

# Fantastic Circuits and Where to Find Them – A Holistic ILP Formulation for Model-Based Hardware Design

NICOLAI FIEGE, University of Kassel, Germany

PETER ZIPF, University of Kassel, Germany

The end of Moore’s law and Dennard scaling emphasizes the need for application-specific computing architectures to achieve high resource and energy efficiency and real-time performance. The concept of a silicon compiler remains an enduring aspiration for design time reduction. In order to generate hardware implementations at register transfer level from behavioral descriptions, design automation tools must address challenging and interdependent problems, including allocation, scheduling, and binding. Additionally, manual intervention by the user is necessary to balance the *resources vs. performance trade-off* via, for example, function inlining or loop unrolling/pipelining. Existing approaches typically solve these problems sequentially, compromising optimality in favor of simplicity and run-time. Here we show how to model the whole model-based design flow as one holistic integer linear programming (ILP) formulation aiming at consistently deriving the optimal microarchitecture for any given application. Incorporating clock gating minimizes the number of useless operations with negligible resource overhead (if any), while always guaranteeing optimal throughput. The unified nature of the proposed ILP model enables implementations unmatched by state-of-the-art approaches in terms of resource efficiency and measured power consumption. These results facilitate a streamlined design flow for highly optimized embedded systems in the context of model-based design.

CCS Concepts: • **Hardware** → **Operations scheduling; Datapath optimization; Resource binding and sharing.**

Additional Key Words and Phrases: Design tools, Electronic design automation, Energy efficiency, Hardware compiler, Optimisation, Reconfigurable systems

## ACM Reference Format:

Nicolai Fiege and Peter Zipf. 2024. Fantastic Circuits and Where to Find Them – A Holistic ILP Formulation for Model-Based Hardware Design. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (November 2024), 36 pages. <https://doi.org/10.1145/3705325>

## 1 INTRODUCTION

Software-based implementations of complex embedded systems regularly fail to meet high standards for resource efficiency, energy consumption, or latency and real-time requirements. To harness computing power effectively, a trend to increasingly specialized hardware implementations can be seen (e.g., a specialized ASIC for Transformer inference [21]). This necessitates intricate register-transfer-level (RTL) descriptions to take advantage of the full potential for energy efficiency and parallel execution. The manual development of RTL code, compared to software design, proves to be a complex and costly task. It is prone to errors and challenging to simulate and debug. Consequently, the adoption of electronic design automation (EDA) concepts becomes imperative to meet expectations in terms of time-to-market and quality.

Decades of research [14, 25, 30, 36] have been dedicated to the automatic transformation of high-level software languages into hardware, paving the way for C-based high-level synthesis (HLS). This method involves synthesizing C/C++/SystemC software descriptions into hardware that

---

Authors’ addresses: Nicolai Fiege, [nfiege@uni-kassel.de](mailto:nfiege@uni-kassel.de), University of Kassel, Kassel, Germany; Peter Zipf, [zipf@uni-kassel.de](mailto:zipf@uni-kassel.de), University of Kassel, Kassel, Germany.

---

© 2024 Copyright held by the owner/author(s).

This is the author’s version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM Transactions on Reconfigurable Technology and Systems*, <https://doi.org/10.1145/3705325>.

replicates the same functionality as the software. HLS, owing to its benefits in quick design-space exploration (DSE), is extensively employed in rapid prototyping and FPGA applications [32, 51].

In parallel, model-based design frameworks, such as MathWorks Simulink HDL Coder, Intel DSPBuilder for FPGAs, and AMD Vitis Model Composer, have become industry standards. These tools facilitate system designers throughout the modeling, implementation, simulation, and testing processes, enabling the realization of more complex designs and reducing time-to-market [50].

While contemporary C-based HLS tools can produce hardware on par with manually coded RTL descriptions [30], commercial model-based hardware design tools lag [46], and there are no academic tools available.

The decision to implement an application in custom hardware stems from diverse requirements, often boiling down to two options:

- (1) Implementing the fastest possible circuit (possibly under tight resource constraints).
- (2) Implementing the smallest/most efficient circuit to achieve a pre-defined throughput.

Current approaches predominantly focus on the former, generating the fastest circuit from the given input description [40]. However, what if system constraints limit useful throughput, such as a fixed data rate from external sensors? In such cases, pursuing higher speed alone may result in energy or resource wastage.

An emerging demand for energy-efficient circuits in both embedded and high-performance computing applications has become evident. This demand, driven by factors like increased battery lifetime and economic considerations, is not fully addressed by current automatic hardware generation approaches, which predominantly focus on the "resources vs. performance" trade-off [10, 22, 39, 40, 45]. We aim to bridge this gap with the following contributions:

- (1) Formalizing the model-based design flow in a holistic integer linear programming (ILP) model for the combined optimization of allocation, scheduling, binding, port assignment, and clock gating targeting FPGAs (Section 4).
- (2) Providing a comprehensive experimental evaluation demonstrating that our proposed approach can handle practical problem sizes. Our algorithm is able to reduce resource and power consumption in resulting FPGA implementations without any performance degradation (Section 5).

Whether one seeks to implement the fastest circuit under tight resource constraints or the smallest circuit to achieve a defined throughput, our proposed approach can generate highly optimized hardware implementations from model-based input descriptions. If an automated procedure always provides the optimal hardware implementation of an algorithm for a given application, are design space explorations even necessary anymore?

## 2 MOTIVATIONAL EXAMPLE

Consider the example Simulink model in Fig. 1. It shows a simple PID controller. Suppose that it shall be integrated into a more complex embedded system running at 200 MHz (e.g., due to a memory interface running at that specific frequency). Inputs and outputs are connected to streaming interfaces, but the input is driven by an analog/digital converter (ADC) with a sample rate of 50 MHz (i.e., one new data sample every four clock cycles). In that case, the most sensible choice is to implement the PID controller with an initiation interval ( $\Pi$ ) of four. This means that the hardware is optimized such that it is able to accept new input data and to generate new output data every four clock cycles. Even if resource constraints would allow for a more parallelized implementation (e.g.,  $\Pi = 2$  or even a fully parallel version with  $\Pi = 1$ ), there would be no benefit as the ADC would still be the bottle neck regarding throughput.

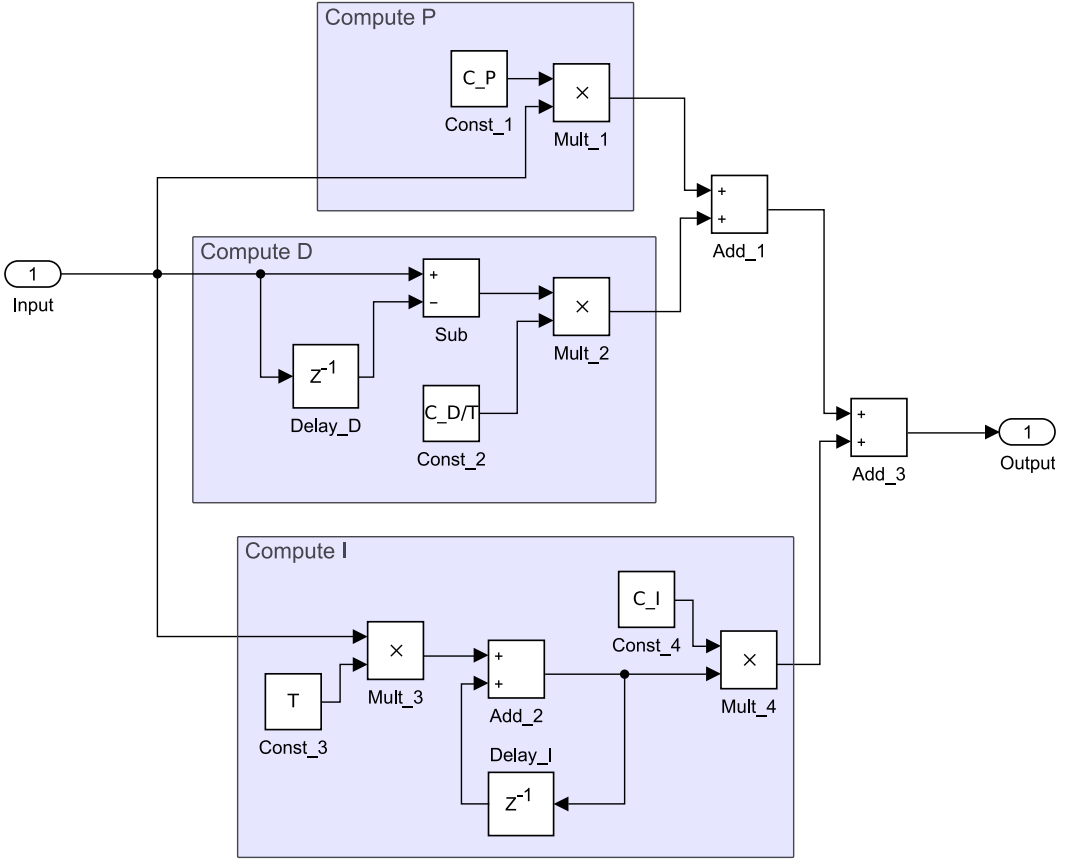


Fig. 1. Example Simulink model: a PID controller

Table 1. Example operator latencies

Type:	Add	Subtract	Multiply
Latency [clock cycles]:	2	2	3

The two delay components in Fig. 1 (i.e., Delay\_D and Delay\_I) operate on a sample-level, which means that they represent a delay by four clock cycles, each. For efficient arithmetic operator implementations, an operator library (e.g., FloPoCo [18]) is usually used. In order to reach the target frequency, it is often necessary to pipeline the operators, leading to a latency regarding their input/output relationship. Example values for operator latencies are given in Table 1.

It suffices to allocate one operator per operator type due to  $\Pi = 4$ . Consequently, the implemented multiplier must be used four times to execute the four multiplication operations shown in Fig. 1 (i.e., Mult\_1...Mult\_4), the adder must be used three times and the subtractor is used once per data sample processed. In general, all operator types  $\omega \in \Omega$  limit the minimum achievable  $\Pi$  via [43]

$$\Pi_{\text{res}}^{\perp} = \min_{\omega \in \Omega} \left\lceil \frac{|O_{\omega}|}{F(\omega)} \right\rceil \tag{1}$$

where  $\Pi_{\text{res}}^\perp$  is the minimum  $\Pi$  caused by operator constraints;  $F(\omega)$  are the allocated hardware units of type  $\omega$ ; and  $O_\omega$  is the set of operations to be executed by an operator of type  $\omega$ . In this example, the multiplication operations limit  $\Pi_{\text{res}}^\perp = 4$ , as  $|O_{\text{mult}}| = 4$  (four different multiplication operations) and  $F(\text{mult}) = 1$  (one multiplier to be implemented in hardware).

This relationship can be deduced from the fact that one operator can only start to execute at most one new operation per clock cycle. This is can be expressed as [38]

$$|\{o_i \in O_\omega : t_i \bmod \Pi = m\}| \leq F(\omega) \quad \forall \omega \in \Omega, m \in [0, \Pi). \quad (2)$$

Here,  $o_i \in O_\omega$  are all operations that must be executed by an operator of type  $\omega$  with limited availability  $F(\omega)$ ; and  $t_i$  is the clock cycle in which  $o_i$  is executed.

It is self-evident that data dependencies must be met when choosing execution times  $t_i$  for all operations. For example, `Add_1` can only begin its execution when results for `Mult_1` and `Mult_2` are available. Formally, this constraint is expressed as [6]

$$t_j - t_i \geq L(Y(o_i)) - d_{i,j} \cdot \Pi \quad \forall e_{i,j} \in E. \quad (3)$$

$E$  is the set of all edges (i.e., data dependencies) in the model;  $d_{i,j}$ , represents the edge weight (i.e., the algorithmic distance on that edge);  $L(\omega)$  is a function which returns the latency of an operator of type  $\omega$ ; and  $Y(o_i)$  returns the operator type associated with operation  $o_i$ . For example, `Sub`'s subtract input depends on the `Input` value from the last sample. Therefore, the corresponding value for  $d_{i,j}$  is one. Cyclic data dependencies (e.g., `Add_2` for sample  $n$  depends on the value for `Add_2` at sample  $n - 1$ ) limit the  $\Pi$  via [6]

$$\Pi_{\text{rec}}^\perp = \min_{\text{cycle} \in G} \left[ \frac{\sum_{e_{i,j} \in \text{cycle}} L(Y(o_i))}{\sum_{e_{i,j} \in \text{cycle}} d_{i,j}} \right]. \quad (4)$$

Here, the term cycle refers to a series of edges forming a closed loop.

In order to prevent excessive chaining of operations with a latency of zero and a propagation delay greater than zero nanoseconds, Oppermann et al. introduced the concept of dedicated chaining edges [38]. These edges are added to the graph prior to scheduling so the scheduler can determine an optimal chaining-aware schedule. In order to support chaining edges, the dependency constraint is changed to

$$t_j - t_i \geq \Delta(e_{i,j}) - d_{i,j} \cdot \Pi \quad \forall e_{i,j} \in E. \quad (5)$$

with

$$\Delta(e_{i,j}) = \begin{cases} 1 & e_{i,j} \in E_c \text{ (i.e., } o_i \& o_j \text{ are not chainable)} \\ L(Y(o_i)) & e_{i,j} \notin E_c \end{cases} \quad (6)$$

where  $E_c \subset E$  represents the set of all chaining-related edges added during pre-processing. Note that the work by Oppermann et al. [38] does not distinguish these additional edges from the original ones since that work is solely concerned with solving the modulo scheduling problem. In this work, however, edges are also used for, e.g., binding decisions and lifetime register counting, so we must differentiate between original edges (for data transfers between operations) and chaining edges (only used to compute an optimal chaining-aware schedule).

Table 2 shows a possible schedule for all operations that adheres to the operator latencies given in Table 1 and to operator and dependency constraints given in (2) and (3), respectively.

Scheduling times for constants can be omitted as they are hard-wired in the final hardware implementation and hence their values do not change over time. It can be seen that a new multiplication operation starts in each time slot modulo  $\Pi$ . Therefore, the multiplier executes a *useful*

Table 2. Example schedule for the Simulink model shown in Fig. 1

Operation	clock cycle	mod 4
Input	0	0
Output	10	2
Add_1	5	1
Add_2	3	3
Add_3	8	0
Sub	0	0
Mult_1	2	2
Mult_2	3	3
Mult_3	0	0
Mult_4	5	1

operation in each clock cycle. The adder, however, is not used to capacity. It only performs a *useful* operation in three out of four clock cycles. If no additional counter measures are taken for its hardware implementation, it might still get new, meaningless input data, the internal logic gates produce switching operations, and dynamic power is wasted. The situation is even more extreme for the subtracter. It only performs one *useful* operation within four clock cycles. Hence, 75 % of its switching activity is caused by useless operations. These situations can be avoided by integrating clock gating into the synthesis procedure: the clock signal is disabled in exactly those clock cycles in which otherwise useless operations would be performed. Since arithmetic operators on FPGAs are usually deeply pipelined to reach high clock frequencies, savings are two-fold: (i) internal pipeline registers do not toggle, and (ii) the combinational logic to perform the operations also does not switch. This reduces overall switching activity and, thus, also the circuit’s dynamic energy consumption.

Note that, depending on the chosen back end, converting the clock gating into clock enables might prove beneficial (e.g., for timing closure). In such cases, our model can also be used to obtain optimal solutions. However, when clock enables are used, the number of clock domains is not a limiting factor anymore, so each allocated functional unit can get its unique “virtual clock domain” with a unique enable-pattern.

### 3 THE IMPLEMENTATION OF MODULO SCHEDULED CIRCUITS

Figure 2 shows the microarchitecture of the implemented systems. It consists of (i) inputs and outputs, (ii) a collection of operators with connected lifetime registers, (iii) a MUX-based interconnect network, (iv) a counter to control the data flow, and (v) clock buffers with decoders for the counter value at their enable-inputs. Note that it is important to drive these clocks glitch-free in order to avoid unwanted clock pulses. This can, for example, be achieved by using BUF<sub>GCE</sub> instances available on AMD FPGAs [2]. Furthermore, we do not need to implement any clock-domain-crossing hardware such as first-in-first-out (FIFO) buffers, since all internal clocks are derived from one master clock (called *clk* in Fig. 2) and the influence from any resulting clock skew can be accounted for by the synthesis tool (e.g., by inserting additional clock buffers during synthesis). Alternatively, modern synthesis tools like AMD Vivado or Intel Quartus also support clock enable conversion in which gating logic is automatically converted to enable logic for the respective registers. In such

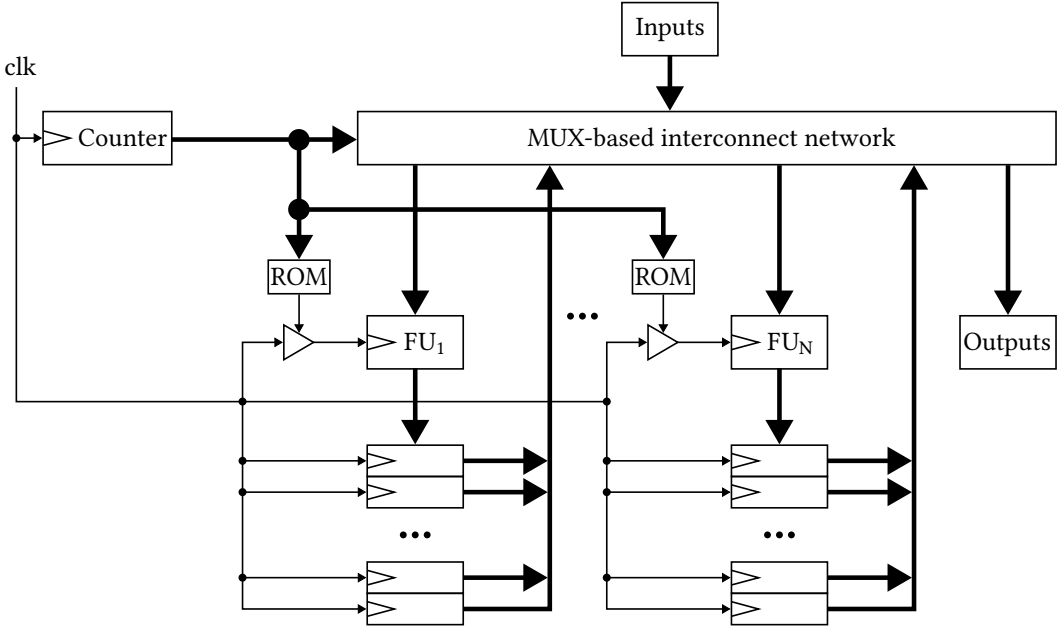


Fig. 2. Microarchitecture: thin lines represent 1-bit signals and thick lines represent data buses; the triangle-shaped blocks driving FU clocks are clock buffers with enable-inputs

cases, the clock buffer in Fig. 2 should be replaced by a simple AND-gate for automatic detection and conversion [1, 28].

Inputs and outputs are implemented as a streaming interface. We expect the surrounding circuit to provide a steady stream of input data at the correct data rate, defined via the system's II. Simultaneously, the system generates output data with the same data rate, delayed by the system's schedule length. Again, we expect the surrounding circuit to be able to accept that data stream.

The operators are implemented and their characteristics (e.g., latency and resource costs for a given back end/operating frequency) are defined by an external operator library. In our experiments, we use the open-source framework FloPoCo [18], but in general, the proposed algorithm is compatible with any given operator library. The lifetime registers are able to store the produced variables until all operations depending on them are started. Register sharing is applied whenever possible, as shown in Fig. 3.

The interconnect network is based on Multiplexers (MUX). In each clock cycle, they pass the correct variables to the operators, controlled by the program counter, which is implemented as a simple modulo-II counter. In cases where multiple MUX inputs are passed to an operator in different time slots modulo II, the respective MUX ports can be shared via a decoder on the select input, as illustrated in Fig. 4.

The clock buffers are controlled by the program counter and a decoder (e.g., implemented using a look-up table). Their purpose is to turn off operators whenever they are not needed due to the scheduling and binding in order to reduce dynamic power consumption. Suppose that an operation is started and not yet finished on any given operator and its clock is turned off. In that case, the operation's latency increases by the number of clock cycles the operator is turned off before the operation traversed the entire operator pipeline. Hence, the clock gating must be compatible

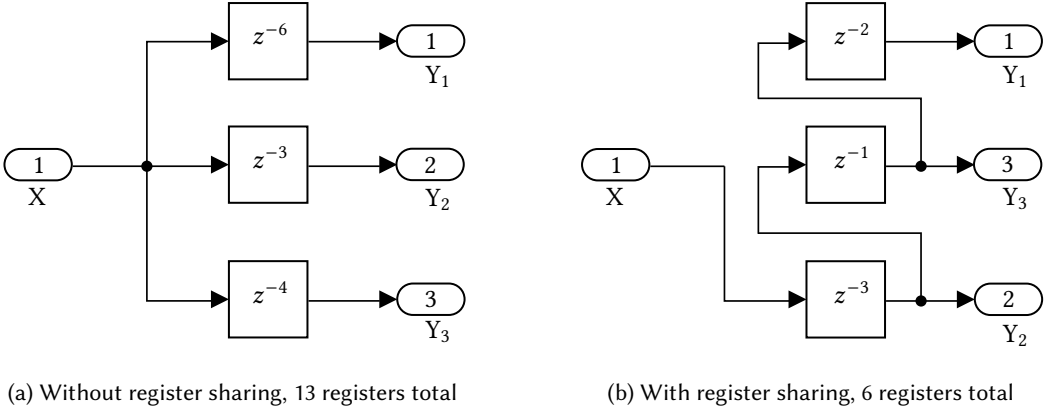


Fig. 3. Register sharing example, illustrated via Simulink blocks

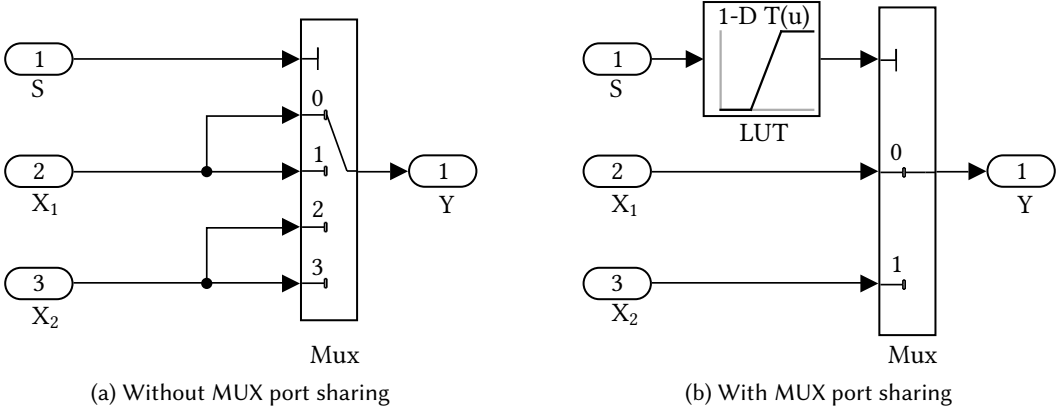


Fig. 4. MUX port sharing example, illustrated via Simulink blocks

with the scheduling and binding such that dependencies between operations are not violated. We therefore handle all these decisions together in a unified ILP model as described in Section 4.2.

### 3.1 Problem Definition

With the aforementioned architecture in mind, there are several degrees of freedom that not only influence each other but also affect throughput, latency, resources and energy consumption of the resulting implementation.

As a reference for the forthcoming problem description, Tables 3–6 show the parameters, upper bounds, functions, and outputs of the combined allocation, modulo scheduling, binding, port assignment, and clock gating problem (ASBPC-P). We use the following notations:

- Sets and lists are denoted via curly brackets:  $\{ \dots \}$ .
- The set of all natural numbers without zero is denoted as  $\mathbb{N}$ .
- The set of all natural numbers including zero is denoted as  $\mathbb{N}_0$ .
- The set of all real numbers is denoted as  $\mathbb{R}$ .
- The set of all non-negative real numbers (including zero) is denoted as  $\mathbb{R}_{\geq 0}$ .
- Intervals including their min/max values are denoted by square brackets:  $[ \dots ]$ .

Table 3. Parameters of the ASBPC-P; it should be noted that—depending on the problem at hand and the optimization criteria—not all parameters must be provided simultaneously

Name	Domain	Explanation
$\Pi$	$\in \mathbb{N}$	Initiation interval
$\#C$	$\in \mathbb{N}$	Number of available clock domains
Res	$= \{\text{LUTs, DSPs, } \dots\}$	Set of all low-level resource types
$r$	$\in \text{Res}$	A specific low-level resource type
$\Omega$	$= \{+, -, \times, \dots\}$	Set of all operator types
$\tilde{\Omega}$	$\subseteq \Omega$	Set of all commutative operator types
$\omega$	$\in \Omega$	A specific operator type
$G$	$= \{O, E, D\}$	Data flow graph
$O$	$= \{o_0, o_1, \dots\}$	List of all operations (i.e., vertices in $G$ )
$O_\omega$	$\subseteq O$	List of all operations that can be executed by an operator of type $\omega$ (i.e., $O_\omega = \{o_i \in O : \Upsilon(o_i) = \omega\}$ )
$E$	$\subseteq O \times O$	List of all precedence relations (i.e., edges in $G$ )
$e_{i,j} = (o_i, o_j)$	$\in E$	A specific edge, represented by a tuple
$E_c$	$\subset E$	List of all chaining-related edges that were added during pre-processing
$D$	$= \{d_{i,j} : e_{i,j} \in E\}$	List of all edge weights

Table 4. Bounds for the ASBPC-P

Name	Domain	Explanation
SL	$\in \mathbb{N}_0$	Schedule length limit (upper bound)
$R$	$\in \mathbb{N}_0$	Lifetime register limit (upper bound)
$P$	$\in \mathbb{N}_0$	Multiplexer input port limit (upper bound)
$S$	$\in \mathbb{R}_0$	Clock gating energy savings limit (lower bound)

- Unless specified otherwise, an interval represents only the integers within the given bounds, e.g.,  $[1, 3] = \{1, 2, 3\}$ .
- Intervals excluding min/max values are denoted by round brackets, e.g.,  $[1, 4) = [1, 3]$ .

A common technique to achieve a given target frequency is pipelining. This means that the operator library influences the achievable  $\Pi$  in presence of recurrences in the model, if operations in cycles are associated to pipelined operators. As an example consider again `Add_2` in Fig. 1. The recurrence has a distance of one (due to `Delay_I`) and a delay equal to `Add_2`'s number of pipeline stages. Following from (4),  $\Pi \geq \# \text{pipeline stages of Add}_2$ .

The allocation defines the number of operators implemented for each operator type. Since an operator can execute at most one operation per clock cycle, a fixed allocation also defines a lower-bound for the achievable  $\Pi$ . Considering again the example model in Fig. 1, allocating one multiplier would result in  $\Pi \geq 4$ , allocating two or three means  $\Pi \geq 2$  and allocating four results in  $\Pi \geq 1$ . When given a model and a target  $\Pi$ , an optimal HLS tool flow would be expected to either allocate the minimum number of operators to reach the requested  $\Pi$  or to notify the user that the inputs are incompatible (e.g., due to insufficient resources in the back end for the requested degree in parallelization).

Table 5. Functions of the ASBPC-P

Name	Domain	Explanation
$\Sigma(r)$	$: \text{Res} \mapsto \mathbb{R}_{\geq 0}$	(Weighting-)Function that maps a resource type to its contribution to the FPGA utilization
$X^\top(r)$	$: \text{Res} \mapsto \mathbb{N}_0$	Function that maps a resource type to the amount of available units on the target FPGA
$X(\omega, r)$	$: \Omega \times \text{Res} \mapsto \mathbb{N}_0$	Function that maps an operator to its low-level resource consumption of type $r$ per implemented operator instance
$L(\omega)$	$: \Omega \mapsto \mathbb{N}_0$	Function that maps an operator to its latency in clock cycles
$F(\omega)$	$: \Omega \mapsto \mathbb{N}$	Function that maps an operator type to the number of allocated hardware units (this function is an <i>output</i> of the modulo scheduling problem if the objective is to find an allocation for minimal back end resource utilization)
$\Phi(\omega)$	$: \Omega \mapsto \mathbb{N}_0$	Function that maps an operator type to its maximum additional latency due to clock gating
$\Lambda(\omega)$	$: \Omega \mapsto \mathbb{R}_{\geq 0}$	Function that maps an operator type to its dynamic energy consumption per clock cycle and per instance
$Y(o_i)$	$: O \mapsto \Omega$	Function that maps an operation to its associated operator type
$Q(e_{i,j})$	$: E \mapsto \mathbb{N}_0$	Function that maps an edge to the associated input port of its sink vertex
$\Xi(\omega)$	$: \Omega \mapsto \mathbb{N}$	Function that maps an operator type to the number of input ports of one of its operators (only relevant for non-commutative operator types)
$\Gamma^\perp(e_{i,j})$	$: E \mapsto \mathbb{N}_0$	Function that maps an edge to its minimum possible lifetime (trivial: $\forall e_{i,j} \in E : \Gamma^\perp(e_{i,j}) = 0$ )
$\Gamma^\top(e_{i,j})$	$: E \mapsto \mathbb{N}_0$	Function that maps an edge to its maximum possible lifetime (trivial: $\forall e_{i,j} \in E : \Gamma^\top(e_{i,j}) = \text{SL} - L(Y(o_j)) - L(Y(o_i)) + \Pi \cdot d_{i,j}$ )

Table 6. Outputs of the ASBPC-P, expressed via functions

Name	Domain	Explanation
$F(\omega)$	$: \Omega \mapsto \mathbb{N}$	Function that maps an operator type to the number of allocated hardware units
$T(o_i)$	$: O \mapsto \mathbb{N}_0$	Function that returns the start time of $o_i$
$B(o_i)$	$: O \mapsto \mathbb{N}_0$	Function that returns the index of the hardware unit which executes $o_i$
$Z(c, \tau)$	$: \mathbb{N}_0 \times \mathbb{N}_0 \mapsto \{0, 1\}$	Function that returns whether clock domain $c$ is turned <i>off</i> in modulo slot $\tau$
$H(\omega, x)$	$: \Omega \times \mathbb{N}_0 \mapsto \mathbb{N}_0$	Function that returns the index of the clock domain that instance $x$ of operator type $\omega$ is connected to; if $H$ returns zero, it is assumed that the <i>default</i> clock domain is used
$Q(e_{i,j})$	$: E \mapsto \mathbb{N}_0$	Function that maps an edge to the associated input port of the operator that executes the edge's sink vertex

Even when optimally allocating operators, it is possible that not all operators are fully utilized due to an imbalanced number of operations in the model to be implemented. In that case, it might be beneficial to use clock gating to turn off operators during their idle times in order to decrease dynamic power consumption. When the number of available clock domains is lower than the number of individually allocated operators, operators with similar idle times can be grouped and turned off jointly. Determining an optimal grouping of operators, however, is non-trivial and interdependent with the schedule and binding.

By increasing the latency of individual operations, the clock gating possibly influences the schedule length (by delaying consecutive operations) and the number of lifetime registers. On one hand, the lifetime register count can be decreased by delaying an operation early in the schedule if its first use occurs several clock cycles later. On the other hand, delaying an operation can also cause the lifetime of other operations with the same successor to increase, if they are finished in an earlier clock cycle.

For each connection in the model, there also exists a connection in the implemented system. However, it is possible to re-use the same connection for two different data transfers in the implemented system whenever the source operator, the sink operator, the lifetime, and the ports are identical. In that case, it is possible to reduce the size of the associated multiplexer in the interconnect network by using a decoder on its select input, as demonstrated in Fig. 4.

### 3.2 Related work

Modulo scheduling has been an active research topic since its inception in the 1980s [43]. Given an operator allocation and a graph, there are many algorithms available that are able to compute a schedule with optimal  $\Pi$  and schedule length [20, 24, 38, 52]. Due to the difficulty of resource-constrained scheduling, alternatively, heuristics can be used. They either, similar to optimal methods, aim at a latency reduction [6, 19, 42], or try to reduce implementation costs by minimizing variable lifetimes [33].

In recent years, Boolean Satisfiability (SAT) has emerged as a powerful competitor to Integer Linear Programming (ILP)-based methods, outperforming previous work in terms of algorithm runtime and solution quality [17, 24]. This comes at an increased complexity to derive a valid model due to SAT's reduced expressiveness. Furthermore, since all constraints are encoded at the bit level, constraints involving floating-point numbers are very cumbersome and inefficient to represent using pure SAT.

Especially in the context of hardware synthesis, it makes sense to leave the allocation of operators to the scheduling algorithm. Hence, a potential user solely requests a target  $\Pi$  for the given graph and the scheduler computes the minimum allocation to reach the  $\Pi$  [39, 52]. Determining *the minimum* allocation then strongly depends on the target platform and the operator library used. Aside from tackling the problem from a different angle (i.e., with slightly different inputs and goals), an allocation aware modulo scheduler can be used for quick design space explorations [39].

After having obtained a valid modulo schedule, the next step is usually to determine which operation is executed by which allocated operator. This is called *binding*. Different bindings lead to varying implementation costs for the interconnect network (i.e., multiplexers and registers) [15, 23, 44]. A dedicated binding algorithm takes a schedule as an input and, based on the data-flow graph and an allocation, computes a binding which reduces lifetime register and multiplexer costs. A natural step to reduce resource demand even further is to include the binding into the scheduling algorithm for *hardware aware* scheduling [22, 45]. This comes at the cost of an increased runtime compared to solving both problems separately [22, 23]. Currently, combinations of scheduling and binding only aim at a reduction of lifetime register costs. Multiplexer minimization is left for dedicated binding algorithms [15, 23, 44] and algorithms to compute a port assignment for binary

commutative operators [5, 7, 13], even though previous work has shown that multiplexers are a non-negligible source of dynamic power consumption in FPGA-based systems [8].

None of the aforementioned algorithms produces optimal results when implementing multiple or nested loops or loops with dynamic memory dependencies. This can be solved by several approaches. The first possibility is computing an optimized static schedule, where accurate operator costs are necessary to fit pipelined circuits onto low-cost FPGAs [40]. A different approach is dynamic scheduling [41], where the schedule is computed by the hardware itself, utilizing handshake signals [12, 29]. This can produce faster circuits, possibly at the cost of increased hardware. Lastly, both approaches can be combined, where portions of the circuit are scheduled statically when dynamic scheduling would not bring a performance increase [9, 16, 49]. Here we focus on model-based hardware design, hence, we restrict our work on the optimal implementation of a single model, assumed to be running indefinitely with fixed throughput.

Despite its relevance, power consumption-aware HLS for FPGAs is only researched superficially. Chen et al. propose a simulated annealing-based algorithm that simultaneously handles scheduling, binding and data path generation based on switching activity and wire length estimations [8]. Furthermore, they adopted a MUX optimization algorithm in order to reduce power consumption even further. Lhairech-Lebreton et al., on the other hand, proposed to clock different parts of the implemented system with different clock frequencies [31]. Their approach is applicable whenever multi-cycle operators can be clocked with a slower frequency without sacrificing performance. The algorithms used in the two aforementioned works are heuristic (i.e., simulated annealing for Chen et al. and a list-based scheduling for Lhairech-Lebreton et al.). To the best of the authors' knowledge, this is the first attempt to derive an optimal algorithm for HLS including all aspects of the generated microarchitecture, with particular emphasis on dynamic power optimization.

## 4 THE PROPOSED ALGORITHM

This section presents HerMan<sup>1</sup>, the proposed ILP model to generate optimized microarchitectures from model-based input descriptions with arbitrary throughput requirements.

### 4.1 The general idea

We propose to solve the ASBPC-P sequentially, as shown in Fig. 5. Whenever the ILP solver fails to provide a feasible solution for one or more of the optimizations, intermediate results can be used as fallback solutions, which are valid but non-optimal.

In the first step, we build an ILP model with a variable operator allocation to find a schedule for the given II, which can be implemented on a given back end (e.g., a PYNQ-Z1 FPGA board). If the ILP solver determines the model to be infeasible, there is no operator allocation which adheres to the back end resource constraints, that is able to support the requested II. In this case, the II is incremented and the updated model is solved, again, until a solution is found. In cases where the user-given II serves as a hard constraint, the solving process can alternatively be terminated early since the throughput requirement is incompatible with the model to be implemented. In these cases, a different back end (e.g., a larger FPGA) or another operator library can be chosen.

Once a minimal operator allocation for the given II is found, the number of functional units is fixed for all subsequent optimization steps (defined via  $F(\omega)$ ). This can be done under the assumption that no system exists with more operators but lower power consumption.

Afterwards, a new ILP formulation is built with the objective of energy savings maximization. To do so, the solver needs information about the number of clock domains  $\#C$ , the energy consumption for each operator per clock cycle  $\Lambda(\omega)$ , and the maximum latency increase per operator type

<sup>1</sup>HerMan: [Holistic ILP Model for Model-Based Hardware Design](#)

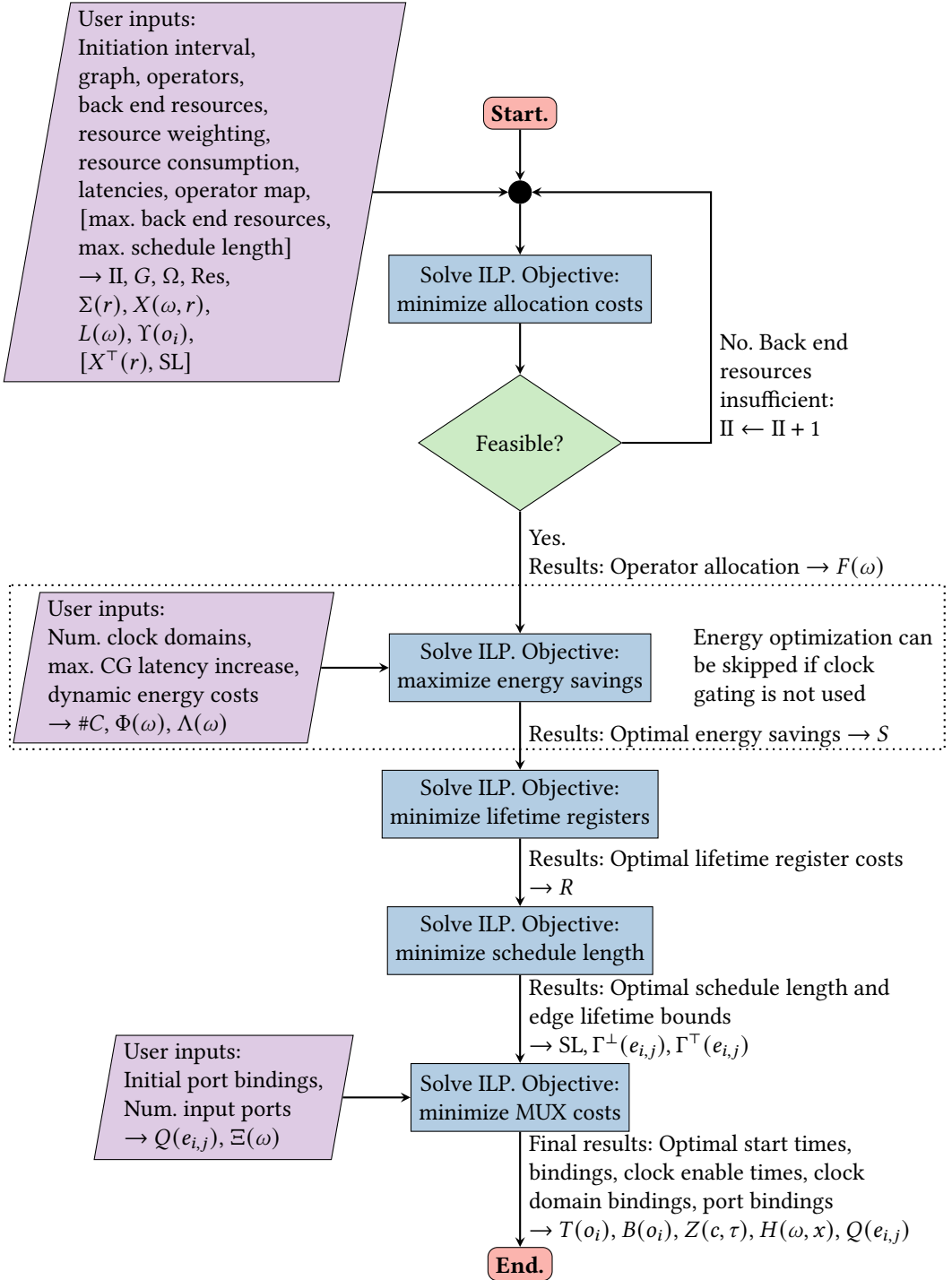


Fig. 5. Flowchart for HerMan

caused by clock gating  $\Phi(\omega)$ . It will be shown in Section 4.2 that  $\Phi(\omega)$  influences the number of variables/constraints needed to model the problem. A high value for  $\Phi(\omega)$  might lead to better results at the cost of a potentially longer solving time. Solving this ILP formulation yields the maximum dynamic energy savings that can be achieved by clock gating, denoted by  $S$ .

Now we can extend the ILP model to also account for lifetime register costs and solve it again with the objective of lifetime register minimization and an additional constraint that permits only solutions with optimal energy savings. This yields the optimal number of lifetime registers.

Our proposed ILP model can also account for MUX costs. To do so, we need the maximum lifetime for each edge  $\Gamma^T(e_{i,j})$ , which can be trivially computed for a bounded schedule length. This is why we solve two more ILP models after having computed optimal lifetime register costs: (i) a model to minimize the schedule length (subject to optimal energy savings and optimal lifetime register costs), and (ii) a model to minimize MUX costs (subject to optimal energy savings, optimal lifetime register costs, and the optimal schedule length). As additional inputs to minimize MUX costs, we need the number of input ports for each operator type,  $\Xi(\omega)$ , which can be obtained from the operator library, and a port mapping for each edge in the DFG,  $Q(e_{i,j})$ . Aside from resource reduction, Chen et al. have shown that MUXs contribute to high power consumption in FPGA-based implementations, so optimizing MUXs is necessary for low-power applications on FPGAs [8].

Finally, we obtain a solution with:

- (1) The minimal  $\Pi$  for the requested back end
- (2) An operator allocation with minimal resource consumption
- (3) Maximum energy savings by clock gating
- (4) Minimal lifetime register costs to store intermediate results
- (5) The minimal schedule length
- (6) Minimal MUX costs in the interconnect network

Note that the optimization performed is hierarchical, and the order in which objectives are optimized can be interchanged depending on the application (e.g., latency-sensitive applications can provide an upper bound for the schedule length and minimize the schedule length first). This means that changing the optimization order might also change individual values for the different objectives. Alternatively, our model can also be used to compute a weighted minimization of the objectives *energy*, *lifetime registers*, *schedule length*, and *MUXs*. To do so, the user would need to provide (meaningful) weighting factors, which is already a non-trivial task by itself.

## 4.2 ILP Formulation

Table 7 gives an overview of all variables of the proposed ILP formulation that are explained and referenced in the following. As described in the previous sections, we seek to solve a multi-criteria optimization problem. All objectives solved with the proposed ILP models are shown in Fig. 6 in (ξ1)–(ξ5) in the order defined by Fig. 5.

A minimization of allocation costs is achieved using (ξ1);  $\Sigma(r)$  defines a pre-defined, constant weighting factor for resource type  $r$  according to the back end used;  $X(\omega, r)$  defines how many resources of type  $r$  are consumed by implementing one instance of an operator with type  $\omega$ ; and  $\chi_\omega$  is an integer-valued variable that represents how many instances of operator type  $\omega$  are allocated to accomplish the  $\Pi$  requested. Note that the weighting factor  $\Sigma(r)$  can, for example, be used to assign equal “importance” to all resource types for heterogeneous back ends such as FPGAs, as shown in Table 8.

In order to correctly model the modulo scheduling problem we need to satisfy (5) for all data and chaining dependencies in the DFG [43]. This constraint states that an operation ( $o_j$ ) can only be started if all operations it depends on ( $o_i$ ) finished executing. By explicitly modeling all operations’

Table 7. Variables of the proposed ILP formulation

Name	Domain	Explanation
$\hat{t}$	$\in \mathbb{N}_0$	Schedule length
$t_i$	$\in \mathbb{N}_0$	Start time of $o_i$
$y_i$	$\in \mathbb{N}_0$	$y_i = \lfloor \frac{t_i}{\Pi} \rfloor$
$m_{i,\tau}$	$\in \{0, 1\}$	$m_{i,\tau} = 1 \iff t_i \bmod \Pi = \tau$
$\chi_\omega$	$\in \mathbb{N}_0$	Number of allocated operator instances of type $\omega$
$b_{i,x}$	$\in \{0, 1\}$	$b_{i,x} = 1 \iff o_i$ gets executed on operator instance $x$ of type $\omega$
$\sigma_{i,j}$	$\in \{0, 1\}$	$\sigma_{i,j} = 0 \implies \nexists \tau \in [0, \Pi) : m_{i,\tau} = 1 \wedge m_{j,\tau} = 1$
$\psi_{i,j}$	$\in \{0, 1\}$	$\psi_{i,j} = 0 \implies \nexists x \in [0, F(\omega)) : b_{i,x} = 1 \wedge b_{j,x} = 1$
$k_{\omega,x}$	$\in \mathbb{N}_0$	Number of lifetime registers after operator instance $x$ of type $\omega$
$f_{\omega,x,c}$	$\in \{0, 1\}$	$f_{\omega,x,c} = 1 \iff$ operator instance $x$ of type $\omega$ is connected to clock domain $c$
$z_{c,\tau}$	$\in \{0, 1\}$	$z_{c,\tau} = 1 \iff$ clock domain $c$ is turned off in modulo slot $\tau$
$h_{\omega,x,\tau}$	$\in \{0, 1\}$	$h_{\omega,x,\tau} = 1 \iff$ the clock for operator instance $x$ of type $\omega$ is turned off in modulo slot $\tau$
$\alpha_{\omega,x,\tau,\phi}$	$\in \{0, 1\}$	$\alpha_{\omega,x,\tau,\phi} = 1 \iff$ the additional latency of all operations executed on instance $x$ of operator type $\omega$ started in modulo slot $\tau$ is equal to $\phi$
$n_i$	$\in \mathbb{N}_0$	Additional latency of $o_i$
$\theta_{i,j,\gamma}$	$\in \{0, 1\}$	$\theta_{i,j,\gamma} = 1 \iff$ the lifetime on edge $e_{i,j}$ is equal to $\gamma$
$a_{\omega_i,x_i,\omega_j,x_j,\gamma,q}$	$\in \{0, 1\}$	$a_{\omega_i,x_i,\omega_j,x_j,\gamma,q} = 0 \implies$ there is <i>no</i> connection necessary between instance $x_i$ of operator type $\omega_i$ and input port $q$ at instance $x_j$ of operator type $\omega_j$ over exactly $\gamma$ lifetime registers
$\pi_{i,j,q}$	$\in \{0, 1\}$	$\pi_{i,j,q} \iff e_{i,j}$ is assigned to input port $q$ of the operator that executes $o_j$

$$\min \sum_{r \in \text{Res}} \Sigma(r) \cdot \sum_{\omega \in \Omega} X(\omega, r) \cdot \chi_\omega \quad (\xi 1)$$

$$\max \sum_{\omega \in \Omega} \Lambda(\omega) \cdot \sum_{x=0}^{F(\omega)-1} \sum_{\tau=0}^{\Pi-1} h_{\omega,x,\tau} \quad (\xi 2)$$

$$\min \sum_{\omega \in \Omega} \sum_{x=0}^{F(\omega)-1} k_{\omega,x} \quad (\xi 3)$$

$$\min \hat{t} \quad (\xi 4)$$

$$\min \sum_{(\omega_i,\omega_j,\gamma,q) \in A} \sum_{x_i=0}^{F(\omega_i)-1} \sum_{x_j=0}^{F(\omega_j)-1} a_{\omega_i,x_i,\omega_j,x_j,\gamma,q} \quad (\xi 5)$$

Fig. 6. Objectives for the proposed ILP formulation

$$\begin{array}{lll}
t_j - t_i \geq \Delta(e_{i,j}) - \Pi \cdot d_{i,j} & \forall e_{i,j} & \text{(T1)} \\
t_j - t_i - n_i \geq \Delta(e_{i,j}) - \Pi \cdot d_{i,j} & \forall e_{i,j} & \text{(T2)} \\
t_i - \Pi \cdot y_i - \sum_{\tau=0}^{\Pi-1} \tau \cdot m_{i,\tau} = 0 & \forall i & \text{(T3)} \\
\sum_{\tau=0}^{\Pi-1} m_{i,\tau} = 1 & \forall i & \text{(T4)} \\
t_i + L(Y(o_i)) \leq \text{SL} & \forall i & \text{(T5)} \\
t_i + L(Y(o_i)) + n_i \leq \text{SL} & \forall i & \text{(T6)} \\
t_i + L(Y(o_i)) + n_i - \hat{t} \leq 0 & \forall i & \text{(T7)}
\end{array}$$

Fig. 7. Timing constraints for the proposed ILP formulation

start times as integer-valued decision variables, we can directly express this constraint in ILP via (T1). Note that (T1) and its modified version for clock gating (T2) are the only ILP constraints relevant for chaining. Hence, all subsequent instances of “ $\forall e_{i,j}$ ” represent “ $\forall e_{i,j} \in E \setminus E_c$ ” instead of “ $\forall e_{i,j} \in E$ ” since they refer to edge lifetimes which relate to the original precedence constraint (3).

Unfortunately, operator constraints (2) require knowledge about the congruence class modulo  $\Pi$  for each operation. Our model for timing constraints is close to the model proposed by Eichenberger and Davidson [20] and shown in Fig. 7. We compute the decomposition of  $t_i$  into the modulo slot and an integer-valued multiple of the  $\Pi$  using (T3). Here,  $y_i$  is an integer-valued decision variable and  $m_{i,\tau}$  is a binary decision variable that represents whether  $t_i \bmod \Pi = \tau$ . Hence, we must make sure that only one  $m_{i,\tau}$  is active per operation via (T4). In presence of an optional user-given upper limit on the schedule length, we also add (T7) to the solver to make sure that the SL limit holds.

This allows us to model (2) for a variable operator allocation defined by  $\chi_\omega$  according to the proposal by Oppermann et al. [39] shown in Fig. 8. We use (A1) to make sure that the solver allocates enough operators of each type to accommodate the execution of all operations. In presence of resource limits due to the chosen back end, we also add (A2) to the solver. Finally, we follow Oppermann et al. [39] and add the redundant constraint (A3) to the solver to help proving optimality for a trivially minimal operator allocation according to (1). Note that this allocation is not necessarily feasible due to the interaction of operator and recurrence constraints. Hence, the solver might allocate more operators if limited operations are part of cyclic dependencies in the DFG or in presence of tight bounds for the schedule length.

From this point forward, we assume that the solver successfully computed the optimal allocation via Objective ( $\xi_1$ ). The next objective is to maximize energy savings for the given allocation. This is necessary because—even for the minimal operator allocation—it is possible that some operators are not fully utilized in each modulo slot. The ILP solver maximizes energy savings via ( $\xi_2$ ) where  $\Lambda(\omega)$  are dynamic energy costs per clock cycle for one instance of operator type  $\omega$ , which have to be determined before-hand by the user; and the binary-valued decision variable  $h_{\omega,x,\tau}$  denotes whether instance  $x$  of operator type  $\omega$  is turned *off* via clock gating in modulo slot  $\tau$ .

To correctly model energy savings, the formulation must be extended towards (i) a fixed allocation which was computed in the previous step, and (ii) a model for clock gating and its influence on energy

$$\begin{aligned} \chi_\omega - \sum_{o_i \in O_\omega} m_{i,\tau} &\geq 0 && \forall \omega, \tau && \text{(A1)} \\ \sum_{\omega \in \Omega} X(\omega, r) \cdot \chi_\omega &\leq X^\top(r) && \forall r && \text{(A2)} \\ \chi_\omega &\geq \left\lfloor \frac{|O_\omega|}{\Pi} \right\rfloor && \forall \omega && \text{(A3)} \end{aligned}$$

Fig. 8. Allocation constraints for the proposed ILP formulation

$$\begin{aligned} \sum_{x=0}^{F(\omega)-1} b_{i,x} &= 1 && \forall i && \text{(R1)} \\ m_{i,\tau} + m_{j,\tau} - \sigma_{i,j} &\leq 1 && \forall i, j, \tau && \text{(R2)} \\ b_{i,x} + b_{j,x} - \psi_{i,j} &\leq 1 && \forall i, j, x && \text{(R3)} \\ \sigma_{i,j} + \psi_{i,j} &\leq 1 && \forall i, j && \text{(R4)} \end{aligned}$$

Fig. 9. Operator constraints for the proposed ILP formulation

consumption and operator latency. The fixed allocation and the resulting operator constraints are shown in Fig. 9, and our clock gating model can be seen in Fig. 10.

For each operation we let the solver compute a binding to one of the allocated operators using (R1). Conflicts in the scheduling and binding arise whenever two operations are scheduled in the same modulo slot and shall be executed by the same operator instance. This case is impossible to implement because each operator can only execute one operation per clock cycle. It is prohibited by (R4), where  $\sigma_{i,j} = 0$  implies  $t_i \bmod \Pi \neq t_j \bmod \Pi$ ; and  $\psi_{i,j} = 0$  implies  $b_i \neq b_j$ . Hence, we must include (R2) and (R3) to ensure that  $\sigma_{i,j}$  and  $\psi_{i,j}$  are correctly set to one whenever  $t_i \bmod \Pi = t_j \bmod \Pi$ , and  $b_i = b_j$ , respectively.

In the proposed clock gating model we compute a clock domain assignment for each operator instance via (C1); binary decision variable  $f_{\omega,x,c}$  denotes whether operator instance  $x$  of type  $\omega$  is connected to gated clock domain  $c$ . We also allow the assignment to no gated clock via the “ $\leq$ ” relation instead of “ $=$ ”. If  $f_{\omega,x,c} = 0 \forall c$  then operator instance  $x$  of type  $\omega$  is connected to the default clock domain which is active in all modulo slots. We use binary variables  $z_{c,\tau}$  to represent that clock domain  $c$  is turned *off* in modulo slot  $\tau$  and  $h_{\omega,x,\tau}$  to represent that the clock for operator instance  $x$  of type  $\omega$  is turned off in modulo slot  $\tau$ . If  $f_{\omega,x,c} = 1$  then, by definition,  $z_{c,\tau} = h_{\omega,x,\tau}$ . This relationship can be realized via the use of the indicator constraint (C2<sup>†</sup>). Alternatively, if the solver does not support indicator constraints, (C2a\*) and (C2b\*), can be used, instead.

Consider the case where all  $f_{\omega,x,c}$  are set to zero to indicate a connection to the default clock domain. This means that all indicator constraints (C2<sup>†</sup>) are disabled, which would mean that the solver would be free to choose any values for the corresponding  $h_{\omega,x,\tau}$  variables, although, by definition, all  $h_{\omega,x,\tau}$  should be zero because the default clock domain is active in all modulo slots.

We enforce this behavior using (C3). It only allows to set  $h_{\omega,x,\tau} = 1$  if at least one  $f_{\omega,x,c}$  is also set to one, indicating a binding to a gated clock.

Recall that the idea of incorporating clock gating into modulo scheduled circuits is to reduce dynamic power for clock cycles in which an operator instance is idle. We say that an operator instance is idle if it does not start the execution of a new operation in a given modulo slot. Consequently, clock gating can only be allowed for a given operator instance for modulo slots in which no operation is executed on that instance. This property is enforced by Constraint (C4).

Finally, we must model a possible latency increase for operations executed on operator instances which are driven by a gated clock. Consider the example where an operator instance  $\hat{x}$  of type  $\hat{\omega}$  has a latency of three clock cycles (i.e., it is realized with three pipeline stages in order to comply with a specific clock frequency), is bound to a gated clock  $\hat{c}$  (i.e.,  $f_{\hat{\omega},\hat{x},\hat{c}} = 1$ ), and  $\Pi = 2$ . If this clock is turned off in modulo slot  $\tau = 0$  and remains active in modulo slot  $\tau = 1$  (i.e.,  $z_{\hat{c},0} = 1$ ,  $z_{\hat{c},1} = 0$ ), then  $h_{\hat{\omega},\hat{x},0} = 1$  and  $h_{\hat{\omega},\hat{x},1} = 0$  caused by (C2<sup>†</sup>). Now also assume that a specific operation  $o_i$  is scheduled in  $t_i = 3$ . This corresponds to modulo slot  $\tau = 1$  (i.e.,  $m_{i,1} = 1$ ). If it is also bound to the given operator instance (i.e.,  $b_{i,\hat{x}} = 1$ ), the operation traverses the operator's first pipeline stage in clock cycle 3. In clock cycle 4, however, the operator instance's clock is turned off, so the second pipeline stage is traversed only in clock cycle 5 and the operator finishes computation in clock cycle 7. This means that the latency of that operation is increased by two.

To model such behavior, we use an additional integer-valued decision variable  $n_i$  per operation which holds the clock gating-induced latency increase. Clearly, that value depends on (i) the operator instance on which the operation is executed (via  $b_{i,x}$ ), (ii) which clock domain drives it (via  $f_{\omega,x,c}$ ), (iii) when this clock domain is turned on/off (via  $z_{c,\tau}$ ), and (iv) in which modulo slot the operation is scheduled (via  $m_{i,\tau}$ ).

We use the binary-valued decision variable  $\alpha_{\omega,x,\tau,\phi}$  to denote whether the latency increase of any operation started in modulo slot  $\tau$  being executed on instance  $x$  of operator type  $\omega$  is equal to  $\phi$ . Obviously, only one of these variables must be active over all possible values that  $\phi$  can take. This is enforced by (C8).

The mapping between  $\alpha_{\omega,x,\tau,\phi}$  and  $h_{\omega,x,\tau}$  is achieved via indicator constraints (C5<sup>†</sup>) or, alternatively, via linear counterparts (C5a\*)–(C5b\*). Reasoning behind (C5<sup>†</sup>) is that after  $\phi + L(\omega) - 1$  clock cycles we accumulated exactly  $\phi$  clock cycles in which the operator instance's clock is turned off. However, it is important that the sum's last element in (C5<sup>†</sup>) is a variable with value zero. Otherwise, the value for  $\alpha_{\omega,x,\tau,\phi}$  could be assigned an incorrect value which would still satisfy these constraints. Therefore, we also add (C7).

Finally, we must also link  $\alpha_{\omega,x,\tau,\phi}$  with  $n_i$  (i.e., set  $n_i = \phi$ ) for all operations executed on instance  $x$  which are started in modulo slot  $\tau$ . This is achieved via indicator constraint (C6<sup>†</sup>), or, correspondingly, linear constraints (C6a\*)–(C6b\*).

Obviously, the latency increase  $n_i$  must also be considered in precedence and schedule length constraints. Therefore, we modify (T1) to (T2) and (T5) to (T6) when modeling clock gating.

For all subsequent optimization runs, we must make sure that we only allow solutions with optimal energy savings (i.e., with an optimal objective value for ( $\xi$ 2)). This is enforced by recording the objective value  $S$  when maximizing energy savings and adding (C9) to the solver for all subsequent optimizations.

The latency increase caused by clock gating can influence the number of lifetime registers needed to implement the circuit by modifying lifetimes for intermediate results. We can exploit this by starting an optimization run to minimize lifetime registers while simultaneously modeling clock gating. The appropriate constraints are shown in Fig. 11 and set up by modifying the model proposed by Sittel et al. [45] to also account for clock gating. The model uses integer-valued decision

$$\begin{aligned}
& \sum_{c=0}^{\#C-1} f_{\omega,x,c} \leq 1 & \forall \omega, x & \quad (C1) \\
& f_{\omega,x,c} \implies z_{c,\tau} - h_{\omega,x,\tau} = 0 & \forall \omega, x, \tau, c & \quad (C2^\dagger) \\
& f_{\omega,x,c} + z_{c,\tau} - h_{\omega,x,\tau} \leq 1 & \forall \omega, x, \tau, c & \quad (C2a^*) \\
& f_{\omega,x,c} + h_{\omega,x,\tau} - z_{c,\tau} \leq 1 & \forall \omega, x, \tau, c & \quad (C2b^*) \\
& \sum_{c=0}^{\#C-1} f_{\omega,x,c} - h_{\omega,x,\tau} \geq 0 & \forall \omega, x, \tau & \quad (C3) \\
& h_{\omega,x,\tau} + m_{i,\tau} + b_{i,x} \leq 2 & \forall \omega, x, \tau, i & \quad (C4) \\
& \alpha_{\omega,x,\tau,\phi} \implies \sum_{\varepsilon=0}^{\phi+L(\omega)-1} h_{\omega,x,(\tau+\varepsilon) \bmod \Pi} = \phi & \forall \omega, x, \tau, \phi & \quad (C5^\dagger) \\
& L(\omega) \cdot (1 - \alpha_{\omega,x,\tau,\phi}) + \sum_{\varepsilon=0}^{\phi+L(\omega)-1} h_{\omega,x,(\tau+\varepsilon) \bmod \Pi} \geq \phi & \forall \omega, x, \tau, \phi & \quad (C5a^*) \\
& -L(\omega) \cdot (1 - \alpha_{\omega,x,\tau,\phi}) + \sum_{\varepsilon=0}^{\phi+L(\omega)-1} h_{\omega,x,(\tau+\varepsilon) \bmod \Pi} \leq \phi & \forall \omega, x, \tau, \phi & \quad (C5b^*) \\
& (b_{i,x} \wedge m_{i,\tau}) \implies n_i - \sum_{\phi=0}^{\Phi(\omega)} \phi \cdot \alpha_{\omega,x,\tau,\phi} = 0 & \forall \omega, x, \tau, i & \quad (C6^\dagger) \\
& \Phi(\omega) \cdot (2 - b_{i,x} - m_{i,\tau}) + n_i - \sum_{\phi=0}^{\Phi(\omega)} \phi \cdot \alpha_{\omega,x,\tau,\phi} \geq 0 & \forall \omega, x, \tau, i & \quad (C6a^*) \\
& -\Phi(\omega) \cdot (2 - b_{i,x} - m_{i,\tau}) + n_i - \sum_{\phi=0}^{\Phi(\omega)} \phi \cdot \alpha_{\omega,x,\tau,\phi} \leq 0 & \forall \omega, x, \tau, i & \quad (C6b^*) \\
& \alpha_{\omega,x,\tau,\phi} + h_{\omega,x,(\tau+\phi+L(\omega)-1) \bmod \Pi} \leq 1 & \forall \omega, x, \tau, \phi & \quad (C7) \\
& \sum_{\phi=0}^{\Phi(\omega)} \alpha_{\omega,x,\tau,\phi} = 1 & \forall \omega, x, \tau & \quad (C8) \\
& \sum_{\omega \in \Omega} \Lambda(\omega) \cdot \sum_{x=0}^{F(\omega)-1} \sum_{\tau=0}^{\Pi-1} h_{\omega,x,\tau} \geq S & & \quad (C9)
\end{aligned}$$

Fig. 10. Clock gating constraints for the proposed ILP formulation

variables  $k_{\omega,x}$  to compute how many lifetime registers are associated to operator instance  $x$  of type  $\omega$ . Obviously, this value must be at least as large as the longest lifetimes of all variables produced by this instance. This is enforced by indicator constraint (L1<sup>†</sup>), or, alternatively, by its linearization (L1<sup>\*</sup>). An optimal lifetime register objective value for following optimizations can be imposed by (L2).

$$\begin{aligned}
b_{i,x} &\implies k_{\omega,x} - t_j + t_i + n_i \geq d_{i,j} \cdot \Pi - L(\Upsilon(o_i)) && \forall e_{i,j}, x && (L1^\dagger) \\
M \cdot (1 - b_{i,x}) + k_{\omega,x} - t_j + t_i + n_i &\geq d_{i,j} \cdot \Pi - L(\Upsilon(o_i)) && \forall e_{i,j}, x && (L1^*) \\
\sum_{\omega \in \Omega} \sum_{x=0}^{F(\omega)-1} k_{\omega,x} &\leq R && && (L2)
\end{aligned}$$

Fig. 11. Lifetime register constraints for the proposed ILP formulation ( $M$  is a big- $M$  constant; if  $\Gamma^\top$  is known, we can set  $M = \Gamma^\top(e_{i,j})$ ; otherwise we can use a max schedule length estimation, e.g., proposed by Oppermann et al. [38])

$$\begin{aligned}
t_i &= 0 && \forall i \in \text{Inputs} && (Q1) \\
t_i + L(\Upsilon(o_i)) - \hat{t} &= 0 && \forall i \in \text{Outputs} && (Q2) \\
t_i + L(\Upsilon(o_i)) + n_i - \hat{t} &= 0 && \forall i \in \text{Outputs} && (Q3)
\end{aligned}$$

Fig. 12. Input/output equalizing constraints for the proposed ILP formulation

When embedding a circuit within a larger system, it might be necessary that all inputs are put into the circuit at the same time and all outputs are produced at the same time in order to avoid implementing additional registers. HerMan directly supports that scenario within its optimization, as this additional design constraint might influence the number of lifetime registers after the corresponding inputs or before the outputs. Fig. 12 shows the ILP constraints to model equalized inputs and outputs. Here, it is assumed that the user provides a set of operations that should be considered as inputs and an additional set for outputs. Constraint (Q1) states that all inputs to the model are scheduled in clock cycle  $t = 0$ , and, depending on whether clock gating is also optimized, (Q2) or (Q3) states that output operations always provide their output values as late as possible. Note that, depending on user-requirements, input scheduling times different from zero are also possible, by changing (Q1) to, for example,  $t_2 = 3$  if  $o_2$  should be scheduled in time step 3. The same can be applied to output schedule times.

The last component in the underlying microarchitecture (Fig. 2), which is not yet optimized, are the multiplexers used to implement the interconnect network between operators and/or ports. An ILP model for MUX optimization, given a schedule, was proposed by Fiege et al. [23]. It also uses commutativity to modify the port assignments for even further MUX reductions. However, it was noted that the schedule can heavily influence the optimization potential in binding. We therefore also include a MUX optimization and a port assignment in our holistic ILP model. Corresponding constraints are shown in Fig. 13. The idea is to model all possible connections between operators in the resulting microarchitecture and let the solver exploit the potential to implement only one connection for the data transfers of multiple edges in the DFG. In the worst case, each edge in the DFG must be implemented using a distinct connection in the circuit. Two edges can share a connection if and only if (i) source operator instances are equal, (ii) destination operator instances are equal, (iii) variable lifetimes are equal, and—for non-commutative operator types—(iv) input ports on the destination operator instance are equal. We use  $a_{\omega_i, x_i, \omega_j, x_j, Y, Q}$  to model whether a

connection from instance  $x_i$  of operator type  $\omega_i$  to input port  $q$  of instance  $x_j$  of operator type  $\omega_j$  over  $\gamma$  lifetime registers will be implemented in hardware. Additionally, we compute a port assignment for commutative operations via  $\pi_{i,j,q}$ . Value  $\pi_{i,j,q} = 1$  indicates that edge  $e_{i,j}$  is connected to port  $q$  of the operator instance which executes  $o_j$ . A port assignment is valid if and only if each incoming edge is assigned to exactly one port and if each port is assigned exactly one incoming edge. This is realized via (M5)–(M6). In the work by Fiege et al. [23], the lifetime for each edge is assumed to be known prior to building the ILP model (i.e., it is built for a given, pre-computed schedule). This is not possible in the present work because scheduling and binding are solved together. Therefore, we create binary-valued variables  $\theta_{i,j,\gamma}$  that represent whether the edge  $e_{i,j}$  has a lifetime of  $\gamma$ . The assignment of exactly one of those variables per edge is enforced via (M2), and the appropriate one is chosen via (M1). Here, (M1) realizes the precedence constraint (3). For each edge, we must make sure that the ILP model correctly accounts for the appropriate connection that must be implemented in hardware in order to realize the data transfer between the operators. To do so, we use variables  $a_{\omega_i, x_i, \omega_j, x_j, \gamma, q}$ , and constraints (M3)–(M4) for non-commutative and commutative operations, respectively. These constraints realize the relation “if  $o_i$  is bound to instance  $x_i$ ,  $o_j$  is bound to  $x_j$ , the lifetime for edge  $e_{i,j}$  is equal to  $\gamma$  (and the edge is assigned to port  $q$ ), then the solver must account for that connection by setting the appropriate  $a_{\omega_i, x_i, \omega_j, x_j, \gamma, q}$  variable to one.”

Summing up all  $a_{\omega_i, x_i, \omega_j, x_j, \gamma, q}$  variables therefore yields the total number of multiplexer inputs in the implemented system. Assuming a linear relationship between the number of MUX inputs and their costs (which is approximately true for FPGA designs [35] and therefore also done in previous work related to binding [23, 44]), minimizing that sum in (ξ5) leads to a minimization of MUX costs caused by the interconnect network shown in Fig. 2.

Even though we do not make use of that feature in our work, it is possible to enforce a specific objective value for subsequent optimization runs via (M7). We use the helper set comprised of 4-tuples

$$A = \{(\omega_i, \omega_j, \gamma, q) \mid \exists e_{i,j} \in E : \omega_i = \Upsilon(o_i), \omega_j = \Upsilon(o_j), \Gamma^+(e_{i,j}) \leq \gamma \leq \Gamma^-(e_{i,j}), \\ ((q = Q(e_{i,j}) \wedge \omega_j \notin \check{\Omega}) \vee (q \in [0, \Xi(\omega_j)] \wedge \omega_j \in \check{\Omega}))\} \quad (7)$$

to store the information between which operator types/ports, and with which lifetimes there might be implemented a connection by the ILP solver. This information can be used to delete unnecessary variables (i.e., those  $a_{\omega_i, x_i, \omega_j, x_j, \gamma, q}$  which cannot be set to one because they represent connections that will never be implemented in hardware).

Finally, note again that the proposed ILP formulation assumes that operators are fully pipelined (e.g., (A1) assumes that an operation only utilizes exactly one modulo slot, and (C5<sup>†</sup>) assumes that an operation traverses one pipeline stage per clock cycle). Enabling multi-cycle operations (i.e., operations with  $\Pi > 1$ ) and/or nested loops requires re-working multiple aspects of our proposed variables/constraints and is therefore out of this work’s scope.

## 5 EXPERIMENTAL RESULTS

We conduct all our experiments on an AMD-EPYC 7443P CPU @ 3.77 GHz. ILP solvers are allowed to use up to 20 threads with a time limit of ten minutes per problem instance. Our FPGA back end model for the PYNQ-Z1 board<sup>2</sup> is given in Table 8. We choose weighting factors such that the weighted number of available units is approximately constant for the different resource types. The operator model (cf. Table 9) is acquired by synthesis experiments for the PYNQ-Z1 board. For simplicity we assume that  $\Lambda(\omega) = 1 \forall \omega \in \Omega$ . This results in an equal weighting of all arithmetic operator types regarding their energy consumption per clock cycle. More accurate results could

<sup>2</sup>The board features a XC7Z020-1CLG400C FPGA from the Zynq-7000 family.

$$\begin{aligned}
& \sum_{\gamma \in \Gamma^\perp(e_{i,j})} \gamma \cdot \theta_{i,j,\gamma} - t_j + t_i + n_i = \Pi \cdot d_{i,j} - L(\Upsilon(o_i)) \quad \forall e_{i,j} & (M1) \\
& \sum_{\gamma \in \Gamma^\perp(e_{i,j})} \theta_{i,j,\gamma} = 1 \quad \forall e_{i,j} & (M2) \\
& b_{i,x_i} + b_{j,x_j} + \theta_{i,j,\gamma} - a_{\Upsilon(o_i),x_i,\Upsilon(o_j),x_j,\Upsilon(o_j),x_j,\Upsilon(o_j),x_j,q}(e_{i,j}) \leq 2 \quad \forall e_{i,j}, x_i, x_j, \gamma : \omega_j \notin \check{\Omega} & (M3) \\
& b_{i,x_i} + b_{j,x_j} + \theta_{i,j,\gamma} + \pi_{i,j,q} - a_{\Upsilon(o_i),x_i,\Upsilon(o_j),x_j,\Upsilon(o_j),x_j,\Upsilon(o_j),x_j,q}(e_{i,j}) \leq 3 \quad \forall e_{i,j}, x_i, x_j, \gamma, q : \omega_j \in \check{\Omega} & (M4) \\
& \sum_{i: \exists e_{i,j} \in E} \pi_{i,j,q} = 1 \quad \forall q, o_j : \Upsilon(o_j) \in \check{\Omega} & (M5) \\
& \sum_{q=0}^{\Xi(\Upsilon(o_j))} \pi_{i,j,q} = 1 \quad \forall o_j, e_{i,j} : \Upsilon(o_j) \in \check{\Omega} & (M6) \\
& \sum_{(\omega_i, \omega_j, \gamma, q) \in A} \sum_{x_i=0}^{F(\omega_i)-1} \sum_{x_j=0}^{F(\omega_j)-1} a_{\omega_i, x_i, \omega_j, x_j, \gamma, q} \leq P & (M7)
\end{aligned}$$

Fig. 13. Multiplexer constraints for the proposed ILP formulation with  $A$  defined in (7)Table 8. FPGA back end model (PYNQ-Z1 board) with  $\Sigma(r)$  chosen such that  $X^\top(r) \cdot \Sigma(r) \approx \text{const.} \forall r \in \text{Res.}$ 

$r \in \text{Res}$	LUTs	FFs	DSPs
$X^\top(r)$	53200	106400	220
$\Sigma(r)$	2	1	483.73

be obtained by using the actual values for energy consumption. But, as seen in the forthcoming evaluation, this simplification already results in dynamic power consumption savings for the resulting circuits. We use BUFGE primitives to realize the clock buffers with enable inputs, which operate in a synchronous fashion. This means that for a low clock-enable value before an incoming rising edge (in order to stop the edge from propagating to the buffer output), any value changes on the clock-enable port during the clock's high-time are ignored by the buffer and, therefore, do not lead to any unwanted clock pulses at the output.

We evaluate our proposed optimization with two different settings:

- (1) HerMan: As depicted in Fig. 5, *including* the energy optimization by clock gating
- (2) HerMan (no ( $\xi$ 2)): As depicted in Fig. 5, but *without* the use of clock gating

We choose benchmark instances from digital signal processing applications, consisting of digital filters and color space conversions.

With the following experiments we seek to answer the following questions:

Table 9. Operator model (results taken from synthesis experiments for the PYNQ-Z1 board)

$\omega \in \Omega$	FP32-Add	FP32-Sub	FP32-Mult	FP32-Div
$L(\omega)$	7	7	2	12
$X(\omega, \text{LUTs})$	343	343	45	1589
$X(\omega, \text{FFs})$	292	292	68	624
$X(\omega, \text{DSPs})$	0	0	2	0
$\Lambda(\omega)$	1	1	1	1
$\Xi(\omega)$	2	2	2	2

- (1) Can clock gating be effectively applied to modulo scheduled circuits in order to reduce power consumption?
- (2) How much computational overhead is necessary to do so?
- (3) Does the incorporation of clock gating also have an impact on other system properties (e.g., the number of lifetime registers)?

We perform three experimental evaluations to answer these questions:

- (1) Pre-synthesis experiments for the whole benchmark suite (Section 5.1): We run Origami-HLS for each benchmark instance and various throughput requirements and evaluate abstract parameters: (i) CPU time to obtain the results, (ii) ratio of *useful* number of operations to *total* number of operations per data sample, (iii) number of lifetime registers, (iv) multiplexer costs.
- (2) Post-place-and-route experiments for selected benchmark instances (Section 5.2): We implement selected VHDL codes generated with Origami-HLS on the PYNQ-Z1 board and evaluate FPGA resource consumption.
- (3) Power measurements on the PYNQ-Z1 board: We create bit streams for selected VHDL designs including a random number generator for input stimulus generation and measure the system's static and dynamic power consumption.

Recall that the proposed approach is developed such that design space explorations become obsolete due to the fact that the user can specify the exact requirements for the resulting circuit. Nevertheless, we perform design space explorations for all benchmark designs to show HerMan's general applicability independent of the actual design constraints. This also means that we have to limit the timeout to a comparatively short number, namely  $t = 10$  min per ILP instance caused by the high number of problems solved. Note that in a practical scenario, significantly more time can be allocated to the optimization procedure since only a single circuit must be generated per system instead of multiple ones. We choose  $\Pi = 32$  as an upper limit for the  $\Pi$  in the design space exploration, since it is larger than the value for  $\Pi_{\text{rec}}^{\pm}$  for all models tested and increasing it further leads to unrepresentative tests due to much smaller values for  $\Pi_{\text{res}}^{\pm}$  in some instances.

Practical values for the number of available clock domains depends on the back end and also on the model to be implemented. On the one hand, a high number of clock domains increases the degrees of freedom for optimization. But on the other hand, using a new clock domain can also lead to an offset in switching activity and, hence, in power consumption. Therefore, we make tests with up to 2, 5 and 10 clock domains. The last input parameter to the ILP model is the maximum additional latency per operator type  $\Phi(\omega)$ . Large values lead to higher switching activity reductions at the cost of an increased ILP complexity and, due to the increased operator latency, a possibly increased schedule length. In the following tests we vary  $\Phi(\omega)$  between 2, 5, 10 and 20 for all operator types.

We implemented HerMan in the open-source C++ scheduling library HatScheT [47] using the open-source ScaLP library [48] as an interface to the Gurobi [26] ILP solver. Note that we chose Gurobi only for performance reasons. The proposed formulation is also compatible with any other standard ILP solver such as CPLEX [27], SCIP [4] or LPSolve [3]. VHDL codes are generated via the open-source tool Origami HLS [37] using FloPoCo [18] as an open-source back end/operator library. All benchmark instances are available as part of the Origami HLS tool.

As a comparison to our holistic approach we use two different approaches from the current state-of-the-art for static scheduling in HLS:

- Scheduling for the given  $\Pi$  with the second objective of schedule length minimization using BLOOP, a Boolean Satisfiability-based scheduler [24]. Results are passed to an optimal binding algorithm using ILP for joint lifetime register and multiplexer optimization [23]. This is called SAT+ILP in the following
- Joint scheduling and binding for the given  $\Pi$  with the second objective of lifetime register minimization using the enhanced Moovac ILP formulation [45]

The remaining HLS flow remains the same. Moovac, BLOOP, and the ILP-based binding algorithm are also implemented in HatScheT, use Gurobi for solving ILP instances, interfaced via ScaLP, and use CaDiCaL for solving SAT instances. Again, Origami & FloPoCo are used for VHDL code generation.

We choose these approaches for the following reasons:

- (1) SAT+ILP represents the traditional, non-holistic approach, where allocation, scheduling and binding are solved optimally, but individually.
- (2) The lifetime-aware version of Moovac is one step further to a holistic approach, since it computes a binding-aware schedule for lifetime register minimization. However, it still neglects MUX costs and energy consumption.

Contrary to HerMan, the other two approaches need a fixed operator allocation as an additional input, since they are not aware of back end resources. For each  $\Pi$  in the design space exploration we therefore allocate the minimum number of operators per type according to Oppermann et al. [39]:

$$F(\omega) = \left\lceil \frac{|O_\omega|}{\Pi} \right\rceil. \quad (8)$$

This ensures that  $\Pi_{\text{res}}^\perp$  is always equal to or smaller than the target  $\Pi$ . However, in presence of recurrences, that operator allocation is not necessarily feasible for the given  $\Pi$  due to the interaction of resource and cyclic dependency constraints [39]. In such cases, a scheduler for dynamic operator allocation like HerMan or the ones by Šůcha and Hanzálek [52] or Oppermann et al. [39] are necessary to find an allocation with minimal resource overhead. However, this case only rarely occurred in our experiments. In such cases, the scheduler fails to find a solution for such a problem instance and instead returns a schedule for the first feasible solution with a larger  $\Pi$ .

We perform the following tests after reaching different design stages to ensure that all circuits are generated correctly:

- (1) Each schedule is validated by checking (2) and (3) for the candidate  $\Pi$  and operator allocation.
- (2) Each binding is checked for correctness by validating whether the algorithm accounted for each edge in the graph by an appropriate connection in the circuit and the correct amount of lifetime registers.
- (3) Each generated VHDL code (including all clock buffers inserted by HerMan) is validated via a behavioral simulation that checks whether it behaves exactly the same—bit per bit—as its Simulink reference (based on a testbench, which is also built in Simulink).

- (4) Each implementation is checked for correctness by verifying that Vivado yields a positive *Worst Negative Slack* (i.e., the timing constraint is met and Vivado was successful at balancing any clock skews produced by the BUFGCE instances).
- (5) Selected implementations are validated for correctness by running them on the PYNQ-Z1 board and letting the on-board CPU core check the output computed by the FPGA.

## 5.1 Pre-synthesis

As a first attempt to answer the given research questions, we evaluate HerMan on the full benchmark suite and analyze pre-synthesis system parameters, namely the ratio of *useful* operations to *total* operations per data sample. An ideal value for this ratio would be 1, as then each operation performed in hardware is a useful operation which is necessary to process the data sample. If no clock gating is applied, each operator performs an operation in each clock cycle. Hence, the number of total operations is given by multiplying the  $\Pi$  with the total number of allocated operators. Clock gating can reduce this number by turning off the clock for an operator in cycles when it should not perform an operation. The number of useful operations is the total number of operations in the model to be implemented. For the example given in Fig. 1 the number of useful operations is 8 (4 multiplications, 3 additions, 1 subtraction). Furthermore, we evaluate the number of lifetime registers to see how many additional flip-flops must be implemented to store intermediate results. The interconnect network complexity depends on the total number of multiplexers and their sizes. As multiplexer resource consumption roughly scales linearly with the number of data inputs [35], we follow previous work [23, 44] and choose the total number of MUX inputs as a metric to estimate their costs (e.g., a system with one 3:1 MUX and two 4:1 MUXs would have  $\# \text{MUXs} = 3 + 2 \cdot 4 = 11$  in the following evaluation).

Results are given in Table 10. Interestingly, Moovac fails to solve the ILP for the majority of resource allocations for `fir_SAM`, while HerMan is always able to compute the minimal resource allocation and finds schedules with significantly fewer lifetime registers compared to the ones where Moovac does not time out. This suggests that manually allocating the minimal number of operators results in harder ILP instances than letting the solver figure it out itself. As expected, the approach of separate scheduling and binding (SAT+ILP) has by far the shortest average runtime (while only using one thread) paired with the highest lifetime register and multiplexer costs. This is also true for `iir_sos8`, where Moovac mostly finds valid but non-optimal solutions. For this special case the SAT+ILP approach has reduced resource consumption compared to Moovac. We exclude all problem instances from our analysis where one of the approaches fails to find a valid solution.

We see that applying clock gating reduces the number of operations performed for all benchmark instances. This comes at a (usually slight) cost increase in Lifetime registers. Multiplexer costs are usually reduced compared to the rest, since HerMan explicitly minimizes MUX costs whereas the others do not. Aside from a few outliers, HerMan without clock gating usually has the lowest LR and MUX count across the whole benchmark suite. Intuitively, one would expect HerMan and Moovac to yield identical results for the number of lifetime registers. But, since Moovac does not explicitly account for input/output schedule time equalization within its ILP model (see Fig. 12) and due to non-optimal results, HerMan sometimes finds cheaper implementations. Additionally, MUX costs are always minimal for HerMan.

Although HerMan usually does not need the full 10 min time per ILP instance, the total time is greatly increased compared to the other approaches. Especially the holistic optimization including clock gating increases runtime significantly, even compared to HerMan without clock gating. For larger benchmark sizes, HerMan frequently times out in one of its optimization steps, yielding no solution within ten minutes. For example, for `iir_sos8`, HerMan often fails to find any valid clock gating pattern, so the microarchitecture is created without any clock gating and the following

Table 10. Pre-synthesis results; all numbers reported are averaged over all IIs and all clock gating settings ( $\#C$  and  $\Phi(\omega)$ ) for the given benchmark model; best results are highlighted in **bold**

Benchmark			HerMan				HerMan (no ( $\xi$ 2))				Moovac [45]				SAT [24] + ILP [23]			
Name	$\Pi_{rec}^\perp$	O	avg. $\frac{\# \text{useful op.}}{\# \text{total op.}}$	avg. # LRs	avg. # MUXs	avg. time [s]	avg. $\frac{\# \text{useful op.}}{\# \text{total op.}}$	avg. # LRs	avg. # MUXs	avg. time [s]	avg. $\frac{\# \text{useful op.}}{\# \text{total op.}}$	avg. # LRs	avg. # MUXs	avg. time [s]	avg. $\frac{\# \text{useful op.}}{\# \text{total op.}}$	avg. # LRs	avg. # MUXs	avg. time [s]
iir_biqu	16	14	<b>0.36</b>	79	15	1235	0.17	<b>48</b>	<b>13</b>	<b>0</b>	0.17	50	15	<b>0</b>	0.17	58	15	<b>0</b>
fir6dlms	7	16	<b>0.35</b>	341	14	1427	0.19	<b>248</b>	<b>11</b>	1	0.19	302	14	<b>0</b>	0.19	281	14	<b>0</b>
iir4	16	26	<b>0.42</b>	548	29	1690	0.23	<b>65</b>	<b>25</b>	1	0.23	182	27	<b>0</b>	0.23	243	<b>25</b>	<b>0</b>
fir_GM	1	16	<b>0.70</b>	323	18	798	0.39	217	<b>16</b>	1	0.39	217	17	<b>0</b>	0.39	<b>200</b>	17	<b>0</b>
iir_sos2	16	26	<b>0.64</b>	171	25	1441	0.35	<b>44</b>	<b>19</b>	1	0.35	82	28	<b>0</b>	0.35	88	27	<b>0</b>
fir_SAM	1	121	<b>0.89</b>	1244	146	1790	0.85	<b>1090</b>	139	1193	0.85	2136	161	600	0.85	2384	<b>131</b>	<b>316</b>
iir_sos4	16	50	<b>0.95</b>	367	45	1308	0.70	<b>44</b>	<b>29</b>	16	0.70	282	50	4	0.70	302	46	<b>0</b>
iir_sos8	16	98	<b>0.74</b>	254	58	1625	0.73	<b>44</b>	52	1023	0.73	1189	83	565	0.73	254	<b>46</b>	<b>1</b>
fir_SHI	1	29	<b>0.91</b>	290	31	1309	0.58	<b>258</b>	<b>24</b>	50	0.58	358	31	2	0.58	376	29	<b>0</b>
rgb_tr	1	24	<b>0.58</b>	468	21	1209	0.34	<b>74</b>	<b>16</b>	<b>0</b>	0.34	222	21	<b>0</b>	0.34	221	21	<b>0</b>
fir_gen	1	15	<b>0.59</b>	47	14	735	0.33	<b>35</b>	<b>12</b>	<b>0</b>	0.33	40	<b>12</b>	<b>0</b>	0.33	87	13	<b>0</b>
splin_pf	1	26	<b>0.84</b>	255	26	1289	0.54	<b>186</b>	<b>21</b>	46	0.54	215	<b>27</b>	1	0.54	336	22	<b>0</b>
fir_hilb	1	14	<b>0.44</b>	218	14	1109	0.25	<b>178</b>	<b>11</b>	<b>0</b>	0.25	203	13	<b>0</b>	0.25	205	12	<b>0</b>
ycbcr_tr	1	22	<b>0.50</b>	140	18	1210	0.29	<b>24</b>	<b>15</b>	1	0.29	59	19	<b>0</b>	0.29	71	19	<b>0</b>
fir_lms	27	15	<b>0.14</b>	114	<b>11</b>	951	0.10	<b>103</b>	<b>11</b>	<b>0</b>	0.10	120	12	<b>0</b>	0.10	125	12	<b>0</b>
fir_srg	1	17	<b>0.37</b>	172	11	1164	0.21	<b>93</b>	<b>10</b>	<b>0</b>	0.21	110	12	<b>0</b>	0.21	117	11	<b>0</b>

optimizations (e.g., lifetime register minimization) do not consider clock gating. In these cases, more computation power (more CPU cores or longer timeout) would be needed to leverage HerMan's full potential.

## 5.2 Post-place-and-route

As our second set of experiments we generate post-place-and-route implementations for the PYNQ-Z1 board using the generated VHDL codes from the pre-synthesis experiments targeting 66.7 MHz. Usually, circuits resulting from an HLS flow are embedded in a larger system including, for example, memory- and sensor-interfaces, which are omitted in our analysis. In order to reduce the I/O count, we include a shift register for the inputs and a multiplexer to select a single output bit. This ensures that all designs fit onto the FPGA regarding the I/O pin count and the overhead due to the shift register and output MUX only skews results minimally compared to full I/O interfaces used in practical scenarios.

Table 11 shows post implementation results. Here we also include circuits generated with the commercial tool HDL coder by MathWorks [34]. The HDL coder differs from the remaining approaches in the following aspects:

- (1) It uses its own operator library, hence, pipeline stages for operators and therefore also  $\Pi_{rec}^\perp$  for the corresponding systems will differ. This is not the case for the remaining approaches, since they all are embedded into Origami HLS and therefore use the same operator library based on FloPoCo.
- (2) The different operator library for the HDL coder yields other LUT/FF/DSP costs compared to the other approaches. This means a difference in resource costs can be caused by the operators as well as the interconnect costs (lifetime registers and multiplexers).
- (3) The HDL coder employs its own allocation, scheduling and binding. So it cannot be guaranteed that it always arrives at the minimal operator allocation.

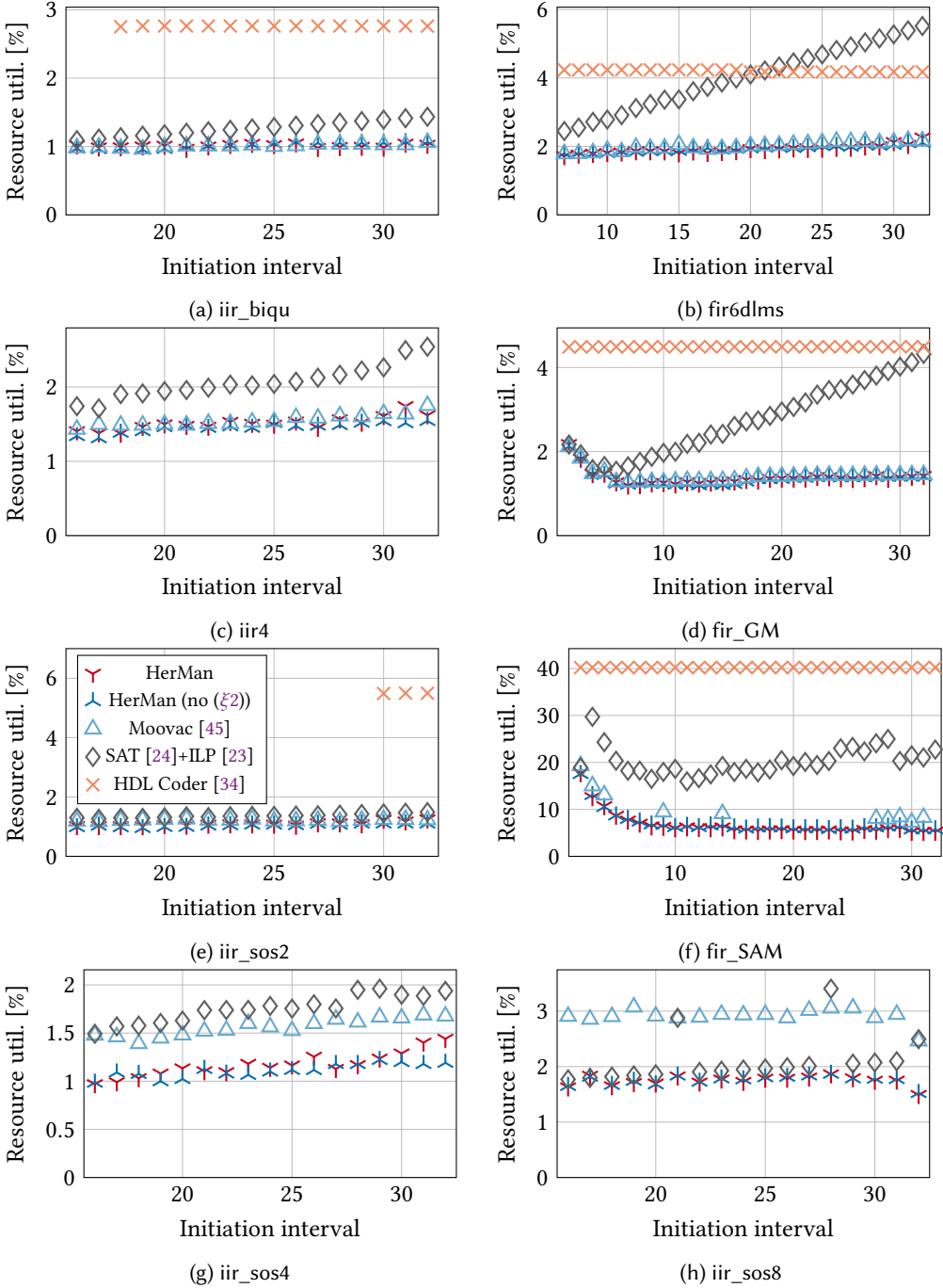


Fig. 14. Post implementation results: II vs. normalized resource utilization (part 1)

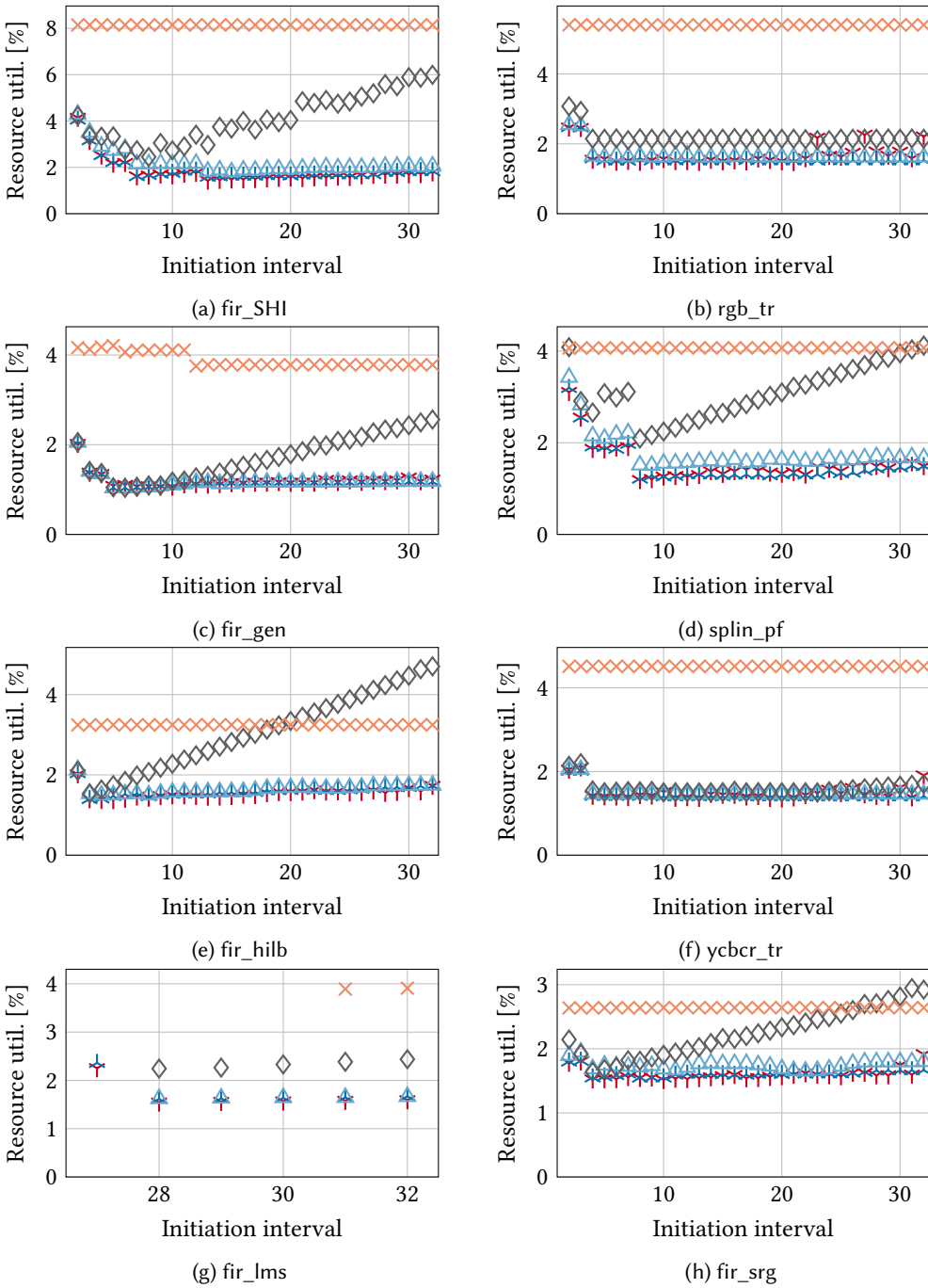


Fig. 15. Post implementation results: II vs. normalized resource utilization (part 2); see Fig. 14 for the legend

Table 11. Post implementation results summary; all numbers reported are averaged over all IIs and all clock gating settings (#C and  $\Phi(\omega)$ ) for the given benchmark model

Benchmark	HerMan				HerMan (no ( $\xi$ 2))				Moovac [45]				SAT [24] + ILP [23]				HDL Coder [34]			
	avg. # LUTs	avg. # FFs	avg. # DSPs	avg. $f_{\max}$ [MHz]	avg. # LUTs	avg. # FFs	avg. # DSPs	avg. $f_{\max}$ [MHz]	avg. # LUTs	avg. # FFs	avg. # DSPs	avg. $f_{\max}$ [MHz]	avg. # LUTs	avg. # FFs	avg. # DSPs	avg. $f_{\max}$ [MHz]	avg. # LUTs	avg. # FFs	avg. # DSPs	avg. $f_{\max}$ [MHz]
iir_biqu	757	1000	2	98	700	921	2	94	702	930	2	93	554	2138	2	92	2347	2188	4	113
fir6dlms	1772	2047	2	93	1655	1954	2	83	1758	2081	2	82	1063	9986	2	80	2917	3721	8	97
iir4	1562	1974	2	96	1198	1369	2	89	1236	1524	2	82	980	3873	2	83	-	-	-	-
fir_GM	1236	1299	2	93	1054	1259	2	91	1102	1323	2	90	638	6618	2	92	3520	3432	8	102
iir_sos2	886	1325	2	95	718	990	2	92	880	1281	2	87	645	2222	2	86	4685	4296	8	101
fir_SAM	6180	7097	7	85	6115	7506	7	80	9541	12568	12	77	3451	54602	7	77	35029	30268	58	86
iir_sos4	1079	1628	2	100	771	1086	2	89	1099	1954	2	83	803	3059	2	83	-	-	-	-
iir_sos8	1378	1676	4	105	1209	1483	4	89	2066	3519	4	81	975	2970	4	89	-	-	-	-
fir_SHI	1573	1795	3	93	1499	1852	3	97	1803	2338	3	95	951	10448	3	92	7274	6624	10	93
rgb_tr	1724	1929	2	99	1310	1443	2	96	1388	1625	2	92	1137	3745	2	92	4640	4107	8	102
fir_gen	899	1203	2	93	783	1125	2	82	792	1160	2	82	691	3213	2	81	2409	2770	10	105
splin_pf	1310	1567	3	94	1149	1361	3	86	1349	1777	3	87	695	7333	3	81	4482	4032	0	93
fir_hilb	1411	1510	2	98	1327	1475	2	101	1346	1565	2	102	924	6984	2	100	2817	2794	4	107
ycber_tr	1437	1679	2	91	1104	1417	2	94	1205	1360	2	93	1085	2019	2	94	3623	3295	8	119
fir_lms	1388	1535	2	96	1429	1637	2	83	1351	1596	2	83	996	4509	2	80	2808	2965	8	96
fir_srg	1441	1657	2	98	1322	1544	2	95	1432	1701	2	94	1000	4342	2	93	2091	2310	4	114
avg.	1627	1933	3	96	1459	1776	3	90	1816	2394	3	88	1037	8004	3	87	6049	5600	11	102

We see that the HDL coder has the highest resource costs across all benchmark instances and fails to generate circuits for selected models altogether. This is most likely due to the reasons mentioned before.

Overall, HerMan without clock gating has the lowest resource utilization across all benchmark instances due to its holistic optimization. SAT+ILP sometimes has reduced LUT count at the cost of a disproportionately high FF increase. This is caused by Vivado, which trades off LUTs for FFs by mapping shift registers into LUTs instead of FFs, if applicable. It is interesting to note that HerMan *without* clock gating yields slightly higher maximal clock frequencies compared to HerMan *with* clock gating. This is most likely caused by clock skews introduced by the BUFGCE instances which Vivado has to compensate during place & route.

Fig. 14 and Fig. 15 show detailed resource utilization for the whole benchmark suite. For each II and each scheduler we display the solution with lowest resource utilization (if more than one solution was found). The resource utilization is displayed as a percentage ratio of weighted utilization over weighted capacity:

$$\text{Resource utilization} = 100\% \cdot \frac{\Sigma(\text{LUTs}) \cdot \# \text{LUTs} + \Sigma(\text{FFs}) \cdot \# \text{FFs} + \Sigma(\text{DSPs}) \cdot \# \text{DSPs}}{\Sigma(\text{LUTs}) \cdot X^{\top}(\text{LUTs}) + \Sigma(\text{FFs}) \cdot X^{\top}(\text{FFs}) + \Sigma(\text{DSPs}) \cdot X^{\top}(\text{DSPs})}. \quad (9)$$

For the PYNQ-Z1 board this means (cf. Table 8):

$$\text{Resource utilization}|_{\text{PYNQ-Z1}} = 100\% \cdot \frac{2 \cdot \# \text{LUTs} + 1 \cdot \# \text{FFs} + 483.73 \cdot \# \text{DSPs}}{2 \cdot 53200 + 1 \cdot 106400 + 483.73 \cdot 220}. \quad (10)$$

The HDL coder consequently fails to exploit operator sharing possibilities when increasing the II and frequently fails to provide solutions for models with recurrences (e.g., the first solution for iir\_sos2 has II = 30 whereas the remaining schedulers provide a solution starting from II = 16). Interestingly, interconnect costs dominate for the non-binding-aware SAT-based modulo scheduler

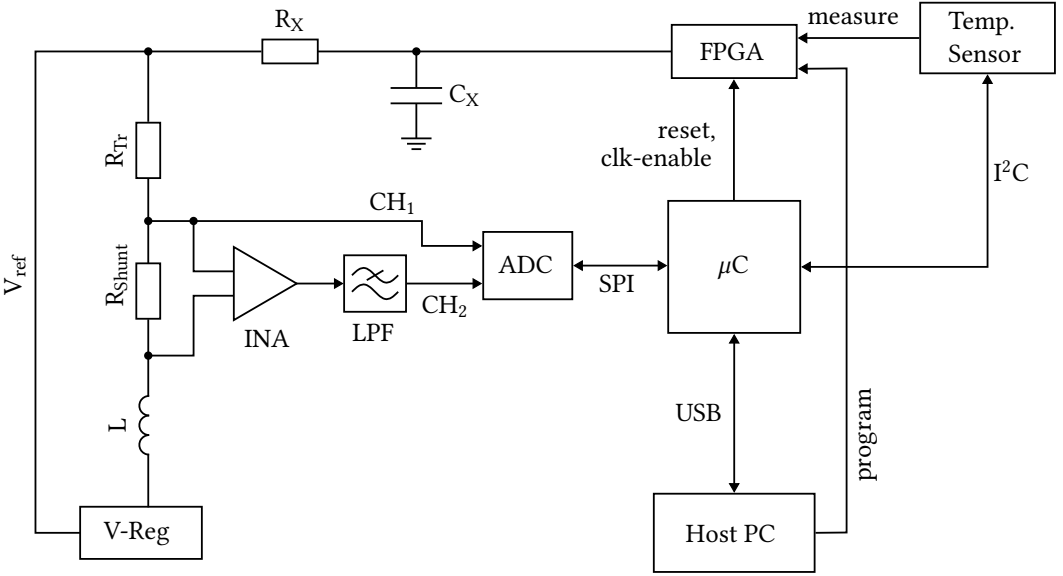


Fig. 16. Block diagram of our measurement setup

with rising IIs once the minimum operator allocation ( $F(\omega) = 1 \forall \omega \in \Omega$ ) is exceeded. Binding-aware schedulers Moovac and HerMan are able to keep interconnect costs virtually constant, even though Moovac fails to schedule the fir\_SAM filter for multiple IIs due to its size.

### 5.3 Power measurements

To validate the practical impact of the proposed optimizations we perform power measurements at the voltage supply of the PYNQ board. To do so, we incorporate a shunt resistor into the voltage supply path between the voltage regulator and its 1 V reference pin in order to not influence the system by our measurement circuit. The complete measurement setup is shown in Fig. 16.

The instrument amplifier (INA) is used for an initial amplification to a voltage domain that can be detected by the 24 bit ADC. The INA is followed by an active low-pass filter (LPF) for anti aliasing and noise reduction. Measurement channel 1 (CH<sub>1</sub>) picks up the FPGA core voltage (ideally, 1 V), and from the voltage measured at CH<sub>2</sub> we can infer the current used by the FPGA logic domain. All measurements are read by a microcontroller via SPI which then sends the data to a host PC for evaluation. Simultaneously, we measure the temperature via an infra red temperature sensor. In our tests, the temperature was kept approximately constant for each benchmark instance. Hence, we do not display it for brevity.

This setup allows us to measure the total power consumption  $P_{\text{total}}$  by the FPGA's logic domain. Using clock gating (a BUF<sub>GCE</sub> instance controlled by the microcontroller interfaced via one of the boards PMOD pins), we can switch off the system clock, which leads to a measurement of the static power  $P_{\text{static}}$ . Subsequently, we calculate the dynamic power  $P_{\text{dynamic}}$  via

$$P_{\text{total}} = P_{\text{static}} + P_{\text{dynamic}} \quad \Longleftrightarrow \quad P_{\text{dynamic}} = P_{\text{total}} - P_{\text{static}}. \quad (11)$$

All bitstreams include the design under test (DUT) and a random number generator for input stimulus generation. DUT outputs are not routed to any I/O pins, so we use the "DONT\_TOUCH" attribute to prohibit Vivado from deleting the whole circuit during optimization.

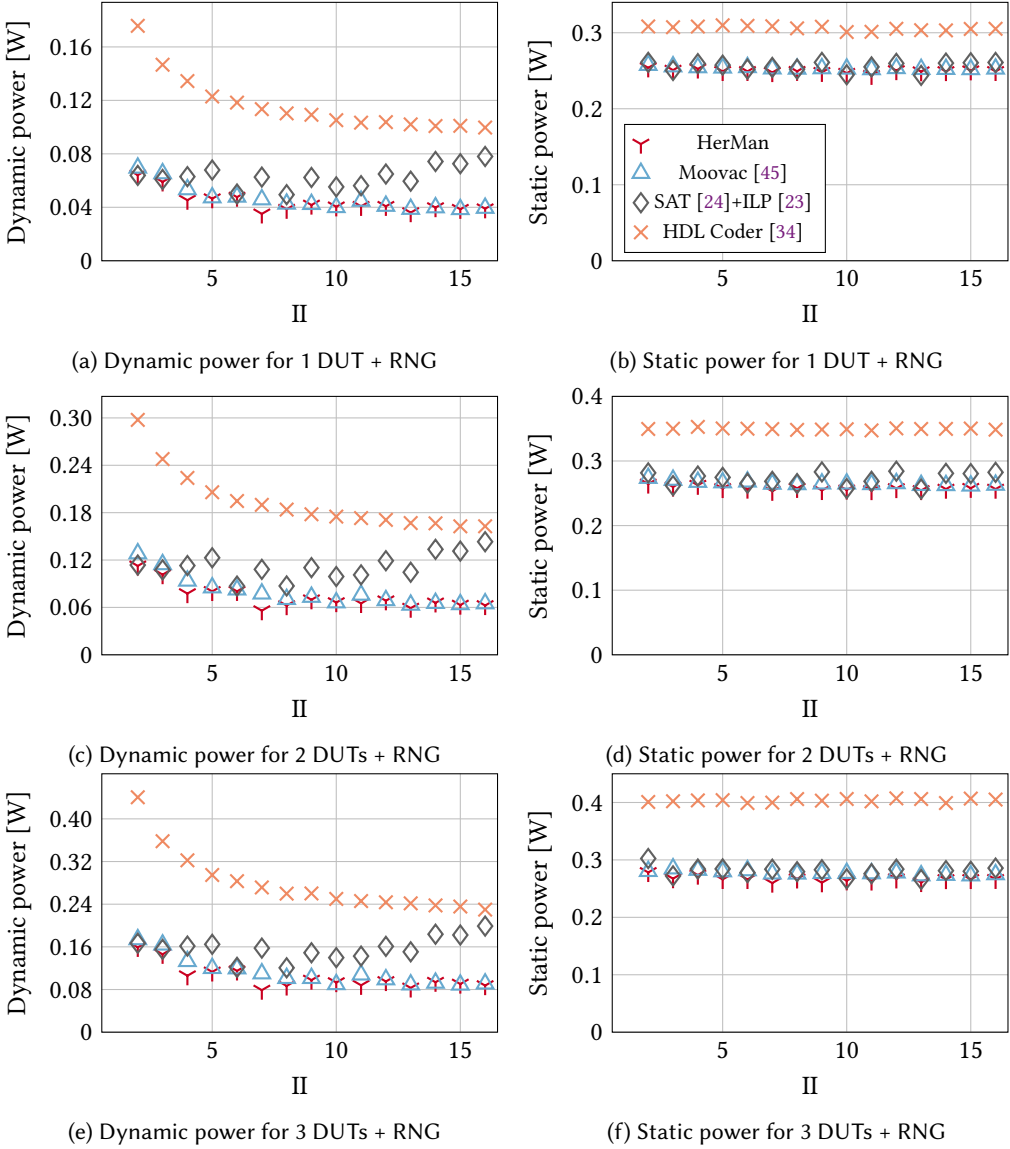


Fig. 17. Power measurement results for fir\_SHI

For each bitstream the DUT implementation together with the RNG consumes only a small portion of the PYNQ board's available resources<sup>3</sup>. Therefore, we perform multiple measurements with multiple DUT counts in parallel to see how scaling the resource consumption translates to power consumption.

Results for the fir\_SHI and iir\_sos8 models are shown in Fig. 17 and Fig. 18, respectively. We give measurement results for one to three DUTs implemented in parallel. We only have access to one setup for the time-consuming power measurement, so performing a measurement for the

<sup>3</sup>HerMan with  $\#C = \Phi(\omega) = 2$  for iir\_sos8 and  $II = 16$  needs 2417 of the available 53200 LUTs (4.54 %)

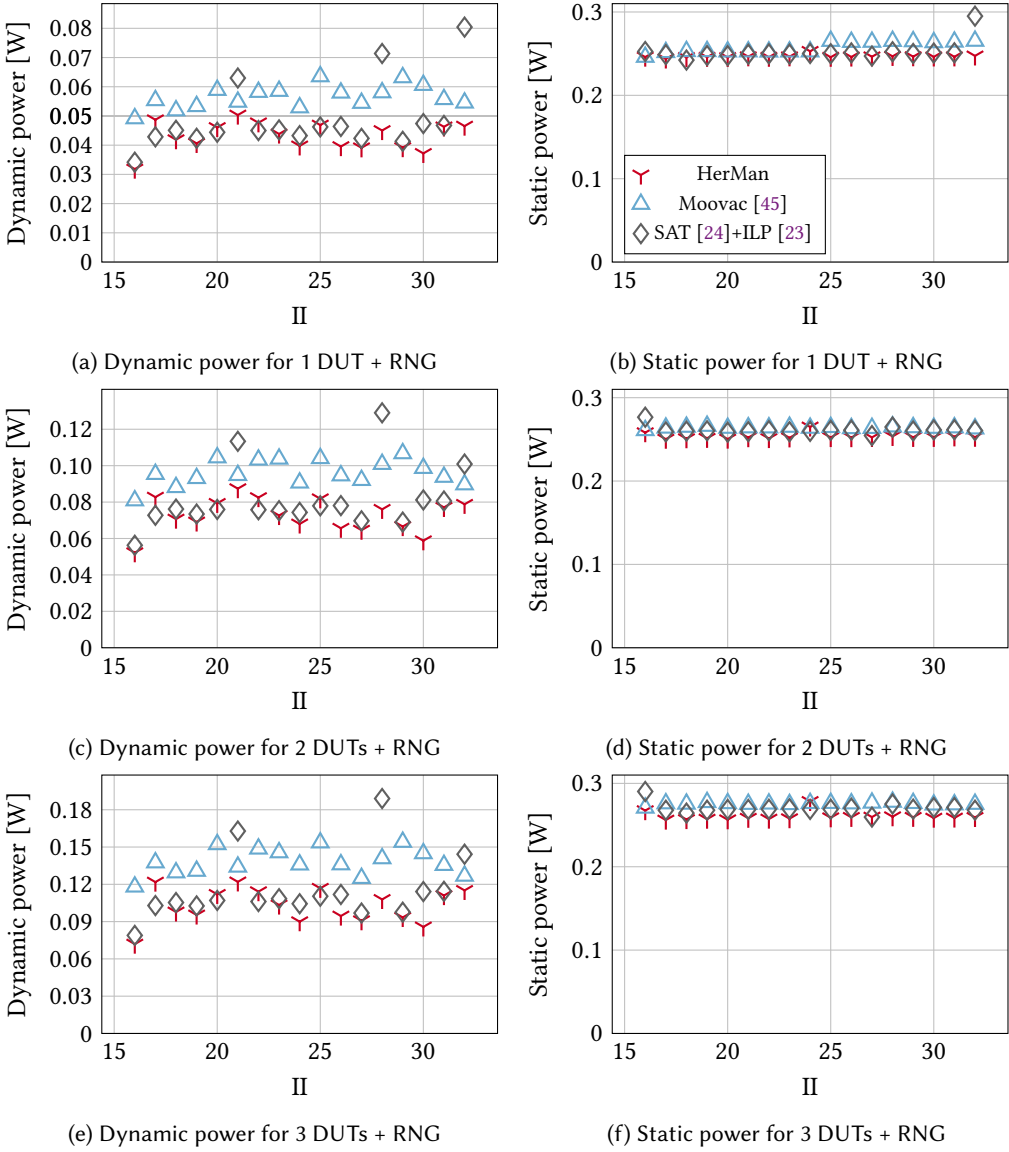


Fig. 18. Power measurement results for iir\_sos8

whole benchmark suite is infeasible. We choose fir\_SHI since it is the largest recurrence-free model for which Moovac successfully computes a valid solution for all IIs. The iir\_sos8 model is chosen because it is the largest model with recurrence, and therefore assumed to be among the hardest benchmark instances in this suite, which is also reflected by the solving times shown in Table 10. From Table 10 it is apparent that HerMan frequently fails to compute clock gating-aware schedules due to the benchmark size. Savings for iir\_sos8 are mainly due to lower resource utilization caused by, e.g., MUX optimizations.

We see that gating operator clocks translates into savings in dynamic power consumption for HerMan compared to the rest. While static power remains largely constant when varying the DUT count, dynamic power rises nearly proportionally (aside from an additive constant which is most likely caused by the RNG) when implementing more DUTs in parallel. From these results we can conclude that the combination of resource- and clock gating-aware scheduling translates to more power efficient circuits. Hence, the proposed approach is primarily suitable for the following cases:

- Multiple small systems are implemented in parallel on the same FPGA.
- Large systems are implemented on the FPGA.
- An FPGA with low static power consumption is used.

AMD suggests using clock enables instead of gated clocks for their FPGAs. Therefore, we perform an additional test where we enable `GATED_CLOCK_CONVERSION` and replace `BUFGCE` instances by simple and-gates to automatically convert the clock gating logic to enable logic. This should allow the synthesis tool to more efficiently utilize the clocking network for improved timing closure since it does not have to account for clock skew caused by the additional clock buffers. Here we study the impact on power consumption. In order to still prevent Vivado from deleting the hardware design during optimization while still allowing the clock-gating-to-clock-enable conversion, we use a MUX to select a single bit from the output data (routed to an FPGA output pin and controlled via some FPGA input pins) instead of the `DONT_TOUCH` attribute. The MUX is used for both the systems with clock gating and those with clock enables for the fairest comparison.

Results are shown in Fig. 19. Power consumption results are approximately equal for `iir_sos8` since clock gating is only rarely employed so circuits with clock gating are mostly equal to circuits with clock enables. This is not the case for `fir_SHI` where dynamic power consumption for clock gating is significantly smaller compared to clock enables until  $\Pi = 5$ . Here, HerMan frequently finds opportunities to disable the operator clocks for power savings.

## 6 FUTURE WORK

Starting from this work, it is natural to investigate the following three topics: (i) scalability, (ii) C/C++ based HLS, (iii) ASIC design.

Regarding scalability, we can either focus on improving the proposed, exact method or tackle the problem heuristically. For exact solutions, other solving frameworks such as Boolean Satisfiability (SAT) or Satisfiability Modulo Theories (SMT) can prove beneficial given the recent success when applied to scheduling in HLS [11, 17, 24]. At the same time, integrating the proposed approach into commercial tools is only feasible if a fall-back heuristic is also available to circumvent prohibitively long CPU times during the stage of rapid prototyping. This can, for example, be accomplished by developing a (heuristic) clock gating-aware modulo scheduler and integrating clock gating into existing binding algorithms.

C/C++ based HLS has some conceptual differences from Simulink models: Simulink models predominantly represent a single loop body, assumed to be running indefinitely. Programs, on the other hand, can consist of multiple, nested loops and function calls, coupled with dynamic memory accesses and control structures like if/else statements. In order to tackle general HLS, HerMan can either be applied to each loop body individually, or the ILP formulation must be extended to also support these features.

HerMan was developed with the intention of mainly targeting FPGA applications. When targeting ASICs, many circumstances change. For example, chaining becomes more relevant since operators do not need to be deeply pipelined in order to reach high frequencies and the clock network is different, causing a change in clock gating constraints. Future work should investigate how these concepts can be integrated into the proposed algorithm.

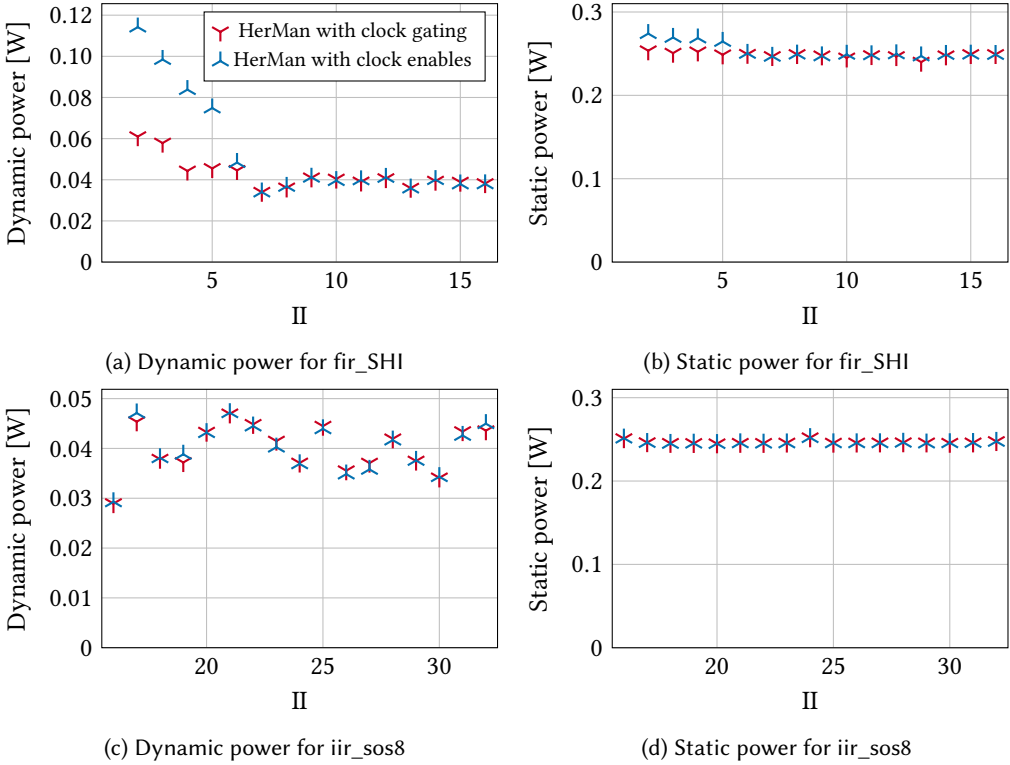


Fig. 19. Power measurement results: clock gating vs. clock enables

## 7 CONCLUSION

This work presented a holistic ILP model for model-based hardware design, which optimizes all aspects of the resulting microarchitecture and also includes clock gating for dynamic power reduction. All circuits generated with the proposed approach achieve the same (or sometimes even better) throughput as those generated with the traditional approach of allocation, modulo scheduling and binding. This is ensured by applying all microarchitecture optimizations as an extension of the original throughput optimization formulation only, thus maintaining all optimal solutions. Since in practical applications, full utilization of operators cannot be guaranteed even with optimal throughput, we use clock gating to save energy by disabling idle operators.

Extensive experimental evaluations show that the holistic nature of our proposed optimization consistently reduces resource consumption compared to separate modulo scheduling and binding [23, 24], binding-aware modulo scheduling [45] and a commercial model-based design tool [34]. The integration of clock gating into the HLS flow, together with the aforementioned resource savings, cause considerable dynamic power savings without any performance degradation in the resulting systems.

## ACKNOWLEDGMENTS

The authors would like to thank Alexander Laber, Martin Hardieck and Patrick Wengel for their work on the power measurement setup, as well as the reviewers for their rigorous and helpful feedback.

## REFERENCES

- [1] AMD. 2024. UltraFast Design Methodology Guide for FPGAs and SoCs (UG949). <https://docs.amd.com/r/en-US/ug949-vivado-design-methodology/Converting-Clock-Gating-to-Clock-Enable>. Accessed: 2024-10-02.
- [2] AMD. 2024. Vivado Design Suite 7 Series FPGA and Zynq 7000 SoC Libraries Guide (UG953). <https://docs.amd.com/r/en-US/ug953-vivado-7series-libraries/BUGFCE>. Accessed: 2024-10-07.
- [3] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. 2004. Open source (Mixed-Integer) Linear Programming system. <https://lpsolve.sourceforge.net/5.5/>. Accessed: 2024-10-02.
- [4] Suresh Bolusani, Ksenia Bestuzheva Mathieu Besançon, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Christoph Graczyk, Katrin Halbig, Ivo Hedtke, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Julian Manns, Gioni Mexi, Erik Mühmer, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Dieter Weninger, and Lixing Xu. 2024. SCIP Solving Constraint Integer Programs. <https://www.scipopt.org>. Accessed: 2024-09-23.
- [5] Philip Brisk and Paolo Ienne. 2009. On the complexity of the Port Assignment Problem for Binary Commutative Operators in high-level synthesis. In *2009 International Symposium on VLSI Design, Automation and Test*. IEEE, New York, NY, USA, 339–342. <https://doi.org/10.1109/VDAT.2009.5158164>
- [6] Andrew Canis, Stephen D. Brown, and Jason H. Anderson. 2014. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, New York, NY, USA, 1–8. <https://doi.org/10.1109/FPL.2014.6927490> ISSN: 1946-1488.
- [7] D. Chen and J. Cong. 2004. Register binding and port assignment for multiplexer optimization. In *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No.04EX753)*. IEEE, New York, NY, USA, 68–73. <https://doi.org/10.1109/ASPDAC.2004.1337542>
- [8] Deming Chen, Jason Cong, and Yiping Fan. 2003. Low-power high-level synthesis for FPGA architectures. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design, 2003. ISLPED '03*. ACM Press, New York, NY, USA, 134–139. <https://doi.org/10.1109/LPE.2003.1231849>
- [9] Jianyi Cheng, Lana Josipovic, George A. Constantinides, Paolo Ienne, and John Wickerson. 2020. Combining Dynamic & Static Scheduling in High-level Synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, Seaside CA USA, 288–298. <https://doi.org/10.1145/3373087.3375297>
- [10] Jianyi Cheng, Lana Josipović, John Wickerson, and George A. Constantinides. 2023. Parallelising Control Flow in Dynamic-scheduling High-level Synthesis. *ACM Transactions on Reconfigurable Technology and Systems* 16, 4 (Sept. 2023), 55:1–55:32. <https://doi.org/10.1145/3599973>
- [11] Jianyi Cheng, John Wickerson, and George A. Constantinides. 2021. Exploiting the Correlation between Dependence Distance and Latency in Loop Pipelining for HLS. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, New York, 341–346. <https://doi.org/10.1109/FPL53798.2021.00066>
- [12] Jianyi Cheng, John Wickerson, and George A. Constantinides. 2022. Dynamic C-Slow Pipelining for HLS. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, New York, NY, USA, 1–10. <https://doi.org/10.1109/FCCM53951.2022.9786096> ISSN: 2576-2621.
- [13] Hao Cong, Song Chen, and Takeshi Yoshimura. 2012. Port assignment for interconnect reduction in high-level synthesis. In *Proceedings of Technical Program of 2012 VLSI Design, Automation and Test*. IEEE, New York, NY, USA, 1–4. <https://doi.org/10.1109/VLSI-DAT.2012.6212613>
- [14] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Transactions on Reconfigurable Technology and Systems* 15, 4 (April 2022), 3530775. <https://doi.org/10.1145/3530775>
- [15] Jason Cong and Junjuan Xu. 2008. Simultaneous FU and Register Binding Based on Network Flow Method. In *2008 Design, Automation and Test in Europe*. IEEE, New York, NY, USA, 1057–1062. <https://doi.org/10.1109/DATE.2008.4484821> ISSN: 1558-1101.
- [16] Steve Dai, Gai Liu, Ritchie Zhao, and Zhiru Zhang. 2017. Enabling adaptive loop pipelining in high-level synthesis. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*. IEEE, New York, NY, USA, 131–135. <https://doi.org/10.1109/ACSSC.2017.8335152> ISSN: 2576-2303.
- [17] Steve Dai and Zhiru Zhang. 2019. Improving Scalability of Exact Modulo Scheduling with Specialized Conflict-Driven Learning. In *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, Las Vegas NV USA, 1–6. <https://doi.org/10.1145/3316781.3317842>
- [18] Florent de Dinechin and Bogdan Pasca. 2011. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design & Test of Computers* 28, 4 (July 2011), 18–27. <https://doi.org/10.1109/MDT.2011.44>
- [19] Leandro de Souza Rosa, Christos-Savvas Bouganis, and Vanderlei Bonato. 2019. Scaling Up Modulo Scheduling for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (May 2019), 912–925. <https://doi.org/10.1109/TCAD.2018.2834440>

- [20] Alexandre E. Eichenberger and Edward S. Davidson. 1997. Efficient formulation for optimal modulo schedulers. *ACM SIGPLAN Notices* 32, 5 (May 1997), 194–205. <https://doi.org/10.1145/258916.258933>
- [21] Etched. 2024. Etched | The World's First Transformer ASIC. <https://www.etched.com>. Accessed: 2024-06-27.
- [22] K. Fan, M. Kudlur, Hyunchul Park, and S. Mahlke. 2005. Cost sensitive modulo scheduling in a loop accelerator synthesis system. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. IEEE, New York, NY, USA, 12 pp.–232. <https://doi.org/10.1109/MICRO.2005.17> ISSN: 2379-3155.
- [23] Nicolai Fiege, Patrick Sittel, and Peter Zipf. 2022. Optimal Binding and Port Assignment for Loop Pipelining in High-Level Synthesis. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, New York, NY, USA, 262–269. <https://doi.org/10.1109/FPL57034.2022.00047> ISSN: 1946-1488.
- [24] Nicolai Fiege and Peter Zipf. 2023. BLOOP: Boolean Satisfiability-based Optimized Loop Pipelining. *ACM Transactions on Reconfigurable Technology and Systems* 16, 3 (July 2023), 49:1–49:32. <https://doi.org/10.1145/3599972>
- [25] Daniel D. Gajski. 1988. *Silicon Compilation*. Addison-Wesley, Boston, MA, USA.
- [26] Gurobi. 2022. Gurobi Optimizer. <https://www.gurobi.com/products/gurobi-optimizer/>. Accessed: 2022-07-22.
- [27] IBM. 2024. IBM ILOG CPLEX Optimizer. <https://www.ibm.com/de-de/products/ilog-cplex-optimization-studio/cplex-optimizer>. Accessed: 2024-09-23.
- [28] Intel. 2024. Intel Quartus Prime Pro Edition User Guide: Design Compilation (1.12.5 Automatic Gated Clock Conversion). <https://www.intel.com/content/www/us/en/docs/programmable/683236/24-2/automatic-gated-clock-conversion.html>. Accessed: 2024-10-02.
- [29] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 127–136. <https://doi.org/10.1145/3174243.3174264>
- [30] S. Lahti, P. Sjövall, J. Vanne, and T. D. Härmäläinen. 2019. Are We There Yet? A Study on the State of High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (May 2019), 898–911.
- [31] Ghizlane Lhahrech-Lebreton, Philippe Coussy, and Eric Martin. 2010. Hierarchical and Multiple-Clock Domain High-Level Synthesis for Low-Power Design on FPGA. In *2010 International Conference on Field Programmable Logic and Applications*. IEEE, Milan, Italy, 464–468. <https://doi.org/10.1109/FPL.2010.94>
- [32] Shuangnan Liu, Francis CM Lau, and Benjamin Carrion Schafer. 2019. Accelerating FPGA Prototyping through Predictive Model-Based HLS Design Space Exploration. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19)*. Association for Computing Machinery, New York, NY, USA, 1–6. <https://doi.org/10.1145/3316781.3317754>
- [33] Josep Llosa, Eduard Ayguadé, Antonio Gonzalez, Mateo Valero, and Jason Eckhardt. 2001. Lifetime-sensitive modulo scheduling in a production environment. *IEEE Trans. Comput.* 50, 3 (March 2001), 234–249. <https://doi.org/10.1109/12.910814>
- [34] MathWorks. 2024. HDL Coder. <https://www.mathworks.com/products/hdl-coder.html>. Accessed: 2024-09-23.
- [35] Konrad Möller, Martin Kumm, Marco Kleinlein, and Peter Zipf. 2017. Reconfigurable Constant Multiplication for FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 6 (June 2017), 927–937. <https://doi.org/10.1109/TCAD.2016.2614775>
- [36] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (Oct. 2016), 1591–1604. <https://doi.org/10.1109/TCAD.2015.2513673> Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.
- [37] Univeristy of Kassel. 2015. Origami HLS. <http://www.uni-kassel.de/go/origami>. Accessed: 2022-01-11.
- [38] Julian Oppermann, Melanie Reuter-Oppermann, Lukas Sommer, Andreas Koch, and Oliver Sinnen. 2019. Exact and Practical Modulo Scheduling for High-Level Synthesis. *ACM Transactions on Reconfigurable Technology and Systems* 12, 2 (2019), 26.
- [39] Julian Oppermann, Patrick Sittel, Martin Kumm, Melanie Reuter-Oppermann, Andreas Koch, and Oliver Sinnen. 2019. Design-Space Exploration with Multi-Objective Resource-Aware Modulo Scheduling. In *Euro-Par 2019: Parallel Processing*, Ramin Yahyapour (Ed.). Springer International Publishing, Cham, 170–183.
- [40] Julian Oppermann, Lukas Sommer, Lukas Weber, Melanie Reuter-Oppermann, Andreas Koch, and Oliver Sinnen. 2019. SkyCastle: A Resource-Aware Multi-Loop Scheduler for High-Level Synthesis. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, New York, NY, USA, 36–44. <https://doi.org/10.1109/ICFPT47387.2019.00013>
- [41] Ian Page and Wayne Luk. 1991. Compiling occam into Field-Programmable Gate Arrays. In *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, Vol. 15. EE&CS Books, Oxford, UK, 271–283.
- [42] Bantwal R. Rau. 1994. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture (MICRO 27)*. Association for Computing Machinery, New York, NY, USA, 63–74. <https://doi.org/10.1145/192724.192731>

- [43] Bantwal R. Rau and Christopher D. Glaeser. 1981. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *ACM SIGMICRO Newsletter* 12, 4 (Dec. 1981), 183–198. <https://doi.org/10.1145/1014192.802449>
- [44] M. Rim, R. Jain, and R. De Leone. 1992. Optimal allocation and binding in high-level synthesis. In *[1992] Proceedings 29th ACM/IEEE Design Automation Conference*. IEEE, New York, NY, USA, 120–123. <https://doi.org/10.1109/DAC.1992.227850> ISSN: 0738-100X.
- [45] Patrick Sittel, Martin Kumm, Julian Oppermann, Konrad Möller, Peter Zipf, and Andreas Koch. 2018. ILP-Based Modulo Scheduling and Binding for Register Minimization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, New York, NY, USA, 265–2656. <https://doi.org/10.1109/FPL.2018.00053> ISSN: 1946-1488.
- [46] Patrick Sittel, Konrad Möller, Martin Kumm, Peter Zipf, Bogdan Pasca, and Mark Jervis. 2017. Model-based hardware design based on compatible sets of isomorphic subgraphs. In *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE, New York, NY, USA, 199–202. <https://doi.org/10.1109/FPT.2017.8280140>
- [47] Patrick Sittel, Julian Oppermann, Martin Kumm, Andreas Koch, and Peter Zipf. 2018. HatScheT: A Contribution to Agile HLS. In *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*. VDE Verlag, Berlin, Germany, 1–8.
- [48] Patrick Sittel, Thomas Schönwälder, Martin Kumm, and Peter Zipf. 2018. ScaLP: A Light-Weighted (M)LP Library. In *Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. Universität Tübingen, Tübingen, 10.
- [49] Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. 2023. Compiler Discovered Dynamic Scheduling of Irregular Code in High-Level Synthesis. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, New York, NY, USA, 1–9. <https://doi.org/10.1109/FPL60245.2023.00009> ISSN: 1946-1488.
- [50] Wei Wu. 2017. *Model-Based Design for Effective Control System Development*. IGI Global, Hershey, Pennsylvania, USA. <https://www.igi-global.com/book/model-based-design-effective-control/www.igi-global.com/book/model-based-design-effective-control/173674>
- [51] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15)*. Association for Computing Machinery, New York, NY, USA, 161–170. <https://doi.org/10.1145/2684746.2689060>
- [52] Přemysl Šůcha and Zdeněk Hanzálek. 2011. A cyclic scheduling problem with an undetermined number of parallel identical processors. *Computational Optimization and Applications* 48, 1 (Jan. 2011), 71–90. <https://doi.org/10.1007/s10589-009-9239-4>

Received 28 June 2024; revised 10 October 2024; accepted 13 November 2024