Behavior-Level Observability Don't-Cares and Application to Low-Power Behavioral Synthesis

Jason Cong
Computer Science Department, UCLA
cong@cs.ucla.edu

Bin Liu
Computer Science Department, UCLA
bliu@cs.ucla.edu

Zhiru Zhang
AutoESL Design Technologies, Inc.
zhiruz@autoesl.com

ABSTRACT

Many techniques for power management employed in advanced RTL synthesis tools rely explicitly or implicitly on observability don't-care (ODC) conditions. In this paper we present a systematic approach to maximizing the effectiveness of these techniques by generating power-friendly RTL descriptions in a behavioral synthesis tool. We first introduce the concept of behavior-level observability and investigate its relation with observability under a given schedule, using an extension of Boolean algebra. We then propose an efficient algorithm to compute behavior-level observability on a data-flow graph. Our algorithm exploits knowledge about select and Boolean instructions, and allows certain forms of other knowledge, once uncovered, to be considered for stronger observability conditions. We also describe a behavioral synthesis flow where behavior-level observability is used to guide the scheduler toward maximizing the likelihood that execution of power-hungry instructions will be avoided under a latency constraint. Experimental results show that our approach is able to reduce total power, and it outperforms a previous method in [15] by 17.7% on average, on a set of real-world designs. To the best of our knowledge, this is the first work to use a comprehensive behavioral-level observability analysis to guide optimizations in behavioral synthesis.

Categories and Subject Descriptors

D.2.8 [**Integrated Circuits**]: Design Aids

General Terms

Algorithms, Experimentation

1. INTRODUCTION

With power dissipation an increasingly critical issue in VLSI design, a number of techniques for power reduction have been developed in advanced RTL synthesis tools. While some techniques try to replace power-hungry devices with their power-efficient counterparts, other orthogonal approaches reduce power by avoiding unnecessary operations, using techniques such as operand isolation and clock gating. Observability don't-care (ODC) conditions, originally introduced by the logic synthesis community (for example, [1]), play an important role in the identification of unnecessary operations in a Boolean network. Isolation cells can be inserted at inputs of a functional unit when its result is not observable [2]. For clock gating, the simplest approach is based on stability conditions [3]: when a register is accepting a value equal to the one already stored in it, its clock can be gated. However, it is recognized that exploiting ODC conditions in clock gating can lead to significantly more power reduction by avoiding unobservable value changes in registers [3–7].

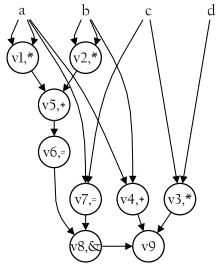
The problem of ODC computation in a sequential RTL model has been approached in a number of ways. Some prior work views the circuit as a finite-state machine (FSM) and calculates the exact ODC condition for every bit using formal methods [5,6]. However, the number of states in an FSM can be exponential in terms of the number of registers. Thus, the exact approach can be prohibitively expensive for moderately large designs. Methods developed in practical systems are often conservative but more scalable, without a thorough analysis of the FSM. The work in [2] assumes that every value stored in a register is observable and only performs analysis on combinational parts of the circuit. The approach in [4] relies on special patterns in the hardware-description language (HDL) to compute ODC conditions, and thus the quality of result depends on the HDL coding style. The algorithm in [7] detects ODC conditions based on datapath topology, using backward traversal and levelization techniques. A more recent work [3] shows that more ODC conditions can be uncovered in the results of [7] by propagating ODC conditions that are already utilized in other parts of the design (possibly discovered manually by the designer). All these methods are reported to be very effective in practice. However, it is not clear how much opportunity for power optimization still exists due to obvious pessimism when computing ODC conditions.

Even with a powerful tool that could calculate the exact ODC condition for every signal in an RTL description efficiently, huge opportunities remain unexploited at a higher level where there is freedom in choosing a good RTL structure among many alternatives of the same functionality. A motivating example is shown in Figure 1, where different schedules with the same latency imply different opportunities for avoiding operations. Note that there is a select instruction in the code (corresponding to a multiplexor in the circuit), and thus the evaluation of some values are not necessary depending on which value is selected as the output. In the first schedule, two multiplications v1 and v2 are always executed. In the second one, when v7 is evaluated as false in the first cycle, v9 will be equal to v3, and values including v1, v2, v5 and v6 are not needed because they will not influence the output. By scheduling instructions intelligently and exploiting ODC conditions in the resulting RTL design, we can effectively restructure the control flow and get different equivalent C codes as shown on the right side; the resulting implementation can have very different power under the same performance constraint. A powerful behavioral synthesis tool could explore such higher-level opportunities to generate and redistribute ODC conditions, whereas an RTL synthesis tool is unable to

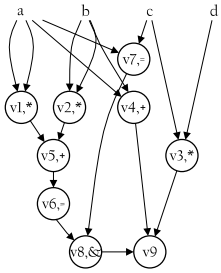
ISLPED'09, August 19–21, 2009, San Francisco, California, USA.

explore this design space — it can only take advantage of available ODC conditions for a fixed schedule.

```
// A data-flow graph in C syntax.
int module(int a, int b, int c, int d) {
  int v1 = a * a;
  int v2 = b * b;
  int v3 = c * d;
  int v4 = a + b;
  int v5 = v1 + v2;
  bool v6 = (v5 == 100);
  bool v7 = (a == c);
  bool v8 = v6 & v7;
  int v9 = v8 ? v3 : v4;
  return v9;
}
```




```
// First schedule.
int module(int a, int b,
           int c, int d) {
  if (a * a + b * b == 100) {
    int v3 = c * d;
    if (a == c)
      return v3;
  }
  return a + b;
}
```




```
// Second schedule.
int module(int a, int b,
           int c, int d) {
  if (a == c) {
    int v3 = c * d;
    if (a * a + b * b == 100)
      return v3;
  }
  return a + b;
}
```

Figure 1: Two schedules of a data-flow graph and the implied control flows when ODC is exploited.

The work in [8] introduced the concept of behavior-level observability and use behavior-level ODC to strengthen conditions for operand isolation and clock gating. However, the algorithm did not capture opportunities for control-flow restructuring (like shown in Fig. 1), and was not used to guide architectural exploration in behavioral synthesis. One common problem associated with optimization at a higher level is that an accurate estimation model is often absent. Fortunately, such problem does not exist for ODC-based power optimization. In fact, all data flows and executing conditions can be analyzed statically in a behavior description and optimized during scheduling. The behavioral synthesis tool can explicitly specify the optimized ODC condition for every register in the RTL description it generates, so that an RTL synthesis tool using low-cost ODC analysis algorithms like [2–7] can work without losing opportunities for power optimization.

Our behavioral synthesis tool first optimizes the input behavioral description using compiler transformations [9], and translates the optimized code into a control-data-flow graph (CDFG). The scheduler decides the control step at which each instruction executes. The result of scheduling can be interpreted as a finite-state machine with datapath (FSMD) [10], which is then translated to an RTL model through a binding process.

In the scheduling process of transforming a CDFG into an FSMD, behavior-level observability conditions are translated into FSMD-observability conditions (observability under a given schedule, more

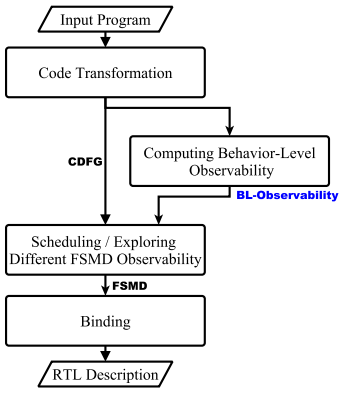


Figure 2: Overview of our behavioral synthesis system.

precisely defined in Section 3). In the example in Figure 1, a behavioral-level observability condition for v1 is $v6 \wedge v7 \wedge v8$; i.e., v1 is only observable when v6, v7 and v8 are all true. However, the evaluation of v1 can never be avoided in the first schedule, because $v6 \wedge v7 \wedge v8$ is always unknown when v1 is evaluated and conservative decisions have to be made to guarantee correctness. The second schedule is better in the sense that we could avoid evaluating v1 when v7 is known to be false — because $v6 \wedge v7 \wedge v8$ will be false then even when v6 and v8 are not yet evaluated. Here we say the FSMD-observability condition of v1 is $v7$. Clearly, different schedules imply different ODC conditions on the FSMD, and it is not always possible to postpone every instruction until its behavior-level observability condition is known, due partly to performance constraints.

The problem of minimizing the cost of executing unnecessary instructions can be described as follows: given a CDFG and profiling information, as well as the cost (average power) for executing each instruction, find a schedule so that the expected total cost is minimized, subject to data-dependency constraints and a latency constraint.

In this work, we present a systematic approach to solve the problem in a behavioral synthesis tool, as shown in Figure 1. We develop a method to identify conditionally unnecessary evaluations, and show how such information can be used to guide the scheduler to generate power-friendly RTL models. The contribution of this paper can be summarized as follows.

- We develop an algebra that generalizes observability in a Boolean network to an arbitrary function and enables observability analysis at a higher level of abstraction. Based on this algebra, we formally introduce several observability measures and explain their relations using the algebra.
- We describe an efficient algorithm to compute behavior-level observability. The algorithm is optimal for acyclic code assuming that inputs and all instructions other than and/or/not/select are viewed as black boxes. The algorithm also provides a mechanism to allow available knowledge about inputs and instructions to be considered.
- We present a behavioral synthesis flow for power optimization by minimizing the number of unnecessary evaluations, guided by behavioral-level observability, and demonstrate the effectiveness of our approach. To the best of our knowledge, this is the first time that behavioral synthesis is guided by a comprehensive observability analysis for power optimization.

2. AN ALGEBRA OF OBSERVABILITY

2.1 Observability

The observability of a function $f(x,y)$ with respect to part of its variables x is a Boolean-valued function of the rest variables y; the observability is true for values of y which makes it possible that changes in x are observable.

DEFINITION 1 (OBSERVABILITY). *For a function $z = f(x,y) : \mathbb{X} \times \mathbb{Y} \to \mathbb{Z}$, where $x \in \mathbb{X}$, $y \in \mathbb{Y}$, an* observability function *of f with respect to x is a function $\mathscr{O}_{\mathbb{X}} f : \mathbb{Y} \to \{0,1\}$, so that $\mathscr{O}_{\mathbb{X}} f(y) \Rightarrow \exists x_1, x_2 \in \mathbb{X}, f(x_1,y) \neq f(x_2,y)$.*

For example, an observability function of the program in Fig. 1 with respect to v3 is v8.

2.2 Observability with Care Set

Knowledge about possible combinations of variable values can be applied to strengthen observability conditions.

DEFINITION 2 (CARE SET). *For the function f as described in Definition 1, a* care set *about y, $\mathbb{M}$, is a subset of $\mathbb{Y}$. Let $\mathcal{F}$ be the set of functions $\mathbb{X} \times \mathbb{Y} \to \mathbb{Z} \cup \{\Phi\}$. An* apply *operator with care set $\mathbb{M}$ about y, $\mathscr{K}^{\mathbb{Y}}_{\mathbb{M}} : \mathcal{F} \to \mathcal{F}$, is defined as*

$$\mathscr{K}^{\mathbb{Y}}_{\mathbb{M}} f(x,y) = \begin{cases} f(x,y), & y \in \mathbb{M}, \\ \Phi, & \textit{otherwise.} \end{cases}$$

Informally, after applying the care set, the resulting function is independent of the input when the input is not in the care set.

THEOREM 1. *$(\mathscr{O}_{\mathbb{X}} f(y) \wedge y \in \mathbb{M})$ is an observability function of $\mathscr{K}^{\mathbb{Y}}_{\mathbb{M}} f(x,y)$ with respect to x.*

Informally, by combining the observability of f with respect to x and the assertion that $y \in \mathbb{M}$, we get the observability of $\mathscr{K}^{\mathbb{Y}}_{\mathbb{M}} f$ with respect to x. This provides a way to obtain stronger observability conditions under a given care set. For the example function in Fig. 1, with the knowledge that $v8 \Leftrightarrow v6 \wedge v7$, we can improve the observability function with respect to v3 from $v8$ to $v6 \wedge v7 \wedge v8$.

2.3 Projection of Observability

When only part of the variables are used for observability computation, we usually need to make conservative decisions about other *invisible* variables by using a necessary condition of the exact observability.

DEFINITION 3 (PROJECTION). *For a Boolean-valued function $h(y_1,y_2) : \mathbb{Y}_1 \times \mathbb{Y}_2 \to \{0,1\}$, the* projection *of h onto $\mathbb{Y}_1$ is a function $\mathscr{P}_{\mathbb{Y}_1} h : \mathbb{Y}_1 \to \{0,1\}$, so that*

$$\mathscr{P}_{\mathbb{Y}_1} h(y_1) = \begin{cases} 1 & \textit{if } \exists y_2 \in \mathbb{Y}_2, h(y_1,y_2) \\ 0 & \textit{otherwise.} \end{cases}$$

Informally, $\mathscr{P}_{\mathbb{Y}_1} h(y_1)$ is the strongest necessary condition for $h(y_1,y_2)$ with respect to y_1. Examples of the projection operation are described in Section 3.

THEOREM 2. *For $g : \mathbb{Y}_1 \times \mathbb{Y}_2 \times \mathbb{Y}_3 \to \{0,1\}$,*

$$\mathscr{P}_{\mathbb{Y}_1}(\mathscr{P}_{\mathbb{Y}_1 \times \mathbb{Y}_2} g) \Leftrightarrow \mathscr{P}_{\mathbb{Y}_1} g.$$

Proofs for most theorems shown above are omitted due to page limitation. The above algebra is a generalization of observability in Boolean networks, and it provides mathematical foundation for observability computation in arbitrary functions, such as a program used in behavioral synthesis.

3. OBSERVABILITY IN A PROGRAM

Consider a program g, let $x \in \mathbb{X}$ be the values under consideration, let V be the set of all the other values in g, among which $B \in \mathbb{B}$ is the set of Boolean values and $C = V - B \in \mathbb{C}$. Let $OUT \in \mathbb{OUT}$ be the set of output values. We view the program as a function $g : \mathbb{X} \times \mathbb{B} \times \mathbb{C} \to \mathbb{OUT}$.

DEFINITION 4 (BL-OBSERVABILITY). *For the program g as described above, a* behavioral-level observability *condition of g with respect to x is $BLO_x \equiv \mathscr{O}_{\mathbb{X}} g$.*

For behavioral synthesis, the target hardware architecture typically evaluates a Boolean function as the *predicate* for an operation [11]. So we can further distinguish between Boolean and non-Boolean values, and only use values in B to compute observability by projecting BLO_x onto $\mathbb{B}$. Conservativeness is introduced in the process because we can only obtain a necessary condition after projection.

DEFINITION 5 (B-BL-OBSERVABILITY). *A behavioral-level observability condition using only Boolean values (*B-BL-observability *condition) of g with respect to x is a function $BBLO_x : B \to \{0,1\}$, $BBLO_x \equiv \mathscr{P}_B BLO_x$.*

For a given schedule s of program g, let $A_s(x)$ be the set of values available when value x is evaluated (i.e., the set of values generated by operations scheduled to finish before the evaluation of x starts), we define FSMD-observability as follows.

DEFINITION 6 (FSMD-OBSERVABILITY). *An* FSMD-observability *condition of g with respect to x under a given schedule s, $FSMDO^s_x : A_s(x) \to \{0,1\}$, $FSMDO^s_x \equiv \mathscr{P}_{A_s(x)} BLO_x$.*

DEFINITION 7 (B-FSMD-OBSERVABILITY). *An FSMD-observability condition using only Boolean values (*B-FSMD-observability*) of g with respect to x under a given schedule s is a function $BFSMDO^s_x : A_s(x) \cap B \to \{0,1\}$, $BFSMDO^s_x \equiv \mathscr{P}_{A_s(x) \cap B} FSMDO^s_x$.*

Using Theorem 2, we have

THEOREM 3. *$BFSMDO^s_x \Leftrightarrow \mathscr{P}_{A_s(x) \cap B} BBLO_x$.*

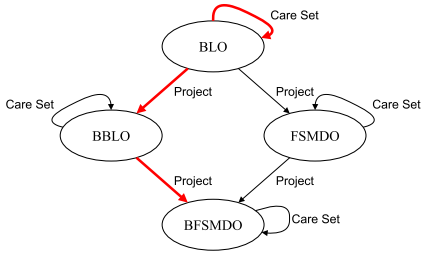


Figure 3: Relations between observabilities.

The conceptual difference between *BLO*, *BBLO*, *FSMDO* and *BFSMDO* lies in the set of values that are used to evaluate observability. All values in the program can be used for behavioral-level analysis, while only available values are meaningful when the schedule is fixed. Theoretically, both Boolean and non-Boolean values can be used, while in practice most architectures support only a Boolean expression as the predicate of an instruction.

Theorem 3 uncovers relations between *BLO*, *BBLO*, *FSMDO*, and *BFSMDO*. Accordingly, different flows for computing *BFSMDO* — the one that can be implemented in a predicated architecture — are possible.

4. OBSERVABILITY ANALYSIS

In our behavioral synthesis flow, we are most interested in *BBLO* and *BFSMDO*. Our flow is illustrated as the bold path in Figure 3: starting from an inexact *BLO*, applying care sets generated from certain instructions, projecting *BLO* to *BBLO* (trivial in our case) and further to *BFSMO* under a give schedule.

4.1 Computing BL-Observability

In general, analyzing the exact observability condition requires nontrivial computational effort. For efficiency, we can simplify the problem by only exploiting knowledge about certain important instructions while treating all other instructions as black boxes. We divide instructions into two categories. (1) *Observability-propagating instructions*: For such an instruction I, we always have $BLO_I \Rightarrow BLO_{src_i(I)}, \forall i$, where $src_i(I)$ refers to the ith operand of I. Typical observability-propagating instructions include xor, add, etc. (2) *Observability-masking instructions*: For such an instruction I, it is possible to have one or more of its inputs to be unobservable while its output is observable. Here we consider the following observability-masking instructions. A select instruction takes three operands, where the first one is a Boolean value used to decide whether the second or the third operand is used as the result.

$$BLO_I \Rightarrow BLO_{src_1(I)}$$
$$BLO_I \wedge src_1(I) \Rightarrow BLO_{src_2(I)}$$
$$BLO_I \wedge \overline{src_1(I)} \Rightarrow BLO_{src_3(I)}$$

For a Boolean and instruction I,

$$BLO_I \wedge src_2(I) \Rightarrow BLO_{src_1(I)}$$
$$BLO_I \wedge src_1(I) \Rightarrow BLO_{src_2(I)}$$

The situation with a Boolean or instruction is similar to that of a Boolean and instruction. Other instructions not discussed above, including bitwise and/or for non-Boolean values, are regarded as observability-propagating instructions. This leads to non-minimum *BLO*, but it allows analysis at a higher level of abstraction efficiently.

For a data-flow graph without loops, the code region can be viewed as a combinational circuit. The *BLO* can be analyzed using a backward propagation process like that used in combinational Boolean networks. For instructions generating outputs, such as return and store, the *BLO* conditions are set to their executing conditions (predicates); for all other instructions, the *BLO* conditions are set to false initially. Instructions are then examined in reverse topological order of the data-flow graph, and the *BLO* condition of each instruction is propagated to its source operands, using rules derived above. The algorithm is shown in Algorithm 1.

Table 1: BL-observability by Algorithm 1.

value	BLO	value	BLO
a	$v6 \vee v7 \vee \overline{v8}$	b	$v7 \vee \overline{v8}$
c	$v6 \vee v8$	d	$v8$
v1	$v7$	v2	$v7$
v3	$v8$	v4	$\overline{v8}$
v5	$v7$	v6	$v7$
v7	$v6$	v8	$true$
v9	$true$		

By applying this algorithm, we can get the *BLO* for instructions in the example code as shown in Table 1. However, the resulting BL-observability condition can be further strengthened by exploiting care sets. Our method exploits care sets from Boolean instructions and/or/not. For the example code, we have $v8 \Leftrightarrow v6 \wedge v7$, or equivalently, $v6v7v8 \vee (\overline{v6} \vee \overline{v7})\overline{v8}$. Using Theorem 1, $BLO_x \wedge (v6v7v8 \vee (\overline{v6} \vee \overline{v7})\overline{v8})$ is a stronger BL-observability with respect to x. The results after applying the care set about all Boolean instructions are illustrated in Table 2.

Algorithm 1 *BLO* computation for a data-flow graph.

for all instruction I **do**
  if I directly generates output **then**
    $BLO_I = predicate(I)$
  else
    $BLO_I = false$
  end if
end for
for all instruction I in reverse topological order **do**
  if I is select **then**
    $BLO_{src_1(I)} = BLO_{src_1(I)} \vee BLO_I$
    $BLO_{src_2(I)} = BLO_{src_2(I)} \vee BLO_I \wedge src_1(I)$
    $BLO_{src_3(I)} = BLO_{src_2(I)} \vee BLO_I \wedge \overline{src1(I)}$
  else if I is Boolean and **then**
    $BLO_{src_1(I)} = BLO_{src_1(I)} \vee BLO_I \wedge src_2(I)$
    $BLO_{src_2(I)} = BLO_{src_2(I)} \vee BLO_I \wedge src_1(I)$
  else if I is Boolean or **then**
    $BLO_{src_1(I)} = BLO_{src_1(I)} \vee BLO_I \wedge \overline{src_2(I)}$
    $BLO_{src_2(I)} = BLO_{src_2(I)} \vee BLO_I \wedge \overline{src_1(I)}$
  else
    for all source operand of I, $src_j(I)$ **do**
      $BLO_{src_j(I)} = BLO_{src_j(I)} \vee BLO_I$
    end for
  end if
end for

Table 2: BL-observability after applying care sets.

value	BLO	value	BLO
a	$v6v7v8 \vee (\overline{v6} \vee \overline{v7})\overline{v8}$	b	$v6v7v8 + (\overline{v6} \vee \overline{v7})\overline{v8}$
c	$v6v7v8 \vee v6\overline{v7v8}$	d	$v6v7v8$
v1	$v7(\overline{v6}\,\overline{v8} \vee v6v8)$	v2	$v7(\overline{v6}\,\overline{v8} \vee v6v8)$
v3	$v6v7v8$	v4	$(\overline{v6} \vee \overline{v7})\overline{v8}$
v5	$v7(\overline{v6}\,\overline{v8} \vee v6v8)$	v6	$v7(\overline{v6}\,\overline{v8} \vee v6v8)$
v7	$v6(\overline{v7}\,\overline{v8} \vee v7v8)$	v8	$v6v7v8 \vee (\overline{v6} \vee \overline{v7})\overline{v8}$
v9	$v6v7v8 \vee (\overline{v6} \vee \overline{v7})\overline{v8}$		

Note that the BL-observability computed using Algorithm 1, and the knowledge exploited are both about Boolean values in the program. It is obvious that Algorithm 1 is the best BL-observability using black-box models for observability-propagating instructions without exploiting knowledge about correlations between Boolean variables. Since all of the observability-masking instructions we consider only use Boolean values for *BLO*, the expression does not actually change after projection to *BBLO*. Under the black-box model, correlations between Boolean values are caused by Boolean operators and/or/not, and all such correlations can be captured by visiting each Boolean instruction in the program, so that no smaller care set is possible without exploiting other instructions. Thus, the resulting *BBLO* is optimal assuming the black-box model.

While we consider the black-box model necessary to enable analysis at a higher level, certain knowledge about instructions, once uncovered, can be employed to strengthen BL-observability and its projections. Our algorithm allows the use of care sets implied by Boolean relations to strengthen the observability condition. For example, `(x == 3)` implies `(x < 10)`. Although capturing exact relations between Boolean values is nontrivial, at least some knowledge can be discovered and exploited. Such techniques have been developed in predicated compilation [12], and can be directly applied in our algorithm.

4.2 Computing Observability on FSMD

According to Theorem 3, under a fixed schedule, *BFSMDO* can be computed from *BBLO* using projection. Obviously, for an in-

struction I, $BFSMDO_I$ can be used to strengthen its predicate to avoid unobservable computation in hyperblocks [13]. It might be tempting to use the strongest $BFSMDO_I$ in the predicate. However, this can lead to an unnecessary increase of predicate complexity, because $BFSMDO_I$ itself may contain redundancy. For example, $BFSMDO_{v9} = v6v7v8 \vee (\overline{v6} \vee \overline{v7})\overline{v8}$, which is always true for the program. In this case, logic simplification should be performed subject to known assertions, using techniques in logic synthesis [1].

Although knowledge introduced earlier might be simplified out later, some knowledge is useful to attain a tighter observability condition after projection. Consider the example data-flow graph in Figure 1, after applying the care set, $BBLO_{v3} = v6v7v8$. For the second schedule, we can project $BBLO_{v3} = v6v7v8$ on $\{v7\}$ and get $BFSMDO_{v3} = v7$. If the naive algorithm is used, we get $BBLO_{v3} = v8$, and $BFSMDO_{v3} = true$ after projection, and thus lose an opportunity for avoiding the evaluation of v3.

Up to now, we have described a systematic way to obtain $BFSMDO$. However, the manipulation of logic expressions, the simplification step in particular, is nontrivial. The BDD-based method can be very efficient using smart strategies, but the exact method is still not scalable in general. Thus, we try to manage the complexity by changing the way care set is applied. We only use the backward propagation algorithm to compute the naive BL-observability. If a Boolean value (or its complement) in the BL-observability of an instruction is evaluated later than the instruction, we try to replace the Boolean value (or its complement) with its necessary condition involving Boolean values evaluated earlier. Obviously, this does not increase the chance that an instruction is executed unnecessarily under a given schedule. For the example design in Figure 1, we have $v8 \rightarrow v6$, $v8 \rightarrow v7$. When $BBLO_{v3} = v8$ is projected onto $\{v7\}$, it is replaced with $v6v7$, leading to $BFSMDO_{v3} = v7$ after projection.

5. ODC OPTIMIZATION IN SCHEDULING

Considering the fact that only Boolean values already evaluated can be used in B-FSMD-observability, the impact of scheduling on ODC-based power management is obvious. To our knowledge, the work in [14] presents the first algorithm designed to create more opportunities for ODC-based power management. This method works as a post-processing step on an existing schedule: it examines multiplexors (select instructions) one by one and tries to move the instruction by computing the Boolean operand earlier if possible. Authors of [14] noticed that their results depended on the order in which multiplexors were examined, and used reverse topological order in their implementation. [15] proposes an improved optimization technique using priority and soft dependency edges. Both [14] and [15] use a very simple method for observability analysis, and the method can only identify a small amount of behavior-level ODC conditions for two reasons. First, the method focuses exclusively on select instructions (multiplexors), without considering other observability-masking instructions such as and/or. Although the method can possibly be extended by viewing and/or as degenerated select, it still cannot capture the information that either operand can mask the observability of the other. Such information is essential for control-flow restructuring when the scheduler exploits different possible speculation/predication schemes under a latency constraint. Second, the analysis method used in [14, 15] does not take advantage of relations between Boolean values, while our method considers opportunities for avoiding the execution of an instruction even when the Boolean value that directly controls observability is not evaluated (Figure 1).

Information about B-BL-observability can be used to guide optimization techniques for scheduling developed in [14, 15]. In our implementation, we use the method in [15], with improved priority computation that considers more opportunities for ODC-based power optimizations. The algorithm can be briefly described as follows: *soft dependency edges* are added between a Boolean-valued instruction B (like cmp/and/or/not) and a normal instruction I whose BL-observability condition contains the Boolean value generated by B. The power saving with each soft dependency edge is estimated using profiling information and the power consumption of one instruction of each type. The algorithm adds soft dependency edges in descending order of estimated power saving under a latency constraint, and finishes when no additional soft dependency edges are feasible.

6. EXPERIMENTAL RESULTS

We have implemented the proposed method in the AutoPilot behavioral synthesis system [16]. The system accepts C/C++ as input description, transforms the code using the LLVM compiler frontend [17], and performs scheduling and binding before generating RTL model in VHDL or Verilog. The generated RTL code is fed to the Magma Talus RTL-to-GDSII toolset [18] to generate gate-level netlist. Gate-level simulation is performed using the Aldec Riviera simulator [19] to obtain power dissipation. All designs are implemented using a TSMC 90nm library in the experiments. Power management is implemented using automatic clock gating by Talus. Further power savings can be potentially achieved with additional low-power techniques (e.g., input isolation, feeding sleep vectors for leakage reduction).

We evaluate our approach on a set of arithmetic and multimedia designs, as described in Table 3. We compare the results obtained

Table 3: Benchmark characteristics.

Name	Description
cordic	efficient trigonometric function evaluation
dfmul	double-precision floating-point multiplier
addr	address translation for MPEG4 decoder
edgeloop	kernel part of a H.264 deblocker
boxmuller	random number generation

in three modes: (1) baseline: traditional scheduling without considering observability; (2) the method described in [15]; (3) the proposed method. Results on estimated area, timing and power after logic synthesis are reported in Table 4.

From the results, we can see that using observability in the scheduler helps to reduce power by an average of 26.2% compared to a traditional scheduler, and by 17.7% compared to the same scheduler with a simpler observability analysis used in [14, 15]. For a design with very simple control flow (like cordic, which does not contain nested if-then-else), our approach and [15] lead to the same result. For designs with more complex control flow, our approach can be significantly better. There is a trend for a slight area increase with our approach as well as [15]; this is because more logic conditions need to be evaluated in order to decide the observability of operations.

7. RELATED WORK

One might think that after partial dead code elimination in the predicated form [20, 21], the predicate of every instruction is equal to its BL-observability condition because no redundant instruction will be executed along every control path. However, this is not true. Consider two Boolean values used in an and instruction, the *BLO* condition for either instruction could contain a term about the other. If *BLO* conditions are applied as predicates, there will be a cyclic dependency between the two instructions, and the code becomes illegal. Thus, from the perspective of BL-observability, one can always find instructions that are unnecessarily executed (unobservable), even after thorough optimization. Since execution of unnecessary instructions cannot be avoided completely, optimizers should try to minimize the cost of unnecessary execution.

Table 4: Experimental results.

name	baseline			[15]				this work			
	area	period	power	area	period	power	%	area	period	power	%
cordic	3001	2.72	0.202	3104	2.73	0.185	91.6%	3104	2.73	0.185	91.6%
dfmul	121546	3.25	3.11	122430	3.31	2.77	89.1%	124001	3.22	1.89	60.8%
addr	4892	2.35	0.353	5020	2.33	0.291	82.4%	5100	2.36	0.246	69.7%
edgeloop	43046	1.99	4.25	46320	2.01	3.74	88.0%	46125	1.98	3.08	72.5%
boxmuller	72127	3.50	2.83	72165	3.51	2.60	91.9%	72258	3.49	2.20	77.7%

Area is in μm^2, period is in ns, and power is in mW. The % column is the comparison of power to baseline.

Notably, using BL-observability to guide scheduling gives us opportunities to unify speculative scheduling and control flow restructuring, as shown in Figure 1. Previous efforts using predicates in hyperblock scheduling also allow speculative scheduling through predicate promotion [13], and have the ability to simplify program decision logic using knowledges about relation between predicates [12], but a post-processing pass like predicated partial dead code elimination is needed to strengthen some predicates after scheduling to fully realize the equivalent transformation. BL-observability could provide more information to the scheduler than predicate, and can be helpful when various tradeoffs are performed by the scheduler under tight constraints like worst-case latency.

Let us, again, consider the example code in Figure 1. In practice, a deliberate designer may optimize the design to one in Figure 4, where a redundant Boolean value is introduced in the hope that it can be used in observability computation for further avoiding multiplications. A technique to add such Boolean guards has been developed in [22] for embedded compilers. We believe that this technique could increase the effectiveness of our approach.

```
int module(int a, int b, int c, int d) {
  if (a == c) {
    if (((a ^ b) & 0xFFFFFFFB) == 0xA)
      if (a * a + b * b == 100)
        return c * d;
  }
  return a + b;
}
```

Figure 4: An even better implementation.

8. CONCLUSION

In this paper we have proposed an algebra about observability, based on which observability is computed and optimized in a systematic way in behavioral synthesis for low power design. Experimental results show that our approach is effective. The analysis using our theory also reveals opportunities in compiler optimization for ODC optimization; we leave it for future work.

Acknowledgments

This work is partially supported by SRC under Task 1879 and NSF under grant CNS-0725354.

9. REFERENCES

[1] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[2] M. Münch, B. Wurth, R. Mehra, J. Sproch and N. Wehn, "Automating RT-level operand isolation to minimize power consumption in datapaths," in *Proc. DATE*, 2000, pp.624–633.

[3] R. Fraer, G. Kamhi and M. K. Mhameed, "A new paradigm for synthesis and propagation of clock gating conditions," in *Design Automation Conf.*, 2008, pp. 658–663.

[4] M. Onishi, A. Yamada, H. Noda and T. Kambe, "A method of redundant clocking detection and power reduction at RT level design," in *Proc. Int. Symp. Low Power Electron. Design*, 1997, pp. 131–136.

[5] L. Benini and G. De Micheli, "Automatic synthesis of low power gated-clock finite-state machines," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 15, pp. 630–643, June 1996.

[6] L. Benini, G. De Micheli, E. Macii, M. Poncino and R. Scarsi, "Symbolic synthesis of clock-gating logic for power optimization of synchronous controllers," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 4, no. 4, pp. 351–375, 1999.

[7] P. Babighian, L. Benini and E. Macii, "A scalable algorithm for RTL insertion of gated clocks based on ODCs computation," *IEEE Trans. on Comput.-Aided Des Integr. Circuits Syst.*, vol. 24, pp. 29–42, Jan. 2005.

[8] Q. Wang and S. Roy, "RTL power optimization with gate-level accuracy," in *Proc. Int. Conf. Computer-Aided Design*, 2003, pp. 39–45.

[9] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann. 1997.

[10] D. Gajski, N. Dutt, A. Wu and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[11] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist and M. Sivaraman, "PICO-NPA: high-level synthesis of nonprogrammable hardware accelerators," *J. VLSI Signal Process. Syst.*, vol. 3, no. 2, pp. 127–142, 2002.

[12] D. I. August, J. W. Sias, J.-M. Puiatti, S. A. Mahlke, D. A. Connors, K. M. Crozier and W. W. Hwu, "The program decision logic approach to predicated execution," in *Proc. the 26th Annual Int. Symp. Computer Architecture*, 1999, pp. 208–219.

[13] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proc. the 25th Int. Symp. Microarchitecture*, 1992, pp. 45–54.

[14] J. Monteiro, S. Devadas, P. Ashar and A. Mauskar, "Scheduling techniques to enable power management," in *Proc. Design Automation Conf.*, 1996, pp. 349–352.

[15] C. Chen and M. Sarrafzadeh, "Power-manageable scheduling technique for control dominated high-level synthesis," in *Proc. Design Automation & Test in Europe*, 2002, pp. 1016–1020.

[16] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang and J. Cong, "AutoPilot: a platform-based ESL synthesis system," in *High-Level Synthesis: From Algorithm to Digital Circuit*, ed. P. Coussy and A. Morawiec, Springer Publishers, 2008.

[17] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation and Optimization*, 2004, pp. 75–86.

[18] `http://www.magma-da.com`.

[19] `http://www.aldec.com`.

[20] R. Bodík and R. Gupta, "Partial dead code elimination using slicing transformations," in *Proc. ACM SIGPLAN 1997 Conf. Prog. Lang. Design Impl.*, pp. 159–170.

[21] S. Ryoo, S. Ueng, W. W. Hwu, "P3DE: profiled-directed predicated partial dead code elimination," Presented at 5th workshop on EPIC Architecture and Compiler technology, March, 2006.

[22] M. A. Ghodrat, T. Givargis and A. Nicolau, "Short-circuit compiler transformation: optimizing conditional blocks," in *Proc. ASPDAC*, 2007, pp. 504–510.