ILP-based Modulo Scheduling and Binding for Register Minimization

Patrick Sittel
Universität Kassel
sittel@uni-kassel.de

Martin Kumm
Universität Kassel
kumm@uni-kassel.de

Julian Oppermann
TU Darmstadt
oppermann@esa.informatik.tu-darmstadt.de

Konrad Möller
Universität Kassel
konrad.moeller@uni-kassel.de

Peter Zipf
Universität Kassel
zipf@uni-kassel.de

Andreas Koch
TU Darmstadt
koch@esa.informatik.tu-darmstadt.de

***Abstract*—A key element for achieving high throughput, e.g. circuits generated with high-level synthesis (HLS) methods and model-based hardware design, is the use of modulo scheduling. Integer linear programming (ILP)-based modulo schedulers are capable of computing schedules that are optimal regarding throughput and latency, while keeping run times to practically usable lengths. However, the generated schedules may lead to an excessive number of registers for storing intermediate values. We propose extensions for ILP-based modulo scheduling that minimizes these registers. The ILP formulation incorporates the elimination of redundant registers by post binding optimization. Extensive experiments on different benchmark sets show average register reductions of 30.4 % compared to commonly used minimum lifetime approaches that reduce register requirements. This comes without any loss in throughput or latency and with less than 4% additional scheduling run time compared to state-of-the-art ILP-based modulo schedulers.**

I. Introduction

Scheduling and binding plays a major role in the design of complex hardware systems. To reduce design time, overall project costs and hardware effort, methods like high-level synthesis (HLS) or model-based hardware design were proposed and became popular in recent years [4], [17]. Due to the constrained resources of hardware implementations, high-level descriptions are typically transformed into a time-multiplexed architecture. To do so, the classical steps of scheduling, allocation, and binding have to be performed. A key element for achieving high throughput is to find a good schedule. Modulo scheduling, where new values (samples) are input into a computation after a fixed number of time steps, called the initiation interval (II), is a commonly used technique for increasing the throughput of an implementation. Given a data flow graph representing the operations to be scheduled, the main task of a modulo scheduler is finding the smallest possible II in order to maximize throughput. Frequently, the secondary objective is the minimization of latency. Note that modulo scheduling is sometimes referred to as cyclic scheduling [7], pipeline scheduling [14] or overlapped scheduling [16]. The resulting schedule is used for the generation of a time-multiplexed architecture. Depending on the context, this architecture generation is called software pipelining [23], loop pipelining [3], or folding [22].

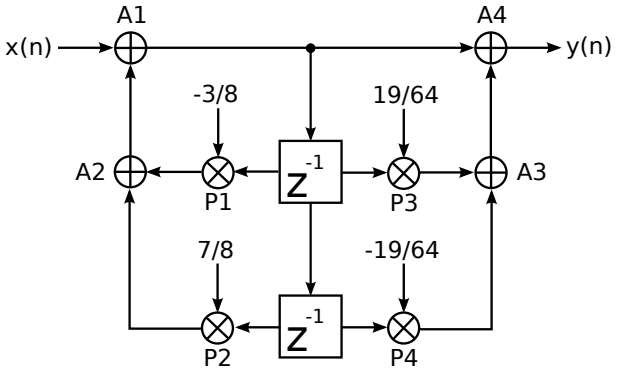
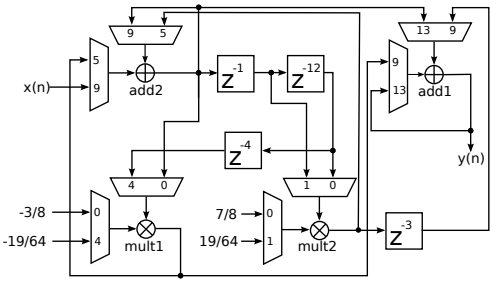


Fig. 1: High-level description of a biquad IIR-filter

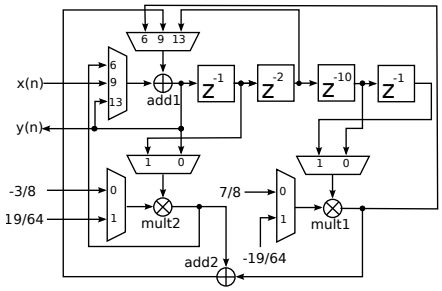
In these time-multiplexed architectures, registers are used to store intermediate results. In the following, we refer to this registers as *lifetime registers*. We consider in this work that these registers will be placed close to the data processing operations to achieve short routing delays. Addtionally, the technique of pipelining is commonly used to fulfill throughput constraints [28]. Consequently, lifetime registers require additional logic resources in field-programmable gate array (FPGA) implementations. Therefore, lifetime registers lead to a resource overhead for application-specific integrated circuits (ASICs) and also pipelined FPGA implementations due to additionally required slices. It can be observed that there exist different schedules and bindings with optimal II and minimal latency that lead to significantly different register requirements in the resulting architecture. This paper extends integer linear programming (ILP) formulations of the modulo scheduling problem minimizing the required registers while keeping the optimal II.

A. Motivational Example

Figure 1 shows the structure of a biquad infinite impulse response (IIR) filter [21]. This high level behavioral description can be seen as a directed labeled graph which consists of vertices $o_i \in O$ (which represent operations) and directed edges $o_i \rightarrow o_j \in E$ (which represent connections).



(a)



(b)

Fig. 2: Time-multiplexed biquad filter using (a) non register-aware modulo schedule (b) minimum register modulo schedule

An important step in the design of the time-multiplexed circuit is the determination of variable lifetimes, which also specifies the number of registers in the implementation. The lifetime of a variable is given by

$$n_{ij} = t_j - t_i - D_i + d_{ij}\text{II}, \qquad \forall o_i \rightarrow o_j \in E \ , \qquad (1)$$

where t_i and t_j are the respective start times, D_i is the latency of node o_i, d_{ij} is the *algorithmic delay* on edge $o_i \rightarrow o_j$ and II represents the initiation interval for the next sample. By *algorithmic delay*, we refer to delays which are part of the algorithm (such as, e.g., the sample delays D1 and D2 in Fig. 1), but not for storing intermediate results in the time-multiplexed circuit, which are called *register* in the following. Note that (1) is also called the folding equation [21]. The value n_{ij} represents the number of time steps the variable represented by edge $o_i \rightarrow o_j$ has to be stored (by using registers). Obviously, $n_{ij} \geq 0$ has to hold for a valid schedule.

For our example, let the latencies of the `add` and `mult` units be four and five, respectively. Also, assume that no more than two `add` and two `mult` hardware units shall be used. Fig. 2a and 2b show two different time-multiplexed architectures, which result from the two different schedules and bindings that are given in Table I. Both schedules provide an optimal II of 13 and a minimum latency (for the given II) of 17. One significant difference between both solutions is that the minimum register approach binds the three operations, A1, A2 and A4, to the same hardware add1, resulting in an implementation that uses fewer lifetime registers (14 instead of 17) after the adder units. Additionally, the mult1 operation is scheduled at time step 1 mod 13 instead of 4 mod 13, which effectively saves three lifetime registers of the mult2 unit that are needed in the non-register aware solution in Fig. 2a. Both architectures use *register sharing*, i.e., registers that originate from the same source are shared as illustrated in Fig. 3. Summing up the total registers including sharing, the solution of Fig. 2a requires 20 registers, while the solution of Fig. 2b only requires 14 registers. Note that the number of 2:1 MUXes remained identical, assuming that a 3:1 MUX is built from two 2:1 MUXes. As the operations are usually pipelined and have an output register, the removal of these registers does not change the critical path delay. Worse solutions like the one in Fig. 2a are common, because state-of-the-art ILP-based modulo schedulers are not aware of the register count and chose any out of many possible solutions with minimum II. This motivates our work on considering the register cost in the ILP-based modulo scheduling.

B. Related Work

Modulo schedulers can be classified into exact approaches that are capable of computing optimal solutions regarding II and often a target-dependent secondary objective (e.g., [16], [8], [7], [27], [1], [19]), and heuristic approaches that cannot give such optimality guarantees, but are chosen for their shorter run times (e.g., [23], [12], [18], [2]). A common secondary objective in the context of very long instruction word (VLIW) compilers is to reduce the register pressure, i.e., to minimize the maximum number of life variables in any time step in the schedule. While it is possible to model this in an exact manner [9], [16], the minimization of the cumulative register lifetimes in the schedule was proposed as a substitute [12], [6], [18], as it can be expressed in a linear form.

HLS tools create application-specific hardware architectures that spatially distribute operations to individual hardware operators. Similar to the VLIW compiler, the absolute minimum number of realized registers in the architecture is given by the maximum number of variables that are alive at the same time [16]. However, the implementation of a corresponding hardware architecture may require a significant overhead for multiplexing data from and to the registers. Instead, HLS tools may instantiate as many parallel registers as needed, which minimizes multiplexing overhead. But, passing data between the hardware operators requires a register for every time step that separates the producer and the consumer of a value, leaving ample opportunity for optimization.

In the extreme case, all operations may be mapped to individual hardware operators (spatially distributing the computation). Then, the problem to minimize the total number of registers required is equivalent to the aforementioned cumulative lifetime minimization. However, this is not a practical assumption. Usually, operations that require access to unique (e.g., memory ports) or expensive (e.g., floating-point operators) on-chip resources are *resource-constrained* in the scheduling problem, and will be time-multiplexed in the final hardware design. The binding decision, i.e., the concrete hardware resource an operation is mapped to, needs to be considered when minimizing the register demand. Specifically, registers from the same hardware operator can be shared, and

TABLE I: Two schedules and bindings of the example in Fig. 1 with II= 13 and latency= 17

	non register-aware		minimum register	
Operation	time slot	bound unit	time slot	bound unit
A1	9	add2	9	add1
A2	5	add2	5	add2
A3	9	add1	6	add1
A4	13	add1	13	add1
P1	0	mult1	0	mult2
P2	0	mult2	0	mult1
P3	1	mult2	1	mult2
P4	4	mult1	1	mult1

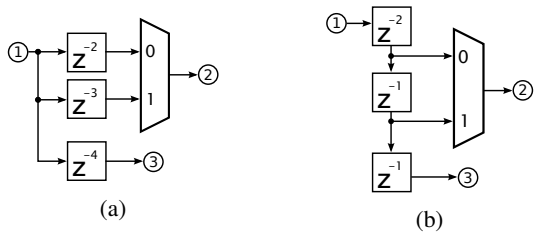


Fig. 3: Example of implementing the registers: (a) without sharing using 9 registers (b) with sharing using 4 registers

only the maximum of the registers required in each time step, over the entire schedule, has to be implemented in the final hardware. To the best knowledge of the authors, none of the existing work considers register sharing in their scheduling and binding [15], [12], [6], [8], [18].

C. Contributions

Previous approaches aim to minimize variable lifetimes for register minimization. This paper proposes an optimal solution for modulo scheduling such that the number of implemented registers is minimized. Using the methods introduced in this work, modulo schedules which lead to register-minimal hardware implementations can be determined without any quality loss regarding throughput or latency, and only little scheduling algorithm run time overhead. We support our claims with detailed scheduling experiments on widely spread applications and problem sizes from C code and model-based design. To explore the bounds of the complete design space we also evaluated a complementary objective that *maximizes* the number of implemented registers.

II. ILP Formulation

In the following, the complete ILP formulation of the state-of-the-art Moovac scheduler [19] is given which was shown to outperform previous modulo schedulers including Modulo SDC [2] used in LegUp. Subsequent, our new approach to model the register cost is given and linear ILP constraints are derived to model the register minimization. For the experimental results provided in Section III, these constraints are used to extend the Moovac formulation, but they are applicable for any ILP-based scheduler.

TABLE II: Constants (top) and variables of Moovac (middle) and the proposed extension (bottom)

Constant/Variable	Meaning
O	Set of operations
$L \subseteq O$	Set of resource constrained operations
C	Set of classes for resource constrained operations (i.e., add, mult, ...)
$L_c \subseteq L$	Set of resource constrained operations of class $c \in C$, i.e., $\bigcup_{c \in C} L_c = L$
E	Edges in the DFG
d_{ij}	Algorithmic delay on edge $o_i \rightarrow o_j$
$a_c \in \mathbb{N}$	No of instances of resource class $c \in C$
$D_i \in \mathbb{N}_0$	Latency (in clock cycles) of operation $o_i \in O$
$t_i \in \mathbb{N}_0$	Start time of operation $o_i \in O$
$m_i = t_i \mod \text{II}$	Modulo start time (congruence class) of operation $o_i \in L$
$y_i = \lfloor t_i / \text{II} \rfloor$	Helper in congruence class computation.
$r_i \in \{0, 1, \ldots, a_c - 1\}$	Resource instance of operation $o_i \in L_c$
$\epsilon_{ij} = \begin{cases} 1 & r_i < r_j \\ 0 & \text{otherwise} \end{cases}$	Overlap variable for resource for pair of operations $o_i, o_j \in L_c$ with $i \neq j$ and $c \in C$. True when o_i's resource index is strictly less then o_j's resource index
$\mu_{ij} = \begin{cases} 1 & m_i < m_j \\ 0 & \text{otherwise} \end{cases}$	Overlap variable for congruence class for pair of operations $o_i, o_j \in L_c$ with $i \neq j$ and $c \in C$. True when o_i's congruence class is strictly less then o_j's congruence class
n_{ij}	Life time of variable on edge $o_i \rightarrow o_j \in E$
R_{cl}, R_i	Number of registers at the output of instance $l \in \{0 \ldots a_c - 1\}$ of resource class c or unconstrained operation o_i (incl. reg. sharing)
$\rho_{il} = \begin{cases} 1 & r_i = l \\ 0 & \text{otherwise} \end{cases}$	Operation $o_i \in L_c$ is bound to instance $l \in \{0 \ldots a_c - 1\}$

A. Moovac ILP Formulation

The input to the Moovac ILP instance is summarized at the top of Table II. The different inputs are constants from the ILP optimizer's point of view. Basically, the operations (O) are divided into resource constrained (L) and unconstrained operations ($O \setminus L$). L is further split up into sets (L_c) of different resource classes ($c \in C$) like add, mult, etc. Each resource class is constrained by a maximum number of instances a_c. Each operation is assigned a latency D_i.

The outputs of the optimization are the start times t_i for each operation for the minimum possible II. As a direct minimization of the II is difficult to realize due to non-linear constraints, the II is considered as a constant and typically several optimization runs are performed starting from a lower bound of II. As the ILP solver is typically fast in proving infeasibility for IIs which are too small, a quick convergence is achieved. In addition to the start times, a number of additional variables are used in the formulation, which are given in the middle part of Table II. Variables y_i and m_i represent the quotient and the remainder in the modulo operation such that $t_i = y_i \text{II} + m_i$. Variables r_i encode which hardware instance (out of a_c) is used to implement node o_i which enables resource constraints. The μ_{ij} and ϵ_{ij} variables are so-called *overlap variables* [29], which model whether two operations overlap in the start times of the same congruence class (μ_{ij})

or use the same resource instance ϵ_{ij}.

In contrast to the original Moovac [19], where the sum of start times was minimized, we aim for the minimization of the overall latency as a secondary objective (in addition to keeping the min II as the primary objective). To realize this, we introduce an artificial node which is called the *super sink* [5] that follows all output nodes. By simply minimizing the start time t_{ss} of this super sink, we minimize the overall schedule length, and thus the latency. The overall Moovac ILP formulation is given as follows:

$$\text{minimize } t_{\text{ss}}$$

subject to

$$\begin{aligned}
&\text{C1:} && t_j - t_i - D_i + d_{ij}\text{II} \geq 0 && \forall o_i \to o_j \in E \\
&\text{C2:} && t_i = y_i\text{II} + m_i && \forall o_i \in L \\
&\text{C3:} && m_i \leq \text{II} - 1 && \forall o_i \in L \\
&\text{C4:} && r_i \leq a_c - 1 && \forall c \in C, o_i \in L_c \\
&\text{C5:} && \epsilon_{ij} + \epsilon_{ji} \leq 1 && \left.\begin{array}{l} \\ \\ \\ \\ \\ \\ \end{array}\right\} \begin{array}{l} \forall\ c \in C, \\ o_i, o_j \in L_c, \\ i \neq j \end{array} \\
&\text{C6:} && r_j - r_i - 1 - (\epsilon_{ij} - 1)a_c \geq 0 && \\
&\text{C7:} && r_j - r_i - \epsilon_{ij} a_c \leq 0 && \\
&\text{C8:} && \mu_{ij} + \mu_{ji} \leq 1 && \\
&\text{C9:} && m_j - m_i - 1 - (\mu_{ij} - 1)\text{II} \geq 0 && \\
&\text{C10:} && m_j - m_i - \mu_{ij}\text{II} \leq 0 && \\
&\text{C11:} && \epsilon_{ij} + \epsilon_{ji} + \mu_{ij} + \mu_{ji} \geq 1 &&
\end{aligned}$$

Constraints C1 represent the precedence relations. Note that the left-hand side is identical to the lifetime (or delay) in (1) which, of course, has to be zero or positive. Constraints C2 and C3 ensure the necessary modulo relations between t_i and m_i. The maximum number of resources for each resource class is defined by constraints C4. Constraints C5–C11 guarantee that every pair of operations o_i, o_j is either assigned to different resources or mapped to different congruence classes, or both. Note that we pre-scheduled the input nodes to start time zero.

B. Preliminaries on Register Modeling

For optimal register minimization, the ILP formulation has to take all lifetimes into account that actually describe registers in the implementation. The variable computed by node o_i has to be stored (delayed) by exactly n_{ij} clock cycles as given in (1), before fed into the hardware unit that computes node o_j. A naive way to implement time-multiplexed circuits is to put n_{ij} registers in each corresponding path as shown in Fig. 3a. The number of registers used in such an architecture is then identical to the cumulative lifetime

$$R_{\text{cl}} = \sum_{o_i \to o_j \in E} n_{ij} \ , \tag{2}$$

as used in [12], [6], [18]. Hence, a straightforward extension is to introduce n_{ij} as an ILP variable and to replace the objective by the minimization of R_{cl} to result in a minimum cumulative lifetime.

As can be seen in Fig. 3b, every time the data from node o_i is computed in the same hardware unit, the corresponding registers are redundant and leave room for optimization. This has been naturally utilized before in the generation of time-multiplexed circuits [22], but never with the objective to minimize the real number of registers, including possible sharing during binding and scheduling. Without changing the critical path, registers can be moved to the same branch, further reducing the number of implemented registers and to increasing the respective workload. Another possibility would be to reduce retiming. In the example that is shown in Fig. 3b, it is possible to move one register to the other side of the MUX. This could be profitable for the resulting clock frequency of the implementation. However, this optimization was not part of this work and will be investigated in the future.

Formally, let E_{cl} be the set of edges of resource-constrained operations which are bound to the same hardware unit

$$E_{cl} = \{o_i \to o_j \in E : r_i = l, o_i \in L_c\} \ , \tag{3}$$

the total number of registers using register sharing is

$$R_{\text{share}} = \sum_{c \in C} \sum_{l=0}^{a_c - 1} \underbrace{\max_{o_i \to o_j \in E_{cl}} n_{ij}}_{=R_{cl}} + \sum_{o_i \in O \setminus L} \underbrace{\max_{o_{i'} \to o_j \in E : i' = i} n_{ij}}_{=R_i} \ , \tag{4}$$

where R_{cl} and R_i denote the registers of resource constrained operations of class c bound to unit l as well as the non-constrained nodes o_i, respectively. Note that R_{share} is always less than or equal to R_{cl}.

C. ILP Formulation for Register Minimization

To make the ILP model aware of registers, we propose the addition of the variables given in the lower part of Table II. Variables R_{cl} and R_i are identical to the max terms in (4). Summing over all R_{cl}'s and R_i's results in our desired minimum register objective

$$\text{minimize } \sum_{c \in C} \sum_{l=0}^{a_c - 1} R_{cl} + \sum_{o_i \in O \setminus L} R_i \ .$$

Now, the following constraints have to be exchanged/added in the ILP formulation to minimize registers:

$$\begin{aligned}
&\text{C1a:} && t_j - t_i - D_i + d_{ij}\text{II} = n_{ij} && \left.\begin{array}{l} \\ \\ \end{array}\right\} \forall o_i \to o_j \in E \\
&\text{C1b:} && n_{ij} \geq 0 && \\
&\text{C14:} && r_i = \sum_{l=0}^{a_c - 1} l \rho_{il} && o_i \in L_c : \forall c \in C \\
&\text{C15:} && \sum_{l=0}^{a_c - 1} \rho_{il} = 1 && o_i \in L_c : \forall c \in C \\
&\text{C16a:} && R_{cl} \geq n_{ij} - (1 - \rho_{il})M && \begin{array}{l} \forall o_i \to o_j \in E, \forall c \in C, \\ l = 0 \ldots a_c - 1 : o_i \in L_c \end{array} \\
&\text{C16b:} && R_i \geq n_{ij} && \begin{array}{l} \forall o_{i'} \to o_j \in E : \\ o_{i'} = o_i \in O \setminus L \end{array}
\end{aligned}$$

Constraints C1a and C1b simply split C1 of the original formulation to access the lifetime n_{ij} as ILP variables. One

TABLE III: Results of the average register counts (avg. regs) and the number optimal solutions found within 5 minutes (opt) using maximum registers (maxReg), Moovac [19], minimum lifetime (minLife) [6] and minimum register (minReg)

		Graph properties				maxReg		Moovac [19]		minLife [6]		minReg (prop.)		Register impr. [%]		
	instance	graph no.	min ops	max ops	avg ops	avg regs	opt	avg regs	opt	avg regs	opt	avg regs	opt	maxReg	Moovac	minLife
MachSuite	aes2	16	29	225	70.8	191.1	14	123.5	14	47.1	14	31.8	14	83.4	74.3	32.5
	bfsqueue	2	105	121	113.0	216.0	2	125.0	2	49.5	2	47.0	2	78.2	62.4	5.1
	fftstrided	2	40	161	100.5	562.0	1	490.5	1	206.5	1	125.0	1	77.8	74.5	39.5
	gemmblocked	5	25	52	36.8	55.8	5	41.8	5	12.4	5	10.2	5	81.7	75.6	17.7
	gemmncubed	3	28	54	40.7	55.0	3	45.3	3	10.7	3	9.7	3	82.4	78.6	9.3
	kmp	4	33	92	58.0	63.8	4	45.8	4	25.0	4	24.5	4	61.6	46.5	2.0
	mdgrid	10	23	302	108.7	467.4	8	365.0	8	91.6	8	68.4	8	85.4	81.3	25.3
	mdknn	2	102	105	103.5	995.0	0	563.5	2	219.0	2	138.5	2	86.1	75.4	36.8
	sortmerge	8	26	80	41.9	38.6	8	24.4	8	11.4	8	9.5	8	75.4	61.1	16.7
	sortradix	15	20	75	36.5	34.1	15	20.5	15	8.0	15	7.7	15	77.4	62.4	3.7
	spmvcrs	2	52	56	54.0	92.5	2	80.5	2	17.0	2	16.0	2	82.7	80.1	5.9
	spmvellpack	2	49	54	51.5	85.5	2	72.0	2	17.0	2	16.0	2	81.3	77.8	5.9
	stencil2d	4	29	43	36.8	24.8	4	19.3	4	5.0	4	5.0	4	79.8	74.1	0.0
	stencil3d	9	25	76	40.9	30.8	9	24.2	9	9.2	9	7.1	9	76.9	70.7	22.8
	viterbi	7	33	81	60.9	131.4	7	104.3	7	34.0	7	30.1	7	77.1	71.1	11.5
CHStone	adpcm	30	20	689	82.7	596.7	28	324.7	28	108.2	28	84.4	28	85.9	74.0	22.0
	aes	22	26	1147	173.9	226.7	21	151.4	21	62.1	21	41.3	21	81.8	72.7	33.5
	blowfish	1	767	767	767.0	7495.0	0	4029.0	0	2968.0	0	2490.0	0	66.8	63.0	16.1
	dfdiv	2	40	43	41.5	18.0	2	15.5	2	8.0	2	8.0	2	55.6	48.4	0.0
	dfsin	3	40	2654	912.3	17338.3	2	9113.0	3	6020.0	2	3419.7	2	80.3	62.5	43.2
	gsm	15	20	194	64.7	76.7	14	51.9	15	22.7	14	21.7	14	71.7	58.2	4.4
	jpeg	113	19	913	99.1	315.0	104	195.7	104	88.3	104	76.3	104	75.8	61.0	13.6
	mips	1	1020	1020	1020.0	11232.0	0	5200.0	0	2116.0	0	858.0	0	92.4	83.5	59.5
	motion	51	32	79	53.2	45.2	51	36.2	51	10.0	51	10.0	51	77.9	72.4	0.0
	sha	25	20	146	70.2	82.8	25	58.7	25	24.9	25	23.9	25	71.1	59.3	4.0
Origami	butterworth	1	32	32	32	247	1	220	1	220	1	155	1	37.2	29.5	29.5
	firlms	1	15	15	15	54	1	46	1	46	1	46	1	14.8	0.0	0.0
	fir6dlms	1	16	16	16	25	1	13	1	8	1	4	1	84.0	69.2	50.0
	fir8tap	1	25	25	25	57	1	41	1	2	1	1	1	98.2	97.6	50.0
	fir16tap	1	49	49	49	114	1	103	1	2	1	2	1	98.2	98.1	0.0
	Hilbertfilter	1	14	14	14	6	1	6	1	1	1	1	1	83.3	83.3	0.0
	iirbiquad	1	14	14	14	20	1	16	1	17	1	14	1	30.0	12.5	17.6
	iirorder4	1	25	25	25	53	1	39	1	34	1	27	1	49.1	30.8	20.6
	PIAntiWindup	1	26	26	26	78	1	61	1	29	1	29	1	62.8	52.5	0.0
	PID	1	14	14	14	15	1	11	1	10	1	6	1	60.0	45.5	40.0
	RGBtoYCbCr	1	23	23	23	15	1	14	1	1	1	1	1	93.3	92.9	0.0
	splineprefilter	1	21	21	21	12	1	6	1	6	1	6	1	50.0	0.0	0.0
	YCbCrtoRGB	1	21	21	21	7	1	4	1	2	1	1	1	85.7	75.0	50.0
	pct	1	25	25	25.0	4	1	3	1	2	1	2	1	50.0	33.3	0.0
Total	MachSuite	91	20	302	59.6	158.8	84	113.7	86	37.7	86	27.9	86	82.4	**75.5**	**26.0**
	CHStone	263	19	2654	104.8	512.5	247	288.5	249	149.1	247	103.1	247	79.9	**64.3**	**30.9**
	Origami	14	14	49	22.9	50.5	14	41.6	14	27.1	14	21.1	14	58.2	**49.3**	**22.1**
	all	368	14	2654	90.5	407.5	345	235.9	349	116.9	347	81.4	347	80.0	**65.5**	**30.4**

problem to compute R_{cl} and R_i in (4) is that the *max* operation is non-linear. The common way to linearize it is to introduce a binary decision variable for each pair of *max* operations, indicating which of the operands is the largest one. Instead, we use a set of binary variables ρ_{il}, which indicate whether resource instance r_i is bound to unit l ($r_i = l$) or not. This relation is modeled by constraint C14. While this leads to the same number of decision variables compared to the common linearization of the *max* operations, it allows the reduction of the search space as exactly one ρ_{il} has to be one for each o_i as formulated by C15. This fixes many other variables whenever one ρ_{il} is set to a binary value in the branch-and-bound tree of the ILP solver. Now, the *max* operation can be modeled by C16a for resource constrained and C16b for non-resource constrained operations. Constraint C16a forces $R_{cl} \geq n_{ij}$ when $\rho_{il} = 1$. For $\rho_{il} = 0$, setting constant M (which is a so-called big-M constant) to a sufficiently large value will lead to a negative value on the right hand side of C16a with the effect that the constraint for this hardware unit is deactivated and the delay can be set to the minimum possible but usually constrained by another hardware unit. For non-resource constrained operations, this distinction is not necessary as given in constraint C16b. Due to the fact that R_{cl} is minimized it is guaranteed that the pure maximum value is selected and not a value larger than the maximum.

III. Experimental Results

In this section, we evaluate the performance of the proposed minimum register modulo scheduling approach. Our work is compared against the state-of-the-art modulo scheduler Moovac [19] which was shown to outperform previous work [8], [2]. As Moovac is not aware of the resulting registers and randomly applies operation binding, the resulting schedule usually requires a non-minimal register count. Additionally, we evaluate the minimum cumulative lifetime approach which is state-of-the-art when targeting register minimization [12],

[6], [18]. Finally, we examine a complementary objective that aims to maximize register numbers. In that way, we show the design space bounds regarding register counts for each modulo scheduling problem. We implemented our approach using Moovac as base. Minimizing R_{cl}, as given in (2), was used as objective. Doing this, we provide an optimal modulo schedule regarding both II (1st objective) and implemented registers (2nd objective). For each scheduling problem, all examined schedulers were constrained to fulfill the same maximum latency D_{max}. Note that this constraint limits the number of registers that the maximum register approach can utilize to a reasonable amount. In theory, an infinite amount of registers could be inserted. Using identical II and maximum latency for each scheduling problem, we examined the worst possible binding regarding register counts. Doing this, the displayed register counts are comparable. All methods were evaluated on 368 graphs from the `C`-based CHStone [11] and MachSuite [24] benchmarks as well as the Matlab/Simulink benchmark of Origami HLS [20].

A. Test Setup

The investigated schedulers were implemented in the open-source Origami HLS framework [20], [25]. We used the `C++` based open-source library ScaLP [26] as ILP interface and the Gurobi ILP solver [10] in single thread mode. For the `C`-based benchmark, a graphml interface to read models from the Nymble HLS compiler [13] was added. The resource limits in the C-based experiments were unlimited for all integer units except the integer division, which was limited to 8 units. Floating point Add/Sub/Mult and Div were limited to 4 and 8 units, respectively. To model common pipelining of floating point operations, the latencies for the model-based Origami benchmark experiments were set to realistic values. In detail, the `add`, `mult` and `sin` operations were set to 4, 5 and 12 clock cycles and resource constraints to 2, 2, 1 units, respectively. All other operations in both experimental setups were unconstrained and set to a single clock cycle latency.

All problems were run on a server system with Intel Xeon CPU E5-2650 v3 processors with 128 GB RAM operating at 2.3 GHz. The time limit of the solver was set to five minutes. Only if an optimal solution was found for the actual II and all lower IIs were proven to be infeasible within this time limit, the solution was counted to be optimal. Otherwise, the best solution found was considered as heuristic solution.

B. Results

The results for all examined applications are presented in Table III. Each row represents one application containing one ore more data flow graphs (given as 'graph no.'). For each of these graph sets, information about the number of operations in the smallest graph, largest graph and on average are given as 'min ops.', 'max ops.' and 'avg ops.', respectively. As an example, the `aes2` instance of the MachSuite benchmark consists of 16 graphs. The smallest graph representation contains 29 and the largest 225 operations. On average, the scheduled graphs of the `aes2` instance have 70.8 operations. The Origami benchmark set consists of several widely-used model-based designs, e.g., (in)finite impulse response filters, controllers and signal transformations.

In total, 368 graphs were used for the scheduling experiments. Each graph was modulo scheduled using four different formulations. As baseline, the Moovac formulation was evaluated. In addition, Moovac with minimum lifetime objective ('minLife') was evaluated to consider state-of-the-art register minimization. The proposed method for register minimization formulation ('minReg') as well as the formulation having the complementary objective ('maxReg') were examined to evaluate our approach as well as the possible range of register counts. Note that the register count of the conventional Moovac always must lie between the minReg and maxReg solutions. The average number of additional registers ('avg regs') for the different data flow graphs and methods are given in each row of Table III. In addition, the number of graphs for which the optimal schedule could be identified within the time limit are given in column 'opt'. The percentage of register improvement ('Register impr.') is given in the last three columns.

As expected, the results show that the minimum lifetime approach performs best in terms of registers from the state-of-the-art methods. However, the proposed minimum register formulation is able to find solutions with significantly reduced register counts in most of the cases. In the other cases, the same register count could be achieved. Remember that the solutions still fulfill the same II and latency for the examined scheduling problems.

Taking the average over all benchmarks, the improvement compared to the minimum cumulative lifetime approach and Moovac is 30.4% and 65.5% respectively. One can observe that larger graphs (like `mips` or `dfsin` in CHStone) seem to offer more optimization potential for register minimization than smaller graphs as less register sharing is possible. Here, minimum registers are often obtained by minLife scheduling and in two cases by Moovac ('firlms' and 'splineprefilter'). For most of the graphs, an optimal solution could be found. If not, even for large problems of $>$1000 nodes like the 'mips' instance good heuristic solutions (83.6% register count reduction compared to Moovac) were found which were already shown to outperform previous heuristics [19]. This shows that the proposed approach is applicable to commonly used large problems.

While the overall run time of Moovac was 4h:39min, it was 4h:55min for minimum cumulative lifetime and 5h:05min with the proposed extensions which corresponds to a run time increase of 9.3% and 3.4%, respectively. However, considering the achieved register savings, this increase seems to be acceptable.

IV. Conclusion & Outlook

An exact register minimization extension to ILP-based modulo scheduling was proposed that is capable to utilize register sharing. While previous solutions generate implementations using an arbitrary number of registers, this

paper shows that register optimal results can be achieved with low run time overhead for benchmark problems of practical size [11], [24]. Future work will be directed towards the integration of real hardware costs. In that way, the ILP-formulation is able to allocate hardware units dynamically and the hardware costs can be constrained using realistic values like the number of FPGA slices used or the area occupied on an ASIC.

REFERENCES

[1] A. Bonfietti, M. Lombardi, L. Benini, and M. Milano. CROSS Cyclic Resource-Constrained Scheduling Solver. *Artificial Intelligence*, 206:25–52, Jan. 2014.

[2] A. Canis, S. D. Brown, and J. H. Anderson. Modulo SDC Scheduling with Recurrence Minimization in High-Level Synthesis. In *IEEE International Conference on Field Programmable Logic and Application (FPL)*, pages 1–8. IEEE, 2014.

[3] L.-F. Chao, A. LaPaugh, and E. H. Sha. Rotation Scheduling: A Loop Pipelining Algorithm. In *Design Automation, 1993. 30th Conference on*, pages 566–572, 1993.

[4] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[5] J. Cong and Z. Zhang. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. In *Design Automation Conference (DAC)*, pages 433–438. IEEE, 2006.

[6] B. D. de Dinechin. Simplex Scheduling: More than Lifetime-Sensitive Instruction Scheduling. Technical Report PRISM 1994.22, 1994.

[7] B. D. De Dinechin. Time-Indexed Formulations and a Large Neighborhood Search for the Resource-Constrained Modulo Scheduling Problem. pages 144–151, 2007.

[8] A. E. Eichenberger and E. S. Davidson. Efficient formulation for optimal modulo schedulers. *ACM SIGPLAN Notices*, 32(5):194–205, May 1997.

[9] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Optimum Modulo Schedules for Minimum Register Requirements. In *International conference on Supercomputing (ICS)*, pages 31–40. ACM, 1995.

[10] Gurobi Optimization, Inc. http://www.gurobi.com, 2018.

[11] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-Level Synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[12] R. A. Huff. Lifetime-Sensitive Modulo Scheduling. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–267. ACM, 1993.

[13] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch. Hardware/Software Co-Compilation with the Nymble System. In *International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–8. IEEE, 2013.

[14] C.-T. Hwang, Y.-C. Hsu, and Y.-L. Lin. *Scheduling for Functional Pipelining and Loop Winding*. ACM, New York, New York, USA, 1991.

[15] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu. A Formal Approach to the Scheduling Problem in High Level Synthesis. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 10(4):464–475, 1991.

[16] R. Ito and K. K. Parhi. Register Minimization in Cost-Optimal Synthesis of DSP Architectures. In *VLSI Signal Processing, VIII*, pages 207–216. IEEE, 1995.

[17] P. Li, P. Zhang, L.-N. Pouchet, and J. Cong. Resource-Aware Throughput Optimization for High-Level Synthesis. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 200–209. ACM, 2015.

[18] J. Llosa, E. Ayguadé, A. González, M. Valero, and J. Eckhardt. Lifetime-Sensitive Modulo Scheduling in a Production Environment. *IEEE Trans. Computers*, 50(3):234–249, 2001.

[19] J. Oppermann, A. Koch, M. Reuter-Oppermann, and O. Sinnen. ILP-Based Modulo Scheduling for High-Level Synthesis. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'16)*, pages 1:1–1:10, 2016.

[20] Origami HLS Website. http://www.uni-kassel.de/go/origami, 2018.

[21] K. K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley & Sons, 2007.

[22] K. K. Parhi, C. Wang, and A. Brown. Synthesis of Control Circuits in Folded Pipelined DSP Architectures. *IEEE Journal of Solid-State Circuits*, 27(1):29–43, 1992.

[23] B. R. Rau. *Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops*. ACM, Nov. 1994.

[24] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks. Machsuite: Benchmarks for Accelerator Design and Customized Architectures. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 110–119. IEEE, 2014.

[25] P. Sittel, M. Kumm, K. Möller, M. Hardieck, and P. Zipf. High-Level Synthesis for Model-Based Design with Automatic Folding including Combined Common Subcircuits. *20. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 103–114, 2017.

[26] P. Sittel, T. Schönwälder, M. Kumm, and P. Zipf. ScaLP: A Light-Weighted (MI)LP Library. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 1–10, 2018.

[27] P. Šůcha and Z. Hanzálek. A Cyclic Scheduling Problem with an Undetermined Number of Parallel Identical Processors. *Computational Optimization and Applications*, 48(1):71–90, 2011.

[28] W. Sun, M. J. Wirthlin, and S. Neuendorffer. FPGA Pipeline Synthesis Design Exploration using Module Selection and Resource Sharing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):254–265, 2007.

[29] S. Venugopalan and O. Sinnen. ILP Formulations for Optimal Task Scheduling with Communication Delays on Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 26(1):142–151, 2015.