10.24425/acs.2021.137426

*Archives of Control Sciences*
*Volume* 31(LXVII), 2021
*No.* 2*, pages 331–346*

# Power-aware scheduling of data-flow hardware circuits with symbolic control

Mete ÖZBALTAN and Nicolas BERTHIER

We devise a tool-supported framework for achieving power-efficiency of data-flow hardware circuits. Our approach relies on formal control techniques, where the goal is to compute a strategy that can be used to drive a given model so that it satisfies a set of control objectives. More specifically, we give an algorithm that derives abstract behavioral models directly in a symbolic form from original designs described at Register-transfer Level using a Hardware Description Language, and for formulating suitable scheduling constraints and power-efficiency objectives. We show how a resulting strategy can be translated into a piece of synchronous circuit that, when paired with the original design, ensures the aforementioned objectives. We illustrate and validate our approach experimentally using various hardware designs and objectives.

**Key words:** symbolic discrete controller synthesis, digital synchronous circuits, power-efficiency

## 1. Introduction

High-level models are often required to reason on synchronous circuits designs, and apply scalable techniques to translate them into new designs that meet various performance goals such as power-efficiency [9]. Among these models, the data-flow family see circuits as actors that communicate through communication channels, and *Kahn Process Networks* (KPN) [8] are a sub-class of such models where each actor (*aka process*) feeds from one or more queue of jobs stored in bounded channels (*FIFOs*). KPNs can be used to describe systems where the amount of data produced and consumed by an process is not statically determined.

We consider designs that implement KPNs, and where each process implementation is described at Register-Transfer Level in a Hardware Description

Copyright © 2021. The Author(s). This is an open-access article distributed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives License (CC BY-NC-ND 4.0 https://creativecommons.org/licenses/by-nc-nd/4.0/), which permits use, distribution, and reproduction in any medium, provided that the article is properly cited, the use is non-commercial, and no modifications or adaptations are made

M. Özbaltan (corresponding author, e-mail: mozbaltan@hotmail.com) is with Erzurum Technical University, Erzurum, Turkey.

N. Berthier is with University of Liverpool, Liverpool, England.

Received 02.02.2020. Revised 27.04.2021.

Language (HDL) such as Verilog. We advance a (mostly-)automated procedure that translates such designs into functionally equivalent ones that in addition enjoy *power-awareness* guarantees, in a bid to reduce their *dynamic power dissipation*. To this end, we compute a *strategy* that implements a *power-aware scheduling policy* by selectively clock-gating [4] each process. Notice we do not seek to reduce the total *energy* consumed (*i.e.*, power integrated over the total computation time). Rather, we seek to reduce the *instantaneous power* (possibly integrated over a small time window), as this kind of power-efficiency policy usually has a positive impact on the lifetime of battery-powered devices [16], and also provides a means to limit chip temperature. In effect, we allow ourselves to degrade timing performance to achieve such goals.

Our approach relies on the construction of *abstract symbolic models* of the designs, and employs *discrete control* techniques to compute a piece of hardware circuit that implements some power-aware scheduling policies specified *in a declarative way*. This piece of circuit can eventually be used to selectively filter the clocks of the processes involved.

**Outline.** The remainder of the paper is organized as follows: Section 2 gives necessary background on symbolic models and control. Section 3 presents our approach, which is experimentally evaluated in Section 4. At last, Sections 5 and 6 review related works and conclude.

## 2. Background on symbolic models and discrete control

Our symbolic models are built upon a finite set of *symbols*, each associated with *finite domains* such as Booleans {true, false}, (signed) fixed-width Integers *Symbolic expressions* are constructed using classical operators from propositional logic ($\neg, \wedge, \vee, \Rightarrow$), linear arithmetic with (in)equalities, and logical operators for fixed-width Integers, possibly guarded with the conditional construct "if *pred* then $expr_1$ else $expr_2$ ". All expressions must be well typed; in particular the two rightmost expressions in conditional constructs must have the same type.

Models are made of *state* and *input* symbols. The values associated to state symbols are initialized with constants, and evolve according to a discrete step (*lock-step*) semantics very similar to that of synchronous circuits: *discrete evolutions* are defined using one assignment $s := e_s$ per state symbol $s$, where $e_s$ is a (well-typed) symbolic expression that determines the valuation memorized in $s$ based on the current valuations for state and input symbols, at each *tick* of an *implicit basic clock*[1].

[1]Throughout the paper, and unlike :=, $s \triangleq e$ denotes the classical formal definition of a left-hand side symbol $s$ with a right-hand side expression $e$.

Solving *symbolic control problems* on such models can be seen as solving a game where, at each tick, one player (the environment) gives a value for a fixed portion of the input symbols, *then* the other player (the *controller*) assigns values to every other input symbol, *and then* the game evolves into a subsequent state according to the discrete evolutions and the inputs given by the players. The *control objectives* assigned to the second player are expressed as logic formulas that involve state and/or input symbols, and the solution of the control problem consists in a *strategy* that this player can follow to win the game by fulfilling all its objectives. The input symbols assigned by the first (resp. second) player are said *non-controllable* (resp. *controllable*).

The control objectives that we use in this work are twofold: First, satisfying *safety control objectives* consists in enforcing a *safety property*. Such properties can be expressed using some temporal logic like LTL [6]. In our case however, we use the same symbolic constructs as for the model to build *stateful observers* that represent the temporal aspects of the properties we need (*e.g.,* sequence, iteration), and can therefore restrict the safety objective formulas to propositional logic. Second, satisfying *optimization control objectives* consists in minimizing a *cost function* (summed) over a sliding window of a given number of ticks. A cost function is typically a total mapping from state and input valuations into some (partially-)ordered set such as the Rationals.

Observe that there does not always exist a strategy that fulfills the desired safety control objectives, and in this case safety control algorithms terminate but produce no output. We shall see that in this work, the absence of a strategy specifically reveals unrealizable objectives regarding the limitation of dynamic power consumption (*w.r.t.* modeling abstractions).

**Tooling.** Few works have addressed the problems of enforcing safety control and optimization objectives on the kind of symbolic models we construct; they mostly derive from the seminal work of [18]. [11] and [12] implemented tools that are suitable for enforcing safety objectives. In turn [10] and [7] implemented solutions for optimization objectives, that essentially consist in symbolic adaptations of Bellman's algorithm for the computation of optimal strategies using dynamic programming [2].

**Strategies as hardware circuits.** Strategies that are computed by algorithms dedicated to operate on symbolic models usually take the form of a *predicate* on state and input symbols. Then, given a valuation for state and non-controllable inputs, a constraint solver needs to be used to find a suitable valuation for controllable inputs that satisfies the strategy. The existence of such a solution is guaranteed by the control algorithm. When this solution is always unique, moreover, the strategy can be translated into a mapping from valuations for state and non-controllable input symbols into valuations for controllable input symbols, which is basically a combinatorial circuit. Strategies suitable for this translation can be obtained by

refinement with the help of a total order on solutions (*e.g.,* with a total order on both the controllable input symbols and their respective domains of definition).

## 3. Models and objectives for power-aware scheduling

### 3.1. Overview of the approach and contributions

We describe in Fig. 1 the work-flows offered by our approach: our goal is to automatically construct the *Clock-gating Logic* (CGL) that implements a power-aware scheduler for all processes involved. A *Design Model* $M$ is first built from the original design. $M$ is made of the synchronous parallel composition of, for each process in the design: (i) a *process model*, which is a symbolic behavioral abstraction of the process, constructed from the HDL description of its implementation; (ii) an *idleness predicate* $idle_{\mathrm{p}}$ that must not hold if the value of any register within pis not strictly equivalent before and after the edge of the clock (*i.e.*, it is an under-approximation); and (iii) a symbolic *power expression* $\mathrm{P}_{\mathrm{p}}$ that gives an estimated measure of the instantaneous dynamic power consumption of the modeled process based on the state and input symbols of p. Each process pis associated with a *clock-inhibition signal*, $inhibit_{\mathrm{p}}$ , which is a *controllable input symbol* that shall hold when the computations of pcan be suspended, *i.e.*, its clock can be inhibited.

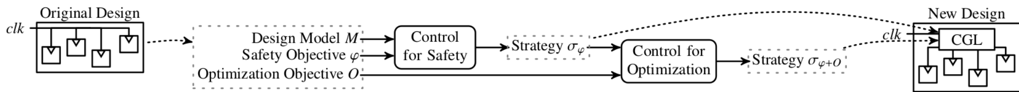

Figure 1: Overview of the possible work-flows for computing the power-aware CGL

Then, a *safety objective* $\varphi$ is built, in the form of a conjunction of propositional formulas involving state symbols of both $M$ plus, possibly, supporting stateful observers specified using additional state symbols. These objectives fall into three categories: *clock-gating constraints* relate each clock-inhibition signal with the model of its process within $M$; *scheduling constraints* restrict the set of eligible control strategies to those that ensure *progress*, *fairness*, and *absence of starvation*; lastly, *peak power constraints* can be used to specify an upper-bound on the sum of all power expressions upon any tick.

At this stage, a symbolic safety control algorithm is used to compute a strategy $\sigma_\varphi$, which is guaranteed to select values for the controllable inputs that ensure that the safety objective $\varphi$ are fulfilled. Since clock-inhibition signals belong to the set of controllable inputs, $\sigma_\varphi$ and $M$ can be used in combination to form a piece of circuit that filters the individual clock signals for the respective processes. Alternatively, the strategy $\sigma_\varphi$ can be improved by using our new symbolic control

algorithm that ensures an additional optimization objective $O$. The cost function that we use to define $O$ basically consists of the sum of power expressions $\mathrm{P_p}$ for all processes. The resulting refined strategy $\sigma_{\varphi+O}$ can be used in the same way as $\sigma_\varphi$ to filter the clocks of each process.

**Other contributions.** We have implemented a set of open-source tools that helps putting our approach into practice[2]. These tools allow designers to: (1) construct the model $M$ and the associated safety and optimization objectives from an HDL description of the original design; (2) compute suitable strategies, and (3) translate them into an HDL description of the CGL (the sought after strategies for clock-inhibition are given in a symbolic form, so their translation into a description of synchronous circuit in an HDL is essentially syntactical); (4) construct a new design that integrates this CGL. Note that only step (3.1) above requires some insight from designers about the design, and all the other stages are fully automated.

We further detail the process abstraction procedure and associated definition of control objectives to obtain a power-aware CGL in the remainder of this section.

### 3.2. Abstracting process implementation behaviors

Our translation and modeling algorithm takes as inputs the set Pof all processes in the original design, and produces a model $M$ along with objectives suitable for computing an implementable CGL by means of symbolic control. The computations within each individual process is described using a module in an HDL, that accepts a dedicated @clk@ signal used to drive the updates of its registers.

#### 3.2.1. Selected HDL variables

Our abstraction algorithm is parametrized with a set of selected HDL variables that make up the portion of state and input spaces that is *precisely represented* in the constructed model of each process. This key aspect allows designers to exploit the knowledge they have on their designs, in particular the usual distinction between command parts and operational parts. Every wire and register that is not explicitly selected is abstracted away and replaced by *oracles*:

#### 3.2.2. Oracle symbols

Indeed, while translating HDL expressions into their symbolic counterpart, our algorithm abstracts away sub-expressions by creating a set of *oracles* to replace them in the models. From the point of view of the constructed models, oracles are *non-controllable input symbols*, which means that the sought after strategy must be computed by assuming they can take any value at any tick.

[2]Each available at https://github.com/mozbaltan/dcs4cgl, https://scm.gforge.inria.fr/anonscm/git/reatk/reatk.git/, and https://gforge.inria.fr/anonscm/git/reatk/ctrl2hdl.git.

From the point of view of the resulting CGL, however, the actual values of the expressions abstracted away with oracles need to be known so as to evaluate the strategy it encodes. To this end, we also produce an "open" HDL implementation of every process, that features additional output wires carrying the values of the oracles (*i.e.*, the value of the expressions they represent). These additional wires are then used to feed the CGL when building up the new design.

Given an HDL expression $e$ on any set of variables, the oracle symbol $\omega_e$ is an unknown input whose value is that of $e$ at every tick. $w_e$ can thus be used to model behaviors where $e$ itself is abstracted away and can take any value in its domain. Every knowledge about the modeled behaviors is not lost however. Indeed, assuming that $e$ and $e'$ admit the same canonical representation $e''$, every occurrence of $e$ and $e'$ can be replaced with the same oracle $\omega_{e''}$, and the equality of valuations for $e$ and $e'$ can still be represented. Then, every expression $e$ that involves a non-selected HDL variable is first translated into a canonical form $e'$, and replaced with $\omega_{e'}$. An efficient way to compute canonical representations consists in using (multi-terminal) binary decision diagrams [5].

#### 3.2.3. HDL traversal procedure

```
module p (input clk, input start, input [31:0] i,
          output reg done, output reg [31:0] o);
   reg r1, r2; reg [31:0] r3;
   initial begin r1=0; r2=0 r3=0; o=0; done=0; end
   always @(posedge clk) begin
      if (start & ~done) r1<=1;
      else if (r1 & ~done) begin
         if (r2 & r3==i) begin o<=r3+i; done<=1; r1<=0; r2<=0; end
         else if (r2)    begin o<=i;    done<=1; r1<=0; r2<=0; end
         else begin r2<=1; r3<=i; end
      end else if (done) done<=0;
   end
endmodule
```

Listing 1: A simple process in Verilog

Our process behavior abstraction algorithm operates on a representation of the module implementation of each process pwhere occurrences of local wires have all been substituted with their respective expression. The algorithm first associates a guard with every clock-triggered assignment to selected registers by traversing every conditional/case constructs of the implementation. Then, for each selected register, it generates a series of cascading conditional constructs, whose leave expressions and predicate conditions are respectively built from values and guards by substituting any expression from the module implementation that involves non-selected HDL variables with oracles. In turn, the *idleness predicate*

$idle_p$ corresponds to the conjunction of the negation of all guards from the above mapping.

A similar process as for selected registers is used to construct the *power expression* $P_p$ , where leaves of cascading conditional constructs denote the amount of potential register bit-flips (*i.e.*, the sum of the width of assigned registers) instead of assigned values or symbols. We claim that this measure gives us cost functions that are suitable for demonstrating the effectiveness of our approach.

$$
\begin{aligned}
done \;:=\;& \text{if } inhibit_p \text{ then } done \text{ else}\\
& \text{if } \neg(start \wedge \neg done) \wedge (r1 \wedge \neg done) \wedge \omega_{r2} \wedge \omega_{i\text{-}r3=0} \text{ then } 1 \text{ else}\\
& \text{if } \neg(start \wedge \neg done) \wedge (r1 \wedge \neg done) \wedge \omega_{r2} \wedge \neg\omega_{i\text{-}r3=0} \text{ then } 1 \text{ else}\\
& \text{if } \neg(start \wedge \neg done) \wedge \neg(r1 \wedge \neg done) \wedge done \text{ then } 0 \text{ else } done\\
idle_p \;\triangleq\;& \neg(start \wedge \neg done) \wedge \neg(r1 \wedge \neg done) \wedge \neg done\\
P_p \;\triangleq\;& \text{if } inhibit_p \text{ then } 0 \text{ else}\\
& \text{if } (start \wedge \neg done) \text{ then } 1 \text{ else}\\
& \text{if } \neg(start \wedge \neg done) \wedge (r1 \wedge \neg done) \wedge \omega_{r2} \wedge \omega_{i\text{-}r3=0} \text{ then } 35 \text{ else}\\
& \text{if } \neg(start \wedge \neg done) \wedge (r1 \wedge \neg done) \wedge \omega_{r2} \wedge \neg\omega_{i\text{-}r3=0} \text{ then } 35 \text{ else}\\
& \text{if } \neg(start \wedge \neg done) \wedge (r1 \wedge \neg done) \wedge \neg\omega_{r2} \text{ then } 33 \text{ else}\\
& \text{if } \neg(start \wedge \neg done) \wedge \neg(r1 \wedge \neg done) \wedge done \text{ then } 1 \text{ else } 0
\end{aligned}
$$

Figure 2: Extracts of symbolic model built from the example process of Listing 1

Let us exemplify our translation procedure by examining the result we obtain from the toy process of Listing 1. Assuming @start@, @r1@, and @done@ constitute the set of selected HDL variables (as they all carry control-flow within @p@), we give in Fig. 2 the discrete evolution that corresponds to @done@, the idleness predicate, and the power expression (the result for @r1@ is similar to that of @done@). State symbol *done* is the counterpart of @done@ in the model. Notice it does not evolve when the computations of pare inhibited. Also, observe that there is a one-to-one correspondence between every portion of guards of conditional constructs and conditions in the HDL code. Further, $P_p$ states that, *e.g.*, 33 bits may flip whenever the HDL expression

```
~(start & ~done) & (r1 & ~done) & ~r2
```

holds, which manifests in the model as a guard involving the oracle $\omega_{r2}$ since @r2@ is abstracted away (@r2@ is not selected).

### 3.3. Abstract process observers and control means

We now describe in this section the additional parts of the constructed model, *i.e.*, observers and control objectives, that allow us to compute a strategy suitable to obtain a power-aware CGL. All these parts are automatically derived from the

original design. For each process p, our construction assumes the availability of the following additional non-controllable input symbols: (i) one *FIFO emptiness* symbol $empty_\mathrm{p}$ , that holds whenever all FIFOs pfeeds from are empty; and (ii) one *termination* symbol $done_\mathrm{p}$ , that holds for one tick whenever pterminates a job. A “slow” global clock *slow-clk* non-controllable input symbol may be used to enforce a preemptive model of concurrency among all processes.

#### 3.3.1. Abstract process observer

The operational status of each process is modeled using a symbolic encoding of a two-state Mealy machine that transitions whenever its input holds. For a process p, this symbolic model is defined as

$$suspended_\mathrm{p} := \text{if } suspended_\mathrm{p} \text{ then } \neg c_\mathrm{p} \text{ else } c_\mathrm{p} \ . \tag{1}$$

The $suspended_\mathrm{p}$ symbol is a Boolean component of the *state space* that holds whenever the operations of pare suspended, whereas the symbol $c_\mathrm{p}$ denotes the Boolean input that drives $suspended_\mathrm{p}$ . $c_\mathrm{p}$ is *controllable*, meaning that it serves as a leaver for the control strategy to suspend or activate pso as to fulfill its objectives. For readability, we additionally define $activate_\mathrm{p} \triangleq suspended_\mathrm{p} \wedge c_\mathrm{p}$ as a symbol that holds iff the process resumes its computations; we also define $activate \triangleq \bigvee_{\mathrm{p} \in M} activate_\mathrm{p}$ that holds iff at least one process of the design is being activated at the current tick.

Observe that the design objectives that we are aiming for (*e.g.,* power-optimization), can straightforwardly be fulfilled by preventing every process involved from computing at all. A very simple strategy that induces such a behavior consists in ensuring that every $suspended_\mathrm{p}$ state variable holds at any tick; one can observe in Eq. (1) that such a strategy always exists. To restrict the set of eligible strategies to those that ensure *progress* and *fairness*, we devise a set of additional *safety objectives* that the target design must satisfy.

#### 3.3.2. Enforcing strict progress

The most basic progress objective states that at least one process must be active at every clock tick unless all FIFOs are empty. It is ensured by means of the predicate $\varphi_{\text{strict-progress}} \triangleq \bigvee_{\mathrm{p} \in \mathrm{P}} \neg suspended_\mathrm{p} \vee \bigwedge_{\mathrm{p} \in \mathrm{P}} empty_\mathrm{p}$ .

#### 3.3.3. Enforcing fairness

In order to express the fairness objective, we symbolically encode *scheduling constraints* as part of safety objectives. To this end, we first augment the set of state components of the model by introducing one *bounded inactivity counter* $q_\mathrm{p}$ per process p: $q_\mathrm{p}$ is reset whenever pis activated, and increases if any process

except p is activated:

$$q_{\mathrm{p}} := \begin{cases} 0 & \text{if } \mathit{activate}_{\mathrm{p}} \\ q_{\mathrm{p}} + 1 & \text{if } \mathit{activate} \wedge \neg \mathit{activate}_{\mathrm{p}} \wedge q_{\mathrm{p}} + 1 < |\mathrm{P}| \\ q_{\mathrm{p}} & \text{otherwise.} \end{cases}$$

Remark that every counter $q_{\mathrm{p}}$ takes its values in the domain $Q \triangleq \{0, \ldots, |\mathrm{P}| - 1\}$. We further declare a *priority list* by using an additional set of symbols $p_i$, for $i \in \{1, \ldots, |\mathrm{P}|\}$. The $p_i$'s also take their values in $Q$, and are constrained according to the invariant $\varphi_{\mathsf{prios}}$ defined as:

$$\varphi_{\mathsf{prios}} \triangleq \bigwedge_{i \in \{1,\ldots,|\mathrm{P}|-1\}} p_i \geqslant p_{i+1} \wedge \qquad \text{[p-sorted]}$$

$$\bigwedge_{i \in \{1,\ldots,|\mathrm{P}|\}} \bigvee_{\mathrm{p} \in \mathrm{P}} p_i = q_{\mathrm{p}} \wedge \sum_{i \in \{1,\ldots,|\mathrm{P}|\}} p_i = \sum_{\mathrm{p} \in \mathrm{P}} q_{\mathrm{p}}. \qquad \text{[p-values]}$$

The above constraint basically states that the list of values associated to the sequence $(p_i)_{i \in \{1,\ldots,|\mathrm{P}|\}}$ is decreasing [p-sorted], and only contains values that belong to the set of all inactivity counters $q_{\mathrm{p}}$'s [p-values]. The $p_i$'s belong to the *controllable input space* of the model: this means that the actual computation of the priority list that these symbols denote (*i.e.*, sorting the values of all activity counters) is *encoded* as part of the target control strategy, and thus eventually within the piece of circuit that computes the CGL.

We eventually express the *fairness* constraint in terms of the above symbols as the conjunction

$$\varphi_{\mathsf{fairness}} \triangleq \varphi_{\mathsf{prios}} \wedge \bigwedge_{\mathrm{p} \in \mathrm{P}} (\mathit{activate}_{\mathrm{p}} \Rightarrow q\mathrm{p} \in \{p_1, \ldots, p_{|\mathrm{P}_{\mathrm{act}}|}\})$$

where $|\mathrm{P}_{\mathsf{act}}| \triangleq |\{\mathrm{p} \in \mathrm{P} | \mathit{activate}_{\mathrm{p}}\}|$ denotes the number of processes that are being activated. $\varphi_{\mathsf{fairness}}$ states that if a process p is activated ($\mathit{activate}_{\mathrm{p}}$), then the value of its inactivity counter must belong to the $|\mathrm{P}_{\mathsf{act}}|$ highest ones. Put another way, $\varphi_{\mathsf{fairness}}$ imposes that if a process is activated, then every other process whose inactivity counter is strictly higher is also activated.

#### 3.3.4. Avoiding starvation by enforcing concurrency

One must also ensure the absence of starvation, which in our case may happen if a process is never suspended. A model of concurrency is therefore declared to ensure that processes are forcibly suspended upon certain circumstances. To clarify the definitions below, we define $\mathit{stalled}_{\mathrm{p}} \triangleq \mathit{suspended}_{\mathrm{p}} \wedge \neg \mathit{idle}_{\mathrm{p}}$ to hold whenever the computations of some process p are suspended while its idleness condition does not hold. We give below two invariants that each correspond to a

model of concurrency: Forcing *cooperation* between processes makes use of the job termination symbol $done_{\mathsf{p}}$ of each process $\mathsf{p}$:

$$\varphi_{\mathsf{coop}} \triangleq \bigwedge_{\mathsf{p}\in\mathsf{P}} \left( term_{\mathsf{p}} \Rightarrow \bigvee_{\mathsf{p}'\in\mathsf{P}\setminus\{\mathsf{p}\}} stalled_{\mathsf{p}'} \Rightarrow \bigvee_{\mathsf{p}'\in\mathsf{P}\setminus\{\mathsf{p}\}} activate_{\mathsf{p}'} \right),$$

where $term_{\mathsf{p}} \triangleq \neg suspended_{\mathsf{p}} \wedge (done_{\mathsf{p}} \vee idle_{\mathsf{p}})$, holds iff at least one stalled process distinct from $\mathsf{p}$is activated whenever $\mathsf{p}$terminates its current job or becomes idle. Alternatively, one can implement *periodic preemption* of processes. This is achieved with the help of the additional non-controllable input symbol *slow-clk* that acts as a "slow" clock, with

$$\varphi_{\mathsf{preempt}} \triangleq slow\text{-}clk \Rightarrow \bigvee_{\mathsf{p}\in\mathsf{P}} stalled_{\mathsf{p}} \Rightarrow \bigvee_{\mathsf{p}\in\mathsf{P}} activate_{\mathsf{p}},$$

which states that at least one process be activated whenever *slow-clk* holds while some process is stalled.

$\varphi_{\mathsf{concurrency}}$ can be defined so as to hold whenever either or both $\varphi_{\mathsf{coop}}$ and/or $\varphi_{\mathsf{preempt}}$ hold, according to the desired model of concurrency.

#### 3.3.5. Putting it all together with clock-inhibition signals

We finally relate the controllable input symbols $inhibit_{\mathsf{p}}$ with the model using the invariant

$$\varphi_{\mathsf{inhib\text{-}suspended\text{-}only}} \triangleq \wedge_{\mathsf{p}\in\mathsf{P}} (inhibit_{\mathsf{p}} \Rightarrow suspended_{\mathsf{p}}),$$

which states that a process must be suspended for its clock to be gated. Overall, the global safety objective that the strategy must ensure by choosing values for all controllable signals (*i.e.*, for each process $\mathsf{p}$, $c_m$ and $inhibit_{\mathsf{p}}$, and the $p_i$ 's) corresponds to the conjunction

$$\varphi_{\mathsf{strict\text{-}progress}} \wedge \varphi_{\mathsf{fairness}} \wedge \varphi_{\mathsf{concurrency}} \wedge \varphi_{\mathsf{inhib\text{-}suspended\text{-}only}}. \tag{2}$$

Besides enforcing that clock-inhibition signals hold at appropriate clock cycles, this objective also enforces progress, fairness, and a suitable model of concurrency, by virtue of the encoded scheduling constraints.

### 3.4. Achieving power-efficiency by control

We define the estimated measure of the total instantaneous power consumption (in number of register flips) of all process from the original design as $\mathrm{P}_{\mathsf{P}} \triangleq \sum_{\mathsf{p}\in\mathsf{P}} \mathrm{P}_{\mathsf{p}}$.

**Optional peak power constraint** The first means of using our derived models to achieve objectives related to power efficiency is to specify that the sought after strategy should ensure a given upper-bound $\mathsf{P}_{\mathsf{max}}$ on the value of $\mathsf{P}_{\mathsf{P}}$ : one can achieve this with the help of the additional safety objective $\varphi_{\mathsf{P}_{\mathsf{max}}} \triangleq \mathsf{P}_{\mathsf{P}} \leqslant \mathsf{P}_{\mathsf{max}}$ , to be appended with a conjunction to Eq. (2).

**Energy minimization.** Further, our models offer an alternative means for enforcing some level of power-awareness of the scheduling induced by the CGL. They can indeed be used to refine a strategy towards minimizing the value of $\mathsf{P}_{\mathsf{P}}$ (*i.e.*, some measure of instantaneous power consumption) summed over a time window, *i.e.*, minimizing the energy consumed.

### 3.5. Enabling dynamic CGL reconfiguration

The invariant of the model as defined above can optionally be refined to support the dynamic reconfiguration of power-awareness policies. For instance, introducing a user-accessible register that switches to/from a clock-gating logic based on idleness conditions only boils down to: (i) add a non-controllable Boolean input symbol $cfg_{\mathsf{idle}}$ to the model; and (ii) replace $\varphi_{\mathsf{inhib\text{-}suspended\text{-}only}}$ in Eq. (2) with $\varphi_{\mathsf{inhib\text{-}configurable}} \triangleq$

$$\wedge_{\mathrm{p}\in\mathrm{P}}\left(inhibit_{\mathrm{p}} \Rightarrow (\text{if } cfg_{\mathsf{idle}} \text{ then } idle_{\mathrm{p}} \text{ else } suspended_{\mathrm{p}})\right).$$

This way, the power-aware scheduling policy can be turned off by setting the input wire of the resulting CGL that corresponds to $cfg_{\mathsf{idle}}$ to @1@, thereby inducing a CGL behavior that corresponds to a classical clock-gating based on idle conditions only.

## 4. Experimental evaluation

We have applied our approach on a series of various designs built from three RTL implementations of widespread signal coding or decoding algorithms: we use a Run-Length Encoder (RLE), a Huffman decoder, and a serial Reed-Solomon (RS) decoder[3]: (a) our first design consists of a pipeline made of the RS decoder followed by the RLE, and then the Huffman decoder; (b) remaining designs comprise a variable number $N$ of RLEs put in parallel (*i.e.*, they receive and output jobs from external ports). Due to the size of the RS decoder, and its internal structure divided into many sub-modules, for this particular process we have applied our implementation abstraction procedure (cf. Section 3.2) on a clearly identifiable sub-module that encodes its control-flow. In addition, in all

[3]Each available at https://github.com/peterqt95/rle, https://github.com/rahuldhameja/Huffman-Decoder, and https://opencores.org/project,reed_solomon_decoder.

cases selected registers were easy to identify, as most control-flow related HDL variables were named "state", "done", etc. Overall, each resulting abstract process implementation model features around 7 Boolean state symbols, and 15 Boolean non-controllable input symbols (oracles).

To experimentally assess the functional correctness and compare the respective dynamic power dissipation of each of the designs at hand (the originals and all the power-aware ones), we first carried out logic synthesis on all of them using the Altera Quartus synthesizer. We then used the Altera ModelSim simulation tool to perform functional simulations using the same benchmark for all circuits originating from the same designs, and checked that the resulting output traces were strictly equivalent.

Table 1: Evaluation results for the original and power-aware designs; the "Cycles" column denotes the total number of clock cycles required for processing the considered test-bench (Device: Cyclone IV, Frequency: 100 Mhz)

| Design Objective | Strategy Computation | | Total Size (bits) | | Power Consumption of Resulting Design | | |
|---|---|---|---|---|---|---|---|
| | Time (s) | Max Memory (MB) | Logic | Registers | Cycles | Cycle Average (mW) | Saving (%) |
| Original | n/a | n/a | 30914 | 23835 | 5969 | 353.68 | – |
| Idleness | 0.08 | 45.56 | 30924 | 23837 | 5969 | 344.81 | 2.51 |
| $P_{max} \leqslant 200$ | 0.40 | 48.12 | 31270 | 23857 | 11935 | 231.49 | 34.55 |
| $P_{max} \leqslant 199$ | 0.25 | 47.27 | n/a | n/a | n/a | n/a | n/a |
| Optim. | 3.96 | 262.87 | 31224 | 23857 | 11932 | 223.20 | 36.89 |
| Cfg.-Idleness | 5.69 | 235.68 | 31305 | 23858 | 5969 | 363.97 | –2.91 |
| Cfg.-Optim. | 5.69 | 235.68 | 31305 | 23858 | 11932 | 226.60 | 35.93 |

The original design (4) was subject to various objectives and configurations, the corresponding results of which we report in Table 1. The "Idleness" objective corresponds to a design where the CGL operates based on idleness conditions only. "$P_{max} \leqslant 200$" and "$P_{max} \leqslant 199$" both seek to impose a strict upper-limit to instantaneous power consumption based on some maximum amount of register flips. "Optim." results from a CGL that achieves a minimization of energy over a sliding window of 3 ticks. Lastly, lines where objectives are prefixed with "Cfg.-" correspond to a single design that incorporates a reconfigurable CGL, as described in Section 3.5.

We first observe that there does not exist a strategy that is able to impose the safety objective $P_{max} \leqslant 199$; this actually reveals that it is not possible to meet both this objective and the scheduling constraints for this particular design. Further, even using a small time window of 3 ticks for the optimization objective,

we can compute a CGL that reduces the (simulated) average power consumption per cycle by about 37%. We have further evaluated the efficiency of the strategy

Table 2: Performance of the strategy computation tool for $N$ RLE instances

| N | Design Objective | Time (s) | Memory (MB) | Feasibility |
|---|---|---|---|---|
| 2 | $P_{max} \leqslant 70$ | 0.08 | 45676 | ✓ |
| | $P_{max} \leqslant 69$ | 0.10 | 45676 | × |
| | $P_{max} \leqslant 70$ +Optim. | 0.22 | 46844 | ✓ |
| | $P_{max} \leqslant 69$ +Optim. | 0.09 | 47328 | × |
| 3 | $P_{max} \leqslant 70$ | 0.47 | 48148 | ✓ |
| | $P_{max} \leqslant 69$ | 0.64 | 50760 | × |
| | $P_{max} \leqslant 70$ +Optim. | 3.54 | 126504 | ✓ |
| | $P_{max} \leqslant 69$ +Optim. | 0.66 | 50840 | × |
| 4 | $P_{max} \leqslant 70$ | 3.54 | 78980 | ✓ |
| | $P_{max} \leqslant 69$ | 23.18 | 225616 | × |
| | $P_{max} \leqslant 70$ +Optim. | 250.62 | 3249848 | ✓ |
| | $P_{max} \leqslant 69$ +Optim. | 23.08 | 225784 | × |

computation tool by applying our approach on original designs (4). We report the results in Table 2, where objectives suffixed with "+Optim." correspond to designs where both an upper-limit on power consumption and a minimization of energy over a sliding window of 3 ticks is desired.

## 5. Related works

Several commercial and academic tools already target automated clock-gating from RTL code. [17] developed an algorithm that automatically introduce the some clock-gating logic into RTL descriptions of circuits, although they focus on the exact computation of idleness conditions for each individual registers. In turn, [1, 13, 14] suggest an algorithm that automatically tries to approximate idleness conditions. [3] detect idleness conditions by using explicit finite-state machines in an approach that targets the conditional activation of individual hardware components using their "enable" signal (an approach similar to clock-gating). At last, [17] exploited conditional statements and case structures within blocks of clock-triggered assignments in HDL languages to determine such conditions. Our approach draws from the latter since it also works based on module-level HDL descriptions, and we build our symbolic models based on the conditional clock-triggered assignments.

[19] develop a formal framework based on data-flow models to analyze hardware circuits, and derive some control logic to drive them towards various performance objectives; our technique operates on similar models, and rely on abstracted versions of the circuits to tackle the state-space explosion problem. More practically, [4, 15] also focus on system-level to suggest a clock gating technique that operates on whole modules.

## 6. Conclusions and future works

We have advanced a tool-supported framework for producing power-aware designs from RTL implementations of KPNs. Our technique permits the automated construction of an abstract symbolic model of the design, as well as associated control objectives. The automatically computed strategy is then translated into a piece of circuit that encodes a clock-gating logic for the design, and guarantees that the specified objectives are met. We plan to design guidelines for using our approach on black-box IPs with user-provided symbolic models. As stated in Section 2, the strategies are usually computed under the assumption that the environment of the model behaves as an adversary: in a sense the strategy is pessimistic. A natural extension of our work is to take some stochastic models of the environment (*e.g.,* inferred from simulation traces) into account to compute strategies that achieve better power efficiency on average.

## References

[1] P. Babighian, L. Benini, and E. Macii: A scalable algorithm for RTL insertion of gated clocks based on ODCs computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **24**(1), (2005), 29–42, DOI: 10.1109/TCAD.2004.839489.

[2] R. Bellman: Dynamic programming and stochastic control processes. *Information and Control*, **1**(3), (1958), 228–239, DOI: 10.1016/S0019-9958(58)80003-0.

[3] L. Benini, P. Siegel, and G. De Micheli: Saving power by synthesizing gated clocks for sequential circuits. *IEEE Design & Test of Computers*, **11**(4), (1994), 32–41, DOI: 10.1109/54.329451.

[4] R. Bhutada and Y. Manoli: Complex clock gating with integrated clock gating logic cell. In *2007 International Conference on Design Technology of Integrated Systems in Nanoscale Era*, (2007), 164–169, DOI: 10.1109/DTIS.2007.4449512.

[5] J. Billon: Perfect normal forms for discrete programs. Technical report, Bull, 1987.

[6] E.M. Clarke, E.A. Emerson, and A.P. Sistla: Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, **8**(2), (1986), 244–263, DOI: 10.1145/5397.5399.

[7] E. Dumitrescu, A. Girault, H. Marchand, and E. Rutten: Multicriteria optimal reconfiguration of fault-tolerant real-time tasks. *IFAC Proceedings Volumes*, **43**(12), (2010), 356–363, DOI: 10.3182/20100830-3-DE-4013.00059.

[8] K. Gilles: The semantics of a simple language for parallel programming. *Information Processing*, **74** (1974), 471–475.

[9] E.A. Lee and T.M. Parks: Dataflow process networks. *Proceedings of the IEEE*, **83**(5), (1995), 773–801, DOI: 10.1109/5.381846.

[10] H. Marchand and M.L. Borgne: On the optimal control of polynomial dynamical systems over z/pz. In *4th International Workshop on Discrete Event Systems*, (1998), 385–390.

[11] H. Marchand, P. Bournai, M.L. Borgne, and P.L. Guernic: Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System: Theory and Applications*, **10**(4), (2000), 325–346, DOI: 10.1023/A:1008311720696.

[12] S. Miremadi, B. Lennartson, and K. Akesson: A BDD-based approach for modeling plant and supervisor by extended finite automata. *IEEE Transactions on Control Systems Technology*, **20**(6), (2012), 1421–1435, DOI: 10.1109/TCST.2011.2167150.

[13] M. Özbaltan: *Achieving Power Efficiency in Hardware Circuits with Symbolic Discrete Control*. PhD thesis, University of Liverpool, 2020.

[14] M. Özbaltan and N. Berthier: Exercising symbolic discrete control for designing low-power hardware circuits: an application to clock-gating. *IFAC-PapersOnLine*, **51**(7), (2018), 120–126, DOI: 10.1016/j.ifacol.2018.06.289.

[15] M. Özbaltan and N. Berthier: A case for symbolic limited optimal discrete control: Energy management in reactive data-flow circuits. *IFAC-PapersOnLine*, **53**(2), (2020), 10688–10694, DOI: 10.1016/j.ifacol.2020.12.2842.

[16] M. Pedram and Q. Wu: Design considerations for battery-powered electronics. In *Proceedings 1999 Design Automation Conference*, (1999), 861–866, DOI: 10.1109/DAC.1999.782166.

[17] N. Raghavan, V. Akella, and S. Bakshi: Automatic insertion of gated clocks at register transfer level. In *Proceedings of the 12th International Conference on VLSI Design*, (1999), 48–54, DOI: 10.1109/ICVD.1999.745123.

[18] P. Ramadge and W. Wonham: The control of discrete event systems. *Proceedings of the IEEE*, **77**(1), (1989), 81–98, DOI: 10.1109/5.21072.

[19] S. Tripakis, R. Limaye, K. Ravindran, G. Wang, H. Andrade, and A. Ghosal: Tokens vs. signals: On conformance between formal models of dataflow and hardware. *Journal of Signal Processing Systems*, **85**(1), (2016), 23–43, DOI: 10.1007/s11265-015-0971-y.