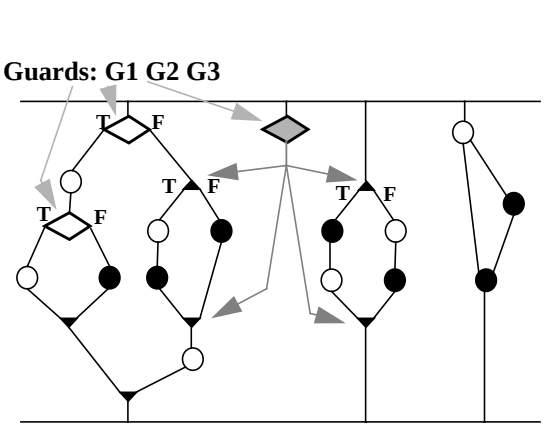
optimality and is run-time efficient. The solution for the complex control case relies directly on having available the totality of solutions for the scheduling problem. Future research includes formulation of operation pre-execution to allow concurrent execution of control branches to improve the performance of the schedules. Extensions to enable limited forms of scheduling for backward loops are planned as well.

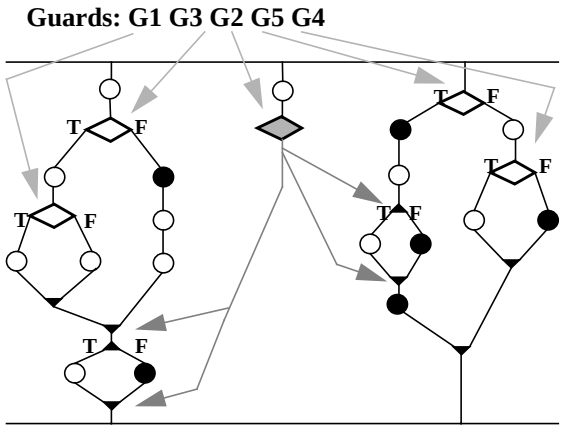
7. References:

1. R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, Vol. C-35, No. 8, August 1986.
2. R.E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams", *ACM Computing Surveys*, Vol. 24, No. 3, September 1992.
3. R. Camposano and R.A. Bergamaschi, "Synthesis Using Path-Based Scheduling: Algorithms and Exercises", *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.
4. M. Fujita, H. Fujisawa and Y. Matsunaga, "Variable Ordering Algorithms for Ordered Binary Decision Diagrams and Their Evaluation", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 12, No. 1, January 1993.
5. S. H. Huang et al. "A Tree-Based Scheduling Algorithm for Control Dominated Circuits", *Proceedings of the 30th ACM/IEEE Design Automation Conference*, 1993, pp. 578-582.
6. C.-T. Hwang, Y.-C. Hsu, Y.-l. Lin, "Optimum and Heuristic Data Path Scheduling Under Resource Constraints", *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.
7. R. Jacobi, N. Calazans and C. Trullemans, "Incremental Reduction of Binary Decision Diagrams", *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1992.
8. T. Kim, J.W.S. Liu, C. L. Liu, "A Scheduling Algorithm for Conditional Resource Sharing", *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1991, pp. 84-87.
9. H. Komi, S. Yamada, K. Fukunaga, "A Scheduling Method by Stepwise Expansion in High-Level Synthesis", *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1992.
10. D. Landskov, S. Davidson, B. Shriver, P. Mallett, "Local Microcode Compaction Techniques", COmputing Surveys v. 12, n.3, Sept. 1980.
11. J.-H. Lee, Y.-C. Hsu and Y.-L. Lin, "A New Integer Linear Programming Formulation for the Scheduling Problem in Data Path Synthesis", *Proceedings of the 26th ACM/IEEE Design Automation Conference*, 1989.
12. S. Malik, A.R. Wang, R.K. Bryant, A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment", *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1988.
13. M. C. McFarland, A. C. Parker and R. Camposano, "The High-Level Synthesis of Digital Systems", Proceedings of the IEEE, Vol. 78, No. 2, February 1990.
14. A. C. Parker, J.T. Pizarro, M. Mliner, "MAHA: A Program for Datapath Synthesis", *Proceedings of the 23th ACM/IEEE Design Automation Conference*, 1986.
15. B. Pangrle D. Gajski, "Slicer-- A State Synthesizer for Intelligent Silicon Compilation", *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1986.
16. C. Papadimitriou and K. Steiglitz, "Combinatorial Optimazation Algorithms and Complexity", Prentice-Hall Inc., New Jersey, 1982. pp. 132.
17. P.G. Paulin and 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.
18. I. Radiovojevic, F. Brewer, "Symbolic Techniques for Optimal Scheduling", UCSB Tech. Report #93-11, May 1993.
19. K. Wakabayashi, T. Yoshimura, "A Resource Sharing and Control Synthesis Method for Conditional Branches", *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1989.
20. K. Wakabayashi, "Global Scheduling Independent of Control Dependencies Based on Condition Vectors", *Proceedings of the 29th ACM/IEEE Design Automation Conference*, 1992, pp. 112-115.
21. L. Yang and J. Gu, "A BDD Model for Scheduling", *Proceedings of CCVLSI*, 1991.

schedule ensemble is four traces (corresponding to all possible values of the guards). In these examples, therea re no limitations on the possible scheduling allowed for the conditionals other than the data-dependencies derived from the CDFG. More complicated problems (Examples G3 and G5) are introduced below as well. In these examples, there is a control correlation between two control fork-join structures running in parallel. This means that the guard representing the path chosen for the correlated forks has the same value in both forks. Thus, an intersection of the sets of conditionals of the two machines is non-empty and corelates to the shaded comparators. This increases the freedom in the scheduling since a smaller number of traces form an ensemble solution. An optimal schedule for these cases must include the correlation to not eliminate those traces which are invalid for an uncoorelated set of forks. This requirement is trivial in the proposed formulation by simply assigning the same guard variable to both forks. Potentially much more complex correlations are clearly possible as are arbitrarily complex conventional control structures. Note that the complexity of the formulation grows with the number of guard variables, not the (possibly exponential) number of traces or control paths. Examples G3 and G5 have 6 and 18 possible control paths respectively, but only 3 and 5 guards respectively are needed. In Table 3, results for the construction and validation of the presented examples is described. For these examples, all paths were scheduled to complete within the time for the written longest cycle, leaving a great deal of freedom for the alternate (shorter) paths. The posted number of traces correspond to the number of *all* components from which valid ensemble schedules can be constructed, the number of possible ensemble schedules is potentially much larger. The posted CPU times are for a DEC Station 5000 machine and a custom C++ BDD package developed at UCSB.



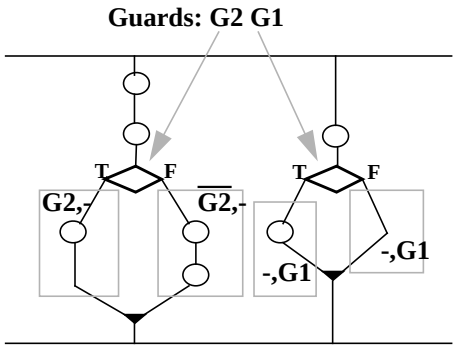
G3 example



G5 example

Table 3. Control-Dependent Results

	Kim	G2	G3	G5
cycles (longest)	8	6	5	7
adders	2	1	1	1
subtractors	1	1	1	1
comparators	1	1	1	1
#control paths	3	4	6	18
#guards	2	2	3	5
#variables	76	27	46	71
#nodes	3934	175	488	2071
#valid traces	6863	35	46	652
CPU time [s]	10.9	1.1	2.9	19.8



G2 example

6. Conclusions and Future Research

In this paper, we have presented a technique for exact scheduling applicable to a wide variety of constraints including register, bus, and resource usage. These techniques are very time competitive with current exact techniques, but also offer the availability of all possible solutions of the scheduling problem in a compact form. This availability lead in a natural way to a technique for scheduling arbitrary forward branching control which retains

be completed for both values of $G(c_k)$. We wish to eliminate all traces from the complete set of schedules which are not part of a valid ensemble schedule (valid for all control paths). This will leave a set of traces which can be combined into all possible ensemble schedules for the instance. Note that the number of valid traces is possibly *far* smaller than the number of ensemble schedules, since a trace may have several matching traces in the ensemble, and the can be exponentially many traces forming an ensemble schedule.

To perform trace validation efficiently, we propose an iterative algorithm based on the OBDD formulation. The idea is as follows: for a set of traces where a conditional c_k is not resolved prior to control step *i* we have to identify those traces corresponding to $G(c_k)$ and $\overline{G}(c_k)$ that match in all preceding control steps, including the current one. This can be done efficiently (since all the matches among all traces are found in parallel) by smoothing out of the OBDD all the variables corresponding to operations in time steps > *i*, and then performing a universal abstraction on the guard corresponding to the value of conditional c_k. For a set of traces in which the conditional c_k is already known (executed prior to *i*) nothing need be done, and we continue to process the rest of the conditionals in a same way. This process must be run on each conditional for each time step in which some conditional can be executed since at that point, knowledge of the conditionals are changed for some traces. The process simultaneously performs the matching on all potential traces, and also ensures that a valid trace exists for each contol point time step since a trace must exist to be matched. However, when trace matching is performed at the subsequent time steps, traces are discarded which potentially invalidate some of the schedules whose completeness was validated during the previous cycles. For this reason, the algorithm is iteratively applied until no more traces are removed. The code fragment below describes the validation algorithm. The following notation is used: S(0) - set of traces generated by scheduler, S(i) - set of traces at iteration *i*, O(i) - set of variables corresponding to time steps > *i*, S' - matching paths, c_k - conditional; $G(c_k)$ - guard associated with a conditional c_k, $\exists_x f = f_x + f_{x'}$ - existential abstraction of a boolean function, $\forall_x f = f_x f_{x'}$ - universal abstraction of a boolean function, $R_k(i)$ - function indicating that a conditional c_k was scheduled prior to time step *i*

$$R_k(i) = \sum_{j<i} C_{jk} \qquad \text{(Eq. 8)}$$

iteration_counter = 0;
do {
  iteration_counter++;
  S(i) = S(i-1);
  for each clock step where at least one conditional can be scheduled {
    $S' = \exists_{O(i)} S(i)$
    for each conditional c_k {
      $S' = S' R_k(i) + \forall_{G(c_k)} (S' \overline{R_k(i)})$
      if (S'==0) { S(i)=0; exit; }
    }
    S(i) = S(i)S';
    if (S(i)==0) exit;
  }
} while (S(i)!=S(i-1));

The number of iterations is bounded in the worst case by the number of conditionals (same as the number of guards). This corresponds to the case where control has a form of an *if-else-if* ladder. On tree-like structures, the number of iterations is bounded by the hight of the tree. However, typically the algorithm converges after only one iteration even on complex control stuctures described in the folowing section. This is due to the relative rarity of trace removal propagating to earlier stages in the validation. It is interesting to note that, due to the very simple control structure and very loose resource constraints, no invalid traces were present in the initial solution obtained from symbolic scheduling of the Kim example.

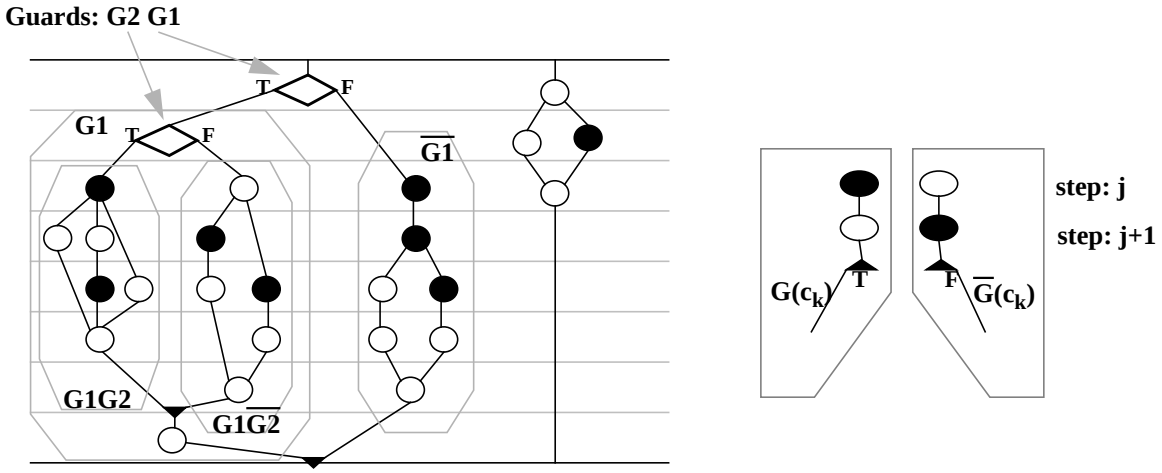
5.2. APPLICATIONS TO PARALLEL CONTROL STRUCTURES

Our model is not limited to those CDFGs which have a conditional tree control structure. Example G2 on the next page is a problem instance where two fork-join pairs are executed completely in parallel, thus complicating the issue of resource availability among the operations. Notice, that the G*i* functions implicitly include some guards variables as 'don't cares', indicating that there is no mutual exclusiveness among the operations belonging to the parallel execution paths, but each path is exclusive with respect to operations on the guard of the opposite sense. Validation of this case uses the same algorithm with convergence in one iteration. Note that a complete

be simultaneously constructed using the same constraints as before. Only Eq.1 requires the following exstension:

$$\forall j \left(\sum_{k \in R_j} C_{kj} \prod_{(i \neq k) \in R_j} \overline{C_{ij}} \right) G_j + \left(\prod_{i \in R_j} \overline{C_{ij}} \right) \overline{G_j} = 1 \qquad \text{(Eq. 1a)}$$

where G_j is a boolean function (defined on the guard variables) that conditions (guards) the execution of operations corresponding to a given control arc. Eq. 1a states that if the condition is fulfilled an operation *j* can be scheduled during one and only one control step -- if the condition is false then operation *j* is not scheduled at all. Consequently, if two operations *i* and *j* are guarded by G_i and G_j whose intersection is empty, then *i* and *j* are mutually exclusive. In general, our solution is a collection of product terms, each one including all the operations and some (not necessarily all) guard variables, reperesenting a possible execution instance for a particular control path. In the figure below is Kim's example[8] in which we assume that one comparator, one subtractor ('black' operations) and two adders ('white' operations) are available. Indicated blocks correspond to operations that share the same function G. Operations belonging to a control-independent portion of CDFG are not guarded and thus belong to all execution paths. Consequently, they can be scheduled simultaneously under all control combinations. Due to very loose resource constraints in this example, there is a large number of possible schedules (see Table. 3), but we were able to find all valid execution traces in less than 11 seconds. It is obvious that the control paths can be scheduled in 7, 8 and 6 steps respectively. However, the fastest possible execution of all the paths is just one possible criterion for optimality of the schedule. For example, to simplify the control complexity it might be advantageous to schedule the control-independent path uniquely in all possible control instances. We were able to verify in just 12.2 seconds that the (7,8,6)-cycle schedule cannot fulfill such a requirement. If we add one control step to one execution path and accept (7,8,7) schedules, it is possible to find a solution (depicted in the example figure) where not only the control-independent path is uniquely scheduled, but the resources used during steps 5 and 7 are same in all three control paths.



Kim's example

5.1. TRACE VALIDATION

After all possible execution traces (for individual paths) are generated, the must be validated, since some of them could not be a part of a valid ensemble schedule. Such a schedule has to be *causal* and *complete* for all control paths. The *causality* requirement dictates that the controller cannot use the knowledge of a *conditional* (an operation that generates a control signal) prior to the time step when it is executed. The figure above illustrates a situation in which two traces corresponding to opposite values of the guard G cannot be chosen to form a valid schedule unless conditional c_k is evaluated prior to step *j*. This is because the knowledge of whether one should execute a multiply or an add requires the knowledge of which path is being executed. Unless the conditional has already been executed, a machine could not determine which operation to execute. The traces corresponding to guard values $G(c_k)$ and $\overline{G}(c_k)$ have to be exactly matching before the conditional is evaluated, otherwise pairing of these two traces violates causality.

The *completeness* requirement states that at any point in the schedule, there must be valid execution traces which correspond to all currently unresolved conditionals. Once the conditional c_k is executed, the causality requirement is no longer required. The schedule simply bifurcates into two different execution traces. However, it is possible that one (or both) execution traces do not exist or are removed when causality is checked for some conditional c_j that is scheduled after c_k. In this case, the potential partial schedule is invalid since the schedule cannot

of the constraints. This approach makes it easy to ‘AND’ together BDDs of comparable sizes and simplifies garbage collection. Since a construction strategy which minimized the run-time did not necessarily minimize the size of the results, it was deemed more important to reduce the run-time and that technique was used in the presented results. This construction starts with an interleaved dependency-based ordering and the BDD lists corresponding to Eq. 3 and Eq. 7 are conjoined and then Eq. 1 and Eq. 2 are conjoined with the result sequentially by adding medium sized conjunctions. Finally, Eq. 4 (register constraints) is applied to the result. This technique attempts to reduce the size of the intermediate functions corresponding to partial products. It was found that constructing intermediate sized conjunctions of Eq. 1 and Eq. 2 before ‘ANDing’ to the other equations increased efficiency. Finally, variables for which all scheduling solutions have either a 1 or a 0 (unate variables) can be iteratively removed resulting in a 40%-50% reduction in the size of the result. This is possible even early on in the construction since even for schedules in which there is a great deal of freedom, many of the spans set by ASAP/ALAP analysis are too large-- leading to a large number of zero variables. Results of this construction for several benchmark examples are shown in the Tables below. HAL and WAVE are the differential equation and fifth-order wave filter respectively. The percentage figures reported in the vars and nodes rows represent the size of the posted value compared to the value without unate varaible extraction. In the figure, the size of the BDD is shown relative to the number of variables squared. It can be seen that typically, the structures are far smaller than n2, where n is the number of variables in the instance. The BDD is a very compessed format for representing these differing schedules, for the 21 cycle case, a BDD with 237 nodes represents all possible 5,149 different schedules for this example.

.

Table 1. HAL (pipelined 2-cycle multiplier)

cycles	6	7	8
adders	1	1	1
multipliers	2	2	1
busses	3	2	2
registers	5	5	5
ordering	PBO(I)	PBO(I)	PBO(I)
#var’s	4 (15%)	28 (74%)	28 (57%)
#nodes	7 (23%)	128 (80%)	81 (60%)
#paths	3	11	36
CPU [s]	0.39	0.82	1.04

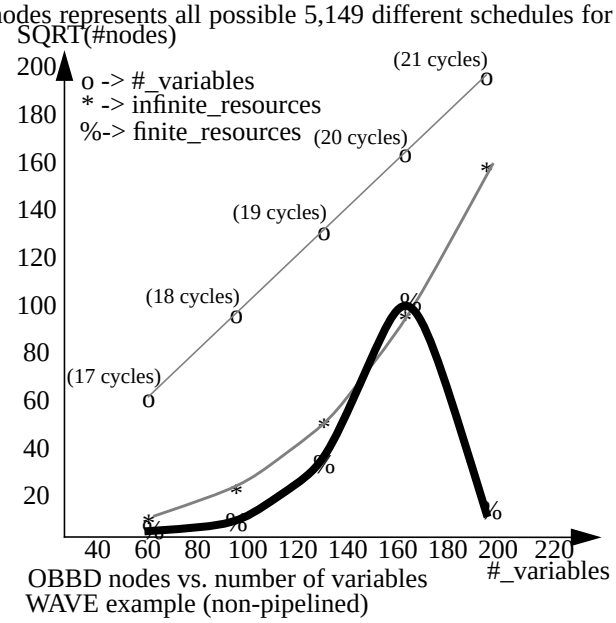


Table 2. WAVE (non-pipelined 2-cycle multiplier)

cycles	17	18	19	20	21
adders	3	2	2	2	2
multipliers	3	2	2	2	1
busses	6	5	4	4	4
registers	10	10	10	10	10
ordering	PBO(I)	PBO(I)	DBO(I)	DBO(I)	DBO(I)
#var’s	9 (15%)	29 (30%)	97 (75%)	147 (90%)	47 (24%)
#nodes	19 (26%)	62 (32%)	1183 (83%)	11430 (94%)	237 (30%)
#paths	18	18	1,071	991,638	5,139
CPU [s]	0.57	3.02	14.48	310.54	40.30

5. Guard Variables and Branches

The Boolean representation can be extended to allow control-dependent scheduling. This requires the introduction of a set of *‘guard’* variables. Each guard labels particular fork/join pair, where the guard is true for one branch and false for the other. Every path through an arbitrary combination of fork/join pairs can be succintly described by a product of the corresponding guard variables. Thus, *all* schedules for *all* forward control paths can

$$i \rightarrow j \Rightarrow \sum_{k \in R_{i,j}} \left(\sum_{(l \geq k) \in R_{i,j}} C_{kj} C_{(l - di + 1)\, i} \right) = 0 \qquad \text{(Eq. 2a)}$$

where d_i is the duration of operation i, and $R_{i,j}$ is $[(\text{ASAP})_j \ldots (\text{ALAP})_i + (d_i - 1)]$. Resource constraints on the number of functional units can then be described by the following symbolic code:

```
foreach C_ij {
    if (pipelined) {
        insert function C_ij into resource constraint for type j, time step i;
    } else {
        foreach k=[i .. (i+(d_j-1))] {
            insert function C_ij into resource constraint for type j, time step k;
        }
    }
}
```

Notice that for a multi-cycle non-pipelined operation j, its variable $C_{i,j}$ is included in d_j type resource equations (Eq. 3). If the operation is pipelined, it affects only one equation (the one corresponding to the control step when the operation starts its execution). Thus, the constraint complexity to describe pipelined functional units is same as that of single-cycle units. Surprisingly, it is a simple process to allow operation chaining as well, even though there is no direct representation of integer time bounds. This is possible since we can (as was done in [9]) augment the list with forward dependencies between operators which specify that j (a successor to i) is the nearest successor to i that cannot be chained into a control step with i given the current clock. This relation is written (i->>j) and has a particularly simple realization as shown in the modified Eq. 2b below.

$$\sum_{k \in R_{i,j}} C_{kj} C_{ki} = 0 \qquad \text{(Eq. 2b)}$$

This is a simpler constraint equation than Eq. 2 above because the data-flow precedence relation is transitive resulting in a simpler constraint for these additional operation dependencies required by potentially chained operations.

This formulation allows encapsulation all the possible legal schedules satisfying specified resource constraints into a single data-structure, and can iteratively be used to perform a minimization process (finding a representative optimal solution in terms of increasingly tight constraints). However, we believe that there are good reasons for retaining *all* solutions meeting the constraints instead of finding one representative solution with respect to some pre-specified cost function early in the design process. In this case, the choice of the scheduling solution is determined before the operation binding, interconnection and placement decisions, which can interact greatly with the schedule, have been made. Unless the scheduling is iteratively recalculated, the system makes a sub-optimal choice of schedule. Alternatively, if the size of the data structure representing all solutions can be kept reasonably small, incremental changes consisting of additional resource or timing constraints can be inexpensively and dynamically tested against the set of schedules. This interaction still admits optimal solution of the ensemble problem since it merely removes schedules that have become infeasible with respect to the additional constraints.

4. BDD Ordering and Construction

Efficient OBDD variable ordering is a notoriously difficult problem [1][2][4][7][12]. An efficient strategy to optimize the orders of all of the constraints is difficult since the optimal orderings for typical ensembles of Eq.1 and those of Eq.3 conflict. Eq.1 tends to subdivide the problem at the boundaries of the operation's span. On the other hand, Eq.3 tends to subdivide the problem at time-step boundaries since all variables occur on the same time step for each instance. For this reason, three simple variable ordering heuristics are proposed and implemented[18]. Dependency-based ordering (DBO) in which successor operations have higher indices than predecessor operations. Mobility-based ordering (MBO) in which BDD indices increase with mobility and Path-based ordering (PBO) in which BDD indices increase with the length of the path an operation is on. Interleaving variables on time steps tends to cluster variables belonging to the same clock cycles and can be expected to have a positive effect on equations that describe resource constraints, but negative effects on the constraints Eq. 1. It was found that DBO with interleaving was typically the most effective strategy in producing the smallest output representations.

Several different strategies for constructing the BDD were investigated. It was found that keeping lists of BDD pointers ordered by the size of the fragment made an effective strategy for ordering the massive conjunction

have good ordering and construction heuristics.

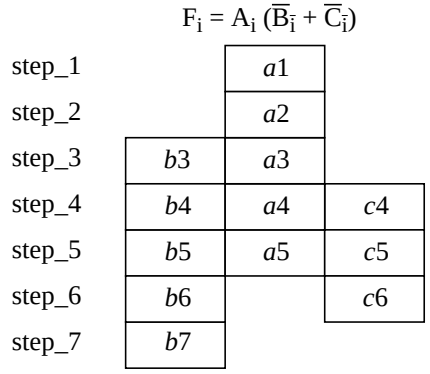
3.3. Register and Bus Resources

A bound on the number of available registers can be implemented using a slightly more complex resource function and then simply plugging this new function into the constructor for Eq. 3. Eq. 5 indicates that if an operation *i* precedes the operations ($j_1..j_n$) at a particular control step *k*, a register is required. This register is required to keep the value of the output of operation i if the successor operations cannot use it immediately.

$$(i \rightarrow (j_1 \ldots j_n)) \Rightarrow F^r_{ki} = \left(A_{ki} \sum_{j=j_1}^{j_n} \overline{B_{kj}} \right) \quad \text{(Eq. 5)}$$

$$B_{kj} = \sum_{l=ASAP_j}^{k} b_{lj}\,,\ A_{ki} = \sum_{l=ASAP_i}^{k} a_{li} \quad \text{(Eq. 6)}$$

Notice that this formulation allows for possible chaining of operations since the constraint predicts that no register is required if the operations are all assigned to the same control step. In practice, each Eq. 5 constraint can simply be plugged into the typed resource constraint equation (Eq. 3). Note that in this case k_r is the number of registers allowed and n_{kr} is set to the number of candidate Eq. 5 functions at the k^{th} control step. The construction



Bi	Ai	Ci
0	$a1$	0
0	$a1 + a2$	0
$b3$	$a1 + a2 + a3$	0
$b3 + b4$	$a1 + a2 + a3 + a4$	$c4$
$b3 + b4 + b5$	1	$c4 + c5$
$b3 + b4 + b5 + b6$	1	1
1	1	1

of the register requirement for an example operation with 2 successors is shown in the figure. The initial operation can be scheduled into one of 5 time steps, F_i is true if a register is required at time step i.

Busses can be treated in a similar fashion. If operation *i* precedes operations ($j_1..j_n$), Eq.7 indicates that at a particular control step *k* a bus may be needed to read an operand (upper part of Eq.7) or write a result (lower part of Eq.7). Notice that the formulation allows a rather complicated situation (same operand used as an input to a number of operations) to be modeled in a simple fashion.

$$\begin{aligned} i \rightarrow (j_1 \ldots j_n) &\Rightarrow F^{br}_{ki} = \sum_{l=1}^{n} C_{kj_l} \\ i \rightarrow (j_1 \ldots j_n) &\Rightarrow F^{bw}_{ki} = C_{ki} \end{aligned} \quad \text{(Eq. 7)}$$

Given the Eq. 7 constraints, we can again treat them as generic resources and plug them into Eq.3 for each time step, since only k_b out of the n_b functions can be active at each particular phase of the time step. The bus constraints apply to for *'read'* and *'write'* phases separately, making no assumption that a number of writes is smaller than the number of read operations at each control step. However, we do assume that read and write transfer phases are interleaved. Similar constructions can be used to constrain the number of other typed resources. It is important to note that this formulation of these constraints does not require the addition of more implementation variables (as is the case for ILP formulations of Bus constraints[5][11]) and can be simply generalized to any Boolean function of the variables as typed resources. The direct OBDD construction allows these resource constraints to be efficiently constructed even for very complex constraints functions.

3.4. Chaining, Multi-cycle and Pipelined Function Units

This formulation can be extended to allow multi-cycle, pipelined and chained operations by making suitable modifications to the constraint equations. If a multi-cycle/pipelined operation *j* can be scheduled at time step *i*, a variable *Cij* is created. Eq.2. (data dependency) has to be modified:

In Eq.1, C_{ij} is true if operation j is scheduled in time step i, R_j is the set of time steps for which operation j can be scheduled based on the ASAP and ALAP bounds (i.e. $[(ASAP)_j$ ... $(ALAP)_j]$). Eq.1 insures that only one instance of a given operation is scheduled in any valid solution. Each operation in the instance requires this constraint. Eq.2 is written for every existing data dependency $i->j$ (operation i immediately precedes operation j) and $R_{i,j}$ is a closed segment of time steps $[(ASAP)_j, (ALAP)_i]$. This equation occurs for each data dependency and insures that all successive dependent operations are either chained ($l>=k$) or are set in subsequent clock cycles ($l>k$). Formally, the conjunction of all of the above constraints creates a function which is true for each input combination of variables corresponding to a valid schedule. Since the OBDD is a graph based representation, each path in the OBDD leading to 1 (the valid node) represents a valid schedule in the sense that all of the data-flow dependencies are satisfied and that the operation in not scheduled redundantly.

3.2. Resource Constraints

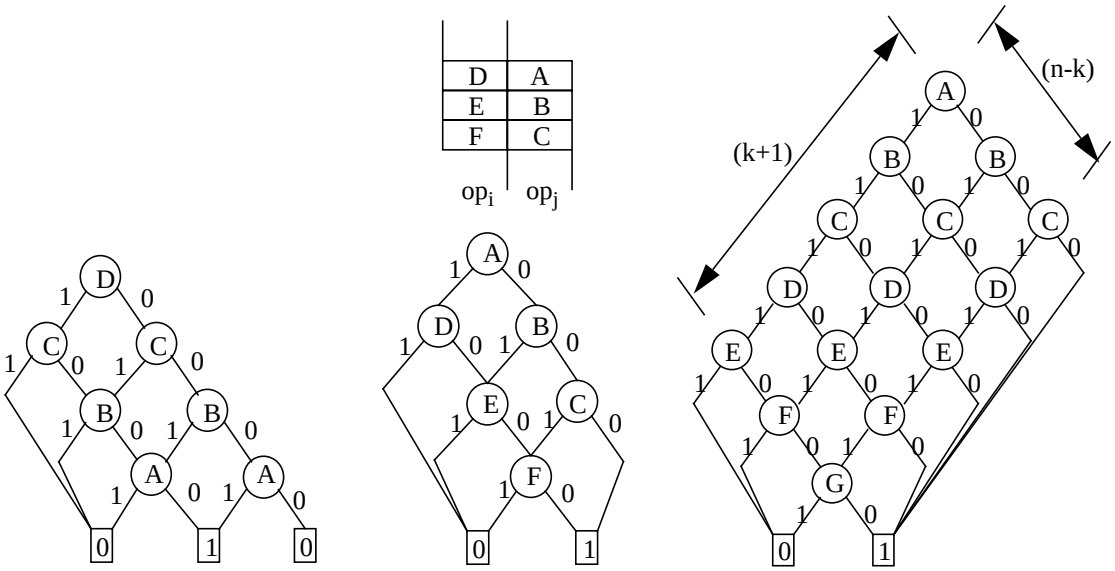
Typed resource constraints can be easily formulated as well. If k_l resources of a certain type r_l (e.g. multipliers, adders, ALUs, registers, busses) are available we can formulate a *'resource-constraint'* Eq.3:

$$\forall i, \forall r_l \sum_{1 \le (lp \ne lq) \le n_{il}} \overline{F_{il_1}}\,\overline{F_{il_2}} \ldots \overline{F_{il_{(n_{il}-k_l)}}} = 1 \qquad \text{(Eq. 3)}$$

F_{il} is a function describing that a resource r_l may be needed during time step i. Eq.3 is written for each time step i and each resource r_l. It indicates that at least $(n_{il}-k_l)$ resources (among n_{il} potential operations in time step i) cannot be scheduled. For example, if $k_m = 2$, single-cycle multipliers are available and there are four potential multiply operations at time step i we can write:

$$\overline{C_{im_1}}\,\overline{C_{im_2}} + \overline{C_{im_1}}\,\overline{C_{im_3}} + \overline{C_{im_1}}\,\overline{C_{im_4}} + \overline{C_{im_2}}\,\overline{C_{im_3}} + \overline{C_{im_2}}\,\overline{C_{im_4}} + \overline{C_{im_3}}\,\overline{C_{im_4}} = 1 \qquad \text{(Eq. 4)}$$

OBDD representations of the Eqs.1-3 are illustrated below. Each constraint has a simple and regular structure,



Eq. 1 (4 time step span) Eq. 2 (3 time step span) Eq. 3 (4 of 7 constraint)

with a small number of nodes. Individual Eq.1 and Eq.3 constraints are symmetric on their support and their OBDD representations are order insensitive (consisting of only *(2n-1)* and *(k+1)(n-k)* nodes respectively). Eq.2 *is* order sensitive - its optimal ordering is the interleaving depicted in Fig.1 (the number of nodes is equal to the number of variables). An attractive property of all these equations (in particular Eq.1 and Eq.3) is that, since we ***know*** their structure, OBDD representations can be built *directly* from CDFG (control/data flow graph). Thus instead of constructing the Boolean constraints as equations and then converting into OBBD format, the OBDD structure is created directly from the input data-flow graph. This process is relatively very fast especially in case of Eq.3 where the number of product terms in a sum-of-products representation is $\binom{n}{k}$ and generates no construction garbage (temporary OBBD nodes that are not referenced in the final solution). It is important to note that although individual equations have very nice spacial complexity, there can be no bound on the complexity of an arbitrary instance os the scheduling problem for any pre-specified ordering since this problem is NP-complete. However, we shall show results that indicate that typical instances, including conventional benchmarks do indeed

mined easily by simple access to the set of solutions. Additionally, since the problem is cast in Boolean form, it should be easy to add complex Boolean control and resource constraints which are difficult in other exact techniques. Finally, knowledge of the type and scope of the constraints can lead to heuristics for their application which potentially has much higher run time efficiency than solution as a general instance of ILP.

2. Previous Work

ILP can provide optimal solutions to the problem of linear scheduling as suggested by Papademitriou and Steiglitz [16], and has been implemented in several variants[6][11]. Komi et al.[9] keep 'near' optimality of solutions but improve the run time by restricting the formulation to a sliding band which is incrementally solved. Finally, a mixed ILP/BDD formulation[21] was proposed to increase the applicability of ILP formulations and simplify the constraints to improve runtime and reduce the number of variables. A prime difficulty of these approaches is the inability to optimally schedule for complex control cases.

Most current scheduling algorithms are based on variations of list scheduling[10]. These are priority based heuristic algorithms which keep a dynamic ready list of operations to be scheduled and using the priority, choose elements from the list to sequentially schedule. The advantages of this technique are simplicity and speed, especially for management of resources. Examples of list scheduling variants are Force-Directed Scheduling[17], Maha[14], and Slicer[15].

Path based scheduling[3] is a heuristic which evaluates all of the possible paths available to a complex schedule through each control branch. Each such path is scheduled as fast as possible, with limitations based on compatibility. This technique has the advantage of providing high performance schedules even in cases when the control overhead is large. However, although the technique is exponential time in the number of paths, it is still a heuristic and cannot determine hard feasibility bounds.

Recently, there has been substantial work on control based scheduling[5][8][19][20]. All of these algorithms are based on heuristic techniques although approaches differ. Wakabayashi introduced the notion of condition vectors qhich allows characterazion of the possible resource sharing consequences in a schedule. Operation pre-execution was utilized to obtain high quality results. Most research is based on the restriction to forward tree-based control structures, although restricted parallel control paths have been described.

3. Linear (Non-Branching) Formulation

In order to develop the general techniques presented later for control path dependent scheduling, we shall first present techniques for the simpler non-branching case of linear scheduling. We shall initially make the simplifying assumption of simple non-pipelined unit time delay for the operations although these restrictions will be removed later. The essential idea is to 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. Each valid linear schedule is then a ***minterm*** of the variables which satisfy all of the constraints. The OBDD represents the logic function of the operation assignment variables and is true for each valid schedule. Since the OBDD is a compressed canonical representation, potentially exponential numbers of schedules (represented by minterms) can be represented in relatively small space. In fact, we shall show that the typical size of the representation (in BDD nodes) is smaller than the square of the number of variables. Since the time complexity of OBDD operations are proportional to their size, these algorithms lead to efficient techniques for scheduling. Eq.1 and Eq.2. below, describe the scheduling constraints when no resource constraints are specified (infinite resources). In this formulation, we make use of the ASAP (as soon as possible) and ALAP (as late as possible) bounds[11] which can be derived by conventional list scheduling[17] to determine the time step spans over which an operation can be scheduled for every feasible schedule. 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.

3.1. Precedence Constraints

$$\forall j \sum_{k \in R_j} C_{kj} \prod_{(i \neq k) \in R_j} \overline{C_{ij}} = 1 \qquad \text{(Eq. 1)}$$

$$i \rightarrow j \Rightarrow \sum_{k \in R_{i,j}} \left(\sum_{(l \geq k) \in R_{i,j}} C_{kj} C_{li} \right) = 0 \qquad \text{(Eq. 2)}$$

Symbolic Techniques for Optimal Scheduling[1]

Ivan Radivojević , Forrest Brewer
Department of Electrical and Computer Engineering
University of California, Santa Barbara
Santa Barbara, CA 93106, U.S.A.
e-mail: ivan@later.ece.ucsb.edu, forrest@ece.ucsb.edu

ABSTRACT: This paper describes a new optimal formulation of the control/data-flow scheduling problem in which data-flow operations are assigned appropriate time steps subject to resource and timing constraints and data and control dependencies. Current solutions to this NP-complete problem involve either heuristic techniques or ILP based optimizations. A property of all of these methods is that once the initial constraints and a cost function are formulated, only a single representative solution is produced. The new formulation instead, generates a closed form solution of the scheduling problem in which all satisfying schedules are encapsulated in a concise BDD representation. This is obviously advantageous for synthesis systems in which certain constraints are not known until subsequent steps of the synthesis are completed. The presented formulation can provide all optimal schedules for arbitrary forward branching (non-looping) control/data paths.

1. Introduction

Operation Scheduling is the process of determining the assignment of operations to time slots of a synchronous system. Each operation is subject to data-flow dependencies, control-flow dependencies and resource dependencies. Data-flow dependencies ensure that the input operands for the scheduled operation are available when the operation is scheduled. Control-flow dependencies allow for removal of some operations when it is known that an alternate control path has been chosen. Finally, resource dependencies restrict the use of operator resources (which are necessary to perform the operations) via implementation constraints. Simple resource dependencies are limits to the numbers of parallel operations which can be performed, although implementations can create much more complex constraints. An additional constraint on the total number of allowed time slots is required to make a finite instance of the problem. A solution is a valid assignmet of the operators to time slots obeying all of the constraints and completely mapping all operations in the instance.

The operation scheduling problem plays a central role in most current high level synthesis methods since it is during scheduling that the performance characteristics of the system are determined.[13] Typically, a set of resource and timing constraints are chosen and the scheduler determines the fastest schedule meeting those requirements. Using a selection of differing resource sets leads to several scheduling solutions from which it is possible to determine the nature of resource/performance tradeoffs for the target system. Practical use of scheduling in commercial design has been almost exclusively limited to hardware implementation of DSP algorithms. This is due to the relative simplicity of the control structure and the use of large numbers of a small set of operands (adds and multiplies) in the data flow. For more general cases, conventional scheduling becomes much more difficult due to control flow complexity and complex external timing and implementation constraints. Current techniques for scheduling are inadequate for this task for three reasons: 1. Although heuristic scheduling (such as path-based[3] or list[17] scheduling) can meet certain kinds of control complexity, it can fail to find solutions in very tightly constrained problems even when such solutions exist. 2. ILP based scheduling[5] is an exact technique but current methods severely limit the kinds of control dependencies which can be accommodated using this technique. 3. Neither technique is useful in cases where the design methodology is iterative and the full set of constraints is not known prior to the scheduling run.

For these reasons, we chose to formulate the scheduling problem using a simple representation based on an OBDD[1][2] which could address all of these issues. Consider a typical ILP formulation of the scheduling problem; There are sets of integer constraints which realize each of the dependency constraints for the scheduling problem. In a typical instance, the overwhelming preponderance of variables are constrained to {0,1}. It seems logical to recast this problem into a Boolean space and to reformulate the actual integer constraints as combinations of Boolean constraints. This is reasonable since typical instances of problems use relatively small integer bounds in problems which are feasible to implement. Using a BDD representation also has the advantage that complex Boolean functions representing *all* possible solutions to a given scheduling problem might be representable in relatively small space. This has several advantages since if all solutions can be represented, the inclusion of an additional constraint is incrementally computable. Thus, the effects of later derived constraints can be deter-

1. This work was sponsored in part by fellowship donation from Mentor Graphics Corp.