Symbolic Scheduling Techniques [1]

Ivan Radivojević, Forrest Brewer

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

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

We propose a new exact formulation of resource-constrained control/data-flow scheduling. Unlike current techniques, a solution is generated in which all satisfying schedules are encapsulated in a compressed representation. The technique supports various forms of code motion and extraction of parallelism not explicit in the input description. In addition, the formulation allows development of set-based heuristics enabling application to large scheduling problems. Currently, exact scheduling of arbitrary forward-branching control structures is supported. Control model generalizations are planned.
1. INTRODUCTION

• *Operation Scheduling* -- assignment of operations to bounded time slots of a synchronous system subject to:
  - Data/control-flow dependencies
  - Resource constraints (functional units, busses, registers)

Additional constraints: timing, synchronization...

• *Current Solutions*:
  - Priority based heuristics
  - Integer Linear Programming (ILP) based optimizations
  - Symbolic techniques
1.1. HEURISTIC SCHEDULERS

- Applicable to wide variety of problems and large CDFGs (e.g. list scheduling, path-based scheduling, force-directed scheduling)

- May fail to find solutions in tightly constrained problems
  - Often very large pool of candidates satisfying pre-specified criteria
  - Typically only one representative is selected
  - Cannot recuperate from suboptimal decisions
  - Potentially expensive to perform lookahead or to backtrack

- Difficult to add new constraints incrementally
1.2. ILP-BASED OPTIMIZATIONS

- Exact ILP methods [Hwang et al.] are computationally very expensive

- Implementations:
  - remapping of constraints and conversion to relaxed LP models [Gebotys]
  - ILP-based heuristics (e.g. zone partitioning, stepwise expansion)

- Problems:
  - no practical model for control-dependent behavior
  - sensitive to the number of formulation variables (problem size)
  - only a single representative solution
  - heuristic approach: subproblems solved optimally, but decisions are local
1.3. SYMBOLIC TECHNIQUES

• Describe scheduling constraints as Boolean functions

• Build BDD (Binary Decision Diagram) encapsulating all feasible solutions

• Recently introduced:
  1. Radivojević, Brewer [SASIMI’93, HLSS’94, DAC’94, IEICE’95]
     - topic of this presentation...
  2. De Micheli et al. [EDAC’94, ICCAD’94]
     - FSM to capture resource/timing/synchronization constraints
     - specification-level formalism restricts code motion
     - potentially exponential number of formulation variables
MOTIVATION

• Encapsulate all feasible schedules

• Why is this advantageous?
  - Constraints can be incrementally applied
  - Increased freedom in subsequent synthesis steps
  - Exact scheduling for arbitrary control
  - For very large problems, allows set-based high-quality heuristics

• What is needed?
  - Flexible control model that can be treated systematically
  - Efficient representation to manipulate large data sets
## SYMBOLIC vs. ILP

- Is there any difference?

<table>
<thead>
<tr>
<th></th>
<th>constraint type</th>
<th>#solution</th>
<th>#variables</th>
<th>support for</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DFGs</td>
</tr>
<tr>
<td>Symbolic</td>
<td>any Boolean function</td>
<td>all</td>
<td>$O\left[(#cycles)\times(#ops) + (#cond)\right]$</td>
<td>yes</td>
</tr>
<tr>
<td>ILP</td>
<td>linear</td>
<td>1</td>
<td>$O\left[(#cycles)\times(#ops) \times 2(#cond)\right]$</td>
<td>yes</td>
</tr>
</tbody>
</table>

#cycles - number of time steps, #ops - number of operations, #cond - number of conditionals.
2. EXPLOITING IMPLICIT PARALLELISM

• Ability to extract parallelism not explicit in the input description dramatically increases scheduling quality
  
  (e.g. irredundant operation scheduling, out-of-order execution of conditionals, speculative execution, global treatment of parallel control structures)

• Problems:
  
  - As the number of control paths increases, it becomes very difficult to keep track of the mutual exclusiveness among the operations
  - Impossible to accurately predict resource usage and availability in a static fashion
IRREDUNDANT OPERATION SCHEDULING

- If operation is *redundant* on a particular control path it need not be scheduled there

- Operation 1 is redundant on ‘False’ path and operation 2 is redundant on ‘True’ path -- this can be used to reduce resource requirements
OUT-OF-ORDER EXECUTION OF CONDITIONALS

• Dynamically resolve execution order of *conditionals* (operations that generate control signals)

resources:
- 2 adders (white)
- 1 subtracter (black)
- 1 comparator

execution time:
- 4 cycles (*1* before *2*)
- 3 cycles (*2* before *1*)

• Faster execution if conditional *2* is evaluated before conditional *1*
SPECULATIVE OPERATION EXECUTION

- Operations from branch arcs executed before the condition is known

- Improves execution time when there are sufficient resources
SPECULATIVE EXECUTION MODEL

- Dependency between the conditional and the *fork* node is removed

- Dependency between the conditional and the *join* node is enforced

- *Restriction*: operations after the join node cannot be executed before the corresponding conditional

  ⇒ Each operation is scheduled at most once per control path (prevents a potential explosion of operator’s instances)

- Solved *exactly* [DAC’94]
CDFG transformation for MAHA [Parker et al., DAC’86]:

- Critical path length bounded by data dependencies
- Execution order of conditionals not dictated by input description
PARALLEL CONTROL STRUCTURES

- *Globally* schedule operations belonging to parallel or correlated trees
- False control path management essential

- Hard problem ⇒ attracted very little attention so far
2.1. FORMULATION

- Scheduling constraints as Boolean functions
  ⇒ build OBDD corresponding to the intersection

- Each variable $C_{ij}$ represents operation $j$ occurring at time step $i$

- Add a ‘Guard’ variable for each conditional
  ⇒ control path is a product of corresponding guards

- Define a ‘Guard’ function $\Gamma_i$ for each operation $i$
  ⇒ identifies control paths where operation $i$ has to be scheduled

- Solution is a collection of traces
  ⇒ trace is a valid execution instance for a particular control path

- Trace is a product term containing the information on a control path (guard variables) and its schedule (0/1 assignment of $C_{ij}$ variables)
EXAMPLE [Kim et al., ICCAD’91]:

- Blocks indicate guard functions $\Gamma$
  - $\Rightarrow \Gamma$’s not restricted to product terms (can handle: goto, exit, case)

- Operations with $\Gamma=1$ scheduled simultaneously under all control combinations
OBDD REPRESENTATION

- Compressed representation of very large number of solutions

- Relatively small number of OBDD nodes ⇒ fast manipulations
WHAT IS INCLUDED IN THE SOLUTION?

- Three possible execution traces exist for the CDFG below (assume: 1 “white” resource available, no speculative execution)

  "True" and "False_1" form a deterministic executable (ensemble) schedule
  "False_2" is an invalid trace -- cannot be paired with "True"

Solution includes only traces belonging to at least one ensemble schedule
TREATMENT OF RESOURCES

• If two operations $i$ and $j$ have guard functions $\Gamma_i$ and $\Gamma_j$ such that $\Gamma_i \Gamma_j = 0$, $i$ and $j$ are pairwise mutually exclusive [SASIMI’93], however...
  - pairwise mutual exclusion not sufficient for optimal use of resources
  - higher-order analysis potentially expensive

• We introduce *Trace Validation* as an alternative to mutex analysis

• Trace Validation imposes causality and completeness on the set of all traces and ensures that each trace is a part of some ensemble schedule
  - *causality* -- traces must match before condition is evaluated
  - *completeness* -- valid execution traces must exist for all control paths
  $\Rightarrow$ implicitly verifies that resource constraints satisfied on individual traces are not violated in the ensemble schedule
**TRACE VALIDATION**

- Proposed algorithm *iteratively* validates all traces for all ensemble schedules *simultaneously*, using only Boolean operations on the OBDD data-structure

- Generates polynomial #constraints (regardless of #traces)

\[ \begin{align*}
\text{B}_i & \quad (\text{code block } i), \quad S_i & \quad (\text{schedule for } B_i), \quad C \quad (\text{conditional}), \quad G \quad (\text{guard corresponding to } C) \\
\text{NOTE: } S_1 \text{ and } S_4' \text{ potentially shared in OBDD representation} 
\end{align*} \]
TRACE VALIDATION

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

- **S** - set of all traces, S(0) - initial set of non-validated traces, S(i) - set of traces at iteration i
- **V** - set of all variables not including guard variables, V'(j) - subset of V corresponding to time steps ≤ j
- **S'** - set of traces from which all variables (V-V'(j)) are removed: S = ∃(V - V'(j))S(i)
- **C** = [c_1, c_2, ..., c_n] - set of all conditionals
- **G** = [G_1, G_2, ..., G_n] - set of guards corresponding to the conditionals
- **R(j)** = [R_1(j), R_2(j), ..., R_n(j)] - resolution vector, set of Boolean functions indicating whether a conditional c_k was scheduled prior to time step j: R_k(j) = ∑ C_{lk} for (l < j)
- **G_{res}** - subset of guards that are resolved prior to time step j
- ∃_x f = f_x + f_¬x - existential abstraction, ∀_x f = f_x f_¬x - universal abstraction
2.2. CONSTRUCTION

• Solution is constructed *iteratively* -- at every time step the following constraints are applied:
  1. Uniqueness
  2. Precedence relations
  3. Resource bounds
  4. Removal of redundantly scheduled operations
  5. Trace validation
  6. Completion test

• *All* of the solutions for *all* control paths are constructed *simultaneously*

• Trace validation ensures that each trace is part of some executable (causal and complete) schedule
WHY ITERATIVE CONSTRUCTION?

- It is possible [SASIMI’93] to construct all constraints during preprocessing and form their intersection.

- In this version *iterative* construction is implemented:
  - constraints incrementally generated during construction
  - easy to test for termination
  - no spurious partial solutions
  - smaller intermediate OBDDs

- Valid partial solutions simplify development of heuristics.
OBDD FORM OF CONSTRAINTS

- BDD representations have regular structure -- built directly from the CDFG (no need for intermediate Boolean equation form)
2.3. APPLICABILITY TO LARGE PROBLEMS

• The main challenge for symbolic techniques -- large solution sets

• “... 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...” (R. E. Bryant, 1992)

• We have shown [HLSS’94] that some standard benchmark instances have billions of optimal solutions

⇒ In such cases, OBDD representation can become too large to be practical
APPLICABILITY TO LARGE PROBLEMS

• It has been generally considered that symbolic techniques are applicable only to small scheduling problems

• We demonstrate [ED&TC’95] that applicability of these techniques can be extended to large DFGs by:
  - using Zero-Suppressed BDDs [Minato]
  - applying the set of interior constraints to reduce size of intermediate solutions
  - implicit application of complex constraints
  - formulation of heuristics that preserve whole sets of partial solutions exhibiting desirable properties
SYMBOLIC ZBDD REPRESENTATION

• Example: 28-cycle Elliptic Wave Filter (EWF) benchmark
  - 34 operations but 437 representation variables
  - OBDD: Every solution includes all 437 variables -- only 34 of them “1”
    ⇒ extremely large solution ( > 130,000 nodes)
  - ZBDD: “0”-variables are implicit (suppressed)
    ⇒ ten-fold reduction in solution size

• Drastically decreased memory requirements

• ZBDD form of constraints can be more complex, but the construction is still efficient -- solutions (both intermediate and final) are smaller
INTERIOR CONSTRAINTS

• Although the size of the final solution is typically very moderate, the partial solutions can become prohibitively large

⇒ Ideally, intermediate size should never exceed the size of the final solution

• Identify and discard partial schedules that cannot contribute to a set of optimal solutions

⇒ at a particular time step such partial schedules cannot terminate for given resources and pre-specified upper bound on execution time

• Set of interior constraints is dynamically generated to prune the OBDD/ZBDD

• Very cost-effective for large DFGs
INTERIOR CONSTRAINTS

- Assume that at the beginning of step $i$ there are:
  - $n$ addition operations that have ALAP bounds in the range $[i \ldots (i+k-1)]$
  - $m$ single-cycle adders available
  \[ \Rightarrow \] At least $(n-km)$ of these additions must be completed prior to step $i$ in a feasible solution

- Similar constraints for pipelined and multicycle functional units

- $k^{th}$-order constraints enable an early detection of many (not necessarily all) partial schedules destined to be discarded within the next $k$ steps

- The completeness of the solution set preserved, hence no impact on optimality
IMPLICIT APPLICATION OF CONSTRAINTS

• It can happen that the partial solution is of a very moderate size, but the constraint to be applied cannot be built

• Similar problem arises in many symbolic applications
  (e.g. in FSM traversal, where transition functions may become prohibitively complex and Characteristic Functions [Coudert et al.] are introduced to alleviate the problem)

• Apply constraints implicitly without attempting to build constraint BDD

• Example: application of register constraints that can be build explicitly for small problems only
**Example**: Generalized resource bound applied to registers

- Vertices in above template can be arbitrary Boolean functions \((f_1, \ldots, f_n)\)
  - typically, constraint *too large* to be built

  (i) introduce a set of auxiliary variables \((y_1, \ldots, y_n)\) corresponding to \((f_1, \ldots, f_n)\)

  (ii) build above template \(T\) using \((y_1, \ldots, y_n)\)

  (iii) compute \(P^0 = \text{And}(P', T)\), where \(P'\) is a partial solution

  (iv) compute new partial solution \(P'' = P^n\) using recursive substitution:

\[
P^i = P^{(i-1)} \mid y_i = f_i
\]
SET-HEURISTICS

• Since valid partial schedules are available after each step, it is possible to devise efficient heuristics

• Utility-based heuristic propagates only the subset of schedules with maximum utilization of resources
  -- Since all such schedules are propagated, this simple heuristic has good behavior

• Set-Heuristics are very robust
  -- Construction pace shows little sensitivity to the upper bound on scheduling latency (initialization parameters)
### 28-cycle EWF: exact and heuristic constructions

- **resources**: 1 single-cycle adder, 1 two-cycle multiplier ( \( > 10^{9} \) solutions)
- **#variables**: 437

- `_zbdd`: exact solution (ZBDD construction), no interior constraints: \( \sim 20.5 \text{ min} \)
- `_ic_zbdd`: exact solution (ZBDD construction) built using interior constraints: \( \sim 12.5 \text{ min} \) \(^1\)
- `_heu_obdd`: utility-based set-heuristic solution (OBDD construction): \( \sim 38 \text{ s} \) \(^2\)
- `_heu_zbdd`: utility-based set-heuristic solution (ZBDD construction): \( \sim 110 \text{ s} \)
- `_obdd`: exact solution (OBDD construction): \( > 1.5 \text{ h} \)

1. For optimal number of registers (10), the size of exact ZBDD solution decreases from \( \sim 14.5 \) to \( \sim 3.5 \) Knodes.
2. \( \sim 9 \text{ s} \) if both utilization and critical path are used as heuristic criteria
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 construction), no interior constraints: could not be constructed
- **_ic_zbdd**: exact solution (ZBDD construction) built using interior constraints: > 1.5 h [1]
- **_heu_obdd**: utility-based set-heuristic solution (OBDD construction): ~ 2 min [2]
- **_heu_zbdd**: utility-based set-heuristic solution (ZBDD construction): ~ 12.5 min
- **_obdd**: exact solution (OBDD construction) -- not constructed (converted from ZBDD solution)

---

1. For optimal number of registers (10), the size of exact ZBDD solution decreases from ~18.5 to ~6.5 Knodes.
2. ~ 30 s if both utilization and critical path are used as heuristic criteria.
2.4. EXPERIMENTAL RESULTS

- Experiments on standard set of benchmarks:
  - EWF, EWF-2, EWF-3 (EWF unfolded 2/3 times), FDCT, Hal, Maha, Kim, Wakabayashi [DAC’89], MulT [Wakabayashi et al., DAC’92]

- CDFGs with up to 102 operations

- Execution times up to 105 time steps

- Exact scheduler: ~ 1,000 formulation variables

- Heuristic scheduler: ~ 6,000 formulation variables

- Very moderate CPU times and memory requirements
EXPERIMENTAL RESULTS

• **EWF (Elliptic Wave Filter, 34 operations):**

<table>
<thead>
<tr>
<th></th>
<th>17</th>
<th>17</th>
<th>18</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>20</th>
<th>21</th>
<th>28</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>#cycles</td>
<td>17</td>
<td>17</td>
<td>18</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>20</td>
<td>21</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>#adders</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>#multipliers</td>
<td>2 (*)</td>
<td>3</td>
<td>1 (*)</td>
<td>2</td>
<td>1 (*)</td>
<td>2</td>
<td>1 (*)</td>
<td>1</td>
<td>1 (*)</td>
<td>1</td>
</tr>
<tr>
<td>#busses</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>#registers</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>#variables</td>
<td>63</td>
<td>63</td>
<td>97</td>
<td>97</td>
<td>131</td>
<td>165</td>
<td>165</td>
<td>199</td>
<td>437</td>
<td>437</td>
</tr>
<tr>
<td>#nodes</td>
<td>82</td>
<td>82</td>
<td>194</td>
<td>209</td>
<td>2,237</td>
<td>2,760</td>
<td>1,905</td>
<td>704</td>
<td>48,649</td>
<td>32,487</td>
</tr>
<tr>
<td>#schedules</td>
<td>18</td>
<td>18</td>
<td>336</td>
<td>18</td>
<td>10,692</td>
<td>52,821</td>
<td>5,142</td>
<td>2,355</td>
<td>4.29e+9</td>
<td>2.63e+8</td>
</tr>
<tr>
<td>CPU time [s]</td>
<td>0.26</td>
<td>0.32</td>
<td>1.14</td>
<td>1.43</td>
<td>7.92</td>
<td>35.12</td>
<td>26.00</td>
<td>7.89</td>
<td>2,420.25</td>
<td>1520.51</td>
</tr>
</tbody>
</table>

2-cycle multiplier and single-cycle adder except: (*) 2-cycle pipelined multiplier.

• **All optimal solutions to all benchmark instances**

• **Number of OBDD nodes significantly smaller than \((#variables)^2\)**
EXPERIMENTAL RESULTS

Benchmarks with branching (exact solutions)

<table>
<thead>
<tr>
<th></th>
<th>Maha</th>
<th>Parker</th>
<th>Kim</th>
<th>Waka</th>
<th>MulT</th>
</tr>
</thead>
<tbody>
<tr>
<td>#cycles(spec)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>longest</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>average</td>
<td>3.31</td>
<td>2.25</td>
<td>2.13</td>
<td>5.75</td>
<td>5.0</td>
</tr>
<tr>
<td>#cycles(non_spec)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>#adders</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>#subtracters</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>#comparators</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#variables</td>
<td>65</td>
<td>49</td>
<td>49</td>
<td>71</td>
<td>55</td>
</tr>
<tr>
<td>#nodes</td>
<td>428</td>
<td>325</td>
<td>220</td>
<td>543</td>
<td>271</td>
</tr>
<tr>
<td>#traces</td>
<td>15</td>
<td>43</td>
<td>12</td>
<td>124</td>
<td>21</td>
</tr>
<tr>
<td>CPU time [s]</td>
<td>13.80</td>
<td>6.32</td>
<td>8.91</td>
<td>7.63</td>
<td>2.84</td>
</tr>
</tbody>
</table>

Comparison with others: average (longest) path

<table>
<thead>
<tr>
<th></th>
<th>Maha</th>
<th>Parker</th>
<th>Kim</th>
<th>Waka</th>
<th>MulT</th>
</tr>
</thead>
<tbody>
<tr>
<td>our</td>
<td>3.31 (5)</td>
<td>2.25 (4)</td>
<td>2.13 (4)</td>
<td>5.75 (6)</td>
<td>5 (7)</td>
</tr>
<tr>
<td>TS [12]</td>
<td>3.31 (5)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4.75 (7)</td>
</tr>
<tr>
<td>CVLS [33]</td>
<td>3.31 (5)</td>
<td>2.38 (4)</td>
<td>2.38 (4)</td>
<td>5.75 (6)</td>
<td>-</td>
</tr>
<tr>
<td>Kim et al. [17]</td>
<td>4.62 (8)</td>
<td>-</td>
<td>6.25 (7)</td>
<td>4.75 (7)</td>
<td>-</td>
</tr>
</tbody>
</table>

- Comparable or superior results using very moderate CPU resources
EXPERIMENTAL RESULTS

• 54-cycle EWF:

**Robustness analysis of the heuristic scheduler (EWF)**

<table>
<thead>
<tr>
<th>cycles</th>
<th>upper bound</th>
<th># vars</th>
<th>utility-based max</th>
<th>utility + CP max</th>
<th>CP U [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>max # nodes</td>
<td>max # nodes</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>54</td>
<td>967</td>
<td>14,328</td>
<td>119</td>
<td>2,210</td>
</tr>
<tr>
<td>55</td>
<td>1,001</td>
<td>14,839</td>
<td>124</td>
<td>2,292</td>
<td>31</td>
</tr>
<tr>
<td>56</td>
<td>1,035</td>
<td>15,559</td>
<td>138</td>
<td>2,392</td>
<td>34</td>
</tr>
<tr>
<td>59</td>
<td>1,137</td>
<td>17,151</td>
<td>159</td>
<td>2,616</td>
<td>36</td>
</tr>
<tr>
<td>64</td>
<td>1,307</td>
<td>19,378</td>
<td>202</td>
<td>2,914</td>
<td>40</td>
</tr>
</tbody>
</table>

2-cycle adder, 2-cycle multiplier

• Set-Heuristics are very robust: the construction pace shows little sensitivity to the upper bound on scheduling latency (initialization parameters)

-- Can be used to derive accurate upper bounds for exact schedulers whose runtime efficiency is more sensitive to such estimates
EXPERIMENTAL RESULTS

- **FDCT (Finite Discrete Cosine Transform, 42 operations):**

    FDCT experiments (heuristic vs. exact)

<table>
<thead>
<tr>
<th>ad</th>
<th>sub</th>
<th>mul</th>
<th>bus</th>
<th>cycles [ZS]</th>
<th>cycles [our]</th>
<th>optimal</th>
<th>reg [h/e]</th>
<th>#vars</th>
<th>max #nodes</th>
<th>CPU [s]</th>
<th>CPU rel</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>yes</td>
<td>11/10</td>
<td>251</td>
<td>1,490</td>
<td>79.1</td>
<td>0.30</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>yes</td>
<td>9/9</td>
<td>229</td>
<td>2,252</td>
<td>65.4</td>
<td>0.15</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>14</td>
<td>13</td>
<td>yes</td>
<td>12/11</td>
<td>377</td>
<td>3,988</td>
<td>52.8</td>
<td>0.01</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>-</td>
<td>14</td>
<td>?</td>
<td>11/-</td>
<td>355</td>
<td>4,451</td>
<td>42.7</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>20</td>
<td>18</td>
<td>yes</td>
<td>11/10</td>
<td>587</td>
<td>12,340</td>
<td>179.3</td>
<td>0.08</td>
</tr>
</tbody>
</table>

- **Heuristic OBDD scheduler**
  
  *(max #nodes)* - max OBDD size during construction, *(CPU [s])* - CPU time, *(reg [h/e])* - heuristic/exact #registers, *(CPU rel)* - heuristic/exact(ZBDD) CPU time ratio

- **Outperforms Zone Scheduling** (ILP-based heuristic) which partitions large problems and solves the subproblems optimally
**EXPERIMENTAL RESULTS**

- *EWF unfolded 2 times (68 operations):*

**EWF-2 experiments (heuristic vs. exact)**

<table>
<thead>
<tr>
<th>ad</th>
<th>mul</th>
<th>bus</th>
<th>cycle</th>
<th>optimal</th>
<th>reg</th>
<th>#vars</th>
<th>max #nodes</th>
<th>CPU [s]</th>
<th>CPU rel</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>33</td>
<td>yes</td>
<td>11/11</td>
<td>135</td>
<td>178</td>
<td>0.7</td>
<td>0.15</td>
</tr>
<tr>
<td>3</td>
<td>2 (*)</td>
<td>6</td>
<td>33</td>
<td>yes</td>
<td>11/11</td>
<td>203</td>
<td>178</td>
<td>0.7</td>
<td>0.15</td>
</tr>
<tr>
<td>3</td>
<td>1 (*)</td>
<td>6</td>
<td>34</td>
<td>yes</td>
<td>11/11</td>
<td>203</td>
<td>203</td>
<td>1.5</td>
<td>0.11</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>6</td>
<td>35</td>
<td>no (34)</td>
<td>11/11</td>
<td>271</td>
<td>291</td>
<td>3.2</td>
<td>0.14</td>
</tr>
<tr>
<td>2</td>
<td>2 (*)</td>
<td>6</td>
<td>35</td>
<td>yes</td>
<td>11/11</td>
<td>271</td>
<td>661</td>
<td>4.3</td>
<td>0.05</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>35</td>
<td>yes</td>
<td>11/11</td>
<td>271</td>
<td>639</td>
<td>4.2</td>
<td>0.06</td>
</tr>
<tr>
<td>4</td>
<td>39</td>
<td></td>
<td></td>
<td>yes</td>
<td>12/11</td>
<td>543</td>
<td>1,770</td>
<td>23.8</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>1 (*)</td>
<td>6</td>
<td>36</td>
<td>yes</td>
<td>11/11</td>
<td>339</td>
<td>686</td>
<td>4.2</td>
<td>0.02</td>
</tr>
</tbody>
</table>
EXPERIMENTAL RESULTS

- *EWF* unfolded 3 times (102 operations):

<table>
<thead>
<tr>
<th>add</th>
<th>mul</th>
<th>bus</th>
<th>cycles</th>
<th>optimal</th>
<th>reg [h/e]</th>
<th>#vars</th>
<th>max #nodes</th>
<th>CPU [s]</th>
<th>CPU rel</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>49</td>
<td>yes</td>
<td>12/12</td>
<td>207</td>
<td>293</td>
<td>1.1</td>
<td>0.104</td>
</tr>
<tr>
<td>3</td>
<td>2 (*)</td>
<td>6</td>
<td>49</td>
<td>yes</td>
<td>12/12</td>
<td>207</td>
<td>293</td>
<td>1.1</td>
<td>0.104</td>
</tr>
<tr>
<td>3</td>
<td>1 (*)</td>
<td>6</td>
<td>50</td>
<td>yes</td>
<td>12/12</td>
<td>309</td>
<td>309</td>
<td>2.4</td>
<td>0.066</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>6</td>
<td>52</td>
<td>no (50)</td>
<td>12/12</td>
<td>513</td>
<td>549</td>
<td>8.8</td>
<td>0.140</td>
</tr>
<tr>
<td>2</td>
<td>2 (*)</td>
<td>6</td>
<td>52</td>
<td>yes</td>
<td>12/12</td>
<td>513</td>
<td>1,263</td>
<td>12.1</td>
<td>0.022</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>58</td>
<td>?</td>
<td>13/-</td>
<td>1,125</td>
<td>3,450</td>
<td>71.9</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>52</td>
<td>yes</td>
<td>12/12</td>
<td>513</td>
<td>1,289</td>
<td>11.9</td>
<td>0.024</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>58</td>
<td>?</td>
<td>13/-</td>
<td>1,125</td>
<td>3,450</td>
<td>70.9</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1 (*)</td>
<td>6</td>
<td>53</td>
<td>yes</td>
<td>12/12</td>
<td>615</td>
<td>1,176</td>
<td>9.5</td>
<td>0.009</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>58</td>
<td>?</td>
<td>12/-</td>
<td>1,125</td>
<td>4,065</td>
<td>74.4</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>59</td>
<td>?</td>
<td>12/-</td>
<td>1,227</td>
<td>2,249</td>
<td>56.9</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1 (*)</td>
<td>4</td>
<td>84</td>
<td>?</td>
<td>15/-</td>
<td>3,777</td>
<td>5,408</td>
<td>186.2</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>102</td>
<td>5</td>
<td>102</td>
<td>?</td>
<td>15/-</td>
<td>5,613</td>
<td>7,762</td>
<td>547.2</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>84</td>
<td>?</td>
<td>15/-</td>
<td>3,777</td>
<td>5,408</td>
<td>176.9</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>105</td>
<td>5</td>
<td>105</td>
<td>?</td>
<td>15/-</td>
<td>5,919</td>
<td>8,010</td>
<td>591.2</td>
<td>-</td>
</tr>
</tbody>
</table>

1-cycle adder, 2-cycle multiplier except: (*) 2-cycle pipelined multiplier
EXPERIMENTAL RESULTS

• Very encouraging results for large DFGs:
  - heuristics are robust and efficient (CPU time, memory usage) while still generating excellent results

• Topics to be investigated:
  - incorporate register cost (not just a bound on the number) in heuristics
  - develop efficient heuristics for CDFGs
  - speed-up ZBDD manipulations
2.5. MODEL GENERALIZATION

- Restriction in current model for speculative execution:
  -- operation after the join node cannot be pre-executed

- Generalized model can lead to an explosion of operation instances due to a node duplication [Percolation Based Synthesis]
  -- is there an efficient implementation?
2.6. LOOP OPTIMIZATIONS

- Goal: increase *throughput* by decreasing loop latency

**FUNCTIONAL PIPELINING**

**LOOP WINDING**
LOOP OPTIMIZATIONS (cont.)

- Exact formulation
  (i) Modification of resource constraints: if latency is $l$, then operations at
time steps $i, i+l, i+2l... \ (1 \leq i \leq l)$ share resources
  (ii) Enforce additional inter-iteration dependencies (loop winding)

- Heuristics
  - greedy set-based heuristics not directly applicable: tend to fill early time
  slots during iterative construction process

- Formulation extensions to allow control-dependent behavior within the
loop body?
  - due to the problem complexity, very little research reported so far...
PUBLICATIONS:


