Symbolic Scheduling Techniques [1]

Ivan Radivojević, Forrest Brewer

Department of Electrical and Computer Engineering

University of California

Santa Barbara, CA 93106, U.S.A.

E-mail: ivan@aurora.ece.ucsb.edu, forrest@ece.ucsb.edu

Phone: (805) 893-8043 Fax: (805) 893-3262

Index terms: Computer Hardware and Design (Special Issue on SASIMI'93)

October 1994.

1 This work was sponsored in part by fellowship donation from Mentor Graphics Corp. and UC-MICRO under project No.92-019.

ABSTRACT -- This paper describes an exact symbolic formulation of resource-constrained scheduling which allows speculative operation execution in arbitrary forward-branching control/data paths. The technique provides a closed-form solution set in which all satisfying schedules are encapsulated in a compressed OBDD-based representation. An iterative construction method is presented along with benchmark results. The experiments demonstrate the ability of the proposed technique to efficiently extract parallelism not explicitly specified in the input description.

1. INTRODUCTION

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 constraints. Priority-based heuristic scheduling [1][2][3] accommodates a variety of control-dependent behaviors, but can fail to find an optimal solution in tightly constrained problems. Applicability of exact ILP methods [4] has been improved by complex remapping of the constraints [5], but current ILP formulations cannot handle complex control-dependent behavior. In a mixed ILP/BDD method [6] data dependencies are captured in an *OBDD* (Ordered Binary-Decision Diagram [7]) form to simplify the ILP execution, but the question of control is not addressed.

Since operation parallelism may not be explicit in the input description, many synthesis systems focus on detection of *mutual exclusiveness* in control/data-flow graphs (CDFGs). Tree scheduling (TS) [8] uses a tree-representation of the execution paths to enable movement of operations. Conditional vector list scheduling (CVLS) [9] uses *condition vectors* [10] 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 [11], but there is no support for speculative execution. These heuristics are restricted to nested conditional branches (conditional tree control structure). Multiple conditional trees are addressed by Wakabayashi [9], but the trees are either scheduled sequentially (using a priority scheme) or conditional tree duplication is performed.

An exact *symbolic* formulation of the resource-constrained scheduling problem was introduced in [12]. Unlike other techniques in which a single representative solution is generated, in this technique *all* feasible schedules are encapsulated in a compressed OBDD form. This is advantageous since the exact effect of additional constraints derived during subsequent synthesis steps is *incrementally* computable. An alternative symbolic formulation using finite automata to capture resource/timing/synchronization constraints is presented in [13].

In this paper, we describe a technique for exact scheduling of arbitrary forward-branching control structures. The technique provides support for speculative operation execution and global treatment of parallel control structures. The paper is organized as follows. In Section 2, we describe several features desirable to improve scheduling quality. The formulation is presented in Section 3. Aspects related to the OBDD construction process are considered in Section 4. Experimental results are discussed in Section 5.

2. ADVANCED SCHEDULING FEATURES

Speculative operation execution -- It is often beneficial to determine the control value simultaneously with branch execution. If sufficient resources are available, operations from both 'true' (T) and 'false' (F) branch paths may be scheduled for execution (pre-executed) before the corresponding conditional value is actually evaluated. A *conditional* is a scheduled operation that generates a control value. Fig. 1(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 five time steps are necessary to execute the CDFG, since the longest dependency chain includes five operations. However, if precedence between the conditional and the fork node is removed, operations from the branch arcs can be pre-executed [2]. Fig.1(b) shows a schedule executing in three cycles using the indicated 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 resolved *after* some other nested conditional. A simple example of this behavior is shown in Fig.1(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* [8] and *CVLS* [9] 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. However, we observe that none of the standard benchmarks (solved in Section 5) need out-of-order execution of the conditionals in minimum-latency solutions.

Irredundant operation scheduling -- Another way to improve scheduling quality is to identify operations that are not redundant in the input description, yet are redundant for some control paths. The importance of such information has been observed and discussed in the literature [8][10].

2 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). Unfortunately, this can lead to an explosion of operator instances for nested complex control.

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.2, a CDFG is shown in which two parallel trees have a correlated control (shaded comparator). The enthusiastic reader can verify that given one adder ('white' operation), one subtracter and one comparator (single-cycle units assumed) a 6-cycle schedule can be found only if the control correlation is properly interpreted (i.e. 'false' paths are not scheduled). As indicated in Fig.2, speculative execution (and additional or more versatile resources) can further improve the execution time.

3. FORMULATION

In this section, an exact scheduling technique supporting all the behaviors described above is formulated. 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. 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 function Γ_j (defined on the guard variables) encodes all the control paths on which *j must* be scheduled.

Shown in Fig.3 is Kim's example [11] in which two guards (G_1, G_2) encode the conditional behavior. There are three possible execution paths: $(G_1G_2, G_1\overline{G_2}, \overline{G_1})$. Indicated blocks $(1, G_1, G_1G_2, G_1\overline{G_2}, \overline{G_1})$ correspond to operations that share the same guard function Γ. Operations which must be scheduled on all control paths have Γ=1. Note that the number of guard variables is not proportional to the number of control paths. In Fig.2, only five guard variables encode 18 control paths.

A solution is 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 both the guard variables (identifying a control path) and operation variables (indicating a schedule for the path). For example, in Fig.3, 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 Γ=1 must be scheduled on that trace. If other operations are scheduled on this trace, they are pre-executed. 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 3.3. Here we just state that the solution OBDD includes only traces belonging to at least one ensemble schedule. Note that the number of ensemble schedules can be much larger than the number of traces.

3.1. Speculative execution model

To incorporate pre-execution, only the control precedence between the conditional and join node is enforced [14]. 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 can be executed at most once regardless of the actual control decisions made when a schedule is executed. Fig.1(b) shows an example where precedences between the conditionals and forks are removed. The critical path length of 5 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 non-pipelined, unit-time operations. Pipelined and multicycle functional units can be accommodated by incorporating execution delay in the equations presented in Sections 3.2 and 3.3 [12]. To model operation chaining, a precedence relation can be added between operations that cannot be chained [4]. $(ASAP)_j$ (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. Symbols "Σ" and "+" correspond to Boolean *Or* function, and "Π" stands for Boolean *And*. Product "*ab*" implies "*a And b*".

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

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

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

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

Eq.1.a states that prior to step $(ALAP)_j$, operation j is not scheduled more than once. On step $(ALAP)_j$ Eq.1.b 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 (pre-executed) or not scheduled at all.

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 $[(ASAP)_j \ldots (ALAP)_i]$

the following must hold:

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

Eq.2.a 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:

$$\left(\overline{C_{sj}} + \sum_{ASAP_i \le l < s} C_{li}\right) + \overline{\Gamma_i} = 1 \tag{2.b}$$

Effectively, Eq.2.a ensures that the operation can be pre-executed only if all of its predecessors have already been executed. An operation after the join node cannot be pre-executed in our model. Thus (Eq.2.b), 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. Eq.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 ($j_1 ... j_n$) are immediate predecessors of the sink node in the CDFG.

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

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 Eq.3.

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

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

F_{sl} is a Boolean function describing that resource r_l is needed during time step *s*. Eq.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*) cannot be scheduled. For functional units, F_{sl} functions reduce to 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, Eq.4 becomes:

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

Eq.4 has $\binom{n_{sl}}{k_l}$ product terms, but its OBDD form is compact (O($n_{sl}k_l$) nodes) and can be built efficiently [12]. Eq.4 applies the resource constraint to all traces simultaneously. *Trace validation* (Section 3.3) 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} [12].

5. Removal of redundantly scheduled 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:

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

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:

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

3.3. Trace validation

A valid *ensemble schedule* is a set of traces which 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. Assume that the conditional c_k (with a corresponding guard G_k) is resolved at step j. Causality requires that the traces corresponding to guard values G_k and $\overline{G}_k$ must match (be identical) for all time steps prior to (and including) j. Completeness ensures that the ensemble schedule includes traces for both G_k and $\overline{G}_k$.

A trace satisfying all of the constraints introduced in Section 3.2 may still not be valid in the sense that it cannot be a member of any set of traces forming an ensemble schedule. *Trace Validation* ensures that each validated trace is part of some executable ensemble schedule. The validation is efficiently preformed by the iterative algorithm shown in Fig.4. 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*; $\forall_x f = f_x f_{\overline{x}}$ - *universal abstraction*
- ***S*** - set of all traces; S(0) - initial set of non-validated traces; S(i) - set of traces at iteration i
- ***V*** - set of all variables not including guard variables
- ***V'(j)*** - subset of ***V*** corresponding to time steps $\leq j$

- ***S'*** - set of traces from which all variables (***V-V'(j)***) are removed: $S' = \exists_{(V - V'(\mathrm{j}))} S(i)$
- ***C*** = [c_1, c_2 ... c_n] - set of all conditionals
- ***G*** = [G_1, G_2 ... G_n] - set of guards corresponding to the conditionals
- ***R(j)*** = [R_1(j), R_2(j) ... R_n(j)] - *resolution vector*

The *resolution vector* ***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*: $R_k(j) = \sum C_{lk}$, for ($l < j$). *S'* is partitioned by ***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* (***G$_{res}$***). The guards from (***G-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 (***G-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 which 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. In the worst case, the number of iterations cannot exceed the number of conditionals. 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.1(a). Assume that the guards G_1 and G_2 correspond to the conditionals *1* and *2*. There are four possible control paths: $(G_1 G_2, G_1 \overline{G_2}, \overline{G_1} G_2, \overline{G_1}\,\overline{G_2})$. 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), but $R_2(2)$ = c_{12} = 1 (i.e. conditional *2* is scheduled at step 1). Thus, matching of traces has to be performed only with respect to still unresolved conditional *1* (i.e. traces for paths $(G_1 G_2, \overline{G_1} G_2)$ 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 3.2 that Eq.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.

4. CONSTRUCTION

The constraints described in Section 3 each have a simple and regular structure [12]. This allows OBDD representations to be constructed *directly* from the CDFG without reference to an intermediate equation form. Although individual equations have efficient orderings, optimal orderings for different equations frequently contradict. However, experimental results indicate that typical instances do have good orderings. The results presented in this paper are generated using the variable ordering where non-guard variables are ordered by increasing time and guard variables are placed on top (i.e. closest to the root of OBDD).

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 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 pre-specify the number of control steps.

5. EXPERIMENTAL RESULTS

The technique described in the paper was implemented in C++ and executed on a Sun SPARC-station10. Reported CPU times correspond to the complete procedure: CDFG analysis, constraint construction, and all OBDD manipulations leading to the reported results. Table 1 summarizes the *elliptic wave filter* (***EWF***) benchmark experiments. We found *all* optimal solutions of each instance using OBDDs whose size was significantly smaller than $(\#variables)^2$. To reduce the size of partial solutions, an auxiliary set of *interior constraints* was generated. 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*... (*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 an 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).

Table 2 and Table 3 show experimental results for benchmarks exhibiting conditional behavior. The rows *#cycles(spec)* and *#cycles(non_spec)* correspond to schedules with and without speculative execution (using the same set of resources). 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 the schedulers which minimize average path length, a subset of solutions with small average path length is generated in a "greedy" fashion.

Benchmarks *Maha* [15], *Kim* [11] and *Waka* [10] are conditional trees, and *MulT* [9] has two parallel trees. *Parker* is *Maha* with addition *A6* converted into a subtraction. The *Maha* solution with one adder and one subtracter is the same as in [8][9]. Allowing more resources (2 adders, 3 subtracters) an improvement of 0.125 (average path length) is made over the best previous result. In *Parker*, this improvement was 0.25. In some 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. As a default, in our system we assume that a single-cycle comparator is used and that its output becomes available only in the successive cycle. This assumption is used in those benchmarks where the number of comparators is indicated in Table 2. Even with this assumption, our technique still derives the same result for *Kim* as in [9]. In *Waka* one path is a cycle longer than that reported in [8]. In *MulT* a one cycle shorter minimum-latency solution was found by exploiting dynamic scheduling of operations belonging to parallel trees.

6. CONCLUSIONS AND FUTURE WORK

We describe a symbolic formulation that allows speculative operation execution and exact scheduling of arbitrary forward-branching control/data paths. The execution order of conditionals is not pre-determined and is dynamically resolved allowing gains in scheduling quality. A trace validation algorithm is introduced which ensures consistency for families of ensemble schedules. An iterative construction method is presented along with benchmark results. In future work, several issues should be addressed: incorporation of control/interconnect costs and extensions to restricted forms of cyclic control. An efficient approach to remove the restriction on our speculative execution model will be considered as well.

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. Since *all* such schedules are propagated, this heuristic has good behavior and is applicable to problems with thousands of formulation variables. Moreover, if the exact scheduler is based on *Zero-Suppressed BDDs* [16] significant improvements in terms of both CPU time and memory usage are observed. The interested reader is referred to the experimental study [17] where larger DFGs are solved.

ACKNOWLEDGMENT

We would like to gratefully acknowledge contributions from Dr. A. Seawright who took part in early discussions and developed the C++ OBDD package extensively used throughout this project. Furthermore, we wish to thank the reviewers for helpful suggestions and comments.

REFERENCES:

[1] R. Camposano, “Path-Based Scheduling for Synthesis”, *IEEE Trans. CAD/ICAS*, vol. 10, no. 1, pp. 85-93, Jan. 1991.

[2] S. Davidson *et al.*, “Some Experiments in Local Microcode Compaction for Horizontal Machines”, *IEEE Trans. Computers*, vol. c-30, no. 7, pp. 460-477, July 1981.

[3] P.G. Paulin and J.P. Knight, “Force-Directed Scheduling for the Behavioral Synthesis of ASIC’s”, *IEEE Trans. CAD/ICAS*, vol. 8, no. 6, pp. 661-679, June 1989.

[4] 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, no. 4, pp. 464-475, Apr. 1991.

[5] C.H. Gebotys and M.I. Elmasry, “Global Optimization Approach for Architectural Synthesis”, *IEEE Trans. CAD/ICAS*, vol. 12, no. 9, pp. 1266-1278, Sep. 1993.

[6] L. Yang and J. Gu, “A BDD Model for Scheduling”, *Proc. CCVLSI*, 1991.

[7] R.E. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation”, *IEEE Trans. Computers*, vol. C-35, no. 8, pp. 677-691, Aug. 1986.

[8] S. H. Huang et al. “A Tree-Based Scheduling Algorithm for Control Dominated Circuits”, *Proc. 30th ACM/IEEE Design Automation Conference*, pp. 578-582, 1993.

[9] K. Wakabayashi and H. Tanaka, “Global Scheduling Independent of Control Dependencies Based on Condition Vectors”, *Proc. 29th ACM/IEEE Design Automation Conference*, pp. 112-115, 1992.

[10] K. Wakabayashi and T. Yoshimura, “A Resource Sharing and Control Synthesis Method for Conditional Branches”, *Proc. 26th ACM/IEEE Design Automation Conf*erence, pp. 62-65, 1989.

[11] T. Kim, J.W.S. Liu, and C. L. Liu, “A Scheduling Algorithm for Conditional Resource Sharing”, *Proc. IEEE International Conference on Computer-Aided Design*, pp. 84-87, 1991.

[12] I. Radivojević and F. Brewer, “Symbolic Techniques for Optimal Scheduling”, *Proc. 4th SASIMI Workshop*, pp. 145-154, Nara, Japan, 1993.

[13] J. Yang, G. De Micheli, and M. Damiani, “Scheduling with Environmental Constraints based on Automata Representations”, *Proc. EDAC*, 1994.

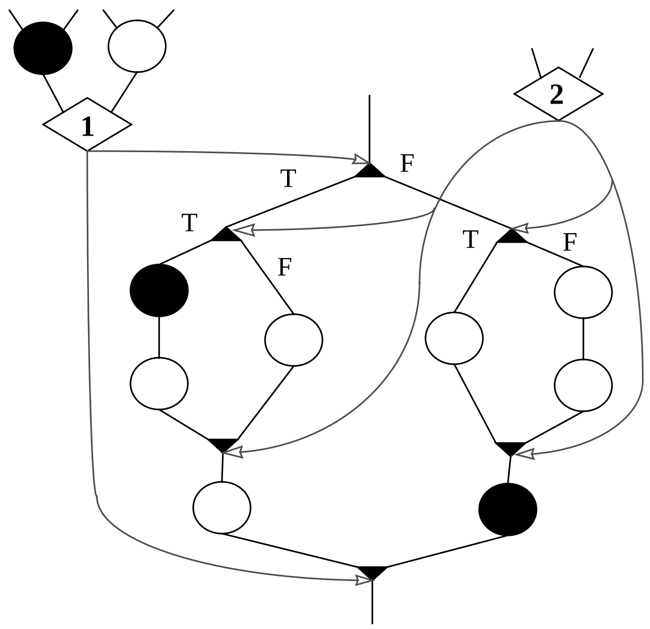
[14] I. Radivojević and F. Brewer, “Incorporating Speculative Execution In Exact Control-Dependent Scheduling”, *Proc. 31th ACM/IEEE Design Automation Conference*, pp. 479-484, 1994.

[15] A. C. Parker, J.T. Pizarro, and M. Mliner, “MAHA: A Program for Datapath Synthesis”, *Proc. 23th ACM/IEEE Design Automation Conference*, pp. 461-465, 1986.

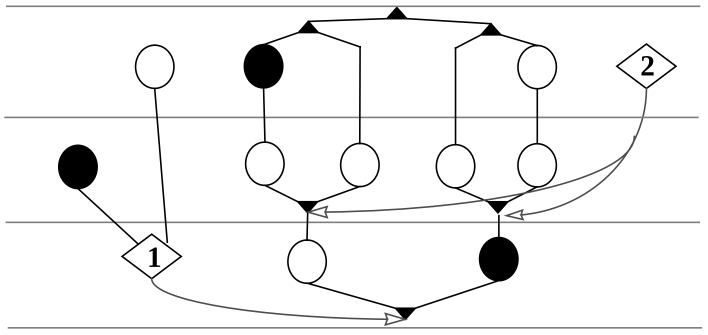
[16] S.-I. Minato, “Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems”, *Proc. 30th ACM/IEEE Design Automation Conference*, pp. 272-277, 1993.

[17] I. Radivojević and F. Brewer, “On Applicability of Symbolic Techniques to Larger Scheduling Problems”, *ECE Tech. Report #94-22*, University of California, Santa Barbara, Sep. 1994.

(a)



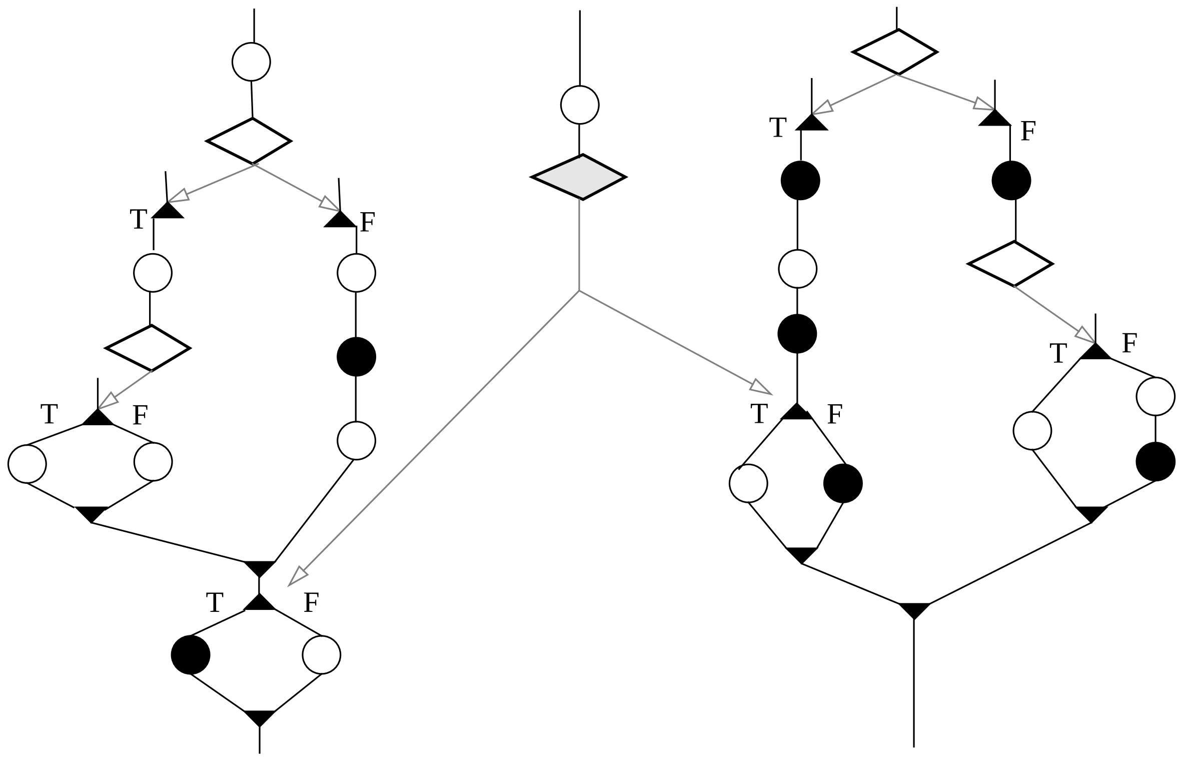
(b)



resources:
- 2 adders (white)
- 1 subtracter (black)
- 1 comparator

execution time:
- 3 cycles (*2* before *1*)
- 4 cycles (*1* before *2*)

Fig. 1. Example CDFG and its schedule



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

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

Fig. 2. CDFG with correlated control

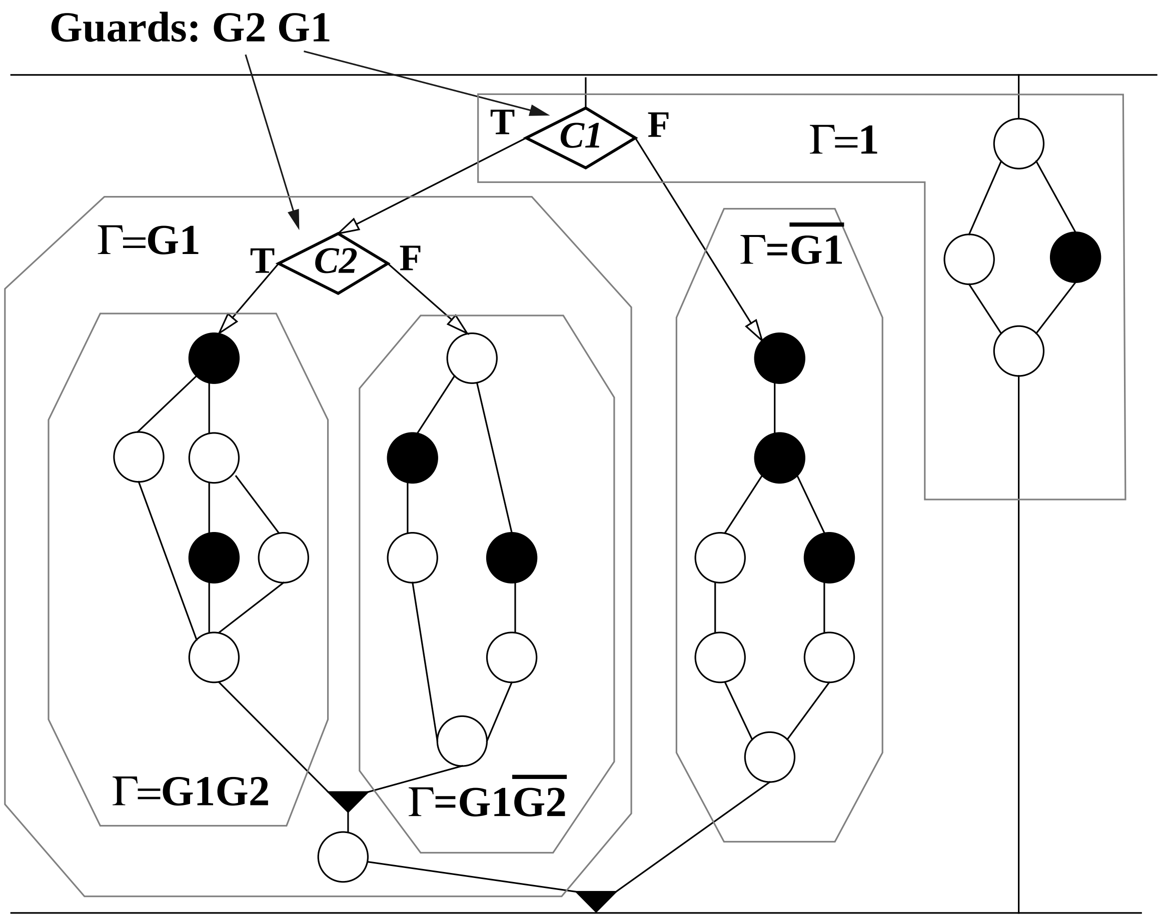


Fig. 3. Kim’s example

```
i = 0;
do {
    i++;
    S(i) = S(i-1);
    for each time step $j$ {
        $S' = \exists_{(V - V'(j))} S(i)$
        for each conditional $c_k$ {
            $S' = S'R_k(j) + \forall_{G_k}(S'\overline{R_k(j)})$
            if (S’==0) { S(i)=0; exit; }
        }
        S(i) = S(i)S’;
    }
} while (S(i)!=S(i-1));
```

Fig. 4. Trace validation algorithm

Table 1: 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	3.8	17.1	12.5	3.9	970.1	587.6
2-cycle multiplier and single-cycle adder except: (*) 2-cycle pipelined multiplier.										

Table 2: 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	7	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]		7.41	4.32	5.68	5.18	2.27	3.52
single-cycle adders, subtracters and comparators assumed							

Table 3: 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 [8]	3.31 (5)	-	-	-	4.75 (7)	-
CVLS [9]	3.31 (5)	2.38 (4)	2.38 (4)	5.75 (6)	-	2.88 (4)
Kim *et al*. [11]	4.62 (8)	-	-	6.25 (7)	4.75 (7)	-

BIOGRAPHY

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 Faculty of Electrical Engineering, 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 VLSI computer architectures.

Forrest Brewer received the Bachelor of Science degree with honors in physics from 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 Corp. and consulted there until 1985. He co-authored 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.