Weighted Control Scheduling

Aravind Vijayakumar and Forrest Brewer
Department of Electrical and Computer Engineering
University of California, Santa Barbara, U.S.A.
{aravind, forrest}@ece.ucsb.edu

Abstract — **This paper describes a practical technique for the optimal scheduling of control dominated systems minimizing the weighted average latency over all control branches. Such a weighted metric is crucial for control dependent scheduling to accommodate practical architectural goals. In contrast to most weighting mechanisms, a non-Bayesian probabilistic measure is used to avoid assumptions of branch independence. The underlying scheduling model allows general FSM-based models for operations, captures several forms of speculative execution and scales well with increasing control complexity.**

I. Introduction

Scheduling is a well studied problem with applications in a wide variety of fields. Of particular interest are applications in High Level Synthesis (HLS), where a behavioral specification is synthesized to a cycle-accurate one. Scheduling is a hard problem, being NP-complete even for resource-constrained, acyclic DFG's. Optimal scheduling techniques have been restricted to relatively simple performance metrics in which worst-case or total latency is optimized. For systems with multiple alternative behaviors or control paths, such metrics can dramatically fail to find sensible solutions. For example, consider a small application specific processor (ASIP) with forwarding, exception handling and queued I/O. A conventional scheduler attempts to minimize the worst case sequences – which might occur only at very low likelihood – to the exclusion of all other costs. A scheduler minimizing every control path separately may miss operation sharing opportunities and so increase the power consumption and design complexity. Instead, an intelligent performance optimization is desired, in which the performance of key instruction sequences (limited by available resources and constraints) are balanced to meet an appropriate weighted metric.

This paper deals specifically with control-dominated data-flow scheduling representing systems with weighted control paths. The data-flow model is acyclic, providing operand based fork and join controls that only affect data dependencies. Data-flow loop restructuring is not addressed, however, general finite-state models for operations are supported. In contrast to current branch probability models, which assume independent probabilities, a non-Bayesian weight model is used. Thus, potentially correlated weights are assigned to subsets of control alternatives. This is important in order to fit the metric to simulation and performance modeling results, which have consistently shown that branch probabilities are highly correlated for branches with temporal locality. Such a model has the issue that the set of weights must be treated as a list instead of a computation tree as in the independent case. Because of this requirement, the problem *specification* can grow exponentially fast with control complexity. However, in practice, the cost of adequate specification is bounded by the accuracy of the simulation analysis and so is controllable.

Given an acyclic control and data dependency specification, a bounded set of FSM-based operator models which map to the specified operations, and a disjoint set of control path weights (which are taken to represent probabilities), a new symbolic algorithm is proposed to generate all minimal weighted latency schedules of operations. Specifically permitted in the analysis are all forms of speculation past branches and branch re-ordering, as well as support for run-time or non-deterministic branches. The model does not allow implicit speculation past variable join points. This technique provides a compromise between implicit scheduling complexity and classical CDFG re-write techniques such as strength reduction, join speculation, and loop unwinding which are not addressed here. Moreover, it provides a bounded state model for scheduling which greatly reduces the complexity of optimal search.

The remainder of the paper is organized as follows: Section II summarises previous work on this and closely related problems. Section III provides a brief overview of the symbolic scheduling model. Section IV outlines the problem formulation. Section V elaborates the algorithm details, and provides a proof of optimal construction. Section VI provides experimental results, and the paper concludes with Section VII.

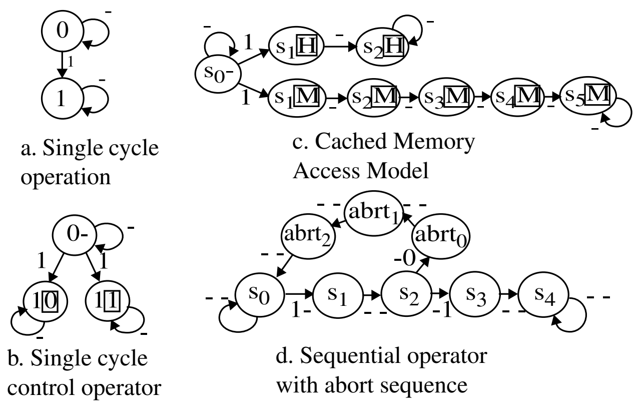
II. Previous Work

Boolean symbolic scheduling techniques supporting control were introduced in [2] and greatly improved in [3]. The first approach, [2], is somewhat akin to an integer linear program (ILP) formulation in that bits are assigned denoting completion of every operation at every time step. Unlike ILP, however, the boolean formulation allows for a relatively inexpensive exact model of control dependent behavior. The technique was inefficient for long schedule lengths, as complete histories were maintained. Ref. [3] presented an exact automata based approach which addressed the issue of lengthy schedules by encoding merely the completion status of operations rather than complete path histories. This was shown to work on a number of large problems, and further to support practical constraints on the control and sequencing of the components. One problem with this technique was that large numbers of possible schedules were equivalent using their metric, leading to poor run-time (and space) performance on under-constrained problems. Recently, Cabodi et al. [5] reformulated the problem as an instance of Bounded Model Checking (BMC) to generate single schedules. This technique gains performance at the cost of additional control dependency overhead. Manolache et al. [9] consider the different problem of scheduling tasks with stochastic delays, and derive efficient heuristics based on approximations of the distributions.

The AFAP or path-based scheduling scheme in [10] minimizes individual control path lengths and tries to construct a controller with minimal control states. The technique has strong restrictions on operators types, and further requires that every path be completely ordered prior to scheduling. In [11], the preceding method is extended to allow operation re-ordering, but only within basic blocks. Neither method supports speculative execution. Minimizing each path length is better than ignoring short paths. However, all control branches need to share the same resources. Given potential control restructuring and speculation, the number of operations and opportunities for function unit sharing are not fixed. Such greedy path length reduction will find fast solutions only at the cost of excess resources.

Bhattacharya et al. [7] introduced expected execution time as a metric, and proposed a heuristic scheduling technique targeted toward reducing it. Lakshminarayana et al. [8] proposed an algorithm with a control representation exposing more parallelism to minimize the expected execution. Both techniques are heuristic, and rely on modeling the branch weights as essentially independent (assuming Bayesian independence in the weights).

Dos Santos et al. [1] use average execution time to guide a code-motion based exploration. Though their method guarantees that at least one optimal solution is always present in the search space, there is no assurance that it will be found. Finally, Kountouris and Wolinski [6] describe a list-based scheduler that uses a priority function based on probability of execution. The control model used in [6] is similar to that of [2], [3] and this work in that no explicit fork nodes are placed, avoiding the ad-hoc clustering of operations into basic blocks.



a. Single cycle operation
b. Single cycle control operator
c. Cached Memory Access Model
d. Sequential operator with abort sequence

Figure 1. FSM operator models

III. Symbolic Scheduling

Automata based symbolic scheduling was introduced in [3]. This section briefly covers the terminology and basis of this technique. A detailed discussion of automata based scheduling techniques can be found in [4].

A. Terminology

The following terms are used in the rest of this work with the stated meanings:

- A **control operation** is one that produces one or more binary control values (**control bits**) which represent a forking of the execution into independent behaviours.
- A **control cube** is a Boolean cube consisting of multiple control bits, representing a set of branch decisions. A cube may be associated with a **weight**, representing the probability of that set of decisions.
- A control cube covering exactly one behavioral sequence of the system is termed a **control case**.
- A **trace** is a scheduled instance of a control case, consisting of a complete ordering of all operations required for the case.
- An **ensemble schedule** is a valid, causal collection of traces covering every control case.
- **Compatible traces** are traces for different control cases that may co-exist in a given ensemble schedule.
- A **resolve label** is a Boolean function associated with each control operation. It is True only in the cycle when the control **resolves** (The value becomes available to the system and controller by the end of that cycle).

B. Operator model

Operations are modeled as small NFA (non-deterministic finite automata) which encode the operation's temporal input and output behavior. Fig. 1 shows a few example operator models. Arcs are labeled by the input requirements, a 1 indicating that an operand must be available in that cycle for the transition to take place. A single-cycle operation that takes one input can be modeled as in Fig. 1a. Note that the non-determinism is used to model the unknown start time of the operation. After it starts, the model executes in a single cycle and ends in its final state. Typically, the operators are encoded with an all-zero start state, and the termination is modeled by a dense (close to all-1s) cube. This choice of encoding heuristically reduces the size of the representaton.

A single cycle *control operation* is shown in Fig. 1b. Execution *traces* are represented by unique sequences of states of the modeling automaton. Since distinct control cases can have distinct traces, every control bifurcation has a corresponding bifurcation in the modeling state space to represent alternative traces. These traces are distinguished by a bit (boxed) whose value indicates the branch selection (true or false path) in the simple single cycle operator model of Fig. 1a.

Much more complex operators may be created to model prescribed sequential behaviors. For example, Fig. 1c shows a non-deterministic 2/5 cycle latency cached memory access. To initiate the access, one operand is needed for the first transition (the address). Since the hit/miss value is known only at run-time, the model has to correctly account for both cases. Thus, this operator behaves exactly like a control operator producing a dynamically computed decision, and correspondingly provides bifurcating states for each case. Fig. 1d models an operation that initiates an abort sequence if it fails to receive inputs in a timely fashion – as the arc labeling indicates, it requires its second input operand to arrive on the third cycle after the first operand.

Such complex operator models are required to provide abstractions of subsystem behavior such as operations forced to execute on a reconfigurable pipeline which is a fixed requirement of the design. In general, an "operation" models the sequential production of a data object which is taken to be subsequently available in the system to satisfy data-flow dependency requirements.

C. CDFG model

The FSM inputs (the arc labels in Fig. 1) controlling the modeling automata are derived from specification control and data-flow dependency relations. Thus, the enabling input to an operation comes from the completion states of its ancestors via data-flow dependencies. Data availability is simply modeled by the completion

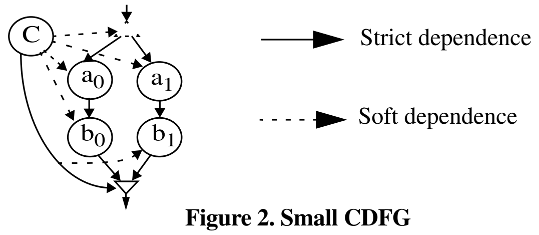


Figure 2. Small CDFG

of a data-available state of the modeling automata. Control dependence is enforced only on operations whose inputs are selected from alternative branches (joins) which are distinguished by one or more control bits in the modeling state. The branch (fork) in Fig. 2 is implicitly captured using soft dependencies on the operations. Soft dependencies only modify the resource constraint mechanism of the schedule: an operation may be started as soon as its input operands (hard dependences) are valid even if it is not known to be necessary. Since control operations are, themselves, subject to this rule, arbitrary speculation past branches and branch re-ordering are implicitly performed. The control marking of branching traces using soft dependences allows for optimal interpretation of mutual independence and resource sharing in the model. Note the strict dependence between the control operation and the join which does prevent speculation past the join boundary as mentioned in the introduction. Join dependence is enforced to prevent an explosion in the number of states since an operation with a different selected operand is effectively a different operation.

D. Scheduling and Validation

The scheduling problem is solved via an implicit reachable state traversal of the modeling automata. The traversal starts from a state where only initial operands are known and ends when states encoding pre-specified termination conditions are encountered. Initially, a series of forward image computations performing breadth-first search of the state space determines the quickest state sequence paths to termination states. Once terminating states are reached, backward pruning is performed to cull those states (and transitions) that do not lie on minimal time paths.

During this backward prune, a validation fixed point computation is performed to ensure that *every* trace is part of *some* ensemble schedule. To see why this is necessary, consider the small CDFG of Fig. 2. Suppose one resource is available to execute the control operation, and one for all the others. Every schedule for this flow consists of two traces, one for each of the possible control values. As scheduling proceeds, terminal states for both paths are reached in 2 time cycles, but there is no causal ensemble schedule of length 2. This is due to the fact that the early termination of each path relies on having speculated differently on that path. For example, to complete the 0 path in 2 cycles, the execution of operations a_0 and C must occur in the first cycle. Since the value of the control operation will not be available until the end of the cycle it executes in, operation a_0 is speculated. The trace that completes the 1 path must execute a_1 in the first cycle – which conflicts with path 0. At the time when the decision has to be made, the value of the control output is unknown. Although there is a trace of length 2 for each branch, the two traces are not compatible within bounded resources.

Succinctly, validation eliminates all states belonging to traces that belong to no valid ensemble schedule. Thus any surviving trace belongs to some ensemble schedule and could be part of a valid optimal solution. The validation process is constructed as a fixed point search on the model since enumerating the constraints is itself exponential. Typically, validation improves the run-time of the model by elimination non-causal states. It should be noted, however, that in contrast to conventional operation scheduling, in this generalized model, there is no guarantee of an ensemble schedule of *any* length.

IV. Problem Formulation

The technique in [3] minimizes worst case latency of the longest path and can heuristically reduce the latency of other paths subject to the worst case. In this work, the scheduling problem is formulated in terms of a weighted average latency metric. This algorithm ensures that **every** trace is part of **some** ensemble schedule of **minimal cost**, the cost being the sum of path weight times path latency. This metric forces changes to the exploration (via new pruning techniques), to validation and to termination, as well as implicit mechanization of the weighting process. This added complexity, however, enables a guided search whose optimality is directly linked with architectural need. In the remainder of the paper, the terms *weight* and *probability* shall be used interchangeably although trace weights are not constrained to behave as probabilities for the algorithm to be exact.

A. Assumptions and Justifications

The input weights are assumed to be obtained from simulation, are all positive, and label disjoint control cubes. All weights are normalized with respect to the smallest possible weight. It should be noted that this may not necessarily be the smallest listed weight - since the weights are from a finite sample of runs, it is possible that some low-frequency paths are not encountered in the simulation. Symbolic scheduling, however, explores *all* paths, and thus will hit such cases. A related issue is the possibility of dead-code or of functionally impossible branches. Since the actual correlated value of branches is not available during scheduling, the model will take branches allowed by the language specification, even if they cannot occur in execution. Usually, such cases account only for a small percentage of the possible paths. To account for these extra resolvers, a per-control probability factor is interpolated from the available weights and used to modulate the path weight to make a well formulated problem.

Path costs are calculated by multiplying the termination latency with the normalized path weight and rounding the product. Thus, the maximum possible *absolute* error due to round-off is 0.5 times the number of weighted paths, and is constant. On the other hand, if an error of ε is assumed in the probability measurement, the *relative* error is ε times the reciprocal of the minimum probability and is constant. (The absolute error in cost at time t for a probability p_k is given by:

$$\delta = t \times \frac{(p_k \pm \varepsilon)}{(p_m \pm \varepsilon)} - t \times \frac{p_k}{p_m} \approx t\frac{p_k}{p_m} \times \left(\pm\frac{\varepsilon}{p_m}\right)$$

ignoring quadratic and higher terms in ε and ignoring p_m with respect to p_k. The relative error is given by the term in the brackets). Since all the weights are positive, the cost is monotonic non-decreasing with increasing exploration cycle depth and the constant relative error dwarfs the constant absolute error within a few cycles.

An important point is that no assumptions of solution continuity are required, as would be the case with classical search techniques. This formulation and algorithm work optimally for arbitrarily sparse solution sets which may have no solutions for some control paths at given latencies. Finally, the restriction of controls to binary values is immaterial, as multi-way branching may be implemented by a binary encoded scheme, mimicking practical hardware implementation.

B. Motivating Example

To motivate the need for a weighted, non-Bayesian approach, consider the CDFG fragment of Fig. 3. Control operation C is as in Fig. 1b, operations a0, a1 (required, respectively, for the 0 and 1 cases of operation C) and addr are as shown in Fig. 1a, the read is as shown in Fig. 1c (flags a hit or miss in the first cycle of its execution)

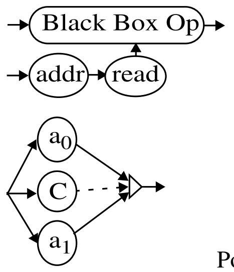


Table 1: Costs

Path	Weight	A	B	C
0, M	0.05	8	11	8
0, H	0.12	6	5	6
1, M	0.03	8	11	11
1, H	0.80	6	5	5
Tot.	1.00	6.16	5.48	5.45

Potential traces

	1	2	3	4	5	6	7	8	9	10	11
H, 6		read H	bbox			▨					
M, 8	addr	M			bbox			▨			
H, 5	C	read H			▨						
M, 11		bbox M						bbox			▨

Figure 3. Motivating Example

and the black-box operator (bbox in the traces) has the sequential behaviour of Fig. 1d.

Though the CDFG is divided into two independent (no data dependency) sections, the presence of a cached memory access means that hit/miss functional correlation is possible with the branches of control operation C. A set of probabilities is shown in Table 1, under the column Weight, for the 4 possible cases arising from C being 0 or 1 and the cache hitting (H) or missing (M). Note that speculative initiation of the black-box may lead to an abort if the cache misses (recall that the operator of Fig. 1d aborts and restarts if the second operand does not arrive on time).

Four of the possible schedule traces for this CDFG are shown in Fig. 3., showing, respectively, sequences for a read hit taking 6 cycles in total, a miss taking 8 cycles, a hit with speculative start of the black box leading to a 5 cycle completion time, and finally, a miss leading to an abort and restart of the black box operation. The boxes are labeled with the operations that are initiated in that time step. The hatched time-steps indicate that a particular trace terminates in that cycle. Each trace shown is compatible with both C=0 and C=1, and the two operations a_0 and a_1 may be scheduled freely in all cases.

Table 1 shows the schedule costs for 3 different ensemble schedules. Schedule A is produced if the optimization metric is minimum worst-case latency or equivalently, minimum sum-of-path lengths. This solution is composed of compatible traces of type H,6 and M,8. As it happens, this schedule never aborts, but in terms of average performance, this is way off target.

If the control operation and the cache hit/miss probabilities are assumed independent and the average latency is optimized, schedule B is obtained. This solution is composed of traces of type H,5 and M,11. This is still not optimal in terms of performance – due to the assumption of branch independence, the scheduler effectively sees a lumped hit-miss distribution of 0.92-0.08 and proceeds accordingly.

It is worth observing, at this point, that the traces H,5 and M,8 are not compatible – to complete the sequence for a hit in 5 cycles, the black box operator must be speculatively started. It will then not line up appropriately with a read miss case and will have to abort and restart. Thus, H,5 may only pair with M,11 and M,8 only with H,6. This means that not only are the branch weights correlated, so are their latencies.

With the non-Bayesian metric, the scheduler is able to speculate only on the correct path, and arrives at schedule C, taking into account all possible side-effects, picking, one representative from each of the traces shown to make the ensemble. The error made in assuming independence (and thus the error in scheduling) grows rapidly with the number of control branches. In this simple example, no resource or dependency constraints span the correlated branches to keep the complexity of the example manageable for discussion.

```
Input: NFA representation of CDFG
1: while not (optimal schedule):
2:     while not (current schedule is valid):
3:         while not (current schedule is complete):
4:             Add time step in forward exploration
5:             Record and weight terminal states
6:         Perform Weighted Validation
7:     if new minimum cost:
8:         Record cost and schedules
```

Figure 4. Simplified Weighted Algorithm

V. Weighted Algorithm

A broad outline of the scheduling technique is shown in Fig. 4. Forward image computation (Section III-D) proceeds via lines 3-5. In line 5, states that mark termination are tagged with a BDD representing the product of the current latency and the weight of the corresponding trace. This is done by conjoining the BDD of the state set with a BDD representing the cost as a binary-encoded integer. Using a common BDD representation for the traces and weights simplifies correspondence and allows implicit computation during the exploration. The default set of BDD variable used to encode the tag are denoted as the x variables. Extra resolvers (Section IV-A), if any, are accounted for here. Line 6, which does the bulk of the work, is shown in greater detail in Fig. 5 and discussed in Section V-A.

A. Weighted Validation

The validation algorithm performs a double fixed point, an outer loop over all time (lines 2-20) and an inner loop (lines 5-17) over all control operations. The inner loop takes a copy of the transitions for each reverse step and loops over all controls until there is no change in the state sets (condition on line 17). Validation removes traces that have no compatible traces, i.e. are not part of any optimal

```
Inputs: State set $S_j$, target T, cost bound BDD B
1: PS = T
2: do:
3:     for each time step j to 0:
4:         Create transition $T_j$ for reverse step $S_j \wedge PS \rightarrow S_{j-1}$
5:         do:
6:             $T_{copy} = T_j$
7:             for each control operation:
8:                 $cvar$ = BDD var. for control val.
9:                 $res$ = BDD func. for resolve label
10:                $T_{jr} = T_j \wedge res$
11:                $T_{jrc0} = T_{jr}$.Cofactor($\neg cvar$)
12:                $T_{jrc1} = T_{jr}$.Cofactor($cvar$)
13:                $T_{jrc1} = \exists x (T_{jrc1} \wedge XtoY)$
14:                $T_{jr} = \exists z (\exists x, y ((T_{jrc0} \wedge T_{jrc1}) \wedge XplYeqZ) \wedge ZtoX) \wedge B$
15:                $T_j = (T_j \wedge \neg res) \vee T_{jr}$
16:                if $Tj = \phi$, exit
17:        while $T_j \neq T_{copy}$
18:        Prune $S_j$ and $S_{j-1}$ using pruned $T_j$
19:        PS = Pruned $S_{j-1}$
20: while some change in states
21: Determine minimum, remove sub-optimals
```

Figure 5. Weighted Validation

ensemble schedule. If a state is present only on traces that are removed in one iteration of the validation loop, that state is removed from the state sets. Since it is possible that it may be the only state that allows some other traces to be compatible, this process has to be taken to a fixed point in the state sets. Bounds on the number of iterations of the validation loops required to reach this fixed point were presented in [2] and [4].

At first glance, the problem of calculating the cost of an ensemble schedule may appear trivial – since adding up the individual trace latencies and multiplying by their weights suffices. Remember, however, that the state traversal effectively forms a breadth-first search, exploring all possible traces in parallel. Thus, the cost of *all* possible ensemble schedules needs to be calculated. It should be clear that enumerating and computing these sums individually is infeasible, as there are, in general, a factorial number of traces and the number of ensembles is exponential in the number of traces. Instead, the computation is performed in parallel symbolically exploiting the logarithmic compression possible in a BDD representation. The cost of this is the size of the BDDs that are built.

The BDD tagging a given state set is somewhat difficult to characterize. In general, the tag consists not of a single number, but a set of numbers. This is because though each **trace** has a unique weight and cost associated with it, each **state** may lie on traces for multiple control cases and may also lie on traces of differing latency for a given control case. The set of numbers tagging the state represent the possible conditional costs of completion given that the automaton is in the current state. Put another way, each state is labelled with the set of sums of costs of compatible traces that pass through it. This is clearly true for the termination states, and this is an invariant on the BDD tags that needs to be preserved. Since the start state is present in every trace, the set of tags associated with the start state are exactly the costs of all possible ensemble schedules.

The ‘resolve’ labeled transitions are extracted (line 10) – these are transitions on which control values become known. As was seen in Section III-B, every control operation has a modeling bit that bifurcates the states on the two paths. This models the freedom to pursue different objectives depending of the result of the branch.

First, the transition is split into the ‘true’ and ‘false’ cases on the control modeling bit (lines 11,12). Next, the tag on one of the cases (the ‘True’ case is shifted from the x variable set to an auxiliary set of bits, the y variables (line 13). Line 14 performs the actual magic – walking from the inside out of the braces, the innermost conjunction annihilates all but the matching cofactors. The next conjunction with the BDD *XplYeqZ* implicitly sums up the x and y variables and places the sum on the z variables. This summation occurs in parallel – a pair of matching cofactors tagged with their respective cost BDDs as mentioned above have every pairwise combination of sums added up in parallel. This generates the correct tag for the parent state, since any schedule leading out from one of the cofactors may be paired with any corresponding schedule leading from the other cofactor and thus, every tag on one cofactor could possibly combine with every tag of the other cofactor.

Finally, the cost is moved back to the original cost bits, the x variables, and pruned for the current bound, B. Overall, three sets of BDD variables are used, viz. x, y and z. The BDD functions *XtoY* and *ZtoX* copy their source variables to the destination variables and *XplYeqZ* places the sum of the x and y variables on the z variables.

Line 15 reassembles the resolving and non-resolving portions of the control. On each backward step (line 18), the tag on each state is propagated backward. Finally, before exiting the procedure, all sub-optimal solutions are removed (line 21).

B. Proof of Optimality

A proof that the procedure outlined above results in optimal schedules is given below. Two preliminary lemmas are proved first, as follows.

Lemma 1: Given an initial, potentially optimal solution, the forward exploration of the modeling automaton needs to be extended only for a finite number (possibly zero) of time steps to obtain a guaranteed optimal solution.

Proof: Let the *n* weighted paths be numbered: *0..n-1*. Associated with each control path *k* is a weight w_k. Further, every path has some earliest cycle of completion, denoted by the integer d_k. Suppose that at an exploration depth of *t* cycles, a solution with cost *C* is found. The only solutions involving traces that terminate *later* than step *t* with lower cost must perforce be paired with traces that finish *earlier* than step *t*, since the cost is monotonic. In the limit, the most optimistic case for a given path *i* occurs when a solution in the future is able to match with all the other traces of length d_k and provide an ensemble with cost less than *C*. Thus, the upper bound on required exploration cycle depth for any path *i* is given by:

$$T_{max_i} = \left\lfloor \frac{C - \sum_{k \neq i} w_k d_k}{w_i} \right\rfloor$$

These path-wise limits are all finite since each w_k is non-zero. Among all the paths, there is some overall maximum value of these path-wise limits, denoted T_{max}. Once exploration has been extended beyond $t{=}T_{max}$, optimality is guaranteed by monotonicity of the cost. This proves the lemma.

It is important to note that reaching all states is **not** a sufficient condition to stop exploration – in fact, the motivating example of Fig. 3 exhibits the optimal solution only after this point is reached. The issue is that although all states have been found, it is still possible to extend a current incomplete low weight path which enables a complete lower cost ensemble. A worthwhile observation is that each time a cheaper solution is found, the current upper bound on exploration depth required, T_{max}, correspondingly decreases.

Lemma 2: The weighted validation algorithm (Fig. 5) computes the solution costs correctly.

Proof: To see this, the termination states for each path are, by construction, weighted with the appropriate cost. When states do not resolve, the back propagation acts to move the costs back to their predecessors on the same path. (i.e. the addition relation works both from addends to sum and in reverse.) Thus, at the point when a control resolves, states that match on the two resolving paths arrive carrying the correct cost of completion. Since the validation cofactoring and conjunction ensures that every state is paired exactly with its control twin, the addition process takes place exactly between the correct resolving branches. This is now propagated backward. Thus, costs are computed correctly when all resolution points are processed and the start state reached.

To prove optimality, it can be seen that applying the weighted validation algorithm once without any extraneous bound will generate the normal solutions, with the correct minimum total cost available, by Lemma 2. Now, given a potentially optimal solution with a known exploration bound, forward exploration is continued till the required maximum exploration depth of Lemma 1 occurs, and overall optimality is assured.

VI. Experimental Results

The algorithms of Section V were implemented using the Colorado University BDD package, CUDD. All experiments were conducted on a 3GHz/2GB Pentium 4 Linux desktop. The memory model for CUDD was set at 1.5GB.

Table 2: Benchmark Details

Benchmark	Operations	Controls	Longest Path
ADPCM-dec	40	10	11
ADPCM-enc	45	11	15
Rotor	28	3	7
Kim-54	54	5	9
Kim-26	26	2	5

Table 2 describes the benchmarks used. The two ADPCM examples are the loop bodies of the encoder and decoder from Medibench [12]. Rotor [3] performs the rotation of a point by an angle, which may lie in any of the 4 quadrants. Kim-26 and Kim-54 are from [13]. For all the benchmarks, + does add, - does subtract, ALU does add, subtract and bit-logic, << is a shifter, == is a comparator, [] an array address decoder and * a multiplier. The multiplier is 2 cycle pipelined, unless mentioned otherwise. All other operations take one cycle. The control value generated by a comparator is **not** assumed to be available in the same cycle the control operation executes - i.e., mutual exclusive sharing is not possible in the same cycle that a control operation is executed, but is possible in the next cycle. Some early papers ignored the causal connection between control activity and control operation execution.

The various algorithms being compared against are labeled Wave (Wavesched, [8]), KW (Kountouris, [6]), DS (Dos Santos, [1]) and SPARK (Gupta, [14]). The optimal algorithm described in this paper is labeled Opt.

A. ROTOR

This example was tested with 3 different weight settings, listed in Table 3.

Table 3: Rotor Quadrant Probabilities

Set	Q1	Q2	Q3	Q4
W0	0.25	0.25	0.25	0.25
W1	0.97	0.01	0.01	0.01
W2	0.6	0.15	0.15	0.1

Table 4: Rotor Latency

Res.	No wt	W0	W1	W2
R1:1[],1ALU,1==,1*	11	10.25(11)	9.05(12)	9.6(12)
R2:1[],2ALU,1==,2*	8	7.75(8)	7.03(8)	7.35(9)
R3:1[],2ALU,2==,1*	9	9(9)	8.04(10)	8.5(10)

As can be seen from Table 4, under any non-trivial weight set, only in one case (R2, W1) does the unweighted solution set contain an optimal solution. The column format is *optimal average latency(longest path)*. For the unweighted case, it is merely the longest path. It is instructive to note that column W0, with equal weights, finds no new solutions - an indication that minimizing the *sum* of all paths may not be effective. All the runs completed in between 0.6s and 2.6s.

B. Kim Benchmarks

These 2 benchmarks are synthetic CDFGs. For the purpose of comparison, the multiply is single cycle. All branches are equi-probable, again, to enable comparison.The results are shown in Table 5.

Table 5: Kim benchmark comparison

CDFG	Resources	Wave[8]	KW[6]	DS[1]	Opt.
Kim-26	2+, 2==, 1-	-	6/6/6	-	5/6/5.5
Kim-26	2+, 1==, 1-	-	6/6/6	-/6/5.75	5/6/ 5.75
Kim-54	2+,2-,2*,1==	10/14/12.6	-	-	7/9/ 7.875

The columns indicate *shortest path/longest path/average latency.* The last column, Opt., shows the results of this algorithm. Blanks indicate that either the example was not used or that the particular datum was not reported. The Kim-26 runs were both under 0.6s, while the Kim-54 example took 2s. The improved results are a combination of the weighted algorithm and the control model. Though its HCDG is not restricted by basic blocks, [6] is unable to find the solution with average latency 5.5.

C. ADPCM benchmarks

The ADPCM coder and encoder both consist of short preambles, followed by a single for loop, and a short post-amble. To compare the encoder with [14], the 3 sections were scheduled separately and the loop latency multiplied by the 10 loop iterations. For the decoder, the values given are for one iteration of the loop.

Table 6: ADPCM comparison

Bench	Resources	Spark	Opt
Dec.	R1: 2 ALU, 1==, 1[], 1<<	-	11.5(13)
Dec.	R2: 1 ALU, 2==, 2[], 1<<	-	12.25(14)
Enc.	R2: 1 ALU, 2==, 2[], 1<<	192	157.75(164)

Table 6 shows the results for this set of experiments. For the optimal algorithm, the column lists *optimum average latency(longest path)*. In the case of the encoder, the longest path is assumed to take the worst case through the loop body each time. Execution details are provided in Table 7. The columns labeled “No wt.” are the statistics for unweighted scheduling. Due to the large number of control cases, the time taken for complete scheduling is dominated by the time to extract a witness from all the optimal ensembles. To give a better idea of the run-time for optimal scheduling, the numbers in brackets indicate scheduling without witness extraction. Peak BDD size is reached during the scheduling process, and does not grow further.

Table 7: ADPCM execution statistics

Bench	Res.	Time(s)		BDD (K nodes)	
		No wt	Opt	No wt.	Opt.
Dec.	R1	5.34(1.42)	5.86(3.04)	41	298
Dec.	R2	8.56(4.3)	13.33(9.18)	71	799
Enc,	R2	9.65(2.3)	12.2(5.09)	144	440

D. Summary

The results show that with modest run times, guaranteed optimal solutions may be found. It was not possible to directly compare results with the methods of [7] and [8] since those techniques include extensive loop optimizations, while the current optimal algorithm operates on bounded-state forward branching CDFGs. It was possible to compare against the work in [14] since the loops in the examples were almost completely on the outside of a data-flow kernel. The results indicate that substantial improvements in expected execution time are possible, even in small examples.

VII. Conclusions, Future Work

In this work, a solution to the problem of optimal scheduling under a weighted average latency metric is described. The solutions are optimal within the uncertainty imposed by errors in path probability measurements. The model for weights can support direct simulation measurement since no assumption of independence is made. The only requirement imposed is that the weights be positive and apply to disjoint control cases. The technique is demonstrated on a variety of benchmark problems with good performance and reasonable time complexity. The authors are unaware of any pre-existing work that generates weighted optimal schedules for even small CDFGs. Future work includes heuristic utilization of the weight information for early pruning and complexity control, and generalizing the method to enable scheduling of loops (such as, for e.g., [10]).

References

[1] L. C. V. dos Santos, M. J. M. Heijligers, C. A. J. van Eijk, J. T. J. van Eijndhoven and J. A G. Jess, "A Code-Motion Pruning Technique for Global Scheduling", *ACM Trans. on Design Automation of Electronic Systems*, 5(3):1-33, Jan 2000.

[2] I. Radivojevic and F. Brewer, "A New Symbolic Technique for Control-Dependent Scheduling", *IEEE Trans. on CAD/ICAS*, 15(1):45-57, Jan 1996.

[3] S. Haynal and F. Brewer, "Efficient Encoding for Exact Symbolic Automata-Based Scheduling", in *Proc. of the 1998 IEEE/ACM Intl. Conf. on CAD*, 1998, pp477-481.

[4] S. Haynal. Automata-Based Symbolic Scheduling. Ph.D. thesis, University of California, Santa Barbara, 2000. http://citeseer.ist.psu.edu/haynal00automatabased.html

[5] G. Cabodi, S. Nocco, S. Quer, L. Lavagno, A. Kondratiev and Y. Watanabe, "A BMC Formulation for the Scheduling Problem in Highly Constrained Hardware Systems", *Electronic Notes in Theoretical Computer Science*, 89(4), 2003.

[6] A. A. Kountouris and C. Wolinski, "Efficient Scheduling of Conditional Behaviours for High-Level Synthesis", *ACM Trans. on Design Automation of Electronic Systems*, 7(3):380-412, July 2002.

[7] S. Bhattacharya, S. Dey and F. Brglez, "Performance Analysis and Optimization of Schedules for Conditional and Loop-Intensive Specifications", in *Proc. of the 31st Design Automation Conference*, 1994, pp491-496.

[8] G. Lakshminarayana, K. S. Khouri and N. K. Jha, "*Wavesched*: A Novel Scheduling Technique for Control-Flow Intensive Designs", *IEEE Trans. on CAD/ICAS*, 18(5):505-523, May 1999.

[9] S. Manolache, P. Eles and Z. Peng, "Schedulability Analysis of Multiprocessor Real-Time Applications with Stochastic Task Execution Times", in *Proc. of the 20th IEEE/ACM Intl. Conf. on CAD*, 2002, pp699-706.

[10] R. Camposano, "Path-Based Scheduling for Synthesis", *IEEE Trans. on CAD*, 10(1):85-93, Jan 1991.

[11] R. A. Bergamaschi, S. Raje, I. Nair and L. Trevillyan, "Control-Flow Versus Data-Flow Based Scheduling: Combining Both Approaches in an Adaptive Scheduling System", *IEEE Trans. on VLSI*, 5(1):82-100, Mar 1997.

[12] C. Lee, M. Potkonjak and W. H. Mangione-Smith. "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems", in *International Symposium on Microarchitecture*, pp330-335, 1997.

[13] T. Kim, N. Yonezawa, J. W. S. Liu, and C. L. Liu, "A scheduling algorithm for conditional resource sharing - a hierarchical reduction approach", *IEEE Trans. on CAD/ICAS*, 13(4):425-438, Apr 1994.

[14] S. Gupta, N. Savoiu, S. Kim, N. Dutt, R. Gupta, and A. Nicolau, "Speculation Techniques for High Level Synthesis of Control Intensive Designs", in *Proc. of the 38th Design Automation Conference,* 2001, pp269-272