Ensemble Representation and Techniques for Exact Control-Dependent Scheduling *

Ivan Radivojević, Forrest Brewer

Department of Electrical and Computer Engineering
University of California, Santa Barbara

Abstract

This paper describes a new exact formulation of control/data-flow scheduling. Unlike current techniques, a closed form solution set is generated in which all satisfying schedules for arbitrary forward branching control/data paths and resource constraints are encapsulated in a compressed OBDD-based representation. A robust, iterative construction strategy is presented along with benchmark results. An advantage of this construction is that initial bounds for the number of clock cycles are no longer needed for exact formulations. This strategy also allows the simple formulation of several set-based heuristics as well as controlling the growth of the BDD representation.

1. Introduction

Heuristic scheduling (path-based[2], list[8], force-directed[12]) can accommodate a wide variety of control dependent behaviors. However, the technique can fail to find solutions in very tightly constrained problems even when such solutions exist. Exact ILP-based scheduling[3][5][9] will always find such solutions, but current approaches are unable to handle complex control-dependent behavior. To improve the run-time efficiency and reduce the number of variables, an ILP-based heuristic[7] and a mixed ILP/BDD formulation were proposed[18].

Recently, there has been substantial work on heuristic scheduling of control dominated systems. Huang [4] uses a representation of the execution paths as a tree. This technique eliminates operations that are not redundant in a program, but are redundant for some of its containing paths. Transformation of a data-flow graph with conditional branches into one without conditional branches is performed in [6]. In order to identify mutually exclusive operations, *condition vectors* are introduced in [16][17]. This technique allows for systematic operation node duplication and pre-execution leading to high quality results. Most current research is restricted to nested conditional branches (*conditional tree* control structures). Scheduling of multiple conditional trees is described in [17], but the trees are scheduled sequentially using a priority scheme. Furthermore, the current scheduling techniques typically produce a single representative solution, forcing the scheduling task to be re-run if constraints found in subsequent synthesis tasks conflict with the current solution.

To address these issues, the scheduling problem was formulated using a compressed representation based on an OBDD (Ordered Binary-Decision Diagrams[1]) representation[13][14]. Using this technique, the complex Boolean functions representing *all* possible solutions to a given scheduling problem are typically representable in a relatively small space. This has the advantage that if all solutions are so encapsulated, the exact effect of inclusion of additional constraints derived during subsequent synthesis steps is *incrementally* computable. Furthermore, the process is exact in that if no schedules are found after some step in the synthesis process, the designer is assured that no schedule exists which satisfies all of the constraints. An elegant related technique is currently under development at Stanford[19] using finite automata to represent resource and timing constraints.

In this paper, we reformulate and extend the symbolic scheduling technique to allow a new iterative construction process. While maintaining the exactness of the solutions, this approach exhibits improved robustness and offers more control over the construction process. The formulation can be extended to incorporate speculative operation execution (pre-execution)-- this work is reported in [15].

2. Formulation

In this formulation we represent all of the scheduling constraints as Boolean equations and build an OBDD cor-

* This work has been supported in part by fellowship donation from Mentor Graphics Corp. and UC-MICRO under project No. 92-019.

responding to their intersection. Each variable in the OBDD describes a particular operation occurring at a particular time step, over a finite set of time steps. A variable is true if the corresponding operation is scheduled during the corresponding time step in a particular solution.

To allow control-dependent scheduling a set of *'guard'* variables[13] is introduced. Each guard G labels a particular fork/join pair, where 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. A Boolean function Γ_j (defined on the guard variables) conditions the execution of operation j in a data/control/data flow graph (CDFG) and encodes directly all the control paths on which j can be scheduled. Furthermore, if two operations i and j are guarded by Γ_i and Γ_j whose intersection is empty, then i and j are mutually exclusive. Using this technique, *all* schedules for *all* forward control paths are simultaneously constructed and are represented in a compressed OBDD form. In general, the solution is a *collection of product terms*, each one including all the variables corresponding to the operations and some (not necessarily all) guard variables, representing a possible execution instance for a particular control path. We call these product terms *traces*[13].

Shown in Fig. 1 is *Kim*'s example[6] in which two guards are introduced to fully describe the conditional behavior. Indicated blocks correspond to operations that share the same guard function Γ. Operations belonging to a control-independent portion of CDFG are not guarded and thus belong to all execution paths. Consequently, they are scheduled in parallel under all control combinations.

The proposed formulation is not limited to CDFGs which have a conditional tree control structure. Example shown in Fig. 2 is a problem instance with a control correlation between two control fork-join structures running in parallel. There is a unique conditional (shaded comparator) and a unique guard determining the flow of control for both forks. Note that the complexity of the formulation grows only with the number of guard variables, not the (possibly exponentially larger) number of traces or control paths. The example in Fig. 2 has 18 possible control paths,

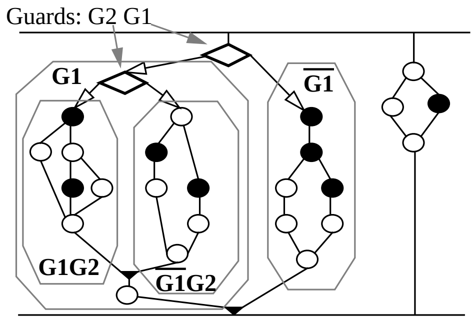


Figure 1: Kim's example

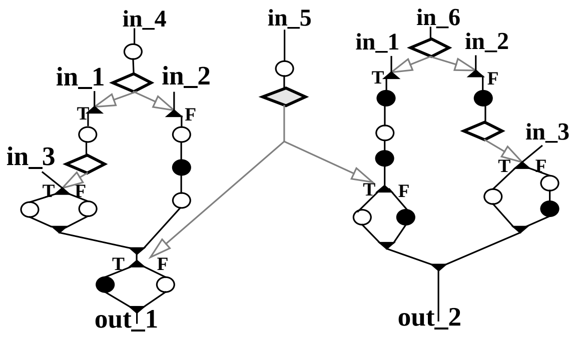


Figure 2: CDFG with correlated control

but only 5 guards need to be defined.

2.1. Precedence constraints

For brevity, we make the simplifying assumption of simple non-pipelined unit time delay for the operations (this restriction is easily removed as reported in [13][14]).

Eq.1 and Eq.2, describe the scheduling problem when no resource constraints are specified. We make use of the ASAP (as soon as possible) and ALAP (as late as possible) bounds to determine the time spans over which an operation can be scheduled. These bounds are not required for correctness, but improve the efficiency of the algorithm by eliminating those variables which cannot be true in any feasible schedule.

Eq. 1 enforces that each operation j is scheduled once and only once on all the paths covered by Γ_j and is not scheduled on other paths. C_{sj} denotes operation j's instance at time step s. If $(ASAP)_j \le s < (ALAP)_j$:

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

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

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

If operation i precedes operation j, for every time step s from the range $[(ASAP)_j \ldots (ALAP)_i]$ the following constraint has to be satisfied:

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

A special *sink* variable is used in the formulation indicate that a particular trace has concluded. Eq. 3 is used as a terminating condition for all traces. The sink variable is initialized to '0', and is set to '1' when the terminating condition is met. The scheduling process can be terminated when *sink* assumes the value '1' on all paths. This adds one Boolean variable to the entire formulation. In these equations, operations $(j_1 \ldots j_n)$ are operations that are immediate predecessors of the sink node in the CDFG.

$$\prod_{l=1}^{n}(R_{sj_l}+\overline{\Gamma_{j_l}}) = 1$$

$$R_{sj_l} = \sum_{k=(ASAP)_j}^{s} c_{kj_l} \qquad \text{(Eq. 3)}$$

2.2. Resource Constraints

If k_l resources of a certain type r_l (e.g. multipliers, adders, ALUs, registers, busses) are available, we can formulate a *'resource-constraint'* Eq. 4:

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

F_{sl} is a Boolean function describing that resource r_l may be needed during time step s. Eq. 4 is applied for each time step s and each resource r_l bounded by k_l. It indicates that at least (n_{sl}-k_l) resources (among n_{sl} potential operations in time step s) cannot be scheduled. For simple constraints, a variable corresponding to a particular instance of an arithmetic/logic operation can be directly substituted in F_{sl}.

If operation i precedes operations (j_1...j_n), Eq. 5 indicates that at a particular control step s, a bus may be needed to read an operand (F^{br}) or write a result (F^{bw}). The bus constraints apply to *'read'* and *'write'* phases separately (i.e. we assume that read/write transfers are temporally interleaved). Note that this formulation easily models the situation in which an operand is used as an input to a number of operations and, contrary to [5], does not require the introduction of additional representation variables.

$$F_{si}^{b_r} = \sum_{l=1}^{n} C_{sj_l}(\Gamma_i+\overline{\Gamma_{j_l}}) \qquad F_{si}^{b_w} = C_{si} \qquad \text{(Eq. 5)}$$

A bound on the number of available registers can be formulated similarly to that shown in reference[13] for the non-branching case.

2.3. Trace validation

A trace which satisfies all of the constraints may still not be a valid execution instance in the sense that it may not be compatible with a complete set of traces forming an executable schedule. A valid schedule must be *causal* and *complete* for all control paths. The *causality* requirement dictates that the schedule cannot use knowledge of a *conditional* (an operation generating a control signal) prior to the time step in which it is scheduled. Fig. 3 illustrates a situation in which two traces corresponding to opposite values of the guard G_k (corresponding to the conditional c_k) cannot be chosen to form a valid schedule unless conditional c_k is evaluated prior to step j. (The decision to execute a 'black' or a 'white' operation requires prior knowledge of which path is being executed). Traces corresponding to guard values G_k and $\overline{G_k}$ must agree before the conditional c_k is resolved. The *completeness* requirement states that a valid trace must exist in each solution for every possible control path.

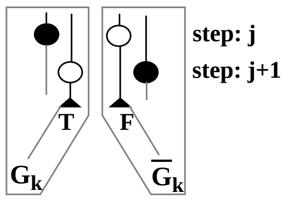


Figure 3: Trace matching

Trace validation ensures that each validated trace is part of an executable ensemble schedule. The validation is efficiently preformed by an *iterative* algorithm shown in Fig. 4. The following notation is used: ***S*** - set of all traces that execute in ***k*** time steps, S(0) - initial set of non-validated traces, S(i) - set of traces at iteration *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*** (a set of Boolean functions indicating that a conditional c_k was scheduled prior to time step j): $R_k(j) = \sum C_{lk}$ for (l<j), $\boldsymbol{G_{res}}$ - set of guards corresponding to the resolved conditionals in $R_k(j)$, ***V*** - set of all variables not including guard variables, ***V'(j)*** - subset of ***V*** corresponding to time steps <= j, S' - set of traces from which all variables representing operation instances after step j are removed: $S' = \exists_{(V-V'(j))}S$, $\exists_x f = f_x + f_{\overline{x}}$ - *existential abstraction*, $\forall_x f = f_x f_{\overline{x}}$ - *universal abstraction*. With respect to *R(j)* the function *S'* can be mapped into a disjoint set of (possibly 2^n) families, corresponding to the subset of guards that are resolved prior to time step j. The guards from ($\boldsymbol{G}$-$\boldsymbol{G_{res}}$) are *don't cares* within the family since at time step j there is no knowledge about how the schedule will look at the successive steps with regard to the future potential values of the unresolved guards. Thus, traces must both *match* and *exist* for

```
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));
```

Figure 4: Trace validation (TV) algorithm

all possible combinations from (***G-G$_{res}$***).

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 TV algorithm repeats until a fixed point is reached. The number of iterations on *i* cannot exceed the number of conditionals in a temporal (precedence) chain of any trace and, in the worst case, is bounded by the number of guards. On tree-like control structures, the number of iterations is bounded by the height of the tree. A formal discussion of the algorithm is reported in [14].

3. OBDD construction

3.1. OBDD structure and ordering

The constraints described in Section 2 each have a simple and regular structure. This allows OBDD representations to be constructed *directly* from the CDFG[13] without reference to an intermediate equation form. This process is fast and generates no construction garbage (nodes that are not referenced in the final solution). Shown in Fig. 5 is the OBDD representation of Eq.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 Eq.4 is $\binom{n}{k}$.

It is important to note that although individual equations have efficient orderings, optimal orderings for different equations contradict. There can be no polynomial bound on the size of an arbitrary instance of the scheduling problem for any pre-specified ordering since this problem is NP-complete[1][10]. However, experimental results indicate that typical instances, including conventional benchmarks, do indeed have good orderings.

All of the results presented in this paper are generated using a simple variable ordering with non-guard variables ordered by increasing time step and guard variables placed on top (i.e. closest to the root of OBDD). This ordering typically results in small OBDDs and accommodates iterative construction. The construction speed can be improved by clustering operations based on their functional dependencies. (For example, the CPU time is reduced by 3x for the example in Fig. 7.) We plan to add this feature to our general implementation.

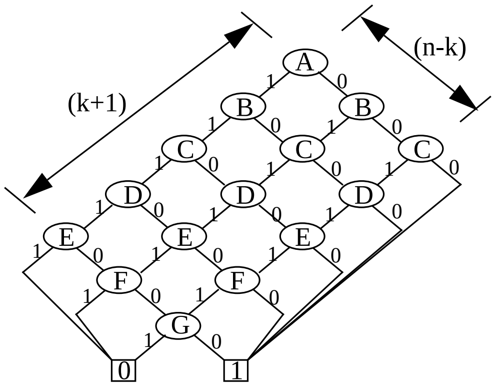


Figure 5: *At-most-k-out-of-n* constraint (k=4, n=7)

3.2. Iterative construction process

In [14] a heuristic procedure to combine all of the scheduling constraints was described. Essentially, the construction combined constraints for which ‘good’ orderings were known first and then sequentially applied the other constraints. Using this technique, the final OBDD typically has relatively small size. However, the size of OBDDs at intermediate stages can be relatively large, resulting in slow construction or large memory requirements.

To improve the robustness of the algorithm, a new iterative construction is proposed. The solution is built on a time-step by time-step basis: only those constraints relevant to a particular time step *j* are generated and applied to the OBDD representing a valid partial solution for the previous (*j-1*) steps. In this way, only partial time sequences of constraints need to be added at each step. This prevents the construction of large set of spurious solutions before all constraints have been applied. We observe that the sizes of intermediate OBDDs are smaller and that generation of ‘garbage’ decreases significantly. Iterative construction generates a larger number of smaller constraints than the earlier process and can be slower for small examples. However, for larger cases it offers far more robust behavior in terms of memory management and allows tighter control over the computation. It also has the advantage that one can detect when schedules have completed obviating the need to accurately pre-specify the number of control steps.

It is possible to further improve the construction process by applying only Eq 1b at ‘ALAP’ time step for operation. Although this allows redundant operator scheduling in intermediate solutions, we observed that the size of the OBBD is typically smaller during construction. Lastly, it is possible to detect when a variable has become unate (has a unique Boolean value in all of the schedules) during the construction. This information can be used to reduce the number and the complexity of applied constraints for non-branching schedules.

3.3. Set heuristic algorithms

Since valid partial schedules are available after each step, this construction can be used to construct efficient heuristic scheduling techniques when the OBDD size becomes large (e.g. 28-cycle elliptic wave filter benchmark has more than 3 billion optimal solutions). The simple heuristic shown in Fig. 6 only propagates the subset of

```
BDDnode   *SetUtilize(partial, sink, step, utility)
BDDnode   *partial, *sink;
int       step, utility; {
    BDDnode  *subset;
    if(step>=minimum_execution_time) { /* check for end */
        subset = And(partial, sink);
        if(subset!=0) return(subset);
    }
    do {
        subset = And(partial,ChooseExactly(utility);
        if(subset==0)       utility--;
        else                return(subset);
    } while(utility>=0);
    return(0);
}
```

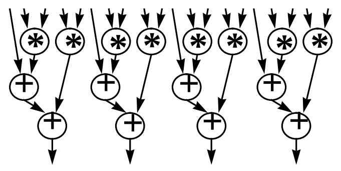
Figure 6: Set heuristic Algorithm

schedules with maximum utilization of resources. Utilization is measured by the number of operations active in each time step. Since *all* such schedules are propagated, this simple heuristic has good behavior. The algorithm can be made less greedy by applying it over a sliding window of several time steps or over a range of utilizations. Finally, the OBDD pruning can be delayed behind the current scheduling step to create 'look-ahead'. These manipulations are surprisingly efficient and consist of repeated use of the construction template shown in Fig. 5. The greedy set heuristic in Fig. 6 optimally solves the linear benchmarks described in detail below with greatly reduced CPU execution times and memory requirements.

4. Experimental results

The expressive power of OBDDs allows encapsulating huge solution spaces using moderate computing resources. Table 1 contains the experimental results corresponding to the elliptic wave filter benchmark. We constructed *all* optimal solutions of each instance and captured all of the solutions in OBDDs whose size was significantly smaller than n^2, where n is the number of variables in the instance. Unate variables were factored out of the final results and it was assumed that coefficient ROMs are directly accessible without the need for global communication (transfer of the operands on shared busses). Table 2. shows the results of the set heuristic described in the previous section. Only solutions with maximal resource utilization were preserved at each step. All cases were solved with larger than optimal time-step bounds to demonstrate the heuristic.

The XMAC example (Fig. 7) represents a *block-matrix multiply-and-accumulate* operation typical for vector units of supercomputers where multipliers and adders are heavily pipelined. The example is more typical for optimizing compilers than for high-level synthesis applications. However, we observe that it may present substantial difficulty for heuristic schedulers (due to the symmetric



$$\begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix} + \begin{bmatrix} b11 & b12 \\ b21 & b22 \end{bmatrix} \times \begin{bmatrix} c11 & c12 \\ c21 & c22 \end{bmatrix} = \begin{bmatrix} r11 & r12 \\ r21 & r22 \end{bmatrix}$$

```
number of controls steps = 19
number of adders: 1; cycle time: 5; pipelined
number of multipliers: 1; cycle time: 6; pipelined
number of registers:2 //I/O registers not counted//
number of busses: 4
...number of obdd nodes: 1022
...number of schedules: 48
...time = 2.35s
```

Figure 7: XMAC example

execution paths, large mobilities of the operations and very tight register constraints). Table 3. corresponds to exact solutions of CDFGs with forward branching control. *Maha*[11] and *Kim*[6] have a conditional tree structure. MulT is a multiple conditional tree introduced by Wakabayashi[17] and *Corr* is the example of CDFG with correlated control introduced in Fig. 2. Solution of these problems was extremely efficient both in terms of a runtime and memory requirements.

All experiments were run on SPARCstation10 using a custom C++ OBDD package. Reported CPU times correspond to the complete procedure: CDFG analysis, constraint construction, and all OBDD manipulations including trace validation generating the final OBDD results.

5. Conclusions and future work

In this paper an incremental construction for the exact solution of resource constrained scheduling problems subject to arbitrary forward control behavior was described. This construction is robust since it enforces constraints only on a time-step basis and also eliminates the requirement that the number of clock steps be pre-specified. (This is a particular problem for control dominated scheduling in which differing paths have widely differing path lengths). Based on this construction, a number of set based scheduling heuristics can be described. These heuristics are particularly useful for weakly constrained problems in which the number of optimal solutions is huge.

In future work, we intend to pursue a greater variety of construction constraints including operation binding, register binding and intercommunication constraints. Further extensions to cyclic and finite state machine based control seems to be viable as well.

Table 1: WAVE filter - exact

#cycles	19	19	20	20	21	28
#adders	2	2	2	2	2	1
#multipliers	1(*)	2	1(*)	2	1	1
#busses	5	4	4	4	4	3
#registers	10	10	10	10	10	n/a
#variables	131	131	165	165	199	437
#nodes	2,111	1,192	5,128	11,582	238	123,143
#schedules	11,466	1,071	65,826	991,638	5,139	3.10279e9
CPU time [s]	6.24	6.69	33.14	57.61	33.78	4920.63
2-cycle non-pipelined multiplier and single-cycle adder except: (*) 2-cycle pipelined multiplier.						

Table 2: WAVE filter - set heuristic

#cycles	19	19	20	21	28	54
#adders	2	2	2	2	1	1(**)
#multipliers	1(*)	2	1(*)	1	1	1
#variables	165	165	199	233	471	1,001
#nodes	162	409	194	328	5,603	9,799
#schedules	1	20	1	5	317,520	423,360
CPU time [s]	2.92	2.98	3.01	6.25	73.77	224.79
2-cycle non-pipelined multiplier and single-cycle adder except: (*) 2-cycle pipelined multiplier, (**) 2-cycle non-pipelined adder.						

Table 3: Control-dependent results

	Maha	Kim	MulT	Corr
#cycles	8	8	4	6
#adders	1	1	-	1
#subtracters	1	1	-	1
#comparators	-	1	-	1
#ALUs (+/-/>)	-	-	3	-
#variables	62	79	22	48
#control paths	12	3	6	18
#guards	6	2	3	5
#TV iterations	1	2	1	2
#nodes	867	576	185	315
#traces	376	68	48	45
CPU time [s]	2.03	2.09	0.16	1.07

6. Acknowledgment

We would like to gratefully acknowledge contributions from Dr. A. Seawright who took part in early discussions and developed the custom C++ BDD package extensively used throughout this project.

7. References:

[1] R.E. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation”, *IEEE Transactions on Computers*, Vol. C-35, No. 8, August 1986.

[2] R.Camposano, A. Bergamashi, “Synthesis Using Path-Based Scheduling: Algorithms and Exercises”, *Proc. 27th ACM/IEEE Design Automation Conference*, 1990.

[3] C.H. Gebotys, “Optimal Scheduling and Allocation of Embedded VLSI Chips”, *Proc. 29th ACM/IEEE Design Automation Conference*, 1992.

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

[5] C.-T. Hwang, Y.-C. Hsu, Y.-l. Lin, “Optimum and Heuristic Data Path Scheduling Under Resource Constraints”, *Proc. 27th ACM/IEEE Design Automation Conference*, 1990.

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

[7] H. Komi, S. Yamada, K. Fukunaga, “A Scheduling Method by Stepwise Expansion in High-Level Synthesis”, *Proc. IEEE International Conference on Computer-Aided Design*, 1992.

[8] S. Davidson, D. Landskov, B. Shriver, P. Mallett, “Some Experiments in Local Microcode Compaction for Horizontal Machines”, *IEEE Transactions on Computers*, Vol. c-30, No. 7, July 1981.

[9] J.-H. Lee, Y.-C. Hsu and Y.-L. Lin, “A New Integer Linear Programming Formulation for the Scheduling Problem in Data Path Synthesis”, *Proc. 26th ACM/IEEE Design Automation Conference*, 1989.

[10] H.-T. Liaw, C.-S. Lin, “On the OBDD-Representation of General Boolean Functions”, *IEEE Transactions on Computers*, Vol. 41, No. 6, June 1992.

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

[12] P.G. Paulin, J.P. Knight, “Force-Directed Scheduling for the Behavioral Synthesis of ASIC’s”, *IEEE Transactions on Computer-Aided Design*, Vol. 8, No. 6, June 1989.

[13] I. Radivojević, F. Brewer, “Symbolic Techniques for Optimal Scheduling”, *Proc. 4th SASIMI Workshop*, Nara, Japan, Oct.1993.

[14] I. Radivojević, F. Brewer, “A New Symbolic Technique for Control-Dependent Scheduling”, *ECE Tech. Report #93-16*, University of California, Santa Barbara, Sep. 1993.

[15] I. Radivojević, F. Brewer, “Incorporating Speculative Execution in Exact Control-Dependent Scheduling”, to appear in *Proc. 31st ACM/IEEE Design Automation Conference*, 1994.

[16] K. Wakabayashi, T. Yoshimura, “A Resource Sharing and Control Synthesis Method for Conditional Branches”, *Proc. 26th ACM/IEEE Design Automation Conference*, 1989.

[17] K. Wakabayashi, “Global Scheduling Independent of Control Dependencies Based on Condition Vectors”, *Proc. 29th ACM/IEEE Design Automation Conference*, 1992.

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

[19] J. Yang, G. DeMicheli, M. Damiani, “Scheduling with Environmental Constraints based on Automata Representations”, to appear in *Proc. EDAC-94*, Paris, France, March, 1994.