Incorporating Speculative Execution in Exact Control-Dependent Scheduling *

Ivan Radivojević Forrest Brewer

Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106

Abstract - This paper describes a symbolic formulation that allows incorporation of speculative operation execution (pre-execution) in an exact control-dependent scheduling of 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. To extract parallelism implicit in the input specification Boolean 'guard' functions are used to identify paths where operations have to be scheduled and the execution order of the conditionals is dynamically resolved. An efficient and systematic iterative construction method is presented along with benchmark results.

I. INTRODUCTION

Heuristic scheduling (path-based[2], list[8], force-directed [12]) accommodates a wide variety of control dependent behaviors. However, it can fail to find solutions in very tightly constrained problems even when such solutions exist. Exact ILP-based scheduling[3][5][9] finds such solutions, but current methods are unable to handle complex control-dependent behavior. To reduce the number of variables and CPU times, an ILP-based heuristic[7] and a mixed ILP/BDD formulation[19] were proposed.

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 to enable a movement of operations. Transformation of a data-flow graph with conditional branches into one without conditional branches is performed in [6]. 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 OBDD (Ordered Binary-Decision Diagram[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 alternative formulation is currently under development at Stanford[18] using finite automata to represent resource and timing constraints.

In this paper, we extend the symbolic scheduling technique to incorporate *speculative operation execution* (*pre-execution*). Speculative execution allows the operations from branch arcs to be executed *before* the branching condition is resolved. This is a very important problem in practice, since speculative execution can improve execution times when there are sufficient resource. It does this by exploiting a exploiting the global parallelism implicit in the problem formulation. The problem is difficult since it can potentially lead to an explosion of operation execution instances. We introduce a restricted model that allows the problem's treatment in a systematic fashion. Several forms of code motion are supported and the resolution of the *conditionals* (operations that generate control signals) is performed dynamically. We are able to solve this model exactly and obtain benchmark results which are equal to or superior to the current published results. In fact, we preserve the *entire* set of feasible solutions thus allowing application of incremental synthesis constraints as before. To our knowledge, this is the only exact technique published for this problem.

II. FORMULATION

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

---

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

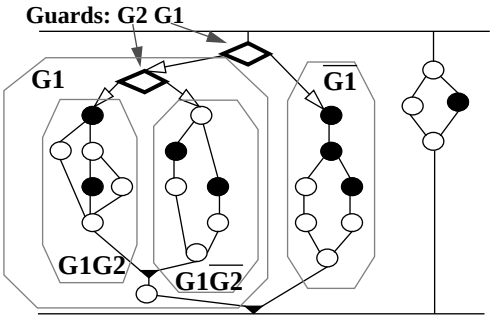

Fig. 1. Kim's example

the corresponding guard variables. A Boolean function Γ_j (defined on the guard variables) conditions the execution of operation *j* in a control/data flow graph (CDFG) and encodes directly all the control paths on which *j must* be scheduled. 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 term including both the variables corresponding to the operations and guard variables. Each term represents a possible execution instance for a particular control path. We call these product terms *traces*.

Shown in Fig. 1 is *Kim*'s example[6] in which two guards 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. Shown in Fig. 2 is a problem instance with a control correlation between two control fork-join structures running in parallel. 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, but only 5 guard variables need to be defined.

A. Speculative execution model

Very often it is beneficial to determine the control value simultaneously with branch execution. If necessary

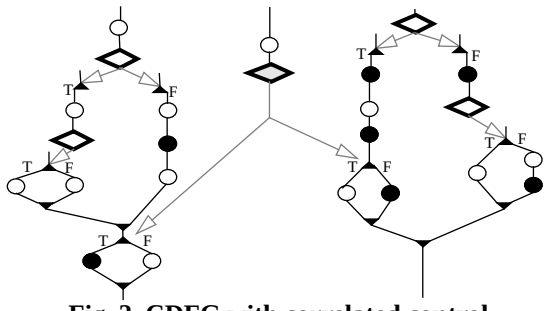

Fig. 2. CDFG with correlated control

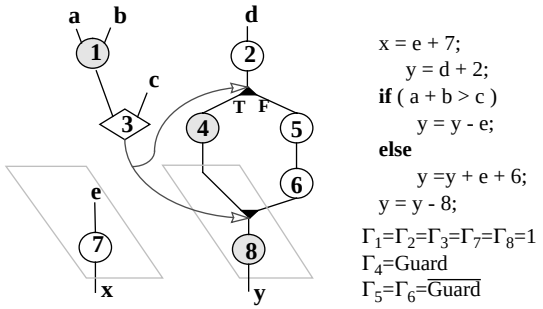

Fig. 3. Speculative operation execution

resources are available, the operations from both 'true' and 'false' branch paths may be scheduled for execution (*pre-executed*) before the conditional value is actually evaluated. Fig. 3 shows an example in which speculative execution allows faster execution using the same set of single-cycle resources (one adder, one subtracter and one comparator). Since they use different resources, operations 4 and 5 can be executed in parallel with a conditional 3. As indicated by the dashed lines, operations 7 and 8 can be scheduled earlier on the 'true' path in order to allow execution in three cycles as opposed to four cycles required by the 'false' path. It is not necessarily beneficial to discard 'non-minimal' solutions early: notice that the schedule where operations 7 and 8 are executed on the fourth cycle in both paths leads to a simpler control structure. Directed arcs in Fig. 3 represent the control dependency between the conditional and fork/join nodes. No speculative operation execution is possible if the dependency between the conditional and the fork is enforced. In general, the dependency between the conditional and the join need not be enforced as well. In this case, (given sufficient resources) the execution time is bounded solely by data dependencies. This can lead to an exponential explosion of operator instances for nested complex control.

To incorporate the pre-execution mechanism in our symbolic scheduling technique, *the control dependency between the conditional and the fork node is removed, and the dependency between the conditional and the join node is enforced instead*. CDFG operations can be scheduled at different time steps on distinct control paths, but cannot be scheduled more than once per path. Using OBDD techniques this model can be solved exactly. The experimental results show that this technique successfully exploits parallelism not explicit in the input specification.

Application of the technique to the *Maha*[11] example is shown in Fig. 4 (directed arcs represent control dependencies and undirected lines correspond to data dependencies). Notice that a great deal of freedom is added to the schedule: e.g. operations A8 and S8 can be executed during an arbitrary time step subject only to resource constraints. Given sufficient resources, a critical path length of 8 in the original graph can be reduced to just 4 (operations: S6, A6, S7, A7). This formulation does not allow for operations following the

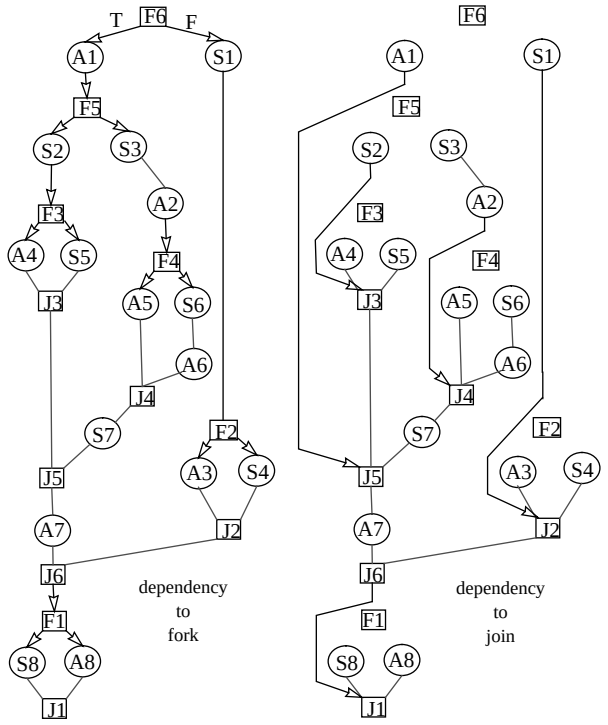

Fig. 4. CDFG transformation for Maha example

join to be executed in a speculative fashion before the corresponding conditional is resolved (e.g. S7 cannot be scheduled in the second cycle due to a dependency from A2). Notice, however, that there is still a lot of freedom to exploit parallelism: since there are no dependencies left among the conditionals, all 12 control paths can start executing *simultaneously.* Furthermore, the *order of execution of conditionals is not restricted to the one prescribed in the input formulation*. It can happen that a top-level conditional cannot be resolved prior to some other nested conditional in input specification. A very simple example of such a behavior is shown in Fig. 5. Assuming that two adders ('white' operation), one subtracter and one single-cycle comparator are available, the schedule executes in three steps with the topmost conditional left unresolved until the end of the very last cycle. The knowledge that the innermost conditional is resolved during the first cycle, however, is essential in order to complete the schedule.

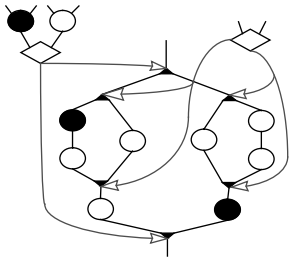

Fig. 5. Reversed conditional resolution

B. Derivation of 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. ASAP (as soon as possible) and ALAP (as late as possible) bounds are constructed to limit 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 variables which cannot be true in any feasible schedule.

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

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

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

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

2. Precedence relations: In case of the speculative execution, care must be taken when to enforce precedence between the operations. If operation i precedes operation j and $\Gamma_i \supseteq \Gamma_j$ (e.g. operations 2 and 4, or 5 and 6, in Fig. 3) then for every time step s in the range $[(ASAP)_j, (ALAP)_i]$ the following constraint must be satisfied:

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

In other cases, when there is a join between i and j (e.g. operations 6 and 8, in Fig. 3), the precedence relation is enforced only on the paths covered by Γ_i:

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

3. Termination: 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 $(j_1 ... j_n)$ are operations that are immediate predecessors of the sink node in the CDFG.

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

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

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

F_{sl} is a Boolean function describing that resource r_l is 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 $(n_{sl}\text{-}k_l)$ resources (among n_{sl} potential operations in time step *s*) cannot be scheduled. By suitable choice of F_{sl}, functional unit, bus and register constraints can be generated.

5. Removal of redundantly scheduled operations: The set of traces obtained in this fashion may include traces where some operations are scheduled in a redundant fashion. (After the conditional is resolved, some operations from paths not taken may still be scheduled if there are available resources). Although this is not a fundamental problem in the schedule, these redundant operations should still be removed to reduce potential power consumption, interconnect and storage requirements. It is relatively straightforward to eliminate such traces from the result which is in OBDD form. Assume that a conditional c_k is resolved prior to time step *j*, and that the guard corresponding to it is G_k. Then all the variables that correspond to operation *i*'s instances scheduled for time steps ≥ *j* in paths where G_k is true have to assume value '0' if:

$$\Gamma_i G_k = 0 \quad (5.a)$$

Similarly, in paths where G_k is false, all the variables that correspond to operation *i*'s instances scheduled for time steps ≥ *j* have to assume value '0' if:

$$\Gamma_i \overline{G_k} = 0 \quad (5.b)$$

C. Trace validation

A trace which satisfies all of the constraints may still not be part of a valid execution instance in the sense that it may not be compatible with any set of traces forming an executable schedule. A valid schedule must be both *causal* and *complete* for all control paths. The *causality* requirement dictates that the schedule cannot use knowledge of the value of a conditional prior to the time step in which it is executed. Fig. 6 illustrates a situation in which two traces corresponding to alternative values of the guard G_k (corresponding to the conditional c_k) are not compatible 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). Compatible 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 combination.

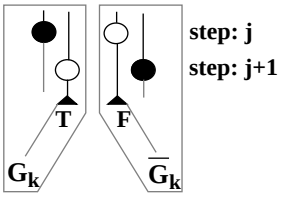

Fig. 6. Trace matching

Trace validation ensures that each validated trace is part of some executable ensemble schedule. The validation is efficiently preformed by an *iterative* algorithm shown in Fig. 7. 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*, $\boldsymbol{C} = [c_1, c_2 \ldots c_n]$ - set of all conditionals, $\boldsymbol{G} = [G_1, G_2 \ldots G_n]$ - set of guards corresponding to the conditionals, $\boldsymbol{R(j)} = [R_1(j), R_2(j) \ldots R_n(j)]$ - ***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 ($\boldsymbol{G\text{-}G_{res}}$) 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 all possible combinations from ($\boldsymbol{G\text{-}G_{res}}$).

The algorithm checks for partial matching up to step *j* for

```
i = 0;
do {
    i++;
    S(i) = S(i-1);
    for each time step j {
        S' = ∃(V−V'(j)) S(i)
        for each conditional c_k {
            S' = S' R_k(j) + ∀_{G_k} (S' R̄_k(j))
            if (S'==0) { S(i)=0; exit; }
        }
        S(i) = S(i)S';
    }
} while (S(i)!=S(i-1));
```

Fig. 7. Trace validation (TV) algorithm

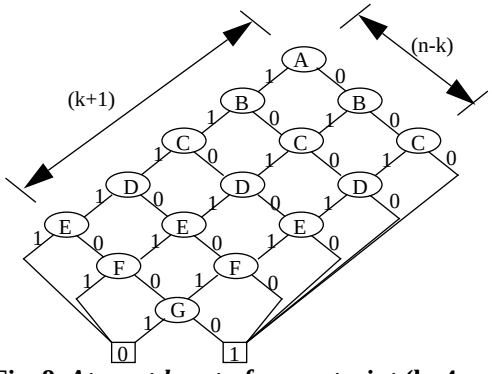

Fig. 8. *At-most-k-out-of-n* constraint (k=4, n=7)

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 iterates 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. A formal discussion of the algorithm is reported in [14].

III. CONSTRUCTION

A. OBDD structure and ordering

The constraints described in Section II 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. 8 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 notice 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.

B. Iterative construction process

Although the final OBDD typically has relatively small size the size of OBDDs at intermediate stages can be relatively large, resulting in slow construction or large memory requirements. Using iterative construction [15] 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 observed that this construction typically results in smaller intermediate OBDDs and very moderate that generation of 'garbage'. 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. Lastly, since a valid partial schedule is available, it is seems possible to formulate simple but efficient heuristics that preserve whole sets of candidates. This can be useful in cases when the size of OBDD becomes too large.

IV. EXPERIMENTAL RESULTS

Tables 1 and 2 show experimental results for several benchmarks. *Maha*[11], *Kim*[6] and *Waka*[16] are conditional trees, *MulT*[17] has two parallel trees. *Parker* is Maha with addition A6 becoming a subtraction.The *Maha* solution with one adder and one subtracter is the same as in [4][17]. Allowing more resources (two adders, three subtracters) an improvement of 0.125 (average path length) was made over the best previous result. In *Parker*, the 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. Note that 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*, but this assumption completely eliminates the need for speculative execution in the *Kim* and *Waka* benchmarks. Note that the dynamically changing values of the guard variables encode the control path being taken at a particular time step. Hence, we normally assume a single cycle comparator which is scheduled and whose output is only available in the successive cycle. This assumption is true for those results in which the number of comparators is indicated below. Given this assumption, our technique still derives the same result for *Kim* as reported in [17]. In *Waka*, however, one path is a cycle longer than that reported in [4]. In *MulT* a one cycle *shorter* solution was found by exploiting dynamic scheduling of conditionals belonging to parallel trees.

In both *Maha* and in the example in Fig. 2, having more than one unit of each type cannot improve the longest path without speculative execution. For Fig. 2, an increase to 3 adders, 2 subtracters, and 2 comparators will improve the longest path from 6 to 4 cycles. This example demonstrates the ability of our approach to perform code motion and

Table 1: Experimental results

	Maha		*Kim*	*Waka*	*MulT*
#cycles	5	4	6	7	3
#cycles(avg)	3.31	2.25	5.75	5.0	3.0
non_speculative	8	8	7	7	4
#adders	1	2	1	1	2
#subtracters	1	3	2	1	1
#comparators	-	-	1	2	1
#variables	65	49	71	55	26
#nodes	428	325	543	271	116
#traces	15	43	124	21	15
CPU time [s]	13.80	6.32	7.63	2.84	3.95
single-cycle adders, subtracters and comparators assumed					

dynamically schedule a complex control instance. We observed that none of the standard benchmarks needed out-of-order execution of the conditionals in the optimal solutions.

Table 2: Comparison with others: average(longest) paths

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

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.

V. CONCLUSION

We described a symbolic scheduling formulation that allows incorporation of speculative operation execution in exact control-dependent scheduling of 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. Boolean 'guard' functions are used to precisely identify paths where operations have to be scheduled and the execution order of the conditionals is dynamically resolved. An efficient and systematic iterative construction method was presented along with benchmark results. Several areas of improvement are the targets for our future work: inclusion of a constraint to implement operation chaining, incorporation of control/interconnect costs in the formulation and extensions to restricted forms of backward loops. An efficient approach to remove the restrictions we assumed in this formulation will be considered as well.

VI. 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.

VII. 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, "Ensemble Representation and Techniques for Exact Control-Dependent Scheduling", *Proc. 7th International Symposium on High Level Synthesis*, 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] J. Yang, G. DeMicheli, M. Damiani, "Scheduling with Environmental Constraints based on Automata Representations", *Proc. EDAC-94*, Paris, France, March, 1994.

[19] L. Yang and J. Gu, "A BDD Model for Scheduling", *Proc. CCVLSI*, 1991.