On Applicability of Symbolic Techniques to Larger Scheduling Problems *

Ivan Radivojević, Forrest Brewer

Department of Electrical and Computer Engineering
University of California, Santa Barbara, U.S.A.

Abstract

It has been generally assumed that recently introduced symbolic techniques are applicable only to small scheduling problems. This report demonstrates that applicability of these techniques can be extended to larger dataflow graphs by: (i) using Zero-Suppressed BDDs, (ii) applying a set of interior constraints that reduce the size of intermediate solutions, (iii) implicit application of complex constraints, and (iv) formulation of set-based heuristics that preserve whole sets of partial solutions exhibiting desirable properties. Both heuristic and exact methods are discussed using standard benchmarks and are compared to the previously published work.

1. Introduction

Operation scheduling is the process of assigning operations to time slots in a synchronous system, subject to data/control-flow dependencies and resource constraints. Practical methods proposed for solving this problem can be divided into three basic categories: *(i)* priority-based heuristics, *(ii)* ILP-based optimizations, and *(iii)* symbolic techniques. Heuristics [4][6][15] are applicable to large schedules but may fail to find an optimal solution in tightly constrained problems. This is primarily because the heuristics cannot recuperate from early suboptimal decisions which typically preserve only one representative from a possibly very large pool of candidates. Applicability of exact ILP methods [9] has been improved by pre-processing and remapping of the constraints [7][8], but the number of variables in the formulation is still a dominant limitation. Heuristic approaches based on ILP [10][11] reduce the number of variables significantly, but suffer from similar deficiencies as general heuristic schedulers.

Symbolic techniques describe scheduling constraints as Boolean functions and build an *OBDD* (Ordered Binary Decision Diagram [2]) encapsulating *all* feasible solutions to a particular problem instance. Such methods provide support for optimal resource-constrained scheduling of non-branching and forward-branching (without speculative operation execution) control/data flow graphs (CDFGs) [16][17], exact resource-constrained scheduling of forward-branching CDFGs with speculative execution [18], and scheduling with environmental constraints [20][1]. The main challenge for symbolic techniques can be summarized by Bryant's observation from [3]: "... In many combinatorial optimization problems, symbolic methods using OBDDs have not performed as well as more traditional methods. In these problems we are typically interested in finding only one solution that satisfies some optimality criterion. Most approaches using OBDDs, on the other hand, derive all possible solutions and then select the best from among these. Unfortunately, many problems have too many solutions to encode symbolically...". It has been shown that some standard benchmark instances have billions of optimal solutions [17]. In such cases, the OBDD representation can become too large to be practical since both its size and CPU runtime increase significantly.

To improve the applicability of symbolic techniques for exact resource-constrained scheduling of larger data flow graphs (DFGs), several novel approaches are proposed in this paper. *(i)* To improve compression of large solution sets, *ZBDDs* (Zero-Suppressed BDDs [14]) are used. *(ii)* To prevent partial solutions from becoming prohibitively large during the construction process a set of auxiliary constraints (*interior constraints*) is used. *(iii)* Complex constraints are applied *implicitly* (i.e. without explicitly building a constraint BDD).

For large problems consisting of thousands of representation variables, we also describe *set-based heuristics* that preserve whole sets of partial solutions exhibiting desirable properties. In this paper, we assume the formulation introduced in [16] and extended in [17][18].

* This work has been supported in part by fellowship donation from Mentor Graphics Corp.

1. A formal approach based on algebra of control-flow expressions is presented in [5].

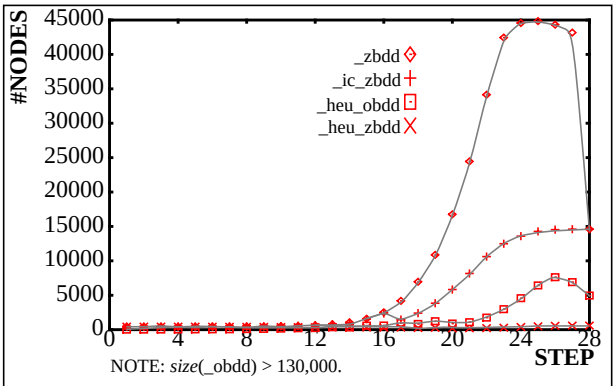

Figure 1. 28-cycle EWF: exact and heuristic constructions

- *resources*: 1 single-cycle adder, 1 two-cycle multiplier (> 10e+9 solutions)
- *#variables*: 437

_zbdd: exact solution (ZBDD), no interior constraints: ~ 20.5 min
_ic_zbdd: exact solution (ZBDD) built using interior constraints: ~ 12.5 min [1]
_heu_obdd: utility-based set-heuristic solution (OBDD): ~ 22 s [2]
_heu_zbdd: utility-based set-heuristic solution (ZBDD): ~ 110 s
_obdd: exact solution (OBDD): > 1.5 h

1. For optimal number of registers (10), the size of exact ZBDD solution decreases from ~14.5 to ~3.5 Knodes.
2. ~ 5 s if both utilization and critical path are used as heuristic criteria

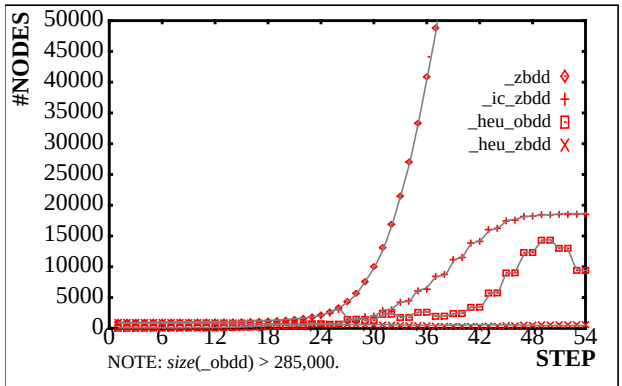

Figure 2. 54-cycle EWF: exact and heuristic constructions

- *resources*: 1 two-cycle adder, 1 two-cycle multiplier (> 10e+13 solutions)
- *#variables*: 967

_zbdd: exact solution (ZBDD), no interior constraints: could not be constructed
_ic_zbdd: exact solution (ZBDD) built using interior constraints: > 1.5 h [1]
_heu_obdd: utility-based set-heuristic solution (OBDD): ~ 73 s [2]
_heu_zbdd: utility-based set-heuristic solution (ZBDD): ~ 12.5 min
_obdd: exact solution (OBDD), not constructed (converted from *_ic_zbdd*)

1. For optimal number of registers (10), the size of exact ZBDD solution decreases from ~18.5 to ~6.5 Knodes.
2. ~18 s if both utilization and critical path are used as heuristic criteria.

2. ZBDD Representation and Manipulations

The advantage of using ZBDDs [14] in exact symbolic schedulers is effectively illustrated by the example of the *Elliptic Wave Filter* (EWF) benchmark. The 28-cycle EWF has only 34 operations but 437 variables are used to fully describe the problem. Every solution (a path to 'true' node in OBDD representation) consists of 437 variables out of which only 34 are '1'. A vast majority of variables are equal to '0' (i.e. a particular operation is not scheduled at a particular time step), but are explicitly represented in the OBDD. Consequently, the representation has more than 130,000 OBDD nodes. Since the 0-variables that belong to the solution are implicit (suppressed) in ZBDDs, much larger compression of the solution set is possible. In fact, our experiments show that for a 28-cycle EWF a nearly ten-fold reduction in size is achieved (Fig. 1). A 15-fold reduction is observed when a 2-cycle adder is used (967 variables, 54-cycle optimal solutions, Fig. 2, [19]). This marked compression of the representation size allows analysis of much larger problems than is possible using OBDDs. For example, the benchmark instances discussed above run in 20 Mbytes of RAM incurring only 2-3% percent CPU time penalty due to garbage collection overhead. In fact, for some examples discussed in Section 6, the number of nodes in the optimal solution sets is occasionally smaller than the number of variables describing the problem.

We observed that, when applied to the scheduling problem, ZBDD manipulations are somewhat slower than similar OBDD manipulations. This seems to be caused by the frequently simpler OBDD form of constraints. The constraint may involve only a few formulation variables and the corresponding OBDD representation is typically small. Such constraint may become more complex when converted to ZBDD because *don't-care* variables are not implicit in the ZBDD representation. Our experience, however, is that very slight modification in the construction strategy can dramatically improve the execution time (discussed in Section 4). Moreover, further performance optimizations are likely since we use a recently developed custom C++ ZBDD package (e.g. currently there is no support for 'inverted-edges' strategy [1][14] that enables faster manipulations and reduces BDD size).

3. Interior Constraints

In the current implementation, the solution is built iteratively and a termination test is performed after all of the constraints relevant to a particular time step are applied. Although the OBDD size of the final solution is typically very moderate, the intermediate solutions can become prohibitively large, resulting in a slower construction and larger memory requirements. Ideally, the intermediate size should never exceed the size of the final solution. In such a scenario, as long as the final solution fits the memory limits of the runtime environment, we should be able to complete a scheduling task.

To alleviate the problems arising from the uncontrolled growth of the intermediate solution, we identify and discard a set of partial schedules that 'hopelessly lag behind' during the construction process and cannot contribute to the set of optimal solutions. This means that at a

particular time step such partial schedules cannot terminate for given resources and a pre-specified upper bound on execution time. This consideration leads to a set of *interior constraints* which is dynamically generated during the scheduling in order to prune the OBDD/ZBDD.

The main strategy is illustrated by the following example: Assume that at the beginning of step *s* there are *n* addition operations that have ALAP (as-late-as-possible) bounds in the range [*s*... (*s*+*k*-1)] and that there are only *m* single-cycle adders available. At least (*n* - *km*) of these addition operations must be completed prior to step *s* in a feasible solution. Selection of a subset satisfying this property is done efficiently using the constraint template shown in Fig. 3 (further discussed in Section 4). Such constraints can be derived for each functional unit type (including multicycle and pipelined units). Interior constraints with *lookahead k* enable an early detection of many (not necessarily all) partial schedules that are destined to be discarded within the next *k* steps. Since the completeness of the solution set is preserved, this does not impact optimality.

Application of interior constraints has its trade-offs: more constraints are generated and applied, but the size of the intermediate solution is kept under better control. For larger problems in particular, interior constraints are very cost-effective. Even if the solution can be built without interior constraint application, this construction requires more CPU time and memory requirements are drastically higher. In the examples shown in Fig. 1 and Fig. 2, the lookahead was set to a value equal to the difference between the upper bound on execution time and critical path latency. The figures indicate controlled growth of the solution in which the intermediate size is never greater than the final size (curves labeled *_ic_zbdd*). Although such ideal behavior is not always achievable, our experiments indicate that the use of interior constraints has a dramatic effect on scheduler efficiency. Without interior constraints, the schedule in Fig. 2 (labeled *_zbdd*) failed to terminate in several CPU hours.

4. Implicit Application of Constraints

In this section we discuss more efficient implementation of two constraints: *generalized resource bound* and *uniqueness* [16][17].

<u>*Generalized resource bound:*</u> The constraint OBDD shown in Fig. 3 is frequently used as a *construction template* in symbolic scheduling. Some applications include: *(i)* selection of solutions that satisfy a particular resource constraint at a particular time step [16], *(ii)* interior constraints described in Section 3, *(iii)* scheduling heuristics (to be discussed in Section 5), and *(iv)* post-processing (after the scheduling is completed, the bounds can be iteratively tightened/identified). The constraint has $O[\binom{n}{k}]$ complexity (in terms of a number of product terms), but the number of nodes is *O(kn)*. The template can be built efficiently using *ite* [1] calls directly. Vertices in this if-then-else template are not restricted to Boolean variables -- complex Boolean functions (f_1, f_2,... f_n) can be inserted into the template (e.g. bus/register constraints, formulated in [16]).

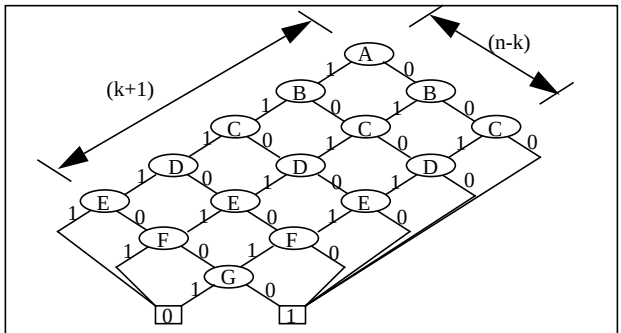

Figure 3. *At-most-k-out-of-n* constraint (k=4, n=7)

However, even when (f_1, f_2, ... f_n) are rather simple, the overall constraint may become extremely large. Consequently, it can happen that the partial scheduling solution is of a very moderate size, but the constraint to be applied cannot be built. However, the scheduling constraint need not be explicitly built. The following can be done instead:

(i) Introduce a new set of auxiliary variables (y_1, y_2, ... y_n) corresponding to the set of functions (f_1, f_2, ... f_n).

(ii) build the template function *T* (shown in Fig. 3) using only (y_1, y_2, ... y_n).

(iii) compute $P^0 = And(P',T)$, where *P'* is a partial solution to which the constraint is applied.

(iv) clearly, a new partial solution *P"* can be obtained using the recursive formula:

$$P^i = \exists y_i \left[And(P^{(i-1)}, Xnor(y_i, f_i)) \right]$$

where $\exists_x f = f_x + f_{\bar{x}}$ **.** This amounts to the standard BDD substitution operation:

$$P^{(i)} = P^{(i-1)}\big|_{y_i \equiv f_i} \qquad (1)$$

Using this approach, in all of the benchmarks discussed in Section 6, we were able to apply register constraints that could not be built explicitly.

<u>*Uniqueness:*</u> This constraint enforces that each operation *j* is scheduled once and only once in all feasible solutions. C_{sj} denotes operation *j*'s instance at time step *s*. If $(ASAP)_j \le s < (ALAP)_j$:

$$\left(\sum_{k \in R_{sj}} C_{kj} \prod_{(i \neq k) \in R_{sj}} \overline{C_{ij}} \right) + \left(\prod_{i \in R_{sj}} \overline{C_{ij}} \right) = 1 \qquad (2)$$

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

$$\left(\sum_{k \in R_{sj}} C_{kj} \prod_{(i \neq k) \in R_{sj}} \overline{C_{ij}} \right) = 1 \qquad (3)$$

This formulation enforces uniqueness explicitly at

every iterative construction step. However, uniqueness can be maintained implicitly (by construction) using the much simpler form of Eq.2:

$$\overline{C_{sj}} + \left(\prod_{i \in R_{(s-1)j}} \overline{C_{ij}} \right) = 1 \qquad (4)$$

where $R_{(s\text{-}1)j}$ = [$(ASAP)_j$, $(s\text{-}1)$]. In some of the largest experiments discussed in Section 6, this simplification reduced CPU time by 35%.

5. Set-Heuristics

Since valid partial schedules are available after each time step, it is possible to devise heuristic scheduling techniques.The simplest *utility-based* heuristic [17] propagates only the subset of schedules with maximum utilization of resources. Utilization is measured by the number of operations active in each time step. The utility-based heuristic is implemented by iterative application of the generalized resource bound. We enforce maximum utilization of functional units, and then iteratively relax this constraint until satisfying partial solutions are found. Since *all* such schedules are propagated, this simple heuristic has good behavior. An additional (second-level) pruning strategy (*utility+CP*) based on the AFAP (as-fast-as-possible) scheduling of the operations belonging to the critical path(s) is possible as well. Essentially, the scheduler favors the partial solutions where the largest number of operations belonging to the critical path(s) have been scheduled. This strategy is effective when the number of operations that can be scheduled simultaneously is very small or when the schedule is expected to take very large number of steps (some of the problems in Section 6 execute in more than 100 cycles).

An accurate estimate of the upper bound on scheduling latency may not be available before scheduling. Unfortunately, the search space increases enormously fast with relaxation of this bound. Set-heuristic scheduling is very robust: the construction pace shows very weak sensitivity to the upper bound used to initialize the scheduler. Although additional constraints are generated (due to an increase in ALAP-ASAP spans for individual operations), the intermediate solution size increases very mildly (Table 1). Furthermore, it is very important that the heuristics be robust, since they can be used to derive accurate bounds for the exact schedulers whose runtime efficiency is more sensitive to the bound estimates.

Table 1: Robustness analysis of the heuristic scheduler

cycles	upper bound	# vars	*utility-based*		*utility + CP*	
			max # nodes	CPU [s]	max # nodes	CPU [s]
54	54	967	14,328	73	2,210	18
	55	1,001	14,839	75	2,292	19
	56	1,035	15,559	81	2,392	20
	59	1,137	17,151	94	2,616	21
	64	1,307	19,378	119	2,914	25
2-cycle adder, 2-cycle multiplier						

Fig. 1 and Fig. 2 indicate that utility-based set-heuristics (curves labeled *_heu_obdd* and *_heu_zbdd*) are far superior to exact schedulers both in terms of CPU time and memory requirements, while still finding representative minimum-latency schedules.

It is to be expected that heuristics based solely on utilization of the functional unit resources will occasionally produce sub-optimal results in terms of register requirements. For example, if the 28-cycle EWF (Fig. 1) is scheduled heuristically with no pre-specified register bound, the solution requires at least 13 registers. However, if a register bound of 10 is enforced during the construction, the utility-based heuristic still produces the fastest possible solutions (28 cycles). The same behavior was observed in the 54-cycle case (Fig. 2) and experiments described in Section 6.

6. Experimental Results

In this section, three examples are used to demonstrate the concepts discussed in this report: *(i)* ***EWF-2*** (*EWF* unfolded two times, 68 operations, Table 2), *(ii)* ***EWF-3*** (*EWF* unfolded three times, 102 operations, Table 3), and *(iii)* ***FDCT*** (*Fast Discrete Cosine Transform* [13], 42 operations, Table 4). The exact scheduler was run using ZBDD representation and interior constraints. Currently, all of the individual constraints are generated as OBDDs and then converted to ZBDDs prior to their intersection. This introduces a very small overhead for large problems, and is beneficial since: *(i)* the OBDD form of constraints is well-understood and they can be built efficiently, *(ii)* OBDD-to-ZBDD conversion is a simple one-pass algorithm, *(iii)* a significant software infrastructure for OBDD-based symbolic scheduling was already available.In contrast, the heuristic was run using OBDDs only, since the number of nodes was kept small (“max #nodes” column). All experiments were run on Sun SPARCstation 10 using custom C++ OBDD/ZBDD packages. The results are compared to the *Zone Scheduling* (ZS) [10]. This method subdivides a large problem in zones and solves the subproblems using ILP technique.

EWF experiments: To compare our results, the scheduler was run with the same constraints on the number of functional units and busses as in [10]. Register bounds (inputs and outputs included) were identified during the post-processing phase for both heuristic and exact scheduler (column “reg[h/e]”). Maximum size of the partial OBDD solution at the end of each iteration step is reported (“max #nodes”), as well as the CPU times (“CPU[s]”) of the heuristic scheduler. Column “optimal” indicates whether the result of the heuristic scheduler

Table 2: EWF-2 experiments

add	mul	bus	**cycles**	optimal	reg [h/e]	#vars	max #nodes	CPU [s]	CPU rel
3	3	6	**33**	yes	11/11	135	178	0.3	0.067
3	2(*)	6	**33**	yes	11/11	203	178	0.4	0.089
3	1(*)	6	**34**	yes	11/11	203	203	0.8	0.062
3	2	6	**35**	no(34)	11/11	271	291	1.7	0.079
2	2(*)	6	**35**	yes	11/11	271	661	2.4	0.033
2	2	6	**35**	yes	11/11	271	639	2.3	0.034
		4	**39**	yes	12/11	543	1,770	13.5	0.005
2	1(*)	6	**36**	yes	11/11	339	686	2.3	0.013
		4	**39**	yes	11/11	543	2,064	14.8	0.008
2	1	4	**40**	yes	11/11	611	1,232	11.8	0.010
1	1(*)	4	**56**	?	14/ -	1,699	2,603	30.5	-
		2	**68**	?	14/ -	2,515	4,128	88.3	-
1	1	4	**56**	?	14/ -	1,699	2,636	30.3	-
		2	**70**	?	14/ -	2,651	4,403	94.4	-
1-cycle adder, 2-cycle multiplier except: (*) 2-cycle pipelined multiplier									

Table 3: EWF-3 experiments

add	mul	bus	**cycles**	optimal	reg [h/e]	#vars	max #nodes	CPU [s]	CPU rel
3	3	6	**49**	yes	12/12	207	293	0.6	0.057
3	2 (*)	6	**49**	yes	12/12	207	293	0.6	0.057
3	1 (*)	6	**50**	yes	12/12	309	309	1.4	0.039
3	2	6	**52**	no(50)	12/12	513	549	4.7	0.075
2	2 (*)	6	**52**	yes	12/12	513	1,263	6.6	0.012
2	2	6	**52**	yes	12/12	513	1,289	6.5	0.013
		4	**58**	?	13/ -	1,125	3,450	39.8	-
2	1(*)	6	**53**	yes	12/12	615	1,176	5.1	0.005
		4	**58**	?	12/ -	1,125	4,065	42.3	-
2	1	4	**59**	?	12/ -	1,227	2,249	31.5	-
1	1 (*)	4	**84**	?	15/ -	3,777	5,408	123.5	-
		2	**102**	?	15/ -	5,613	7,762	356.7	-
1	1	4	**84**	?	15/ -	3,777	5,408	118.2	-
		2	**105**	?	15/ -	5,919	8,010	372.9	-
1-cycle adder, 2-cycle multiplier except: (*) 2-cycle pipelined multiplier									

could be verified by the exact scheduler. A question mark in that column means that we were not able to construct all minimum-latency schedules before exceeding the time limit (one CPU hour). Column “CPU rel” indicates the ratio of the execution time for the heuristic and exact constructions. The CPU times for the exact constructions were generated using interior constraints as aggressively as possible (i.e. all possible lookaheads were allowed at each scheduling step).

- *EWF-2:* We were able to solve exactly the instances with up to 611 variables in the formulation. The largest instance solved heuristically introduced more than 2,500 variables. Heuristics found minimum-latency solutions in all cases but one, with greatly reduced runtime and memory requirements. Solutions for 2-cycle (non-pipelined) multipliers are the same as in [10]. However, no information on registers is included in [10] and there are no results for pipelined units. Exact solution sizes range between 274 and 4,331 ZBDD nodes.

- *EWF-3:* Benchmark instances with up to 615 variables were solved exactly, and up to 5,919 variables (105-cycle case) heuristically. The heuristic failed in one case to find the optimal execution time (however, as in the case of *EWF-2*, that problem instance was solved exactly). *EWF-3* results were not provided by [10] [2]. Exact solution sizes are 293 to 1,311 ZBDD nodes.

Pre-specified register bounds can be used during the construction to minimize the number of registers needed in the heuristically scheduled results. We ran the heuristics with fixed register bounds of 11 (all *EWF-2* instances) and 12 (all *EWF-3* instances) and in all cases the solutions that required the same number of cycles as those presented in Table 2 and Table 3 were found. However, making an accurate estimate on the register bound is a difficult problem, and a further work to directly incorporate a register cost (not just a bound on the number) is planned. This is important for exact scheduling as well, since register constraints can dramatically reduce the solution space. For example, even using OBDDs, all schedules for the 28-cycle EWF with 10 registers can be found in 690 CPU seconds (7x faster than the unconstrained version, Fig. 1).

FDCT experiments: Although *FDCT* has a relatively moderate number of operations, we include it in this report for two reasons: *(i)* it comes from a practical application, and *(ii)* due to its highly symmetric nature (which should lead to huge solution sets), it is likely to be rather challenging task for exact schedulers. Table 4 presents the results for some larger FDCT instances. As before, the scheduler was run with the same constraints on the number of functional units and busses as in [10]. Register bounds were determined during the post-processing phase for the approach which used both utilization and critical path heuristic. To constrain the solution space, the exact scheduler was run using a pre-specified register bound. The heuristic found the fastest schedules in all cases and performed quite well in terms of the number of

Table 4: FDCT experiments

add	sub	mul	bus	**cycles [*our*/ZS]**	opt.	reg [h/e]	#vars	max #nodes	CPU [s]	CPU rel
2	2	2	10	**10/10**	yes	11/10	251	1,490	47.1	0.179
2	2	2 (*)	10	**11/ -**	yes	9/9	229	2,252	39.6	0.093
1	1	2	8	**13/14**	yes	12/11	377	3,988	35.1	0.007
1	1	2 (*)	8	**14/ -**	?	11/ -	355	4,451	29.7	-
1	1	1	6	**18/20**	yes	11/10	587	12,340	117.0	0.055
1	1	1	4	**18/ -**	yes	11/10	587	5,486	56.7	0.034
1	1	1 (*)	6	**19/ -**	yes	10/9	565	15,346	175.3	0.102
1	1	1 (*)	4	**19/ -**	yes	10/9	565	7,216	91.3	0.063
single-cycle units assumed except: (*) 2-cycle pipelined multiplier										

2. To our knowledge, the only reference to this problem is in [7], where a result for the instance with 1 pipelined multiplier and 3 adders is presented. There is no information on the number of registers and busses.

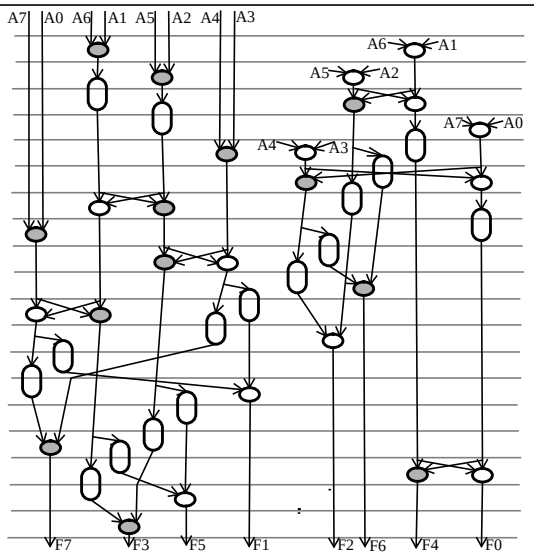

Figure 4. 19-cycle FDCT with pipelined multiplier

registers (typically, off by 1). As can be seen in rows 3 and 5, our heuristic scheduler outperforms *ZS*. This can be explained by the fact that we preserve a complete set of solutions satisfying the heuristic criteria. Even for 2-cycle pipelined multipliers our results are equal (row 4) or better (row 7) that those reported for single-cycle units in *ZS*. Moreover, in rows 6 and 8, we indicate that the problem can be solved with the reduced number of busses (4 instead of 6). The *FDCT* instance with 1 adder, 1 subtracter, 1 pipelined multiplier and 4 busses (row 8) is frequently used to evaluate scheduling results for functional pipelining. However, to our knowledge, the best reported results so far required latency (iteration interval) of 20 cycles [12]. One randomly selected 19-cycle schedule is shown in Fig. 4.

7. Summary

This report demonstrates that the applicability of symbolic techniques can be extended to larger scheduling problems by: using Zero-Suppressed BDDs, applying a set of interior constraints that reduce the size of intermediate solutions, and implicit application of complex constraints. Furthermore, we describe efficient set-heuristics that preserve whole sets of partial solutions exhibiting desirable properties. Not only are some large benchmarks solved exactly for the first time, but the related heuristics demonstrate excellent behavior. This justifies further research in symbolic techniques, development of CDFG scheduling heuristics and the inclusion of loop optimization concepts. A further improvement in runtime efficiency can be expected if execution interval analysis [19] is used for search space reduction. Interior constraints can be viewed as a subset of such analysis.

Symbolic techniques introduce significant flexibility to a design process. Since sets of solutions are preserved, it is possible to explore the solution space after the scheduling is done and apply additional constraints incrementally. The process can be made adaptive -- we can try to preserve as large as possible set of solutions based on acceptable speed of the construction. Thus, a wide range of solutions, from heuristic to optimal, can be generated. Experiments to do this in a fully automated fashion are under way.

Acknowledgment -- We thank A. Crews, C. Monahan, and A. Stornetta for helping improve the C++ BDD packages.

References:

[1] K.S. Brace, R.L. Rudell, and R.E. Bryant, "Efficient Implementation of a BDD package", *Proc. 27th DAC*, 1990.

[2] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. Computers*, vol. c-35, no.8, Aug. 1986.

[3] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams", *ACM Computing Surveys*, vol.24, no.3, Sep. 1992.

[4] R.Camposano, "Path-Based Scheduling: for Synthesis", *IEEE Trans. CAD/ICAS*, vol.10, no.1, Jan. 1991.

[5] C.N. Coelho Jr. and G. De Micheli, "Dynamic Scheduling and Synchronization Synthesis of Concurrent Digital Systems under System-Level Constraints", *Proc. ICCAD*, 1994.

[6] S. Davidson, D. Landskov, B. Shriver, and P. Mallett, "Some Experiments in Local Microcode Compaction for Horizontal Machines", *IEEE Trans. Comp*. vol. c-30, no.7, July 1981.

[7] C.H. Gebotys and M.I. Elmasry, "Global Optimization Approach for Architectural Synthesis", *IEEE Trans. C*AD/ICAS, vol.12, no.9, September 1993.

[8] C.H. Gebotys, "Throughput Optimized Architectural Synthesis", *IEEE Trans. VLSI Syst.*, vol.1, no.3, Sep. 1993.

[9] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A Formal Approach to the Scheduling Problem in High Level Synthesis", *IEEE Trans. C*AD/ICAS, vol.10, no.4, Apr. 1991.

[10] C.-T. Hwang and Y.-C. Hsu, "Zone Scheduling", *IEEE Trans. CAD/ICAS*, vol.12, no.7, July 1993.

[11] H. Komi, S. Yamada, and K. Fukunaga, "A Scheduling Method by Stepwise Expansion in High-Level Synthesis", *Proc. ICCAD*, 1992.

[12] T.-F. Lee, A.C.-H. Wu, D. Gajski, and Y.-L. Lin, "A Transformation-Based Method for Loop Folding", *IEEE Trans. CAD/ICAS*, vol.13, no.4, April 1994.

[13] D. J. Mallon and P.B. Denyer, "A New Approach To Pipeline Optimisation", *Proc. EDAC*, 1990.

[14] S.-I. Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems", *Proc. 30th DAC*, 1993.

[15] P.G. Paulin and J.P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Tran*s. CAD/ICAS, vol.8, no.6, June 1989.

[16] I. Radivojević and F. Brewer, "Symbolic Techniques for Optimal Scheduling", *Proc. 4th SASIMI*, Nara, Japan, 1993.

[17] I. Radivojević and F. Brewer, "Ensemble Representation and Techniques for Exact Control-Dependent Scheduling", *Proc. 7th Intl. Symp. High Level Synthesis*, 1994.

[18] I. Radivojević and F. Brewer, "Incorporating Speculative Execution In Exact Control-Dependent Scheduling", *Proc. 31th DAC*, 1994.

[19] A.H. Timmer and J.A.G. Jess, "Execution Interval Analysis under Resource Constraints", *Proc. ICCAD*, 1993.

[20] J. Yang, G. De Micheli and M. Damiani, "Scheduling with Environmental Constraints based on Automata Representation", *Proc. EDAC*, 1994.