional schedulers.

2 Background

In this section, we discuss the basic concepts used in our work. First, we describe our CDFG model with an example. Next, we describe state transition graphs (STGs), which help to describe schedules. This is followed by an introduction to the concept of concurrent loop optimization and an example illustrating the power of this technique. We then present a general technique for keeping track of inter- and intra-loop dependencies, which allows a unified treatment of loops and other conditional operations.

2.1 Preliminaries

High-level synthesis starts with the compilation of the input behavioral description, usually written in a high-level hardware description language like VHDL or Hardware C, into a CDFG. A CDFG is a directed graph, whose nodes represent operations, and

[*]This work was supported in part by NSF under Grant No. MIP-9319269 and in part by Alternative System Concepts, Inc. under an SBIR contract from Air Force Rome Laboratories.

edges represent dependencies between operations. In CFI descriptions, the dependencies are of two types: data and control. An edge represents a data dependency if the source node of the edge produces data that the sink node consumes. Existence of a control dependency between nodes implies that the execution of the sink node depends on the outcome of the execution of the source node. Figures 1 and 2 show a behavioral description written in a

```
int TEST1 ( int $k1$, int $k2$, int $a$, int $b$){
  int $x1, x2, var1, var2, var3, var4, var5$;
  $x1 = 10$;
  $x2 = 0$;
  for (int $count = 0$; $count \leq k1$; $count{+}{+}$) {
  // $<= 1$ and $++1$
    if ($x1 < 100$) { // $<1$
      $var1 = a + x1$; // $+1$
      $var2 = 5 + var1$; // $+2$
      $x1 = var2 \times 3$; // $*1$
    } else {
      $var3 = a + x1$; // $+3$
      $x1 = var3 \times 7$; //$*2$
    }
  }
  for (int $count1 = 0$; $count1 \leq k2$; $count1{+}{+}$) {
  // $<= 2$ and $++2$
    $var5 = b + x2$; // $+4$
    $x2 = 6 \times var5$; // $*3$
  }
  return ($x1 + x2$); // $+5$
}
```

Figure 1: A fragment of code, TEST1, written in a high-level language

high-level language and the CDFG corresponding to the description, respectively. The comment at the end of each statement indicates the operation in the CDFG it corresponds to. Variable declarations and initializations do not correspond to operations in the CDFG. Data dependencies in the CDFG are indicated by continuous arcs, and control dependencies, by broken arcs. The **then** and **else** branches of an **if** statement are represented by placing $+$ and $-$ symbols, respectively, adjacent to operations executed in these cases. For example, operation $+1$, which is in the **then** branch of the **if** statement <1, has an incoming control dependency marked by a $+$, and operation $+3$, which is in the **else** branch of the same **if** statement, is fed by a control dependency marked by a $-$ symbol. Note that the value that gets assigned to *x1* depends upon whether operation <1 evaluates to *true* or *false*. This is expressed by means of a *select* operation, which assigns to its output, either of two values, at its *l* and *r* input ports, depending on the value at *s*. Though, in general, an operation can execute only if all its data dependencies are known, a select operation can execute if the value at port *s* is known, and data are available at the selected input port. For the CDFG shown in Figure 2, variable *x1* is the output of select operation **Sel1** whose input ports *l*, *r*, and *s* are fed by $*1$, $*2$, and <1, respectively. An edge is also annotated with the initial value of the variable it corresponds to. Edges in a CDFG are of two types: reverse and forward edges. Reverse edges are those that represent data and control dependencies between operations belonging to different iterations of the same loop body, and forward edges represent dependencies within an iteration of a loop. Conditional edges in the CDFG are annotated with their probability of occurrence. For example, operation <1 evaluates to *true* with a probability of 0.5, and operation $<=1$ evaluates to *true* with a probability of 0.98.

Variables *x1* and *x2* in the CDFG do not directly feed operation

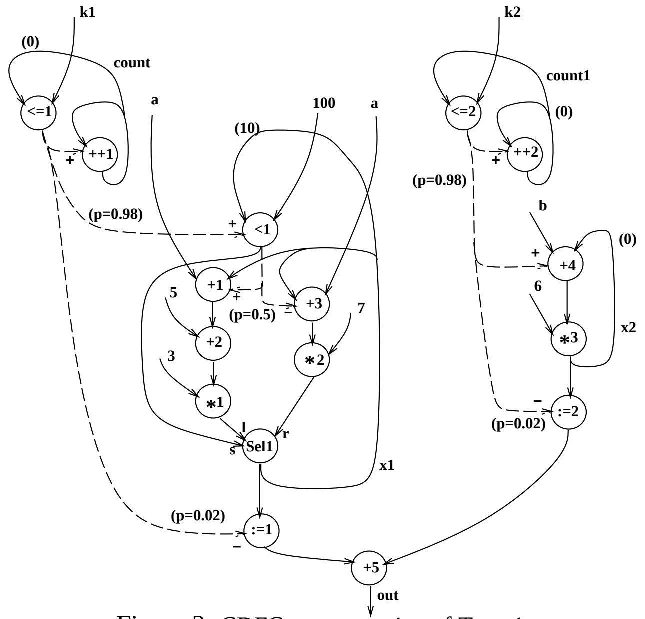

Figure 2: CDFG representation of TEST1

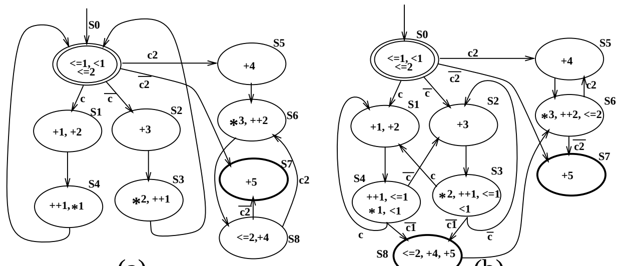

Figure 3: (a) A schedule produced by PBS, and (b) a schedule produced by LDS

$+5$, but feed operations $:= 1$ and $:= 2$ instead. These intermediate operations are control-dependent on operations $<= 1$ and $<= 2$, respectively. They ensure that operation $+5$ can only execute when both loops have finished execution. In general, when an operation outside a loop body (sink) is dependent on the result of an operation inside the loop (source), the result of the source operation feeds an intermediate operation which is control-dependent on the operation that performs the loop test. The sink operation is sourced by the intermediate operation. These intermediate operations are referred to as *endloop* operations. These operations do not have a physical implementation, and are incorporated into the CDFG for ease of representation and handling.

On completion, the scheduler outputs an STG describing the schedule. The STG is a directed graph whose nodes represent states and edges represent transitions between states. Nodes in the STG have information about the operations executed in the corresponding state, and edges capture the conditions under which a state transition takes place.

2.2 Concurrent loop optimization

In this subsection, we describe concurrent loop optimization, a technique which helps overlap the schedules of the independent loop bodies which share resources. To motivate this technique, we present schedules derived using some existing algorithms, without the benefit of this technique and contrast them with the schedule produced by our scheduler *Wavesched*. We consider the scheduling of the CDFG shown in Figure 2 under the following allocation constraints: a multiplier of type *mult1*, two adders of type *add1*, three comparators of type *comp1*, and two incrementers. Each of

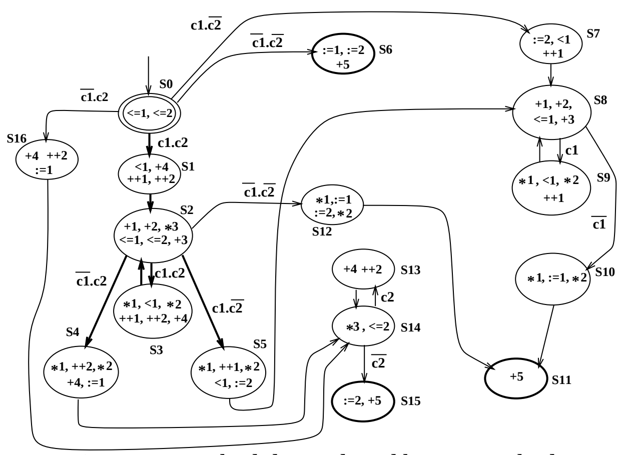

Figure 4: A schedule produced by Wavesched

these functional units is assumed to take one cycle. Suppose that the following chains of operations can be performed in one cycle: (a) two additions, (b) a multiplication followed by a comparison, (c) a comparison followed by an increment, (d) a comparison followed by an addition, (e) an addition followed by a comparison, (f) two comparisons, (g) a comparison followed by a multiplication, and (h) an increment followed by a comparison.

Figures 3(a), 3(b) and 4 represent the STGs obtained by path-based scheduling (PBS) [4], loop-directed scheduling (LDS) [5] and *Wavesched*, respectively, under the above constraints. Start states are represented by double ellipses, and terminal states, by bold ellipses. The method given in [5] for deriving the expected number of cycles for schedules, which is applicable to CFGs, was modified to make it applicable to CDFGs [14]. The expected number of cycles for the schedules produced by PBS, LDS, and *Wavesched*, respectively, computed using this modified method, are 248 *cycles*, 199.94 *cycles*, and 120.09 *cycles*.

The reason for the above difference is that the two loops in CDFG TEST1, which are actually independent, and can be executed in parallel, are sequentialized in the CFG. As a result, for the STGs shown in Figures 3(a) and 3(b), the execution of the second loop can begin only after the execution of the first is terminated. However, in the schedule produced by our algorithm, the execution of these parallel loops is overlapped in time. With both loops executing in parallel, our schedule would complete one iteration of each loop in two cycles by appropriately skewing the execution of the operations in the loops to ensure that resource constraints are satisfied. The bold edges in Figure 4 represent transitions between states where both loops execute concurrently. The schedules produced by PBS and LDS, however, cannot take advantage of this effect. Therefore, one iteration of each loop still takes two clock cycles, but these iterations are sequential as opposed to parallel, resulting in a 1.7-fold slowdown in performance for LDS, relative to our method.

The above example illustrates that the performance of schedulers based on the CFG model is heavily dependent on the structure of the CFG. If a CFG, which parallelized the execution of loops, were used as input to the LDS or PBS algorithms, the schedule produced would have been similar to the one produced by our algorithm. However, we are not aware of any algorithm that derives the "best" CFG for a given behavioral description, under a given set of allocation constraints. Therefore, a scheduling technique that can effectively use the flexibility afforded by the CDFG representation is essential in exploiting parallelism in the input description. Details of our scheduling algorithm, which satisfies the above criterion, can be found in Section 3. Note that, in this case, the algorithm described in [6], while partially addressing the limitations of CFG-based scheduling, would not significantly improve the performance of the schedule over that of LDS because it is restricted to reordering of operations within basic blocks.

At this stage, we would like to point out that some synthesis systems do not allow control-dependency chaining, *i.e.* an operation cannot be scheduled in the same state that generates the condition necessary for its execution. If specified, our scheduling algorithm can either enforce this restriction, or benefit from the flexibility that results from its absence. For the ease of exposition, this restriction is assumed to be in effect for schedules produced by *Wavesched*. However, this restriction is not enforced for the schedules shown in Figures 3(a) and (b).

Our algorithm supports *implicit unrolling* of loops. The user can specify a bound on the number of times, U, a loop can be unrolled. This implies that the scheduler has the option of unrolling the loop up to U times during the course of its execution. If the CDFG contains nested loops, different loops in the nest can be unrolled as deemed appropriate. A state could, therefore, potentially contain operations belonging to several different loops and to different iterations of the same loop body. Hence, we need a mechanism to keep track of the position of an operation in the loop hierarchy and handle inter-iteration dependencies induced by loop unrolling. Section 2.3 presents a technique we have devised for this purpose.

2.3 Handling loop-induced dependencies

From the example presented in the previous section, it is clear that the ability to perform inter- and intra-loop optimizations is a desirable feature for a CFI scheduler. Incorporating these techniques into a scheduling algorithm requires a general technique to keep track of loop-induced dependencies. The techniques presented here are geared towards simplifying loop handling by making loops appear "just like" other conditional dependencies.

In this subsection, we present (a) a numbering convention to keep track of different iterations of an operation (presented in Section 2.3.1), and (b) a means of deriving fanin-fanout relationships between operations augmented with the derived loop iteration numbers (presented in Section 2.3.2).

To motivate the analysis in this subsection, we consider the following problem, which is common to most scheduling algorithms: that of identifying the schedulable successors of an operation which has just been scheduled. If loop unrolling is not allowed, this procedure can be carried out through structural analysis of the CDFG. First, we derive the set, *Succ_set*, of fanouts of the scheduled operation. If all fanins of an operation, *succ* $\in$ *Succ_set*, have been scheduled, *succ* is considered schedulable. If loop unrolling is allowed, structural analysis of the CDFG is not sufficient to perform this task. This is because different iterations of the same operation can co-exist, and in order to schedule a specific iteration of an operation, we need to ensure that the correct iterations of its fanins have been scheduled. For example, for the CDFG shown in Figure 5, operation $*1$, which occurs in the third iteration of the outer loop and the second iteration of the inner loop is schedulable only if the third iteration of operation $+1$, and the second inner loop iteration of the third outer loop iteration of operation > 3 have been scheduled.

2.3.1 Keeping track of loop iterations

Loops can be viewed as inducing a hierarchical structure on the CDFG. Operations which are not part of any loop have a hierarchical level of 0, operations which are embedded in exactly one loop have a hierarchical level of 1, and so on. This hierarchical structure can be represented as a tree.

Definition 1 *The* loop number *of an operation is defined as an ordered set consisting of the numbers assigned to different loops that this operation is a part of. The set is ordered by increasing hierarchical level.*

For instance, the loop number of operation $++3$ in Figure 5, which is a part of loop $L3$, numbered 0, at hierarchical level 1, and loop $L1$, numbered 0, at hierarchical level 0, is $< 0, 0 >$. The number of elements in an operation's loop number is the same as

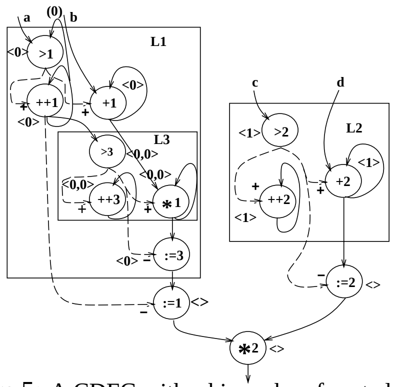

Figure 5: A CDFG with a hierarchy of nested loops

its hierarchical level. Every operation in Figure 5 is annotated with its loop number.

The hierarchical level of an endloop operation is one less than that of the loop it helps terminate. For example, operation $:= 3$, which serves to transfer the result of operation $*1$ outside loop $L3$, takes on loop number $< 0 >$. This represents a hierarchical level of 0, while operations in the body of loop $L3$ have a hierarchical level of 1.

Each loop that contains an operation, can be unrolled an arbitrary number of times (bounded by a user-specified constant). Operations in different iterations of the same loop are distinguished by their *loop counts*.

Definition 2 *If an operation embedded in n loops, numbered a_1, a_2, …, a_n in increasing hierarchical order, represents the b_ith iteration of the loop numbered a_i, its loop count is given by $< b_1, b_2, \ldots, b_n >$.*

2.3.2 Identifying operation fanin-fanout relationships

Section 2.3.1 introduced a means of representing different iterations of the same operation. In this subsection, we seek to infer inter- and intra-iteration dependencies among operations. The most natural way to represent such relationships is through a directed graph, called the *composite CDFG*, whose nodes are referred to as *composite operations.* A composite operation possesses information about the operation it represents, its loop number, and its loop count. It is represented as $\{op,\ ln,\ lc\}$ where *op* identifies the operation, and *ln* and *lc* its loop number and loop count, respectively. Function $OP(C_{op})$ returns the operation that composite operation C_{op} corresponds to.

We now present a method to construct the edges of the composite graph. First, we formally state the problem we wish to solve. We then relate the loop number of a composite operation to that of its fanins. This result is utilized to establish a relationship between the loop count of a composite operation and that of its fanins.

Problem statement: Given composite operations C_{op1} and C_{op2}, find the conditions under which C_{op1} is a fanin of C_{op2}, given that $OP(C_{op1})$ is a fanin of $OP(C_{op2})$.

Lemma 1 *If $C_{op1} = \{op1, ln1, lc1\}$ sources $C_{op2} = \{op2, ln2, lc2\}$, then either (a) ln1 is a prefix* [1] *of ln2, or (b) ln2 is a prefix of ln1 and C_{op2} has a hierarchical level one less than C_{op1}.*

Lemma 2 *If C_{op1} sources C_{op2} and the hierarchical level of C_{op1} is n, then (a) the first $n-1$ elements of the loop counts of C_{op1} and C_{op2} are identical, and (b) the nth element of the loop count of C_{op2} exceeds by one (is equal to) the nth element of the loop count of C_{op1}, if the edge connecting $OP(C_{op1})$ to $OP(C_{op2})$ is a reverse (forward) edge.*

Proofs of these lemmas are available in [14].

[1] ln_i is a prefix of ln_j if the hierarchical level of $ln_i \leq$ hierarchical level of ln_j and the first p elements of ln_i are identical to the first p elements of ln_j, where p is the hierarchical level of ln_i.

Table 1: Table of successors

	loop number(Y)	*N.I.*	*loop count(Y)*
1	$< a_1, a_2, \ldots, a_n >$	*no*	$< b_1, b_2, \ldots, b_n >$
2	$< a_1, a_2, \ldots, a_n >$	*yes*	$< b_1, b_2, \ldots, b_n + 1 >$
3	$< a_1, a_2, \ldots, a_m >$ *where* $(m > n)$	*yes*	$< b_1, b_2, \ldots, b_{n-1}, b_n + 1, b'_{n+1}, \ldots, b'_m >$
4	$< a_1, a_2, \ldots, a_m >$ *where* $(m > n)$	*no*	$< b_1, b_2, \ldots, b_{n-1}, b_n, b'_{n+1}, \ldots, b'_m >$
5	$< a_1, a_2, \ldots, a_{n-1} >$	*n/a*	$< b_1, b_2, \ldots, b_{n-1} >$

Table 1 gives the loop count of a fanout operation Y of operation X. X is assumed to have a loop number of $< a_1, a_2, \ldots, a_n >$ and a loop count of $< b_1, b_2, \ldots, b_n >$. *N.I.* stands for next iteration, and a *yes* (*no*) in this column indicates that Y is connected to X by means of a reverse (forward) edge. In rows 1 and 2 of the table, X and Y belong to the same loops and, therefore, have the same loop numbers. The loop count of Y is either the same as the loop count of X (row 1) or exceeds the loop count of X by one in the nth position (row 2), where n is the hierarchical level of X, depending upon whether X and Y belong to the same loop iteration, or are one iteration apart. Rows 3 and 4 correspond to a case where the loop number of X is a prefix of the loop number of Y, *i.e.* Y belongs to every loop that X is a part of, but the converse is not true. In this case, the last $m - n$ elements of the loop count of Y cannot be inferred from the loop count of X and the nth element of the loop count of Y either equals or is one greater than the nth element of the loop count of X, depending upon whether or not X and Y belong to the same iteration of the deepest loop to which both belong. In the last row, Y represents an endloop operation of the deepest loop X belongs to. By Lemma 2, the first $n-1$ elements of the loop counts of X and Y are equal, thus completely specifying the loop count of Y.

3 The Algorithm

In this section, we describe our scheduling algorithm in detail. We first outline the algorithm and illustrate its application to a CDFG. We then describe its constituent parts.

3.1 Outline

Figure 6 shows our scheduling algorithm. The algorithm accepts as input, a CDFG, an allocation constraint, a clock cycle time, and a bound on the number of times a loop can be unrolled. Control dependency edges in the CDFG are annotated with their probabilities of occurrence. The allocation constraint specifies the numbers and types of functional units available to the scheduler. Module selection (assignment of a functional unit type to each operation in the CDFG) is assumed to have been performed prior to scheduling. The scheduler returns an STG which represents the schedule.

We illustrate the scheduling process using the CDFG shown in Figure 2. The allocation, clock cycle, and chaining constraints are the same as those used for the example in Section 2.2. We assume for this example that a loop can be unrolled no more than once.

Definition 3 *The frontier of a state is the set of all composite operations which are immediately schedulable upon leaving the state.*

Our basic approach is constructive: we construct an STG from a CDFG by packing operations from the CDFG into states in the STG, beginning with the first level of operations in the CDFG. A state receives a set of operations to schedule; these operations constitute a subset of any one of its immediate predecessor's frontier. The frontier grows as operations are scheduled in the state (statements **11** to **16** in the pseudocode), because the scheduling of operations renders more operations schedulable. When a state is fully packed, the operations in its frontier are stored in the array *Unscheduled_immediate_successors*, which is indexed by state. States whose immediate successors have not been identified are stored in a queue, *State_q*. Statement **3** illustrates the formation of the queue. States are successively dequeued (**24**). When a state, *s*, is dequeued, the operations on its frontier are packed into states which represent the immediate successors of *s*. If the fully grown state is not identical to any known state, then it is enqueued (**22**) and the composite operations on its frontier are stored in the array *Unscheduled_immediate_successors* (**21**). This state will be dequeued in turn. When the queue is empty (**23**), we have a complete STG which describes the schedule.

The first step, during scheduling, is the identification of the set of operations, *initial*, that can be scheduled initially. The elements of this set are operations whose first iteration depends only upon primary inputs. In our example (see Figure 2), operations $<= 1$ and $<= 2$ fall into this category. Keeping track of the iteration number of every operation embedded in a loop is accomplished by working with composite operations (defined in Section 2.3.2). Statement **5** handles the conversion of the operations in *initial* into composite operations. Since we are dealing with the first iteration of operation $<= 1 (<= 2)$, whose loop number is $< 0 >$ $(< 1 >)$, the corresponding loop count is $< 0 > (< 0 >)$.

At this stage, we capture the conditions under which different subsets of *initial_composite* execute, and create states which are responsible for scheduling the corresponding subsets. This is done by first identifying the control dependency edges feeding the composite operations in *initial_composite* (**7**). Each of these control edges can evaluate either to *true* or *false*. Different combinations of truth values on the control dependency edges result in activation of different subsets of *initial_composite*. Statement **8** identifies different combinations of truth values on the control dependency edges and statement **9** extracts the subset of *initial_composite* that is activated on this combination of truth values. Suppose that, in this case, when *condition_inputs* is an empty set, the statement "**foreach** combination of conditions (*condition*, *condition_inputs*)" allows exactly one iteration through the following loop. Then *S_condition* evaluates to *initial_composite* because no composite operation in *initial_composite* has any incoming control dependency edges. Once a subset of composite operations has been identified, we form a state, *new_st*, which shares with its successors, the responsibility of scheduling the composite operations in *S_condition* (**10**). The state grows with the scheduling of composite operations present in *S_condition* and their successors. This expansion continues until no more composite operations can be added to a state. The expansion of a state (statements **11** through **16**) can be halted by allocation, clock cycle, and loop unrolling constraints. A three-stage process controls the scheduling of composite operations to a state:

1. Selection of a composite operation, *new_C*, from *S_condition*: the selection process should identify the "best" composite operation whose inclusion in *new_st* respects allocation, clock cycle, and loop unrolling constraints (**12**). In our example, composite operation $s0 = \{<= 1, < 0 >, < 0 >\}$ is selected.
2. Addition of the schedulable successors of *new_C* to *S_condition*: when an operation is scheduled, immediate successors of this operation, whose predecessors have been scheduled, become eligible for scheduling. In our example, completion of the composite operation $\{<= 1, < 0 >, < 0 >\}$ implies that its immediate successors, $s1 = \{+ + 1, < 0 >, < 0 >\}$, $s2 = \{:=1, <>, <>\}$, and $s3 = \{<1, < 0 >, < 0 >\}$, can be scheduled. *S_condition* is augmented by the addition of *s1*, *s2*, and *s3* (**15**). Therefore, *S_condition* constitutes a frontier of composite operations which sweeps over the surface of the CDFG as scheduling progresses.
3. Removal of *new_C* from *S_condition* (**16**).

One more iteration through the innermost loop schedules composite operation $\{<= 2, < 0 >, < 0 >\}$ in *new_st* and adds $\{+ + 2, < 0 >, < 0 >\}$, $\{+4, < 1 >, < 0 >\}$, and $\{:= 2, <>, <>\}$ to *S_condition*. At this point, no additional composite operations can be scheduled in *new_st* because of the assumption related to control-dependency chaining. Therefore, the **break** statement at line **13** is executed, transferring flow of control outside to loop. We check to see if *new_st* is identical to any existing state (**18**). The process of inferring equality between states is described in [14]. Since this is not the case, we add an arc from the parent state, *S0*, to *new_st*, labeled 1, which implies that there is an unconditional transition from *S0* to *new_st* (**20**). The immediate succes-

```
Wavesched (CDFG G, ALLOCATION_CONSTRAINT C,
        CLOCK_PERIOD clk, UNROLL_BOUND U, STG S){
0  SET<OPERATION> initial = get_level_1_operations(G);
   //level 1 operations are defined as those operations which,
   //when all loop edges (reverse edges) in G are removed,
   //depend only on primary inputs
1  STATE S0; //create a start state
2  STATE parent_state = S0;
3  QUEUE<STATE> State_q; //create a queue of states
4  ARRAY<SET<COMPOSITE_OPERATION>> Unscheduled_-
   immediate_successors;//An array, indexed by state, which
   // stores the composite operations which are immediately
   // schedulable upon leaving the state
5  SET<COMPOSITE_OPERATION> initial_composite = Make_-
   composite(initial);//construct composite operations from
   //the operations in initial with loop counts initialized to 0.
6  loop_forever() {
7    SET<COMPOSITE_OPERATION> condition_inputs =
     composite operations whose outputs are control
     dependency edges feeding operations in initial_composite;
8    foreach combination of conditions (condition,
     condition_inputs) {//condition corresponds to the
       //assignment of a specific set of truth values to the
       //results of the composite operations in condition_inputs
9      SET<COMPOSITE_OPERATION> S_condition =
         under_condition(condition, initial_composite);
       //Extracts all operations in initial_condition that will
       //execute under condition
10     STATE new_st;
11     loop_forever() {
12       COMPOSITE_OPERATION new_C =
           Select_composite_operation(S_condition, C, clk, U);
13       if (new_C == NULL) break; //no more operations
         can be scheduled in this state
         else{
14         add_composite_operation_to_state(new_C, new_st);
15         add_schedulable_successors(new_C, S_condition);
16         remove_composite_operation(S_condition, new_C);
         }}
17     S_successors = S_condition;
18     if (new_st is identical to an existing state, P)
19       add an arc in S, labeled condition,
         from parent_state to P;
       else {
20       add an arc in S, labeled condition,
         from parent_state to new_st;
21       Unscheduled_immediate_successors[new_st]
           = S_condition;
22       append(new_st, State_q);
       }}
23   if (is_empty(State_q) == 1) break;//STG complete, exit
24   STATE s = dequeue_top(State_q);
25   Initial_composite = Unscheduled_immediate_successors[s];
}}
```

Figure 6: Our scheduling algorithm

sors of *new_st* are invested with the responsibility of scheduling the composite operations remaining in *S_condition*. *new_st* is then appended to *State_q* and immediately dequeued. Its frontier constitutes *initial_composite* which is handled in the manner described above.

3.2 Selecting the best composite operation

Since the process of selecting the "best" composite operation can be computationally intensive, we use a heuristic. Our method is based on the fact that operations in the CDFG which feed primary outputs through long paths are more critical (*i.e.* less mobile) than operations which feed primary outputs through short paths and, therefore, the former need to be scheduled earlier. The length of a path is measured as the sum of the delays of its constituent operations.

In data-dominated descriptions, with no loops and conditional operations, the longest path between any pair of operations is fixed. In CFI descriptions, some paths could be input-dependent. Therefore, the longest path between a pair of operations must be defined with respect to a given input. For example, the longest path between operations <1 and $+5$ in Figure 2 depends on the number of iterations of the loop containing <1, which depends on the value of $k1$. Since our scheduling algorithm is geared towards minimizing the average execution time, we use the expected length of the longest path from a composite operation to a primary output as a metric to rank different composite operations. We next outline a procedure to compute the expected length of the longest path and illustrate its application to a CDFG.

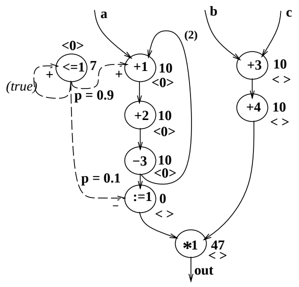

Figure 7: Illustration of composite operation selection

Example 1: Consider the CDFG shown in Figure 7. Control dependency edges are annotated with their probabilities of occurrence, and operations with the corresponding functional unit delays and loop numbers. For example, operation $+1$ is assigned to an adder type which takes $10ns$ to execute and has a loop number $< 0 >$. Suppose we are faced with the question of selecting a composite operation from the set $\{\{+1, < 0 >, < 3 >\}, \{+3, <>, <>\}\}$. We first evaluate the expected length of the longest path assuming that no loop unrolling is performed and then incorporate the effect of unrolling. Assuming that no unrolling is performed:

$$\begin{aligned} E(\lambda(+1, out)) &= 10 + E(\lambda(+2, out)) \\ &= 20 + E(\lambda(-3, out)) \\ &= 30 + 0.9 \times E(\lambda(+1, out)) + 0.1 \times 47 \\ \Rightarrow E(\lambda(+1, out)) &= 347ns \end{aligned} \quad (1)$$

where $E(\lambda(op1, op2))$ is the expected length of the longest path connecting operations *op1* and *op2*. Note that this delay corresponds to the length of the longest path connecting the *zero*th iteration of operation $+1$ to *out*. Since we are dealing with the third iteration of operation $+1$, we need to subtract, from our estimate, the time taken to perform three iterations of the loop, which turns out to be $30 \times 3 = 90ns$. This yields an expected maximum path length of $257ns$ for the composite operation $\{+1, < 0 >, < 3 >\}$. Adopting a similar procedure for operation $\{+3, <>, <>\}$ results in an expected longest path length of $57ns$. Therefore, operation $\{+1, < 0 >, < 3 >\}$ is chosen to be scheduled in the state under consideration. We can see that this would be a good choice because operation $+1$ is on the critical path while operation $+3$ is not. ∎

3.3 Synthesizing compact STGs

The techniques presented in this section help keep the size of the STG small, without significantly compromising the average execution time of the schedule. These techniques are based on identifying probable and improbable threads of execution and optimizing the probable threads for speed and the improbable threads for STG compactness.

Consider statement **8** in Figure 6. This statement extracts all possible assignments of truth values for the composite operations that feed the operations of *initial_composite* through control dependency edges. Statement **9** packs all operations which are activated under a given combination of conditions into *S_condition*, which is passed on to a newly created state for scheduling. Not all combinations of conditions are equi-probable. We identify all combinations of conditions whose probability falls below a certain user-specified threshold, t_p. The subsets of *initial_composite* which are activated under any one of these combinations are fused into a single set *S_improbable*, which is considered for scheduling. The state created for scheduling these composite operations is tagged as a *compact state*. Note that, in doing so, we have traded the option of separately optimizing the threads of execution represented by the combinations of conditions represented in *S_improbable* for STG compactness. The children of a compact state are tagged compact. For the frontiers of compact states, the algorithm presented in Figure 6 is modified by replacing statements **7** and **8** by the corresponding statements in Figure 8. The techniques presented in this example can reduce the size of the controller by a significant amount without much compromise in the average schedule length of the design. For the example TEST1 presented in Section 2, the STG shown in Figure 4 is synthesized assuming $t_p = 1$, which represents a schedule optimized for compactness. The number of states in the schedule is 17, eight less than the number of states that would be obtained if the CDFG were synthesized with $t_p = 0$. In this case, both schedules have the same average length; in general, however, a large value of t_p would result in a longer schedule.

```
7   SET<COMPOSITE_OPERATION> loop_tests = loop test of
    operations whose outputs feed operations of
    initial_composite through control dependency edges
8   foreach combination of conditions (condition, loop_tests) {
      //condition corresponds to the assignment of a specific
      //set of truth values to the results of the composite
      //operations in loop_tests
```

Figure 8: Optimizing for compactness

4 Experimental results

The techniques described in this paper were implemented in a program called *Wavesched*, written in C++. We evaluated this program by using it to produce schedules for several commonly available benchmarks. These schedules were compared against those produced by LDS [5] and PBS [4], with respect to the following metrics: (a) expected number of cycles, (b) number of states in the STG produced, and (c) the smallest and largest number of cycles taken to execute the behavioral description. In general, finding the largest number of cycles taken to execute a behavioral description is a hard problem. However, for the examples considered in this paper, static analysis of the description was sufficient to find the number. For *Wavesched*, a value of 0.2 was used for t_p (defined in Section 3.3).

Table 2 summarizes the results obtained. The columns labeled *E.N.C.*, *#states*, *best-case*, and *worst-case* represent, respectively,

Table 2: Expected number of cycles, number of states, best- and worst-case number of cycles results

CDFG	*Clk*	*E.N.C.*			*#states*			*best-case*			*worst-case*		
		ws	LDS	PBS	*ws*	LDS	PBS	*ws*	LDS	PBS	*ws*	LDS	PBS
GCD	13	13.9	14.4	18.2	6	5	5	1	1	1	386	386	513
Maha	13	3.3	6.0	6.0	12	10	10	3	5	5	4	7	7
Cordic	13	41.0	60.0	79.0	9	8	8	3	3	3	515	771	1026
Test1	13	178.6	301.7	349.8	24	12	12	1	1	1	766	1538	1791
Saxpy	13	104.0	501.0	501.0	8	6	6	104	501	501	104	501	501
Paulin	13	7.0	11.0	11.0	7	11	11	7	11	11	7	11	11

the expected number of cycles, number of states in the STG produced, smallest number of cycles, and largest number of cycles taken to execute the STG. Minor columns *ws*, LDS, and PBS represent schedules produced by *Wavesched*, LDS, and PBS, respectively. We used a library of functional units (their delays given in brackets) which consists of (a) an adder *ad1* (10*ns*), (b) a subtractor *sb1* (10*ns*), (c) a multiplier *mlt1* (23*ns*), (d) a less-than comparator *cmp1* (10*ns*), (e) an equality comparator *eq1* (5*ns*), (f) an incrementer *inc1* (5*ns*), (g) a multi-bit OR gate *or1* (3*ns*), and (h) a shifter *s1* (10*ns*). We allowed all schedulers to take advantage of multicycling and data chaining, and allowed control chaining. The following chains of operations were assumed to execute in one cycle: (a) two additions, (b) two subtractions, (c) a comparison followed by either an increment or an addition or a subtraction or a shift, and (d) two comparisons. The following chains of operations were assumed to execute in two cycles: (a) a comparison followed by a multiplication, and (b) a multiplication followed by a comparison. The cycle time was fixed at 13 *ns* for all examples. Note that the chain of two operations frequently executes in much less time than the addition of their execution times. The allocation constraints for an example can be found by looking up the entries in Table 3. For example, the allocation constraint for *GCD* is one subtractor, one less-than comparator, three equality comparators, and one multi-bit OR gate.

The results obtained indicate that *Wavesched* produced an average expected schedule length speedup of 2.1 over schedules obtained using LDS, and 2.2 over schedules obtained using PBS. The average price paid in terms of the number of states was a 26% increase for *Wavesched* compared to LDS and PBS. Note that an increase in the number of states need not be reflected in a similar increase in circuit area. We believe that the improvement in performance will be worth paying the small price in area, if any. For the example *Test1*, the schedule produced by *Wavesched*, upon synthesis with an in-house system, resulted in a controller-datapath circuit whose layout area was only 1.7% more than that produced using *LDS*, though *Wavesched* incurred a 100% overhead in the number of states in the schedule. We also note that, for *Wavesched*, the number of cycles in the shortest and longest paths is smaller than or equal to the corresponding numbers for LDS and PBS. The CPU times, measured on an SGI Indy workstation, with a MIPS R4400 CPU running at 175 *MHz*, were under 30 seconds for all the examples.

Table 3: Allocation constraints for the examples in Table 2

Circuit	*ad1*	*sb1*	*mlt1*	*cmp1*	*eq1*	*inc1*	*or1*	*s1*
GCD	-	1	-	1	3	-	1	-
Maha	2	2	-	2	-	-	-	-
Cordic	2	2	1	1	1	2	-	2
Test1	2	-	2	3	-	2	-	-
Saxpy	2	-	2	-	2	2	-	-
Paulin	2	2	2	-	-	-	-	-

5 Conclusions

In this paper, we presented a novel scheduling technique which is geared towards minimizing the expected execution time for CFI behavioral descriptions. It uses a CDFG model which preserves the parallelism inherent in the application. It can perform optimizations within and across loop boundaries and implicitly unroll loops to improve the average schedule length. This is made possible through a general loop representation technique which we have devised. It also performs concurrent loop optimzation. Experiments performed on several benchmarks show expected schedule length speedups of upto 4.8 over schedules produced by existing schedulers.

References

[1] D. D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Boston, 1992.

[2] W. Wolf, A. Takach, C. Huang, and R. Mano, “The Princeton university behavioral synthesis system,” in *Proc. Design Automation Conf.*, pp. 182–187, June 1992.

[3] D. Ku and G. D. Micheli, “Relative scheduling under timing constraints,” *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 696–718, June 1992.

[4] R. Camposano, “Path-based scheduling for synthesis,” *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 85–93, Jan. 1991.

[5] S. Bhattacharya, S. Dey, and F. Brglez, “Performance analysis and optimization of schedules for conditional and loop-intensive specifications,” in *Proc. Design Automation Conf.*, pp. 491–496, June 1994.

[6] R. Bergamaschi, S. Raje, I. Nair, and L. Trevillyan, “Control-flow versus data-flow scheduling: Combining both approaches in an adaptive scheduling system,” *IEEE Trans. VLSI Systems*, vol. 5, pp. 82–100, Mar. 1997.

[7] Y. Fann, M. Rim, and R. Jain, “Global scheduling for high-level synthesis applications,” in *Proc. Design Automation Conf.*, pp. 542–546, June 1994.

[8] S. Amellal and B. Kaminska, “Functional synthesis of digital systems with TASS,” *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 537–552, May 1994.

[9] T. Kim, N. Yonezawa, J. W. S. Liu, and C. L. Liu, “A scheduling algorithm for conditional resource sharing- a hierarchical reduction approach,” *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 425–438, Apr. 1994.

[10] K. Wakabayashi and T. Yoshimura, “A resource sharing and control synthesis method for conditional branches,” in *Proc. Int. Conf. Computer-Aided Design*, pp. 62–65, Nov. 1989.

[11] A. Aiken, A. Nikolau, and S. Novack, “Resource-constrained software pipelining,” *IEEE Trans. Parallel and Distributed Systems*, vol. 6, pp. 1248–1269, Dec. 1995.

[12] D. M. Lavery and W. W. Hwu, “Modulo scheduling of loops in control-flow intensive non-numeric programs,” in *Proc. Int. Symp. Microarchitecture*, pp. 126–137, Dec. 1996.

[13] B. R. Rau and J. A. Fisher, “Instruction-level parallel processing: History, overview, and perspective,” *J. Supercomputing*, vol. 7, pp. 9–50, July 1993.

[14] G. Lakshminarayana, K. S. Khouri, and N. K. Jha, “Wavesched: A novel scheduling technique for control-flow intensive behavioral descriptions,” Tech. Rep CE-J97-001, EE Dept., Princeton Univ.