A New Symbolic Technique for Control-Dependent Scheduling

Ivan Radivojević and Forrest Brewer

Abstract—**This paper describes an exact symbolic formulation of control-dependent, resource-constrained scheduling. The technique provides a closed-form solution set in which all satisfying schedules are encapsulated in a compressed OBDD-based representation. This solution format greatly increases the flexibility of the synthesis task by enabling incremental incorporation of additional constraints and by supporting solution space exploration without the need for rescheduling. The technique provides a systematic treatment of speculative operation execution in arbitrary forward-branching control/data paths. An iterative construction method is presented along with benchmark results. The experiments demonstrate the ability of the proposed technique to efficiently exploit parallelism not explicitly specified in the input description.**

I. Introduction

RESOURCE-CONSTRAINED operation scheduling is the process of determining the assignment of operations to time slots of a synchronous system, subject to data/control-flow dependencies and resource (e.g., functional units, buses, registers) availability. We say that scheduling is control-dependent if some operations from the control/data flow graph (CDFG) are executed conditionally due to the presence of control-flow constructs such as *if-then-else, goto, case, exit*, etc. Such scheduling plays an important role in high-level synthesis (HLS) of digital systems [7], [24]. There are two difficult issues in a formal treatment of control-dependent, resource-constrained scheduling: i) concise formulation of the conditional behavior and ii) treatment of resources. An efficient formulation should not generate an excessive number of constraints and formulation variables. Moreover, a formal evaluation of resource availability in the face of conditional execution is required. This is particularly difficult when movement of operations across basic code block boundaries is not prohibited. It has been demonstrated that the ability to perform speculative operation execution leads to superior schedules [35], [38], [43].

Current practical methods for solving this NP-complete problem involve two basic approaches: i) heuristics and ii) integer linear programming (ILP). Priority-based heuristic scheduling (e.g., [5], [26], [28]) can accommodate a variety of control-dependent behaviors but may fail to find an optimal solution in tightly constrained problems. The reason for this is that heuristic schedulers cannot recuperate from early suboptimal decisions that typically preserve only one representative from a possibly very large pool of qualified candidates. Conventional ILP methods [15] can solve scheduling exactly but suffer from exponential time complexity and the inability to efficiently formulate control constraints. General applicability of these ILP methods has been improved by remapping the constraints [11], [12], a mixed ILP/BDD method [47], and heuristic approaches based on ILP [14], [18]. However, with the exception of [6] (discussed below), no ILP-based technique provides support for conditional behavior. Similarly, a recent branch-and-bound technique [42] based on execution interval analysis [41] has been applied only to acyclic DFG's.

Many HLS systems *prohibit* code motion in order to avoid problems related to evaluation of resource availability and causality of the solutions. An alternative strategy is to *explicitly* write constraints describing global movement of operations, but such approaches reduce to exhaustive enumeration of potential execution scenarios. In the formulation described in this paper, code motion is allowed *implicitly*—there is no need to describe freedom already available (although implicit) in a CDFG.

As an example, we consider the formal approach based on algebra of control-flow expressions (CFE's) [6]. In that work, the timing and synchronization requirements for communicating machines are encapsulated in finite-state machine (FSM) description. From this, scheduling constraints are derived and subsequently solved using a BDD-based O/1 ILP solver. The FSM description is constructed from an algebraic CFE specification that implicitly restricts code motion. Consider, for example, the code segment shown in Fig. 1. A possible CFE specification for this fragment is $p(c{:}r + \bar{c}{:}s)$. This requires that p be executed before c and c before either r or s. An alternative specification is $c{:}pr + \bar{c}{:}ps$, which allows c to be executed before p. If c depends on p, only the first statement is correct. However, if c and p are independent, then *both* behaviors are legal. It is possible to create a *specification* that lists all correct execution scenarios, but the number of such scenarios and the size of the specification grow dramatically as the program complexity increases. In contrast, in our approach, only data dependencies are used to impose the execution order of p and c. In fact, if the data dependencies allow such motion, r and/or s may be executed before c and potentially before p as well. Thus, these potential execution scenarios are implicitly supported by the formulation.

Since operation level parallelism may not be explicit in the input description, some heuristic schedulers focus on detection

Manuscript received September 20, 1993; revised February 22, 1995. This work was supported in part by a fellowship donation from Mentor Graphics Corp. and UC-MICRO under Project 92-019. This paper was recommended by Associate Editor K. Keutzer.

The authors are with the Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106 USA.

Publisher Item Identifier S 0278-0070(96)01343-7.

```
p;
if (c) r;
else s;
```

Fig. 1. Conditional behavior.

of mutual exclusiveness in CDFG's. Tree scheduling (TS) [13] uses a tree-representation of the execution paths to enable movement of operations. Conditional vector list scheduling (CVLS) [43] uses *condition vectors* [44] to dynamically track mutual exclusiveness of the operations that can be executed in a speculative fashion (i.e., pre-executed). Transformation of a CDFG with conditional branches into one without conditional branches is performed in [17], but there is no support for speculative execution. Furthermore, these heuristics are restricted to nested conditional branches (conditional tree control structure). Multiple conditional trees are addressed by Wakabayashi [43], but the trees are either scheduled sequentially (using a priority scheme), or conditional tree duplication is performed.

Some synthesis systems emphasize treatment of behavioral level timing specifications. However, either a predefined order of operation is enforced before the scheduling [5], or the treatment of resource constraints is not fully considered [19]. The PUBSS system [45] forms a product machine of individual behavior FSM's (BFSM's) to statically schedule I/O communication between the components. PUBSS supports a variety of timing constraints. However, parallelism increasing techniques [9] are applied in a static fashion (before BFSM collapsing and scheduling). The issue of resource constraints is either not formally discussed [40], [48], or the formulation of exclusivity constraints requires an excessive number of 0/1 ILP variables [39].

To our knowledge, the first attempt to address the scheduling problem using symbolic computations was made by Kam in [16]. There, several CAD applications of MDD's (multivalued decision diagrams) were described. Scheduling of acyclic DFG's with function unit constraints was formulated using multivalued variables, but the approach seemed to be practicable only for tightly constrained problems. Unfortunately, too few experimental results were left documented to make a critical assessment of that approach. An exact *symbolic* formulation of the control-dependent, resource-constrained scheduling problem was introduced in [31]. Unlike other approaches in which a single representative solution is generated, in this technique, *all* feasible schedules are encapsulated in a compressed Ordered Binary Decision Diagram [4] (OBDD) form. This is advantageous since the exact effect of additional constraints derived during subsequent synthesis steps is *incrementally* computable. Also, there is the additional benefit of being able to explore the solution space without the need to reschedule the problem instance. An alternative symbolic formulation [46] uses finite automata to capture resource/timing/synchronization constraints. A product automaton is built that satisfies the specified behavior. Its OBDD representation is then traversed to find a minimum-latency schedule. However, similar to [6], the technique lacks support for various forms of a parallelism extraction to be described in Section II.

In this paper, we describe a symbolic technique for exact resource-constrained scheduling of arbitrary forward-branching control structures. Scheduling is performed with the assumption that the allocation of resources is known. The technique supports speculative operation execution and global treatment of parallel control structures. To allow a systematic treatment of the problem, a flexible control representation based on *guard variables, guard functions,* and *traces* is introduced. A *trace validation* algorithm is proposed to enforce causality and completeness of the set of all feasible solutions. The scheduling technique presented in this paper supports arbitrary Boolean constraints as well as conventional timing constraints. Scheduling of multirate interacting FSM's is not addressed in this paper. Similarly, we do not discuss optimizations based on algebraic transformations [30].

The paper is organized as follows. In Section II, we describe several approaches to resource-constrained control-dependent scheduling, as well as some features desirable to improve scheduling quality. The formulation is presented in Section III. Aspects related to the OBDD construction process are considered in Section IV. Experimental results are discussed in Section V. Finally, in Section VI, we present conclusions as well as the questions to be addressed in the future.

II. High-Performance Scheduling Issues

Our scheduling technique assumes an input in the form of a CDFG specification. The CDFG describes both dataflow and control dependencies between the operations and is similar to the one used by Wakabayashi [44]. Fig. 2 contains an pseudocode example and its CDFG representation. Operation nodes are atomic actions potentially requiring use of hardware resources (e.g., arithmetic/logical operations, read/write cycles). Conditional behavior is specified by means of fork and join nodes. An operation node generating a control signal for a fork/join pair is called a *conditional*. Directed arcs establish a link between the conditional and a related fork/join pair. In Fig. 2, the conditional labeled op_2 tests the result of the addition (op_1) and determines the flow of control (i.e., whether "true" (T) or "false" (F) branches should provide operands for op_6).

Fig. 2(a)–(c) shows three different ways to schedule the example assuming that only one resource of each type is available. The schedule in Fig. 2(a) uses the knowledge that after a conditional (op_2) is executed, operations belonging to "T" and "F" branch arcs are mutually exclusive. However, the join node is treated as a synchronization point: op_6 cannot be scheduled until both the "T" and "F" branch are executed. This leads to inefficient schedules, since the execution times for alternative branch arcs may differ widely. Consequently, in this example, it takes five cycles to execute the schedule no matter what decision is made by the conditional. This approach corresponds to that used by traditional ILP schedulers (e.g., [15]).

The schedule shown in Fig. 2(b) improves the "average" execution time to 4.5 cycles by scheduling op_6 on the fourth cycle at the "F" branch. Note that the operation execution

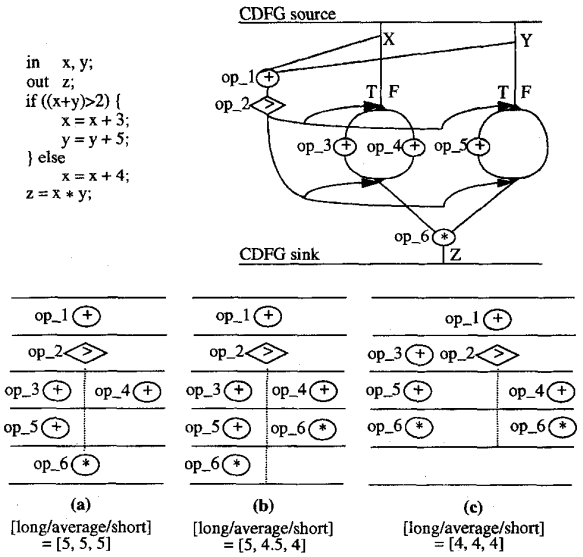


Fig. 2. Example CDFG and its schedules.

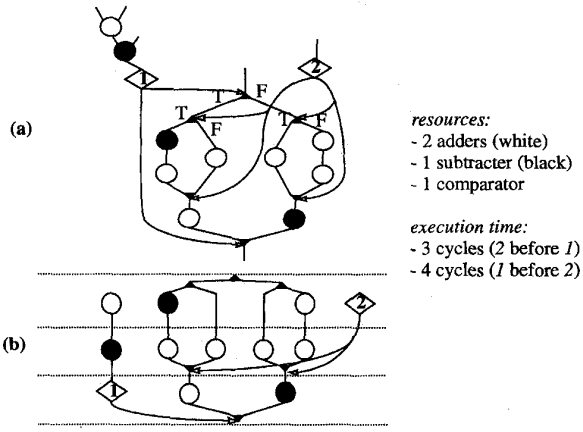


Fig. 3. Speculative operation execution.

order is predetermined before scheduling (e.g., *op_2* before *op_3*, although no data dependency exists between these two operations in the CDFG). This approach is supported by a number of heuristic schedulers (e.g., [5]) and by one recent exact technique [6].

The schedule from Fig. 2(c) not only further improves the average execution time but reduces the longest execution path to four cycles as well. This is done by scheduling *op_3* on the second cycle in a speculative fashion (i.e., before the corresponding conditional *op_2* is resolved). Note that the resource requirements cannot be predicted in a static fashion. For example, if more adders are available, *op_4* can be executed in a speculative fashion as well. The mutual exclusion of *op_3* and *op_4* must be evaluated dynamically by taking into account when the corresponding conditional (*op_2*) is scheduled. This kind of scheduling is supported by several heuristics ([13], [29], [37], [43]).

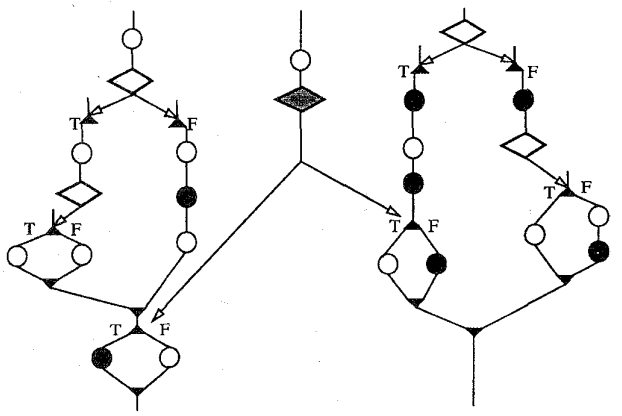
There are several ways to improve the scheduling quality by exploiting parallelism implicit in the CDFG representation.

Speculative Operation Execution: It is often beneficial to determine the control value simultaneously with branch execution. Operations from branch arcs that are executed before the corresponding conditional value is evaluated are said to be *pre-executed*. Such speculative operation execution allows more flexibility in using given hardware resources. A *conditional* is a scheduled operation that generates a control value. Fig. 3(a) shows a CDFG where the control dependencies between the conditionals (comparators 1 and 2) and the corresponding fork/join pairs are explicitly indicated. Speculative operation execution is not possible if the control precedence between the conditional and the fork node is enforced. In this case, at least six time steps are necessary to execute the CDFG, since the longest control/data dependency chain includes six operations. However, if precedence between the conditional and the fork node is removed, operations from the branch arcs can be preexecuted. Fig. 3(b) shows a schedule executing in three cycles using the indicated resources. In general, precedence between a conditional and join node need not be enforced either. In this case, the execution time is bounded only by data dependencies (given sufficient resources).

Out-of-Order Execution of Conditionals: It can happen that a faster schedule is obtained if the top-level conditional (in the input specification) is evaluated *after* some other nested conditional. A simple example of this behavior is shown in Fig. 3(b). The schedule executes in three cycles with the conditional 1 left unresolved until the end of the very last cycle. The knowledge that conditional 2 is resolved during the first cycle is essential to properly interpret resource usage. Both *TS* [13] and *CVLS* [43] rely on a conditional-tree representation of the control and cannot accommodate out-of-order execution of the conditionals without dynamically modifying the tree structure.

Irredundant Operation Scheduling: Another way to improve scheduling quality is to identify operations that are not redundant in the input description but are redundant for certain control paths. The importance of such information has been observed, and the algorithms to detect such operations have been discussed in the literature [13], [44].

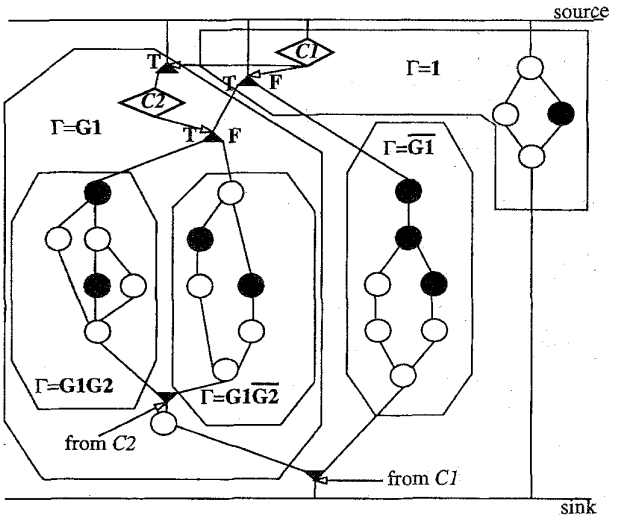
Applications to Parallel Control Structures: Control structures that are either fully parallel or have correlated control introduce additional scheduling challenges. As the number of control paths increases, it becomes difficult to keep track of the mutual exclusiveness among the operations. Ideally, the scheduler should evaluate and maintain this information for all control paths. In Fig. 4, a CDFG is shown in which two parallel trees have a correlated control (shaded comparator). The reader can verify that, given one adder ("white" operation), one subtracter ("black" operation) and one comparator (single-cycle units assumed), a six-cycle schedule can be found only if the control correlation is properly interpreted (i.e., "false" paths are not scheduled). As indicated in Fig. 4, speculative execution (and additional or more versatile resources) can further improve the execution time. Although not typical for conventional structured programs, parallel control structures are likely to result from program transformations performed by parallelizing compilers (e.g., loop unrolling where a conditional behavior is present within the loop body) [35].



no speculative execution:
- 6 cycles (3ALU or 1add/1sub/1comp)

speculative execution:
- 5 cycles (3ALU or 2add/1sub/1comp)
- 4 cycles (5ALU or 3add/2sub/2comp)

Fig. 4. CDFG with correlated control.



Guards: G1 (corresponding to C1 decisions)
G2 (corresponding to C2 decisions)

Fig. 5. Kim's example.

The formulation presented in this paper supports all of the advanced scheduling features discussed above. The execution delay of the longest path of a scheduled CDFG is frequently referred to as the *minimum latency* of the schedule. Our goal is to find *all* minimum-latency schedules, given a CDFG specification and resource constraints. By using OBDD's, we can encode all feasible solutions to a particular problem instance.

III. Formulation

In this formulation, all scheduling constraints are represented as Boolean functions, and an OBDD corresponding to the intersection is built. Each variable C_{sj} describes operation j occurring at time step s. C_{sj} is true iff operation j is scheduled at time step s in a particular solution. We assume a unique mapping from operation type to function unit type. To represent control-dependent behavior, a set of *guard variables* is introduced. Each guard G represents a control-flow decision by a particular conditional—the guard is true for one branch and false for the other. Every control path through an arbitrary combination of fork/join pairs is described by a product of the corresponding guard variables. For each operation j, a Boolean *guard function* Γ_j (defined on the guard variables) encodes all the control paths on which j *must* be scheduled.

Computation of Γ *Functions*: Assume that operation i has n successors $(j_1, j_2, \cdots, j_n)$ and that none of the successors is a join node. Then a guard function Γ_i can be simply computed as a Boolean *Or* of the successors' guard functions Γj_k $(k = 1, 2, \cdots, n)$. This means that operation i has to provide operands to all of its successors. If a successor of i is a join node, then its contribution to Γ_i is equal to $\Gamma_{join} G_k$ or $\Gamma_{join} \overline{G}_k$ (depending whether i belongs to the "T" or "F" branch). Guard functions corresponding to all of the nodes can be computed by a one-pass traversal of the CDFG that starts from a sink node whose guard function is initialized to "1" (tautology).

Shown in Fig. 5 is a CDFG fragment of Kim's example [17] in which two guards (G_1, G_2) encode the conditional behavior. There are three possible execution paths: $(G_1 G_2, G_1 \overline{G_2}, \overline{G_1})$. Indicated blocks $(1, G_1, G_1 G_2, G_1 \overline{G_2}, \overline{G_1})$ correspond to operations that share the same guard function Γ. Operations that must be scheduled on all control paths have $\Gamma = 1$. Note that the number of guard variables is not proportional to the number of control paths. (In Fig. 4, only five guard variables encode 18 control paths). Furthermore, we observe that Γ's are not restricted to product terms (thus, they can handle constructs such as: *goto, exit, case*). A more detailed discussion of the guard-based model is available in [34].

In many aspects, the guard-based model is similar to execution conditions from *path analysis* [2]. In that approach, however, Boolean conditions are used in the hardware allocation phase (after AFAP scheduling is performed). Nevertheless, that research demonstrated that OBDD's efficiently represent control signals in large scale problems. In fact, similar guard-based representations have been used in areas other than HLS—for example, to perform "if-conversion" in experimental vectorizing compilers [1] and simplify code generation for VLIW and superscalar machines supporting predicated execution [8], [22], [36]. A fundamental difference in our approach is that we dynamically consider when the guard becomes known, not just what its value is on a particular control path.

The technique presented in this paper generates a solution in the form of a collection of *traces*. A *trace* is a possible execution instance for a particular control path. In OBDD form, traces correspond to product terms of the Boolean function. Each trace includes the guard variables (identifying a control path) and operation variables (indicating a schedule for the path). For example, in Fig. 5, each trace corresponding to the "false" branch of conditional C_1 contains $\overline{G_1}$, as well

as 0/1 assignment of C_{sj} variables. Operations with $\Gamma = \overline{G_1}$ or $\Gamma = 1$ must be scheduled on that trace. If other operations are scheduled on this trace, they are preexecuted.

The *ensemble schedule* is a set of traces forming a complete deterministic schedule. Conditions for the existence of such a schedule are discussed in Section III.3. The solution OBDD includes only traces belonging to at least one ensemble schedule and implicitly incorporates all feasible ensemble schedules. Note that the number of ensemble schedules can be much larger than the number of traces.

3.1. Speculative Execution Model

In our speculative execution model, only the control precedence between the conditional and join node is enforced. CDFG operations can be scheduled at different time steps on distinct control paths but cannot be scheduled more than once per trace. Each operation from the CDFG is executed at most once regardless of the actual control decisions made when the schedule is executed. For example, this means that in the current model the following scenario is prohibited: i) operation j executes in a speculative fashion using operands A and B and generates result R, ii) a control decision is made and R is discarded, and iii) operation j executes using a different set of input operands (e.g., C and D) and a correct value of R is recomputed.

Fig. 3(b) shows an example where precedences between the conditionals and forks are removed. The critical path length of 6 in the original CDFG is reduced to just 3. All four possible control paths may start executing simultaneously.

3.2. Derivation of Constraints

For brevity, we assume nonpipelined, unit-time operations. Pipelined and multicycle functional units can be accommodated by incorporating execution delay in the equations presented in Sections III.2 and III.3 [31]. To model operation chaining, a precedence relation can be added between operations that cannot be chained [15]. $(ASAP)_j$ (as soon as possible) and $(ALAP)_j$ (as late as possible) bounds are constructed to limit the time spans over which an operation j can be scheduled. These bounds are not required for correctness but improve the efficiency of the construction. C_{sj} denotes operation j's instance at time step s. Fork (join) nodes are not explicitly used in the formulation. Precedences to fork (join) nodes are translated in a transitive fashion to the successor nodes of the fork (join). Symbols "Σ" and "$+$" correspond to Boolean *Or* function, and "Π" stands for Boolean *And*. Product "ab" implies "*a And b*."

1) Uniqueness: Equations 1 enforce unique scheduling of operations from the CDFG at time step s. If $(ASAP)_j \leq s < (ALAP)_j$:

$$\sum_{k \in R_{sj}} \left(C_{kj} \prod_{i \neq k \in R_{sj}} \overline{C_{ij}} \right) + \prod_{i \in R_{sj}} \overline{C_{ij}} = 1 \qquad \text{(1a)}$$

where R_{sj} is the range $[(ASAP)_j \cdots s]$. If time step $s = (ALAP)_j$:

$$\sum_{k \in R_{sj}} \left(C_{kj} \prod_{i \neq k \in R_{sj}} \overline{C_{ij}} \right) + \left(\prod_{i \in R_{sj}} \overline{C_{ij}} \right) \overline{\Gamma}_j = 1. \qquad \text{(1b)}$$

Equation (1a) states that prior to step $(ALAP)_j$, operation j is not scheduled more than once. On step $(ALAP)_j$, (1b) ensures that operation j has been executed on all paths covered by Γ_j. On paths not covered by Γ_j, operation j can be either uniquely scheduled (preexecuted) or not scheduled at all.

The constraint formulated in (1a) can be simplified. An iterative form of (1a) that enforces uniqueness implicitly (by construction) is formulated in the following equation:

$$\overline{C_{sj}} + \left(\prod_{i \in R_{(s-1)j}} \overline{C_{ij}} \right) = 1 \qquad \text{(1c)}$$

where $R_{(s-1)j}$ is the range $[(ASAP)_j \cdots (s-1)]$.

2) Precedence Relations: If operation i precedes operation j (i.e., there is a dependency arc from i to j in the CDFG) and $\Gamma_i \supseteq \Gamma_j$ (Γ_i covers Γ_j) then for every step s in the range $[(ASAP)_j \cdots (ALAP)_i]$ the following must hold:

$$\left(\overline{C_{sj}} + \sum_{ASAP_i \leq l < s} C_{li} \right) = 1. \qquad \text{(2a)}$$

Equation (2a) states that either operation i has to be scheduled before step s, or operation j cannot be scheduled at step s. The case "Γ_i covers (but is not equal to) Γ_j" ($\Gamma_i \supset \Gamma_j$) occurs when the dependency from i to j goes through a fork node. When $\Gamma_i \not\supseteq \Gamma_j$ (Γ_j not contained in Γ_i—e.g., the dependency from i to j goes through a join node), the precedence relation is enforced only on the paths covered by Γ_i:

$$\left(\overline{C_{sj}} + \sum_{ASAP_i \leq l < s} C_{li} \right) + \overline{\Gamma}_i = 1. \qquad \text{(2b)}$$

Effectively, (2a) ensures that the operation can be preexecuted only if all of its predecessors have already been executed. An operation after the join node cannot be preexecuted in our model. Thus (2b), the dependencies to its predecessors are enforced only conditionally.

3) Termination: A single sink variable is used in the OBDD representation to indicate that a particular trace has concluded. It is initialized to "0" and is set to "1" when the terminating condition for the trace is met. Equation (3) is used as a terminating condition for all traces in parallel. The scheduling process can be terminated when *sink* assumes the value "1" on all paths of an ensemble schedule. In these equations, operations $(j_1 \cdots j_n)$ are immediate predecessors of the sink node in the CDFG

$$\prod_{l=1}^{n} (R_{sj_l} + \overline{\Gamma_{j_l}}) = 1, \quad \text{where} \quad R_{sj_l} = \sum_{k=(ASAP)_j}^{s} C_{kj_l}. \qquad \text{(3)}$$

Function R_{sj_l} is true if operation j_l is scheduled prior to or at step s. The fact that execution of j_l is mandatory only on paths covered by Γ_{j_l} is reflected by (3).

4) Resource constraints: If k_l resources of a certain type r_l (e.g., multipliers, adders, ALU's, registers, buses) are available, we formulate a "*generalized resource bound*" (4)

$$\sum_{1\le(l_p\neq l_q)\le n_{sl}} \overline{F_{sl_1}}\,\overline{F_{sl_2}}\cdots\overline{F_{sl_{(n_{sl}-k_l)}}} = 1. \tag{4}$$

F_{sl} is a Boolean function stating that resource r_l is needed during time step s. Equation (4) is applied at each step s for each resource r_l. It ensures that at least $(n_{sl} - k_l)$ resources (among n_{sl} potential candidates at step s) will not be scheduled. For functional units, F_{sl} functions are simply the operation variables. For example, if at step s operation instances $C_{sm_1}, C_{sm_2}, C_{sm_3}$ and C_{sm_4} are candidate multiplications and there are only $k_m = 2$ multipliers available, (4) becomes

$$\overline{C_{sm_1}}\,\overline{C_{sm_2}} + \overline{C_{sm_1}}\,\overline{C_{sm_3}} + \overline{C_{sm_1}}\,\overline{C_{sm_4}} + \overline{C_{sm_2}}\,\overline{C_{sm_3}} + \overline{C_{sm_2}}\,\overline{C_{sm_4}} + \overline{C_{sm_3}}\,\overline{C_{sm_4}} = 1.$$

Equation (4) applies the resource constraint to all traces simultaneously. *Trace validation* (Section III.3) ensures that there are no resource violations in any ensemble schedule. Bus and register constraints are generated for linear schedules by suitable choice of F_{sl} [31].

5) Removal of RedundantlyScheduled Operations: Assume that a conditional has executed, and the "true" branch is selected. Operations from the "false" branch may still be scheduled on the trace corresponding to the "true" branch if there are available resources. Such traces are identified and removed. Assume conditional c_k (whose corresponding guard is G_k) is resolved prior to time step s. Then all the variables that correspond to operation j's instances scheduled for time steps $\geq s$ have to assume value "0" on traces where G_k is true if

$$\Gamma_j G_k = 0. \tag{5a}$$

Similarly, on traces where G_k is false, all the variables that correspond to operation j's instances scheduled for time steps $\geq s$ have to assume value "0" if

$$\Gamma_j \overline{G_k} = 0. \tag{5b}$$

6) Timing Constraints: Since C_{sj} denotes operation j's instance at time step s, it is possible to describe a variety of timing constraints using Boolean functions. For example, assume that operation i precedes operation j and that both of them execute in a single cycle. Furthermore, assume that operation i can be scheduled at steps 1, 2, and 3 (corresponding variables are C_{1i}, C_{2i}, and C_{3i}), and that j can be scheduled at steps 2, 3, and 4 (C_{2j}, C_{3j}, and C_{4j}). Then, a constraint "j has to be scheduled exactly one cycle after i" can be written as

$$C_{1i}C_{2j} + C_{2i}C_{3j} + C_{3i}C_{4j} = 1. \tag{6}$$

Minimum/maximum constraints can be represented similarly. For example, a constraint "j has to be scheduled at least two cycles after i" amounts to a Boolean function

$$C_{1i}C_{3j} + C_{1i}C_{4j} + C_{2i}C_{4j} = 1. \tag{7}$$

An iterative formulation of the constraints is possible as well. For example, (6) can be applied at step s ($s = 2, 3, 4$) using

$$C_{(s-1)i} + \overline{C_{sj}} = 1. \tag{8}$$

Together with the uniqueness constraint (1), (8) enforces the timing constraint implicitly (by construction). If a timing constraint has to be conditionally enforced, a modification similar to that in (2b) is necessary. Since we use arbitrary Boolean functions to represent constraints, more complex timing behavior can also be conveniently described.

The formulation described throughout Section III is also applicable to scheduling without speculative operation execution. Essentially, a control dependency between the conditional and fork node in the CDFG can be enforced as a hard precedence relation. However, a slightly modified set of the constraints is used to improve efficiency [32]. In addition, timing constraints can be used to enforce precedence between the operations and prohibit speculative execution on individual basis. This is because, in our formulation, precedence constraints are simply a special case of timing constraints.

3.3. Trace Validation

A trace satisfying all of the constraints introduced in Section III.2 may still not be valid in the sense that it cannot be a member of any set of traces forming an ensemble schedule. The example CDFG in Fig. 6 demonstrates that resource-constrained scheduling of all individual control paths is not sufficient for a proper treatment of control-dependent behavior. Both the "True" and "False" control paths can be scheduled individually in two time steps assuming one single-cycle resource of each type ("white," "black," comparator). However, observe that the execution traces shown in the figure cannot be combined into an executable schedule meeting the stated resource constraints. Since the decision regarding which path to execute is not known until the end of the first step, the "True" and "False" paths are indistinguishable during that cycle. This means that both *op_1* and *op_5* as well as *op_3* and *op_6* must be executed simultaneously, violating the resource constraint. (A decision to exclusively execute *op_1* and *op_5* or *op_3* and *op_6* depends on knowledge not available until the end of the first cycle!) In fact, no two-cycle schedule is possible, although both control paths can be individually scheduled in two time steps.

A valid *ensemble schedule* is a minimal set of traces that is both *causal* and *complete*. The *causality* requirement dictates that the schedule cannot use knowledge of the value of a conditional prior to the time when the conditional is executed (resolved). *Completeness* requires that a trace must exist for every possible control combination. An ensemble schedule is a minimal set in the sense that if any trace is removed, the set is no longer complete. Assume that the conditional c_k is resolved at step j. Causality requires that the traces corresponding to guard values G_k and $\overline{G_k}$ must be identical (match) for all time steps prior to and including j. Completeness ensures that the ensemble schedule includes traces for both G_k and $\overline{G_k}$.

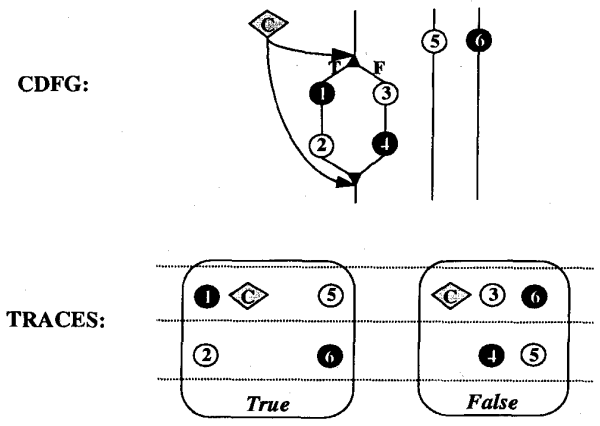


Fig. 6. Ensemble schedule counterexample.

```
i = 0;
do {
    i++;
    S(i) = S(i-1);
    for each time step j {
        S' = ∃(V-V'(j)) S(i)
        for each conditional ck {
            S' = S'Rk(j) + ∀Gk (S'R̄k(j))
            if (S'==0) { S(i)=0; exit; }
        }
        S(i) = S(i)S';
    }
} while (S(i)!=S(i-1));
```

Fig. 7. Trace validation algorithm.

Trace validation ensures that each validated trace is part of some ensemble schedule. The validation is efficiently preformed by the iterative algorithm shown in Fig. 7. The following notation is used:

f_x ($f_{\overline{x}}$)	*positive (negative) cofactor* of a Boolean function f with respect to a variable x,
$\exists_x f = f_x + f_{\overline{x}}$	*existential abstraction* is $\forall_x f = f_x f_{\overline{x}}$ is *universal abstraction*,
$\boldsymbol{S}$	set of all traces; $S(0)$—initial set of non-validated traces; $S(i)$—set of traces at iteration i,
$\boldsymbol{V}$	set of all variables not including guard variables-,
$V'(\boldsymbol{j})$	subset of $\boldsymbol{V}$ corresponding to time steps $\leq j$,
$\boldsymbol{S'}$	set of traces from which all variables ($\boldsymbol{V} - \boldsymbol{V'(j)}$) are removed: $S' = \exists_{(V-V'(j))} S(i)$,
$\boldsymbol{C}$	$= [c_1, c_2 \cdots c_n]$is the set of all conditionals,
$\boldsymbol{G}$	$= [G_1, G_2 \cdots G_n]$ is the set of guards corresponding to the conditionals,
$\boldsymbol{R(j)}$	$= [R_1(j), R_2(j) \cdots R_n(j)]$ is the *resolution vector*.

The *resolution vector* $\boldsymbol{R(j)}$ is a set of n Boolean functions (one for each conditional), where each function $R_k(j)$ indicates whether a conditional c_k was scheduled prior to time step j: $R_k(j) = \Sigma C_{lk}$, for $(l < j)$. S' is partitioned by $\boldsymbol{R(j)}$ into a disjoint set of as many as 2^n families, corresponding to the subset of guards that are resolved prior to time step j ($\boldsymbol{G_{res}}$). The guards from ($\boldsymbol{G} - \boldsymbol{G_{res}}$) (i.e., the unresolved guards) have to be *don't cares* within the family since at time step j there is no knowledge about the future values of the unresolved guards. Traces must both *match* and *exist* for all possible combinations from ($\boldsymbol{G} - \boldsymbol{G_{res}}$), to ensure causality and completeness of the ensemble schedule. The algorithm checks for partial matching up to step j for all traces in parallel. However, it is possible that a trace that matched up to time step j is invalidated in subsequent steps. Thus, its set of matching traces may no longer be complete. The trace validation algorithm iterates until a fixed point is reached. The number of iterations cannot exceed the number of conditionals. Thus, the algorithm generates a polynomial number of constraints regardless of the number of traces.

The intuition behind the trace validation algorithm can be provided by means of the schedule from Fig. 3(b). Assume that the guards G_1 and G_2 correspond to the conditionals 1 and 2. There are four possible control paths: $(G_1G_2, G_1\overline{G_2}, \overline{G_1}G_2, \overline{G_1}\ \overline{G_2})$.At the first step resolution vector components $R_1(1)$ and $R_2(1)$ are both zero since neither conditional is scheduled prior to step 1. To have a causal ensemble schedule, traces for all four control paths must match at the first step. At the next step, $R_1(2)$ is still zero since conditional 1 is not scheduled prior to step 2. However, $R_2(2) = c_{12} = 1$ since conditional 2 is scheduled at step 1. Thus, the matching of traces has to be performed only with respect to conditional 1 (i.e., traces for paths $(G_1G_2, \overline{G_1}G_2)$ must match for the first two steps, as well as the traces for $(G_1\overline{G_2}, \overline{G_1}\ \overline{G_2})$). The same argument holds for step 3.

Trace validation implicitly verifies that the ensemble schedules do not violate resource constraints. We indicated in Section III.2 that (4) prevents such violations from occurring on individual traces. Since traces match before the conditional is resolved, resource bounds are met. After the conditional is resolved, the traces are mutually exclusive with respect to that particular conditional, and no verification is necessary.

3.4. Treatment of Loops

If a loop body does not contain conditional behavior, our formulation can be extended (similar to the ILP technique described in [15]) to incorporate loop optimization techniques such as loop winding and functional pipelining. The resource constraint procedure has to be modified to capture the fact that operations at time steps $s, s+l, s+2l \ldots$ share resources. Variable l represents the *latency* (iteration interval). In the case of loop winding, additional care has to be taken to preserve inter-iteration data dependencies.

The technique can also accommodate the approach to cyclic control adopted in path-based scheduling (i.e., loop cycles are broken, execution is trapped in the last operation of a loop body and, after the scheduling is completed, transitions

TABLE I
RELATION TO ILP

	constraint type	*#solutions*	*#variables*
Symbolic	*any Boolean function*	*all*	**O**[(*#cycles*)∗(*#ops*)] + (*#cond*)
ILP	*linear*	*1*	**O**[(*#cycles*)∗(*#ops*) ∗$2^{(\#cond)}$]

#cycles - number of steps, *#ops* - number of operations, *#cond* - number of conditionals

are added in the control finite state machine). However, the systematic treatment of speculative execution for parallel branching control with cycles is an open research problem.

3.5. Relation to ILP

Table I illustrates some differences between our technique and ILP formulations of resource-constrained control-dependent scheduling. In the symbolic approach, *any Boolean function* can be used as a constraint. Unlike ILP techniques, we can efficiently generate and store *all* feasible solutions to a particular problem instance. More importantly, this requires a very little overhead in terms of formulation variables when compared to the formulation of nonbranching scheduling. In the worst case, the number of variables in our formulation is proportional to the product of the number of time steps and the number of operations in the CDFG. In contrast, an identical problem instance formulated using ILP [6] requires, in the worst case, an exponentially larger number of variables. We observe that conventional ILP techniques [11], [15] essentially do not provide support for control-dependent scheduling. In such approaches, a CDFG operation has to be scheduled on the same cycle on all appropriate control paths.

IV. CONSTRUCTION

The constraints described in Section III have a simple and regular structure [31]. This allows OBDD representations to be constructed *directly* from the CDFG without reference to an intermediate equation form. Shown in Fig. 8 is the OBDD representation of (4). It is used as a general construction template for all of the typed resource constraints. Note that the number of product terms in a sum-of-products representation of (4) is $\binom{n}{k}$. However, its OBDD form is compact ($O(nk)$ nodes) and can be built efficiently using *ite*, [3] (if-then-else) calls. Vertices in this if-then-else template are not restricted to Boolean variables—complex Boolean functions $(f_1, f_2, \cdots f_n)$ can be inserted into the template (e.g., bus/register constraints, formulated in [31]).

However, even when $(f_1, f_2, \cdots f_n)$ are rather simple, the overall constraint may become extremely large. Consequently, it can happen that the partial scheduling solution is of moderate size, but the constraint to be applied is prohibitively large. However, the scheduling constraint need not be explicitly built [33]. The following can be done instead:

1) Introduce a new set of auxiliary variables $(y_1, y_2, \cdots y_n)$ corresponding to the set of functions $(f_1, f_2, \cdots f_n)$.

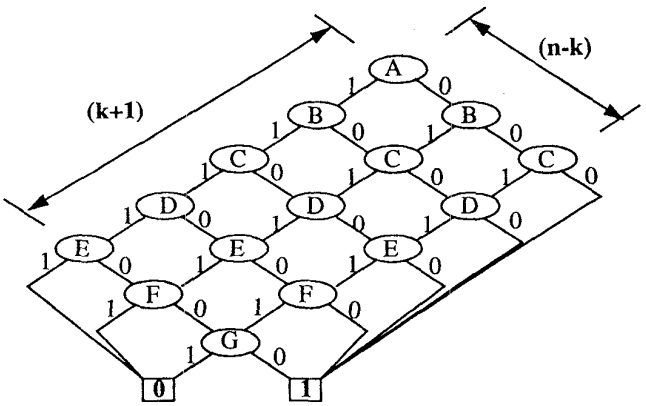


Fig. 8. At-most-k-of-n constraint ($k = 4, n = 7$).

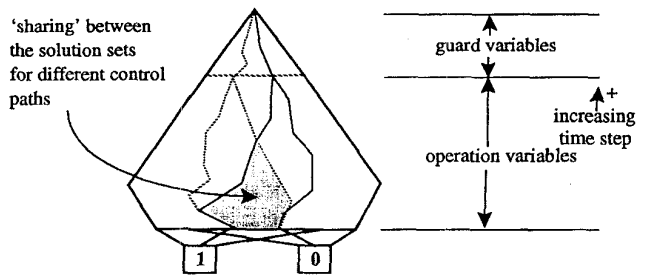


Fig. 9. OBDD representation.

2) Build the template function T (shown in Fig. 8) using only $(y_1, y_2, \cdots y_n)$.
3) Compute $P^0 = And(P', T)$, where P' is a partial solution to which the constraint is applied.
4) Clearly, a new partial solution P'' can be obtained using the recursive formula

$$P^i = \exists y_i [And(P^{(i-1)}, Xnor(y_i, f_i))]$$

where $\exists_x f = f_x + f_{\overline{x}}$. This amounts to the standard BDD substitution operation

$$P^{(i)} = P^{(i-1)}|_{y_i \equiv f_i}. \tag{9}$$

Using this approach, in the benchmarks discussed in Section V.1, we were able to apply register constraints that could not be built explicitly because of memory limitation.

Although individual equations have efficient orderings, optimal orderings for different equations frequently contradict. (In fact, optimal OBDD variable ordering problem is known to be NP-complete [4], [10], [21].) However, experimental results indicate that typical instances do have good orderings. The results presented in this paper are generated using the variable ordering shown in Fig. 9, where nonguard variables are ordered by increasing time step, and guard variables are placed on top (i.e., closest to the root of OBDD). This ordering typically results in small OBDD's and accommodates iterative construction.

Using *iterative* construction, the solution is built on a time-step by time-step basis; only those constraints relevant to a particular time step s are generated and applied to the

TABLE II
EWF EXPERIMENTS

#cycles	17	17	18	18	19	20	20	21	28	28
#adders	3	3	3	2	2	2	2	2	1	1
#multipliers	2(*)	3	1(*)	2	1(*)	2	1(*)	1	1(*)	1
#buses	6	6	6	6	6	4	4	4	4	4
#registers	10	10	10	10	10	10	10	10	10	10
#variables	63	63	97	97	131	165	165	199	437	437
#nodes	82	82	194	209	2,237	2,760	1,905	704	4.9e4	3.2e4
#schedules	18	18	336	18	1.1e4	5.3e4	5,142	2,355	4.3e9	2.6e8
CPU time [s]	0.2	0.2	0.5	0.6	3.4	14.0	12.5	3.5	624.7	391.5

2-cycle multiplier and single-cycle adder except: (*) 2-cycle pipelined multiplier

TABLE III
EWF WITH LOOP WINDING

	non-pipelined multiplier				*pipelined multiplier*				
#multipliers	3	2	2	1	3	2	2	1	1
#adders	3	3	2	2	3	3	2	3	2
latency	16	16	17	19	16	16	17	16	17
delay	18	18	19	21	18	18	19	18	19
#variables	97	97	131	199	97	97	131	97	131
#nodes	776	465	689	1,788	799	776	878	258	189
#schedules	2,055	674	108	19,498	2,160	2,055	144	77	19
CPU [s]	4.0	1.2	9.2	7.1	4.4	4.0	9.4	0.5	3.5

TABLE IV
BENCHMARKS WITH BRANCHING

		Maha	*Parker*	*Kim*	*Waka*	*MulT*
#cycles(spec)	longest:	5	4	4	6	7	3
	average:	3.31	2.25	2.13	5.75	5.0	3.0
#cycles(non_spec)		8	8	8	8	7	4
#adders		1	2	2	2	1	2
#subtracters		1	3	3	1	1	1
#comparators		-	-	-	1	2	1
#variables		65	49	49	71	55	26
#nodes		428	325	220	543	271	116
#traces		15	43	12	124	21	15
CPU time [s]		5.9	3.6	4.7	4.1	2.0	3.3

single-cycle adders, subtracters and comparators assumed

TABLE V
COMPARISON WITH OTHERS: AVERAGE (LONGEST) PATH

	Maha		*Parker*	*Kim*	*Waka*	*MulT*
our	3.31 (5)	2.25 (4)	2.13 (4)	5.75 (6)	5 (7)	3 (3)
TS [13]	3.31 (5)	-	-	-	4.75 (7)	-
CVLS [43]	3.31 (5)	2.38 (4)	2.38 (4)	5.75 (6)	-	2.88 (4)
Kim *et al.* [17]	4.62 (8)	-	-	6.25 (7)	4.75 (7)	-

TABLE VI
S2R EXPERIMENTS

execution_type		*#cycles*	*CPU_time* [s]
speculative	parallel	8	108.8
	serial	10	177.9
non_speculative	parallel	11[a]	56.2
	serial	16[b]	13.5

- memory constraint: 1 single-port look-up table
- pipelined control delay = 2 cycles
- resource constraints: 3 single-cycle ALUs (+,-), 2 two-cycle pipelined multipliers

a. can be achieved with 2 single-cycle ALUs as well (55.2 s)
b. can be achieved with 1 single-cycle ALU as well (12.4 s)

OBDD representing a valid partial solution for the previous $(s-1)$ steps. This prevents the construction of large sets of spurious intermediate solutions. It also has the advantage that schedule completion can be easily detected, obviating the need to accurately prespecify a bound on the number of control steps.

To verify the existence of an ensemble schedule, trace validation *must* be done when a termination test is performed on a set of traces that have concluded execution. Note that this set is typically significantly smaller than the set of all traces in the intermediate solution. To reduce the number of traces (and thus, potentially reduce the intermediate OBDD size), it is possible to perform trace validation at the end of every iteration. This can, however, lead to somewhat increased CPU time and more intensive garbage collection. We enforced trace validation only when the intermediate OBDD size exceeded a prespecified threshold. Similarly, the uniqueness constraint for operation j can be applied just at step $(ALAP)_j$ (i.e., application of only (1b) is sufficient). This diminishes the number of constraints that have to be applied and typically increases the speed of construction. However, note that the speed-up techniques described in this paragraph may produce intermediate solutions that temporarily contain invalid traces.

V. EXPERIMENTAL RESULTS

The technique described in the paper was implemented in C++ and executed on a Sun SPARCstation10. Reported CPU times correspond to the complete procedure: CDFG analysis, constraint construction, and all OBDD manipulations leading to the reported results. We apply the technique to three typical problem types. Table II is an application to scheduling of acyclic DFG's. Table III demonstrates the ability of our technique to perform loop winding on cyclic DFG's. Finally, Tables IV–VI discuss the scheduling of acyclic CDFG's. The results are compared to the optimal or best known results. No other work reports competitive results for all three problem types.

5.1. Acyclic DFG's

Table II summarizes the *elliptic wave filter* (***EWF***) benchmark experiments. We found *all* optimal solutions of each instance using OBDD's whose size was significantly smaller than $(\#variables)^2$. To reduce the size of partial solutions, an auxiliary set of "*interior constraints*" was generated [33]. The basic strategy is as follows: Assume that at the beginning of step s there are n addition operations that have ALAP bounds in the range $[s \cdots (s+k-1)]$ and that there are only m single-cycle adders available. Clearly, at least $(n-km)$ of these addition operations must be completed prior to step s in a feasible solution. This observation enables early detection of many (not necessarily all) partial schedules that are destined to be discarded within the next k steps. Similar constraints can be applied for each functional unit type including multicycle and

pipelined units. A further improvement in runtime efficiency is possible if execution interval analysis [41], [42] is used for search space reduction. Interior constraints can be viewed as a subset of such analysis.

Some problems may have extremely large solution sets, decreasing the efficiency of OBDD manipulations. Nevertheless, since valid partial schedules are available after each construction step, runtime-efficient heuristics *based on sets* can be devised. For example, we can propagate only the subset of schedules with maximum utilization of resources at each step (*utility-based* heuristic [32], [33]). Since *all* such schedules are propagated, this heuristic has good behavior and is applicable to problems with thousands of formulation variables. Using this technique, problems with 102 operations in DFG, 5919 formulation variables, and 105 cycles have been solved. The largest benchmark instances in Table II (28-cycle EWF's) run in less than 15 CPU s, while still finding minimum-latency schedules. Moreover, if the exact scheduler is based on *Zero-Suppressed BDD's* [25], significant improvements in terms of both CPU time and memory usage are observed. The interested reader is referred to the recent experimental study [33] where larger DFG's (*EWF* unfolded two and three times, *FDCT* [23]) are scheduled. An *FDCT* instance with one adder, one subtracter, one pipelined two-cycle multiplier, and four buses is frequently used to evaluate schedulers. We are able to solve this problem that includes 565 formulation variables both heuristically (60 CPU s) and exactly (523 CPU s). A randomly selected optimal nineteen-cycle schedule (using nine registers) is shown in Fig. 10. To our knowledge, no exact technique has reported a solution to this problem instance, and the best heuristic results so far required latency (iteration interval) of twenty cycles [20]. Furthermore, it took us less than 100 CPU s to verify infeasibility of a ten-cycle *FDCT* instance with five two-cycle nonpipelined multipliers and three ALU's. Due to its very large symmetric search space, this *FDCT* instance is reported to be an extremely hard problem [42]—both ILP and branch-and-bound techniques take more than two CPU h to reach the infeasibility conclusion.

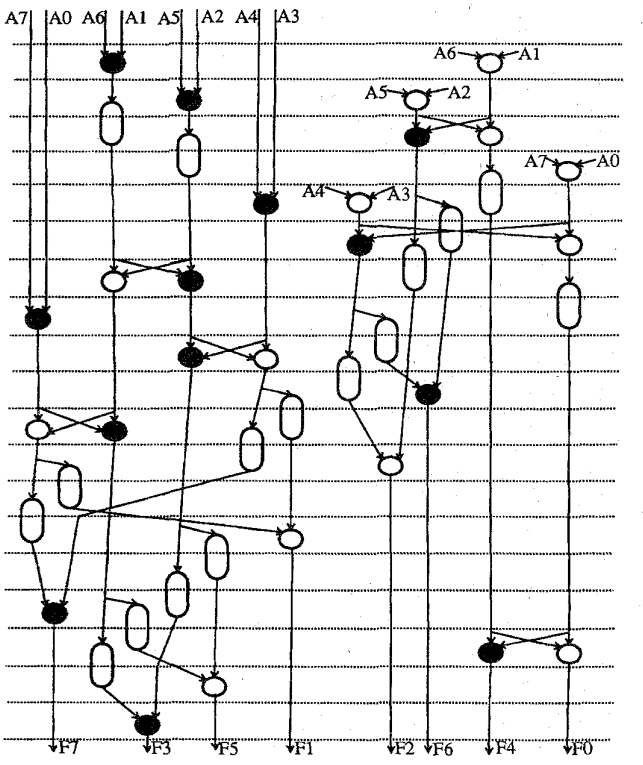


Fig. 10. Nineteen-cycle FDCT with pipelined multiplier.

5.2. Cyclic DFG's

Loop winding results for *EWF* are indicated in Table III. All optimal schedules, both in terms of latency (iteration interval) and delay (iteration time), are constructed using very moderate computing resources. Several ILP techniques (e.g., [12] and [15]) report results equivalent to those presented in Tables II and III, with the difference that we provide all optimal schedules.

Direct comparison of CPU times is misleading due to machine differences and to the fact that only ILP execution times without preprocessing are typically reported. Similarly, the efficient branch-and-bound technique [42] does not report the time for execution interval analysis.

5.3. Acyclic CDFG's

Tables IV and V show experimental results for benchmarks exhibiting conditional behavior. The rows *#cycles(spec)* and *#cycles(non_spec)* correspond to scheduling with and without speculative execution using the same set of resources and demonstrate the benefits of performing such code motion. The scheduler terminates when all minimum-latency ensemble schedules are found. The number of cycles for the longest control path is indicated as "*longest*." To compare our results with schedulers that minimize average path length, a subset of solutions with small average path length is generated in a greedy fashion. Benchmarks *Maha* [27], *Kim* [17], and *Waka* [44] are conditional trees, and *MulT* [43] has two parallel trees. *Parker* is *Maha* with addition *A6* converted into a subtraction. Our results are compared to the best published results. The *Maha* solution with one adder and one subtracter is the same as in [13] and [43]. Allowing more resources (two adders, three subtracters), an improvement of 0.125 (average path length) is made over the best previous result. In *Parker*, this improvement was 0.25. In most previous work, it is assumed that the comparators incur a small delay within a clock cycle and that the operations following the branch on "true" and "false" paths are mutually exclusive during the *same* cycle. This treatment of the conditionals requires increased cycle time, additional multiplexing, and restricts pipelining of the control. Our results reflect this model in *Maha* and *Parker* only, but this assumption completely eliminates the need for speculative execution in the *Kim* and *Waka* benchmarks. By default, we assume that a single-cycle comparator is used and that its output becomes available for control only in the successive cycle. Even with this assumption, our technique still derives the same result for *Kim* as in [43]. In *Waka*, one path is a cycle longer than that reported in [13]. In *MulT*, a one cycle

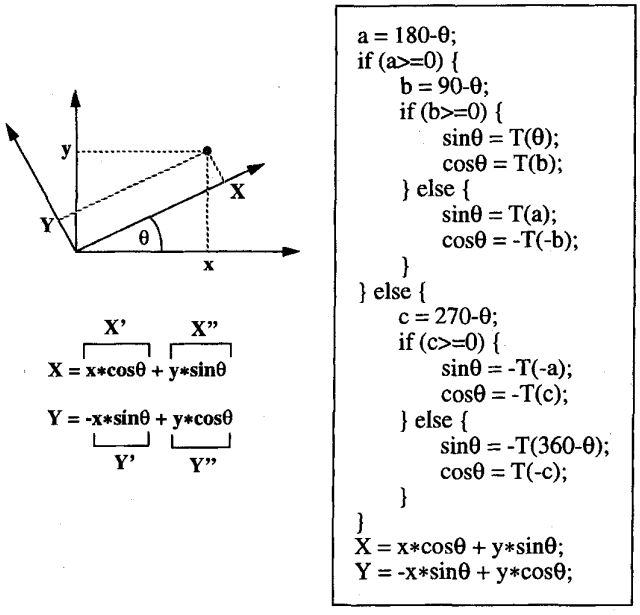
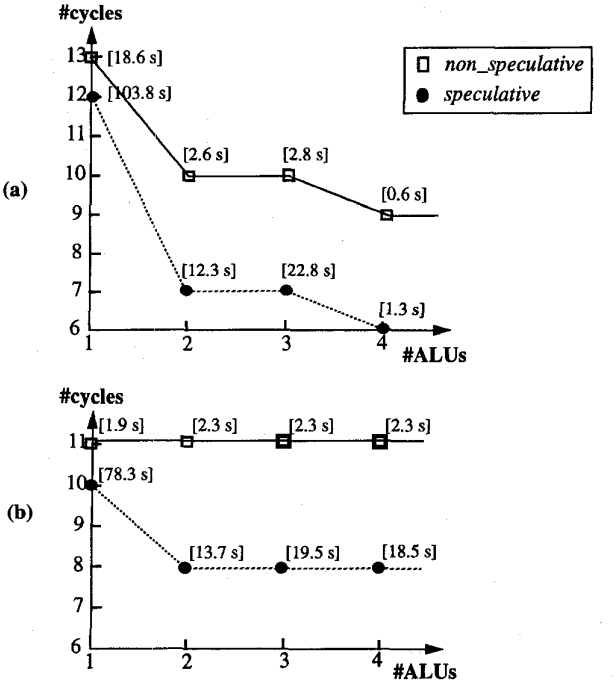


Fig. 11. ROTOR example.



- memory constraint: 1 single-port look-up table
- pipelined control delay = 2 cycles
- resource constraints:
(a) single-cycle ALU (+, -, *)
(b) single-cycle ALU (+, -), 2 two-cycle pipelined multipliers

Fig. 12. ROTOR experiments.

shorter minimum-latency solution was found by exploiting dynamic scheduling of operations belonging to parallel trees. There is no information on execution times for the results reported in [13], [17], and [43].

The *ROTOR* example (Fig. 11) performs a rotation of coordinate axes by angle θ. This transformation is used in many applications (e.g., graphics applications and positional control systems). The example requires computation of trigonometric functions ($\sin\theta$ and $\cos\theta$). In high-performance applications, a typical approach is to precompute the value of sine and cosine functions and store the sampled values in corresponding tables. However, if high numerical accuracy is required, the size of the storage tends to become rather large. A compromise approach amounts to storing values for only a quadrant of one trigonometric function (e.g., sine values for arguments $0° \leq \Theta \leq 90°$). It is straightforward to use such a look-up table for obtaining values for both sine and cosine for all possible input arguments ($0° \leq \Theta \leq 360°$).

A pseudocode description of the coordinate rotation using only the first quadrant of the sine function is presented in Fig. 11. "T(angle)" corresponds to a table read at a location "angle." Similarly, "-T(angle)" corresponds to a table read followed by a negation. We assume that only one single-port look-up table is available and that every "read table" takes one cycle to complete. Although it is possible to simultaneously perform subtraction and comparison of two operands, in the example, we assume *pipelined control*, which introduces a two-cycle delay. For example, if operation $(a = 180 - \theta)$ is executed at step s, result a is available at the beginning of step $(s+1)$, but control flow is affected by the comparison at the beginning of step $(s+2)$.

To simplify interpretation of the results, in Fig. 12(a), we assume that the available ALU's can perform all arithmetic/logical operations (add, subtract/negate, multiply) in a single cycle. The minimum number of cycles to execute the schedule is presented for cases with and without speculative execution. We observe that, given the same resource constraints, speculative execution enables much faster schedules. In Fig. 12(b), a more realistic assumption is made. Single-cycle ALU's perform addition and subtraction. Multiplication is performed by two two-cycle pipelined multipliers. In this case, adding more ALU's cannot improve the performance unless speculative execution is allowed. In Fig. 12, CPU run-times are indicated in brackets. By allowing speculative execution, an average improvement in minimum latency of 25% is achieved using the same resources.

Fig. 13 shows an eight-cycle ensemble schedule (two ALU's, Fig. 12(b)). Operations executed in a speculative fashion are represented using thick lines. If the input angle Θ belongs to the first quadrant, the computation is performed in seven cycles. However, since all ensemble schedules are implicitly encapsulated in an OBDD, the user can search for solutions having other properties. It is relatively straightforward to look for similarities among the traces in order to simplify the control. For example, if the first-quadrant computation takes eight cycles as well, it is possible to have the same schedule for operations X', X'', Y', Y', X, and Y for all control paths during the fifth, sixth, seventh, and eighth cycles. This sort of design space exploration can be performed without rescheduling the problem instance.

In Fig. 14, we introduce the $S2R$ example that translates spherical coordinates $[R, \Theta, \Phi]$ into the Cartesian (rectangular)

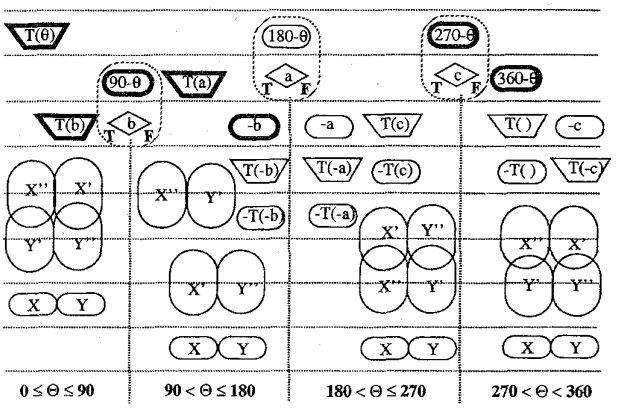


Fig. 13. Eight-cycle ROTOR schedule.

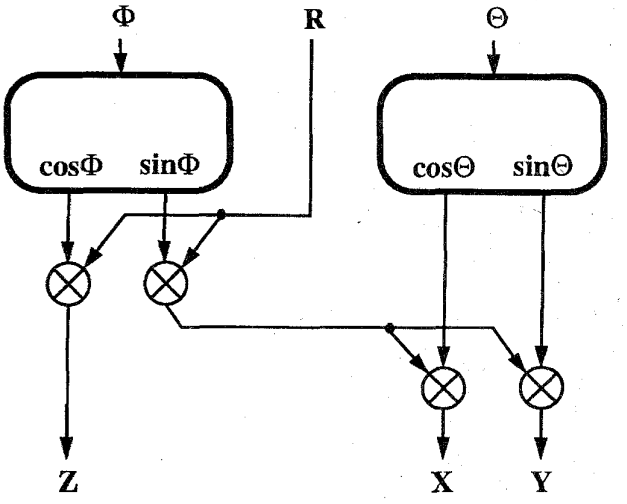


Fig. 14. S2R example.

coordinate values $[X, Y, Z]$. The problem includes computation of trigonometric functions (as described in the *ROTOR* example) for two input angles (Θ, Φ). There are 42 operations in the CDFG representation and, if executed in a speculative fashion, as many as 64 execution paths. If a single-port look-up table is used, the scheduling of parallel trees (corresponding to computations for Θ and Φ) has to be done *simultaneously*. This means that the schedule *guarantees* synchronization of the memory accesses without busy/waiting hardware handshaking.

Shown in Table VI (*#cycles*) are *S2R* latencies using one single-port look-up table, three ALU's and two two-cycle pipelined multipliers. All solutions are exact and correspond to execution with and without speculative execution. In each case, two values are included. An unconstrained version ("*parallel*") allows both trees to be scheduled and executed in parallel. For comparison, we provide the latencies for a "*serial*" version of the problem that imposes an execution order (Φ-tree executed before Θ-tree). The results clearly indicate the benefit from being able to schedule parallel computations in a speculative fashion. None of the results can be further improved by increasing hardware resources.

VI. Conclusions and Future Work

We describe a symbolic formulation that allows speculative operation execution and exact resource-constrained scheduling of arbitrary forward-branching control/data paths. To our knowledge, no other work has been reported on exact techniques supporting speculative execution. The presented technique provides a closed-form solution set in which all satisfying schedules are encapsulated in a compressed OBDD-based representation. An advantage of the formulation is that there is no need to explicitly describe freedom present in the input CDFG description. The execution order of conditionals is not predetermined and is dynamically resolved allowing gains in scheduling quality. To allow a systematic treatment of the problem, a flexible control representation based on guard variables, guard functions, and traces is introduced. The trace validation algorithm is proposed to enforce causality and completeness of the solution set. An iterative construction method is presented along with benchmark results. The results demonstrate the ability of the technique to efficiently exploit operation-level parallelism implicit in the input description.

In future work, several related synthesis issues are to be addressed. These include incorporation of control/interconnect costs and extensions to allow a more complex mapping from operation types to function units. Also, extensions to general forms of cyclic control and an efficient approach to remove the restriction from the current speculative execution model are planned. Finally, to further improve the efficiency, additional work is needed to identify tighter operation bounds for the control-dominated case.

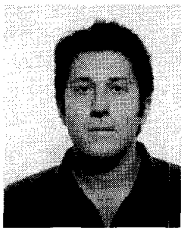
Acknowledgment

The authors would like to gratefully acknowledge contributions from Dr. A. Seawright who took part in early discussions and developed the original C++ OBDD package extensively used throughout this project. Their special thanks go to A. Crews, C. Monahan, and A. Stornetta for recent efficiency improvements while reimplementing the package. Finally, they would like to thank the reviewers for helping to clarify the presentation of this paper.

References

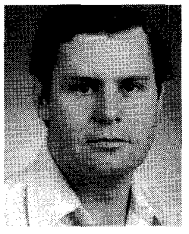
[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proc. 10th Annual ACM Symp. Principles of Programming Languages,* pp. 177–189, Jan. 1983.
[2] R. A. Bergamaschi, R. Camposano, and M. Payer, "Allocation algorithms based on path analysis," *Integration, the VLSI J.,* vol. 13, no. 3, pp. 283–299, Sept. 1992.
[3] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Proc. 27th ACM/IEEE Design Automation Conf.,* pp. 40–45, 1990.
[4] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.,* vol. C-35, pp. 677–691, Aug. 1986.
[5] R. Camposano, "Path-based scheduling for synthesis," *IEEE Trans. CAD/ICAS,* vol. 10, pp. 85–93, Jan. 1991.
[6] C. N. Coelho, Jr. and G. De Micheli, "Dynamic scheduling and synchronization synthesis of concurrent digital systems under system-level constraints," in *Proc. IEEE Int. Conf. Computer-Aided Design,* pp. 175–181, 1994.
[7] G. De Micheli, *Synthesis and Optimization of Digital Circuits.* New York: McGraw-Hill, 1994.

[8] J. C. Denhert and R. A. Towle, "Compiling for the Cydra 5," *J. Supercomputing,* vol. 7, pp. 181–227, Jan. 1993.
[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Programming Languages and Syst.,* vol. 9, no. 3, pp. 319–349, July 1987.
[10] S. J. Friedman and K. J. Supowit, "Finding the optimal variable ordering for binary decision diagrams," *IEEE Trans. Comput.,* vol. 39, pp. 710–713, May 1990.
[11] C. H. Gebotys and M. I. Elmasry, "Global optimization approach for architectural synthesis," *IEEE Trans. CAD/ICAS,* vol. 12, pp. 1266–1278, Sep. 1993.
[12] C. H. Gebotys, "Throughput optimized architectural synthesis," *IEEE Trans. VLSI Syst.,* vol. 1, no. 3, pp. 254–261, Sept. 1993.
[13] S. H. Huang *et al.,* "A tree-based scheduling algorithm for control dominated circuits," in *Proc. 30th ACM/IEEE Design Automation Conf.,* pp. 578–582, 1993.
[14] C.-T. Hwang and Y.-C. Hsu, "Zone scheduling," *IEEE Trans. CAD/ICAS,* vol. 12, pp. 926–934, July 1993.
[15] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Trans. CAD/ICAS,* vol. 10, pp. 464–475, Apr. 1991.
[16] T. Y. K. Kam and R. K. Brayton, "Multi-valued decision diagrams," Univ. California, Berkeley, Memo UCB/ERL M90/125, Dec. 1990.
[17] T. Kim, J. W. S. Liu, and C. L. Liu, "A scheduling algorithm for conditional resource sharing," in *Proc. IEEE Int. Conf. Computer-Aided Design,* pp. 84–87, 1991.
[18] H. Komi, S. Yamada, and K. Fukunaga, "A scheduling method by stepwise expansion in high-level synthesis," in *Proc. IEEE Int. Conf. Computer-Aided Design,* pp. 234–237, 1992.
[19] D. Ku and G. De Micheli, "Relative scheduling under timing constraints," in *Proc. 27th ACM/IEEE Design Automation Conf.,* pp. 59–64, 1990.
[20] T.-F. Lee, A. C.-H. Wu, Y.-L. Lin, and D. D. Gajski, "An effective methodology for functional pipelining," *IEEE Trans. CAD/ICAS,* vol. 13, no. 34, pp. 439–450, Apr. 1994.
[21] H.-T. Liaw and C.-S, Lin, "On OBDD-representation of general Boolean functions," *IEEE Trans. Comput.,* vol. 41, pp. 661–664, June 1992.
[22] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," in *Proc. 22th Int. Symp. Computer Architecture,* pp. 138–150, June 1995.
[23] D. J. Mallon and P. B. Denyer, "A new approach to pipeline optimization," in *Proc. European Design Automation Conf.,* pp. 83–87, 1990.
[24] M. C. McFarland, A. C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proc. IEEE,* vol. 78, pp. 301–318, Feb. 1990.
[25] S.-I. Minato, "Zero-suppressed BDD's for set manipulation in combinatorial problems," in *Proc. 30th ACM/IEEE Design Automation Conf.,* pp. 272–277, 1993.
[26] B. M. Pangrle and D. D. Gajski, "Slicer: A state synthesizer for intelligent compilation," in *Proc. IEEE Int. Conf. Comput. Design,* pp. 42–45, 1987.
[27] A. C. Parker, J. T. Pizarro, and M. Mliner, "MAHA: A program for datapath synthesis," in *Proc. 23th ACM/IEEE Design Automation Conf.,* pp. 461–465, 1986.
[28] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Trans. CAD/ICAS,* vol. 8, pp. 661–679, June 1989.
[29] R. Potasman, J. Lis, A. Nicolau, and D. Gajski, "Percolation based synthesis," in *Proc. 27th ACM/IEEE Design Automation Conf.,* pp. 444–449, 1990.
[30] M. Potkonjak and J. Rabaey, "Optimizing resource utilization using transformations," *IEEE Trans. CAD/ICAS,* vol. 13, no. 3, pp. 277–292, Mar. 1994.
[31] I. Radivojević and F. Brewer, "Symbolic techniques for optimal scheduling," in *Proc. 4th SASIMI Workshop,* Nara, Japan, pp. 145–154, 1993.
[32] ——, "Ensemble representation and techniques for exact control-dependent scheduling," in *Proc. 7th Int. Symp. High Level Synthesis,* pp. 60–65, 1994.
[33] ——, "On applicability of symbolic techniques to larger scheduling problems," in *Proc. European Design and Test Conf.,* pp. 48–53, 1995.
[34] ——, "Analysis of conditional resource sharing using a guard-based control representation," in *Proc. IEEE Int. Conf. Comput. Design,* pp. 434–439, 1995.
[35] B. R. Rau and J. A. Fisher, "Instruction-level parallel processing: History, overview, and perspective," *J. Supercomputing,* vol. 7, pp. 9–50, Jan. 1993.
[36] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental computer: Design philosophies, decisions and trade-offs," *IEEE Comput.,* vol. 22, pp. 12–34, Jan. 1989.
[37] M. Rim, Y. Fan, and R. Jain, "Global scheduling with code motions for high-level synthesis applications," *IEEE Trans. VLSI,* vol. 3, no. 3, pp. 379–392, Sept. 1995.
[38] E. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Trans. Comput.,* pp. 1405–1411, Dec. 1972.
[39] A. Takach and W. Wolf, "Scheduling constraint generation for communicating processes," *IEEE Trans. VLSI Systems,* vol. 3, no. 2, pp. 215–230, June 1995.
[40] A. Takach, W. Wolf, and M. Leeser, "An automaton model for scheduling constraints in synchronous machines," *IEEE Trans. Comput.,* vol. 44, pp. 1–12, Jan. 1995.
[41] A. H. Timmer and J. A. G. Jess, "Execution interval analysis under resource constraints," in *Proc. IEEE Int. Conf. Computer-Aided Design,* pp. 454–459, 1993.
[42] ——, "Exact scheduling strategies based on bipartite graph matching," in *Proc. European Design and Test Conf.,* pp. 42–47, 1995.
[43] K. Wakabayashi and H. Tanaka, "Global scheduling independent of control dependencies based on condition vectors," in *Proc. 29th ACM/IEEE Design Automation Conf.,* pp. 112–115, 1992.
[44] K. Wakabayashi and T. Yoshimura, "A resource sharing and control synthesis method for conditional branches," in *Proc. 26th ACM/IEEE Design Automation Conf.,* pp. 62–65, 1989.
[45] W. Wolf, A. Takach, C.-Y. Huang, R. Manno, and E. Wu, "The Princeton University behavioral synthesis system," in *Proc. 29th ACM/IEEE Design Automation Conf.,* pp. 182–187, 1992.
[46] J. C.-Y. Yang, G. De Micheli, and M. Damiani, "Scheduling and control generation with environmental constraints based on automata representations," *IEEE Trans. CAD/ICAS,* to appear.
[47] L. Yang and J. Gu, "A BDD model for scheduling," in *Proc. CCVLSI,* 1991.
[48] T.-Y. Yen and W. Wolf, "Optimal scheduling of finite-state machines," in *Proc. IEEE Int. Conf. Computer Design,* pp. 266–369, 1993.



Ivan P. Radivojević received the B.S.E.E. degree from the University of Belgrade, Yugoslavia, in 1987 and the M.S.E.E. degree from Drexel University, Philadelphia, PA, in 1990.

From 1987 to 1989, he worked as a Research Engineer at the University of Belgrade, where he contributed to the design of numerous microprocessor and DSP-based real-time systems. During the 1990–91 academic year, he was a Teaching Fellow at the ECE Department, Drexel University, Philadelphia. Currently, he is a Ph.D. candidate at the ECE Department, University of California, Santa Barbara. His research interests include high-level synthesis, logic design, and hardware/software issues in superscalar and VLIW architectures.



Forrest Brewer received the Bachelor of Science degree with honors in physics from the California Institute of Technology, Pasadena, in 1980 and the M.S. and Ph.D. degrees in computer science in 1985 and 1988, respectively, from the University of Illinois, Urbana-Champaign.

Since 1988, he has served as an Assistant Professor with the University of California, Santa Barbara. From 1981 to 1983, he was a Senior Engineer at Northrop Corporation and consulted there until 1985. He coauthored Chippe, which was the first closed loop high level synthesis system. Recently, his research work has been in the application of logic synthesis techniques to high level synthesis, specification and scheduling of control-dominated designs.