Periodica Polytechnica
Electrical Engineering and Computer Science

Hierarchical Pipelining of Nested Loops in High-Level Synthesis

58(3), pp. 81-91 (2014)
DOI: 10.3311/PPee.7610

Gergely Suba

RESEARCH ARTICLE

RECEIVED 15 JANUARY 2013; ACCEPTED 28 APRIL 2014

Abstract

Pipelining of the nested loops is very important in increasing the throughput of a system developed by a high-level synthesis tool. The most pipelining methods can handle only single loops. Therefore, nested loops are converted into a single loop, called loop flattened loop. In consequence, i.e. the sequential loops cannot be implemented in separate pipeline stages. This constraint limits the throughput. In this paper, a novel method are presented for nested loops by implementing to avoid this limitation. The method has the advantage that the desired restart time of the whole system can be given as an input parameter. The necessity of the pipeline scheduling on each loop hierarchy level can also be determined by this method. A novel multi-rate dataflow graph is also introduced for modeling the nested loops in an easy and abstract way.

Keywords

HLS · SDF · multi-rate · nested loops · pipeline scheduling

1 Introduction

High-level synthesis is based on many optimization methods in order to ensure a desired performance in speed, area and cost. The loops are essential parts of the algorithms to be implemented by a high-level synthesis tool. Therefore, the proper loop handling is unavoidable in increasing the throughput in a pipeline system. In achieving a given pipeline throughput (as the reciprocal of the initiation interval or restart time[1]), even the loops with constant trip count may set limit. Therefore, efforts have to make for decreasing the latency or restart time of the loop.

In this paper, a novel method is presented for increasing the pipeline throughput of nested loops with constant trip count. By this method, the pipeline scheduling can be performed on more than one level of the loop hierarchy simultaneously in contrast with the usual solutions. Compared with the previous works, this simultaneous scheduling has advantage, if the loop hierarchy contains also sequential loops. The method can map the sequential loops into successive pipeline stages[2], which may increase the throughput of a system significantly.

Contrary to the methods applied by the most HLS tools, the desired restart time (initiation interval) of the whole system can be given as an input parameter for the method presented in this paper.

The method can be used if every loop in the loop nest has constant trip count (number of iterations). Otherwise a transformation is needed to make the trip counts constant. For example, if an upper bound can be defined or estimated for the trip count of a loop, this bound can be used as constant trip count. In this case, some extra control solution ensure that the loop body is executed in the same number of times as the original trip count. Although this transformation increases the latency, and so it can cause some performance loss, in many cases the benefits of pipelining sequential loops is greater than the drawback of this performance loss.

Gergely Suba
Department of Control Engineering and Information Technology,
Faculty of Electrical Engineering and Informatics,
Budapest University of Technology and Economics
Magyar tudósok krt 2., H-1117 Budapest, Hungary
e-mail: sugergo@iit.bme.hu

1 Further the restart time (R) and Initiation Interval (II) will be used as synonyms

2 Let P_j^k denote the data processed by the operation e_j at the k-th initiation (restart). In the set of operations $E = \cup S_i$, the disjoint subsets S_i can be called pipeline stages, if $\forall k : P_s^k = P_r^k$ holds for $\forall e_s, e_r \subset S_i : \forall i$.

It is beneficial to perform the pipeline scheduling method on a dataflow representation, because these are formal and abstract models of a system. The operations inside a loop body are executed more times than the program itself, therefore a so-called multi-rate dataflow model [14, 10, 4] can be applied to represent nested loops. The well known multi-rate dataflow graphs (e.g. SDF, discussed later) cannot represent the nested loops in hierarchical way. Therefore, a novel dataflow graph based on existing single-rate dataflow models is introduced.

The paper is organized as follows. In Section 2, the previous work in the area of nested loops pipelining and the related dataflow models are reviewed. In Section 3, a novel dataflow graph, the so-called MR-HSDFG is introduced. In Section 4, the latency and restart time calculation is presented in the MR-HSDFG model. Section 5 discusses the optimization of these two parameters. In Section 6, experimental results are presented. The conclusion is summarized in Section 7.

2 Previous Work

In this section, the most relevant methods are evaluated regarding the pipeline scheduling of nested loops. Further on, the dataflow graph representations are compared, which are suitable for modeling nested loops.

One of the approaches is the hierarchical reduction [12] method. In this way, the program (represented in dataflow graph) is scheduled hierarchically, starting with the innermost loop. After this scheduling, the whole loop is substituted by a single operation. The same scheduling method will be executed for an outer loop, if it does not contain inner loops, only operations (every loop has already been substituted by operations). At the end of this procedure, the entire program is reduced to a single operation. The paper [12] contains only the main concept of the reduction, and it does not discuss the precise algorithm.

Another hierarchical approach is the hierarchical pipelining [3]. In this method, four levels of the hierarchy are defined: system, behavior, loop and operation. The description of the levels are not uniformed; each level has its own description graph (CFG, CDFG or DFG), and the handling of the levels of the hierarchy is different, which is a drawback of the method. The throughput constraint, given as input of the method, determines the duration time of the stages. The procedure starts with filling the stages node by node. A node is assigned to the same stage as the predecessor node, if the duration time of the stage will not be exceeded by this assignment. Next, the throughput constraint of each pipeline stage is distributed among the nodes (i.e. loops) within the stage. These distributed constraints limit the latency of the nodes. This heuristic, which is a substantial part of the approach, does not guarantee an optimal solution [3] in contrast to the method presented in this paper, nevertheless this is the nearest approach to the basic objectives of our method.

Another approach to perform pipelining is to flatten [6] the loop nest first, then the resulted single, non-nested loop can be pipelined using the single loop pipelining methods [11, 12]. The main idea of the loop flattening is to emulate the execution of the original loop nest by a single (called flattened) loop, where the trip count is equal to the total sum of inner loop trip counts. The method calculates that which iterations of the original loops should be executed for each iteration of the flattened loop. Several commercial HLS tools apply this approach, e.g. Calypto Catapult C [5] or Xilinx Vivado [1].

The advantage of the loop flattening is that it can handle also some types of loops having non-constant trip counts. The disadvantage of the loop flattening regarding the pipelining is that the original inner loops disappear. Therefore, the pipeline scheduling cannot be executed level by level (e.g. the sequential loops cannot be mapped into successive pipeline stages). Thus, the throughput of the whole system may be decreased.

Another way is to use a self-timed ring representation to perform the pipeline scheduling of nested loops [7]. This method is dedicated to asynchronous pipeline mode where a handshake control is assumed in each stage. This solution makes the run-time pipeline loop scheduling easier, but the necessary control overhead is significant. The self-timed ring method can be applied only for asynchronous systems, which is out of the scope in this paper.

The method presented in this paper is based on the hierarchical reduction, i.e. in contrast with the flattening way, it is made level by level in the loop hierarchy.

In handling the nested loops, the dataflow representation is widely used. The main types of such models can be classified as follows.

For digital signal processing, the application of Synchronous Data Flow (SDF) [13] is typical. In SDF the nodes represent operations (called actors), and the edges represent communication channels realized by FIFO queues. The FIFOs connect to the ports of the actors. For each port of an actor the number of produced or consumed tokens is defined. An input port consumes tokens from the predecessor FIFO, and an output port produces tokens to the successor FIFO. The number of produced and consumed tokens of an actor can differ, because the frequency of the fires (executions) of the actors may vary. Therefore, the SDF can be considered as a so-called multi-rate [14] dataflow graph.

The Homogeneous SDF (HSDF) is a simplified variant of the SDF, where the number of produced and consumed tokens of each actor is always 1. Therefore, the HSDF can be considered as a so-called single-rate [14] DFG.

As the HSDF is a very simplified model, it is easier to analyze than the generic SDF model. Besides, the SDF model has disadvantage that the most analyzing and optimizing algorithms have to start with transforming the SDF into HSDF. These transformations increase the number of nodes significantly. [13] Therefore, the SDF representation is practically not applicable in case of great number of operations.

An approach resembling the HSDF is the so called Elementary Operation Graph (EOG) [2]. This model is also single-rate, but it contains also timing parameters in contrast with the HSDF. The nodes are considered as elementary (not separable) operations. For each operation, the duration (execution) time is given, where duration time 0 refers to combinatorial unit. For the elementary operations of the EOG, the following assumptions are made:

1. operation v_i is started only after having finished $\forall v_i \in V$, where v_j is the direct predecessor operation of v_i and V is the set of the operations
2. operation v_i requires all its input data during the whole duration time of the v_i (this duration time is denoted by t_{vi} in the following)
3. operation v_i may change its output during the whole duration time
4. after the output of v_i shows, holds its current output stable until its next start

The EOG supports the pipeline scheduling and allocation algorithms. For calculating and optimizing the restart time, there are algorithms in [2] that will be reused in this paper. These algorithms handle the loops as single elementary operations, without defining the inner behavior of them. Therefore, the inner operations remain hidden and so there are no way to involve them into the pipelining algorithms. The aim of this paper is to represent and handle each level of the loop hierarchy separately.

3 MR-HSDFG - a novel multi-rate dataflow graph

In this section, a novel dataflow model, the Multi-Rate Homogeneous Synchronous Dataflow Graph (MR-HSDFG) is introduced, which can be considered as an extension of the EOG and HSDF model. The main purpose is to represent nested loops in an abstract and formal way for performing the pipeline scheduling efficiently. The dataflow models overviewed as previous work do not solved this problem for practical application. The HSDF and the EOG are single-rate DFG, so they cannot represent nested loops for an efficient pipeline scheduling. Although, the SDF can represent nested loops, but it is always converted the model to HSDF for analyzing and scheduling. This step increases the number of the operations significantly.

The MR-HSDFG is a finite and directed graph. Each node represents an operation, which has zero[3] or more input and zero or one output (but, both of them cannot be zero). Each edge is a dataflow channel, representing the data transfer between operations.

The assumptions of the MR-HSDFG differ partially from the EOG:

1. the same as in the EOG (see Section 2)
2. the same as in the EOG (see Section 2)
3. operation v_i must not change its output during the whole duration time
4. after the output of v_i shows, holds its actual output stable until the next output value is calculated and sampled by the clock

Based on these assumptions, the duration time of a non-pipelined operation is identical with its so called busy time (denoted by q_{vi} for the operation v_i) contrary to [2]. In MR-HSDFG pipelined operations also may occur. For such operations the duration and busy times may be given separately as parameters.

The model has five types of operations:

- Elementary operation: an operation, which is considered as atomic (inseparable). In EOG this is the only operation type.
- Loop operation: an operation that represents a loop. The behavior of the loop body is defined by an inner dataflow graph (discussed later), which is also a MR-HSDFG, so it can contain loop operations, too.
- Constant operation: an operation, which produces a constant value in runtime (it has no input dataflow channel).
- Input operation: represents an input itself of the dataflow graph (it has no input dataflow channel).
- Output operation: represents an output itself of the dataflow graph (it has no output dataflow channel).

The duration time of the constant, input and output operations are 0 by definition.

The loop operations are essential for modeling and handling the nested loops. The general structure of the loop operation is shown in Figure 1, where there are special fixed operations (Counter and Selector) and a task dependent subgraph, called inner graph representing the loop body. (this inner graph is illustrated by a single node in Figure 1)

In order to describe the inner graph as a MR-HSDFG, some transformations are shown in Figure 2. The Counter, which counts the number of the loop body executions, can be considered as an input operation of the inner graph. The Selector as an output operation enables in a downsampling way which value from the loop body should be transferred to the direct successor operation. When the enable input is on, the value will be buffered to the output of the Selector operation, and will be held until the next enable sign. The input data of the loop operation arriving from other operations are assumed to be stable during the whole loop operation. Therefore, the input data are considered as constant operations.

A loop operation can be formally defined as a pair $\langle G, tc \rangle$, where:

- G is the inner graph (MR-HSDFG), which represents the loop body
- $tc \in N$ is the trip count of the loop, where $tc > 1$ (otherwise there wouldn't be a loop). While the loop operation is performed once, G should be restarted tc times, i.e. tc is used as a parameter of the Counter.

Such a MR-HSDFG representation fits to the nested loops written in a programming language by applying the structure of

3 It is a difference between MR-HSDFG and EOG, because in EOG every operation has at least one input and one output.

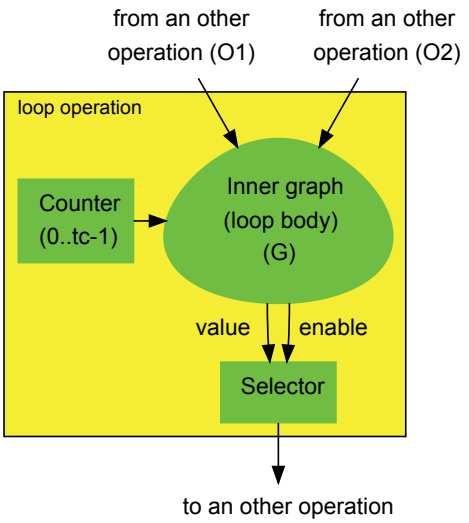


Fig. 1. General construction of the loop operation (in this figure the loop operation has two predecessor operations)

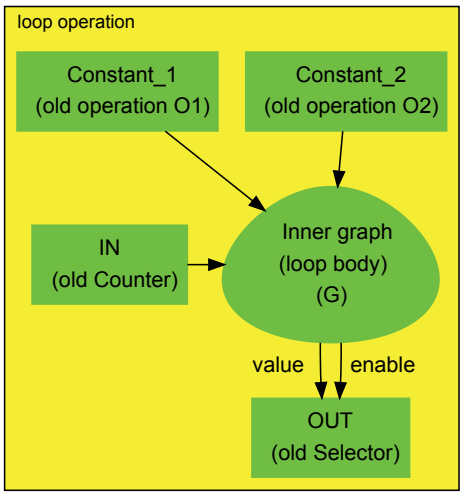


Fig. 2. Inner MR-HSDFG of the loop operation

Figure 1, loop by loop separately. The main (top-level) function can also be described by an MR-HSDFG. The loops inside the main function can be mapped to loop operations as shown in Figure 1. The other parts of the main function can be represented as elementary and constant operations, because these parts run just once during each execution of the main function. The main function parameters and the return statement are represented as input and output operations.

The *tc* is the trip count of the loop, considered as a parameter of the Counter operation for defining the upper bound of the counting. If the trip count is data dependent, the user should define an upper bound for *tc*. The commercial software tools (e.g. Catapult C) generally also assume this user interaction. In such cases, it is an additional task for the user to ensure a proper function if the defined *tc* is exceeded.

The MR-HSDFG is illustrated by an example. Let's consider the following expression as the task to calculate $f(a, n)$:

$$f(a,n) = \prod_{i=1}^{n} (a+i), n \in [1, n_{max}], n \in \mathbb{N} \qquad (1)$$

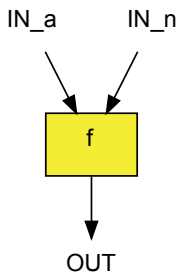


Fig. 3. EOG representation of the task

Inputs a and n are assumed to be given repeatedly. In this case, the operation $\prod_{i=1}^{n}$ should be represented by a loop.

In a poor EOG representation the f function is practically handled as a simple elementary operation (shown in Figure 3) where the inner behavior is hidden. In case of this representation, pipeline scheduling cannot be performed for the inner behavior of f. However, the f function is represented as a loop operation in MR-HSDFG, where the inner graph represents the behavior of the loop body as shown in Figure 4. The inner graph without its environment is illustrated in Figure 5. Since this inner graph doesn't contain loop operation, it can be considered as a regular EOG. Therefore, the analyzing and optimizing algorithms developed to the EOG [2] can be applied on it by some extension. The following sections introduce these extended algorithms.

4 Calculation of the minimal restart time on MR-HSDFG

In this section, an algorithm is introduced for calculating the minimal restart time of the system under three constraints.

If the MR-HSDFG does not contain loop operation, the model will be similar to the EOG.

In this case, the latency and the restart time can be calculated in EOG as shown in [2] by assuming that the EOG is acyclic. In the following, these algorithms are extended loop operations by applying MR-HSDFG.

In order to form an acyclic graph, the operations inside each cycle[4] has to be substituted by a single operation first.

Let $L(G)$ the latency of graph G (as an MR-HSDFG) as the sum of the duration times of the longest path in G. The calculation of the minimal restart time is shown for the three cases as follow:

- **Non-pipelined mode:** the system is restarted when the previous input data has already arrived at the output. In this case, the throughput can be low, but the necessary area may be decreased by proper scheduling and allocation.

$$R_{np}(G) = L(G) = \sum t_{v_i} \qquad (2)$$

4 Cycle in the graph, that is directed path goes from a node to the same node.

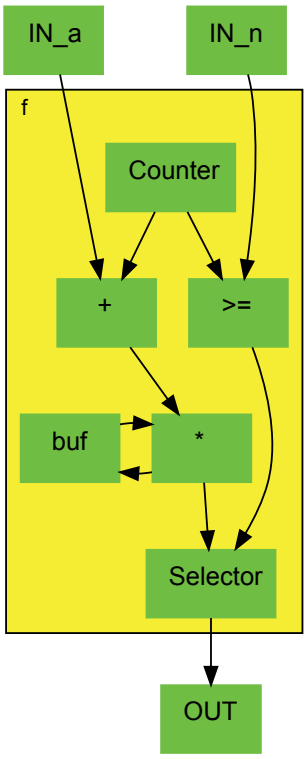


Fig. 4. MR-HSDFG representation of the task (the =, +, <=, ∗ and buf operations represent the loop body, called inner graph)

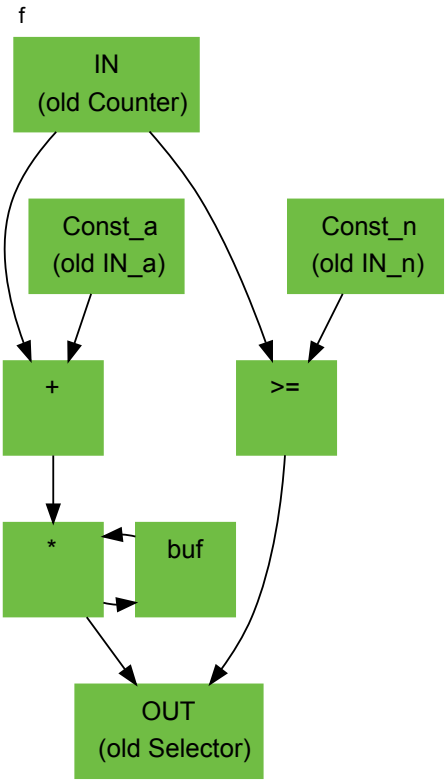


Fig. 5. MR-HSDFG representation of the inner graph

where $\sum t_{vi}$ denotes the sum of all duration time in the critical (longest) data path.

- **Pipeline mode without replication of operations:** in this case, the required area may be larger because of stronger allocation constraints than in the non-pipelined mode.

$$R_{pmr}(G) = max(q_{v_i}) \quad \forall v_i \in V \tag{3}$$

- **Pipeline mode assuming replicated operations:** in this case, the $R_{min}(G) = 1$ can be achieved, if each operation is allowed to be replicated. By extending the calculation in [2]:

$$R_{min}(G) = max\left(q'_{v_i}\right)$$
$$q'_{v_i} := \begin{cases} 1 & \text{if } v_i \text{ can be replicated} \\ q_{v_i} & \text{if } v_i \text{ cannot be replicated} \end{cases} \tag{4}$$
$$\forall v_i \in V$$

where V is the set of the operations, $t_{v_i} \in \mathbb{N}$ is the duration time (number of clock periods required for execution) and $q_{v_i} \in \mathbb{N}$ is the busy time of the v_i operation. Based on the above definitions

$$R_{np}(G) >= R_{pmr}(G) >= R_{min}(G) \tag{5}$$

holds.

For the calculations (2), (3) and (4), the duration and busy time of each operation is required. For most of the operations, these values are given by parameters. To determine the duration and busy $\langle G, tc \rangle$ time of a loop operation, extra calculations are needed. The inner behavior of the loop operation v_i is defined by the pair. For the sake of simplicity, let be assumed that a loop operation cannot be restarted more frequently than its latency. In this case, the duration time and the busy time are identical. The calculation can be formulated as follows:

$$q_{v_i} = t_{v_i} = (tc_i - 1) * R(G_i) + L(G_i) \tag{6}$$

where $R(G_i)$ is the restart time of the inner graph inside the loop operation v_i, and $L(G_i)$ is the latency of that. Expression (6) can be explained as detailed below.

The inner graph G_i of the loop operation v_i is also MR-HSDFG, and its restart time $R(G_i)$ can be defined by the Expressions (2), (3) or (4). While the loop operation is executed once, the inner graph is restarted tc times, therefore between the first and last restart $(tc_i - 1) * R(G_i)$ time elapsed. After the last restart, the loop operation is still busy for the time $L(G_i)$ (this time is needed to process the last input). Therefore, this value should be added in the expression, to get the whole duration and busy time of the loop operation.

For nested loops, this calculation should be applied in a bottom-up recursive way beginning at the innermost loop.

5 Obtaining a desired restart time based on MR-HSDFG

The method in [2] based on EOG operates with buffer insertion and operation replication. This is a so called RESTART algorithm, which is adapted to MR-HSDFG as follows. In

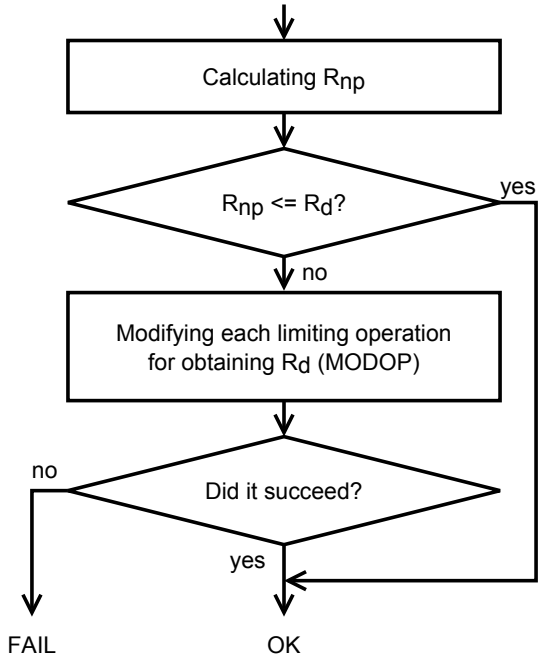


Fig. 6. Extended RESTART algorithm

MR-HSDFG, the modified version of this optimization method can be used, which will be introduced in this section.

For obtaining a desired restart time R_d, the extended RESTART algorithm handles the loop hierarchy in a top-down recursive way.

The overview of the achieving algorithm is illustrated in Figure 6. The first step is to calculate the restart time in non-pipelined mode (R_{np}, details in Section 5.1). If the resulted R_{np} does not exceed R_d, the whole algorithm can be stopped successfully. If it is not the case, each limiting operation has to be modified by algorithm MODOP (details in Section 5.2). The extended RESTART algorithm may fail to obtain R_d, if $R_d < R_{min}$.

5.1 Calculating the non-pipelined restart time (R_{np})

The first step is to calculate the duration time of each loop operation, in order to determine the non-pipelined mode restart time. The non-pipelined mode offers the most freedom in allocation. (it can result the least resource using and the largest amount of allocating) Expressions (2) and (6) in Section 4, detail the calculation for determining R_{np}.

5.2 Modifying the loop body

If R_d is smaller than the calculated R_{np}, then each such operation which limits the restart time has to be modified one by one. This modifying algorithm (MODOP) is illustrated for a single operation v_i in Figure 7.

If v_i is a loop operation, then the replication of the whole operation is attempted to avoid by pipelining the loop body (calculation details in the following subsection).

5.2.1 Calculation the required loop body restart time $R_d(G_i)$

Pipelining the loop body can reduce the duration time of the whole loop operation, if the latency of the loop body is greater than 1.

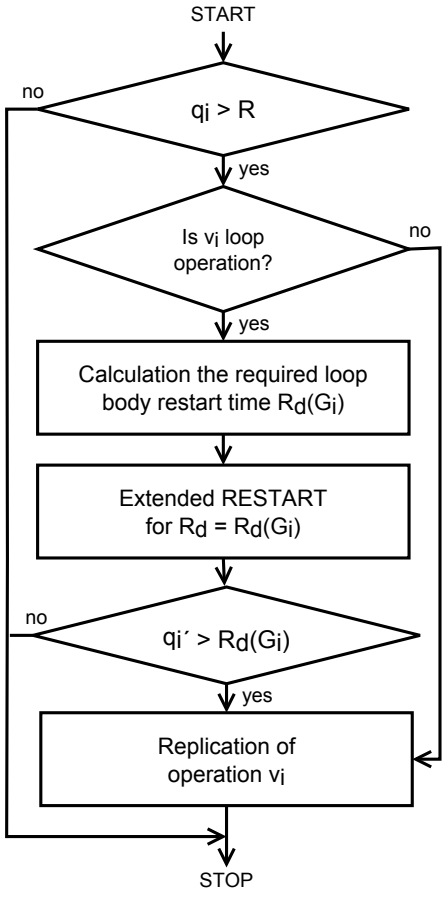


Fig. 7. Algorithm MODOP in details

In case of a loop operation, which doesn't satisfy the desired restart time R_d, an effective way is to pipeline schedule the inner graph of the loop operation. This inner scheduling can decrease the duration time (which is calculated for the non-pipelined mode first as illustrated in Section 5.1) of the loop operation. In the following, the pipeline scheduling of this inner graph will be discussed.

The aim is to set

$$q_{v_i} \leq R_d \tag{7}$$

preferably by pipelining the loop body without replication. By substituting Expression (6) into Expression (7):

$$(tc_i - 1) * R(G_i) + L(G_i) \leq R_d \tag{8}$$

By expressing $R(G_i)$,

$$R(G_i) \leq \frac{R_d - L(G_i)}{tc_i - 1} \tag{9}$$

holds. Since a restart time can only be a positive integer, the required restart time of the loop body can be expressed by applying Expression (4) as follows:

$$R_d(G_i) = max\left(\left\lfloor \frac{R_d - L(G_i)}{tc_i - 1} \right\rfloor, R_{min}(G_i)\right) \tag{10}$$

As the inner graph is an MR-HSDFG, the algorithm illustrated in Figure 6 has to be executed formally in a top-down way in case of nested loops.

The resulted duration time t_{vi} and busy time q_{vi} of the whole loop operation v_i can be calculated according to Expression (6) by substituting $R_d(G_i)$ instead of $R(G_i)$.

5.2.2 Replication of v_i

If the operation is not a loop operation, or the inner pipelining step (Section 5.2.1) cannot ensure the desired restart time (R_d), then the only way to achieve R_d is the replication. The required number of copies (c_i) for v_i can be determined as follows[5]:

$$c_i = \left\lceil \frac{t_i}{R} \right\rceil \tag{11}$$

6 Experimental results

In this section, two tasks as examples are analyzed. The first one (edge detection algorithm) demonstrates the advantage of the MR-HSDFG modeling for calculation the non-pipelined restart time. The second one (divisor calculation) illustrates this advantage in ensuring the desired restart time also for loops.

For comparison, Catapult C has also been applied in both tasks.

6.1 Edge detection algorithm

The algorithm is based on the Roberts operator [15]. A 64 x 64 pixel size grayscale bitmap is assumed. This bitmap is transformed to a same size bitmap for enhancing the edges of the original bitmap. The highest intensities in the transformed bitmap represent the edges.

The data interface of the algorithm consists of two streams, one for the input (original) bitmap and one for the output (transformed) bitmap.

According to the algorithm [15], the intensities $(p_1 \ldots p_4)$ of four input pixels are required for calculating the intensity $P_{\text{out}}(x, y)$ as a function of the output pixel coordinates:

$$\begin{aligned} p_1 &= P(x-1, y-1) \\ p_2 &= P(x+1, y-1) \\ p_3 &= P(x-1, y+1) \\ p_4 &= P(x+1, y+1) \end{aligned} \tag{12}$$

where (x, y) denotes the pixel coordinates.

The calculation is based on the Roberts operator [15]:

$$P_{out}(x, y) = max\left(|\, p_1 - p_4 \,|, |\, p_2 - p_3 \,|\right) \tag{13}$$

As it can be seen, the two adjacent rows has to be used for calculating the intensity in a given row. These two rows should be stored simultaneously besides the current row, thus altogether three rows.

As the access of the bitmap is sequential (the data arrives in a stream), these 3 rows should be buffered.

The Catapult C source code of the algorithm is shown in Figure 8, where there are three for loops. Loop I iterates through the rows (between 0 and 63), and loops J1, J2 iterate through the columns (between 0 and 63 too). In each iteration of loop I, one input row is processed, and one output row is produced. The task of loop J1 is to read the input stream, and to store the data in a temporary buffer. Loop J2 calculates the next output pixel one by one using the input stream and the values stored in the temporary buffer.

The MR-HSDFG constructed from this code is shown in Figure 9. The results are summarized in Table 1. In the first and fourth row the restart time of the outer loop I seems to be approximately equal to the sum of the duration time of the operations inside the loop body:

$$R_3 = \sum t_{v_i} \tag{14}$$

where the duration time t_{v_i} of the loop operations can be calculated by applying the Expression (6). In this case, $R_3 \approx (63R_1 + L_1) + (63R_2 + L_2)$.

For the second and the third row contain the results obtained from Catapult C by running it with flattening. In this case, loops J1, J2 disappeared.

The parameter $II_1 = 1$ (in Catapult C II denotes the initiation interval, which is the same as restart time) in the third row cannot be satisfied, because the minimal restart time R_{min} of the original (without flattening) loop J2 is 2, because the two Read operations connect to the same buffer.

The results in the fourth row are provided by Catapult C by running it for the inner loops separately without flattening. In this case, the desired restart time for the inner graph of loop J1 has been 1, and 2 for the inner graph of loop J2. This throughput result (12609) seems to be the best provided by Catapult C.

The last row represents the result obtained by applying the method presented in this paper. In this case, J1 and J2 are assigned to different pipeline stages, therefore better result (8384 cycle instead of 12609) is achieved compared with the Catapult C.

6.2 Divisor calculation

This task as an example calculates the average of 64 integer input values and determines the number of divisors of this average, considered as a lower integer. The Catapult C code represents the algorithm is shown in Figure 10. The code contains two for cycles, the first one (J1) deals with the calculation of the average, and the second one (J2) calculates the number of divisors.

The algorithm is illustrated by MR-HSDFG in Figure 12 where J1 and J2 are loop operations.

If the user defines R_d as the minimal pipeline restart time without replication, then the following calculation provides the result. By applying Expression (6), the duration time (and busy time) for the loop operation J1 is $(64 - 1) * 1 + 3 = 66$ (for the inner graph G_{J1}, $R_d = R_{pnr}(G_{J1}) = 1$ can be satisfied according to Expression (3)). For the loop operation J2, Expression (6) provides the duration time (and busy time) $(256 - 1) * 5 + 7 = 1282$

5 In [2], the required number of copies is calculated as $\left\lceil \frac{t_i+2}{R} \right\rceil$. Because of the MR-HSDFG assumption 3 (each output is buffered), +2 can be neglected.

Tab. 1. Results for the edge detection algorithm (R_i is the restart time of the inner graph, L_i is the inner latency of the given loop body. The notations II_I, II_{J1}, II_{J2} refer to the Catapult C terminology corresponding to the restart time of the inner graph GI , GJ1 and GJ2.)

	Tools and mode	Loop J1 (tc_1=64)	Loop J2 (tc_2=64)	Loop I (tc_3=64)	Throughput cycles
1	Cat. without pipelining	R_1=2, L_1=2	R_2=4, L_2=4	R_3=386	24705
2	Cat. II_I=2	flattened	flattened	R_3=2 (tc_3=8128)	16260
3	Cat. II_I=1	flattened	flattened	-	*cannot be satisfied*
4	Cat. II_I=1, II_I=2	R_1=1, L_1=2	R_2=2, L_2=4	R_3=197	12609
5	MR-HSDFG R_d=R_{pnr}	R_1=1, L_1=3	R_2=2, L_2=4	R_3=131	8384

Tab. 2. Results (R_i is the restart time of the inner graph, L_i is the inner latency of the given loop body, The notations II_{J2}, II_M refer to the Catapult C terminology corresponding to the restart time of the inner graph G_{J2}, II_M.)

	Tools and mode	Loop J1 (tc_1=64)	Loop J2 (tc_2=256)	Throughput cycles	Area
1	Cat. without pipelining	R_1=1, L_1=1	R_2=6, L_2=6	1603	589
2	Cat. II_{J2}=1	R_1=1, L_1=1	R_2=1, L_2=6	329	2880
3	Cat. II_M=1	*flattened*	*flattened*	(L_M=8) 319	3071
4	Cat. II_{J2}=1, U_{J2}=2	R_1=1, L_1=1	R_2=1, L_2=6 (tc_2=128)	202	5733
5	Cat. II_M=1, U_{J2}=2	*flattened*	*flattened* (tc_2=128)	(L_M=10) 191	5968
6	Cat. II_M=1, U_M=2	*flattened* (tc_1=32)	*flattened* (tc_2=128)	*cannot be satisfied*	
7	MR-HSDFG R_d=R_{pnr}	R_1=1, L_1=3	R_2=5, L_2=7	1282	622
8	MR-HSDFG R_d=262	R_1=1, L_1=3	R_2=1, L_2=7	262	2447
9	MR-HSDFG R_d=R_{min}	R_1=1, L_1=3	R_2=1, L_2=7	66	9789

(for the inner graph G_{J2}, $R_d = R_{pnr}(G_{J2}) = 5$ can be satisfied according to Expression (3)) This situation is outlined in Figure 11(a).

Let it be assumed now that the user wants to apply $R_d = R_{min}$ for the whole algorithm. According to Expression (4) $R_{min} = 66$, because the replication of the operation Stream In is impossible. Therefore, the replication of the whole loop J1 is excluded. Thus, the only obstacle in achieving $R_d = R_{min} = 66$ is loop J2 with its $R(G_{J2}) = 1282$ in Figure 11(a). In order to eliminate this obstacle, the required restart time for the inner graph of the loop J2 can be calculated by applying Expression (10):

$$R_d(G_{J2}) = max\left(\left\lfloor \frac{66-7}{256-1} \right\rfloor, 1\right) = 1 \qquad (15)$$

Inside the inner graph G_{J2}, only the operation % is to be replicated, where the number of copies is $\left\lceil \frac{5}{1} \right\rceil = 5$ (Expression (11)). In this case, the $R_d(G_{J2}) = 1$ can be satisfied and the duration time of the loop operation J2 will be $(256 - 1) * 1 + 7 = 262$ (Expression (6)). Thus, the loop operation J2 has to be replicated in $\left\lceil \frac{262}{66} \right\rceil = 4$ copies (Expression (11)) to achieve $R_d = 66$ for the whole algorithm. This case is outlined in Figure 11(b).

The Catapult C and MR-HSDFG results are summarized in Table 2. Row 1 shows the Catapult results, if pipelining is not aimed. In Row 2, loop J2 is pipelined, because it seems to be the bottleneck in Row 1. Row 3 shows the results of flattening the main loop. This result is similar to the previous row, because pipelining the loop J1 is ineffective. For further increasing the throughput, loop unrolling [8, 9] method is applied (Rows 4-6), but the area cost will be increased significantly in this cases. The requirement in Row 6 cannot be fulfilled, since loop J1 cannot be unrolled, because of the operation Stream In.

The results of the method presented in this paper are summarized in Row 7-9. In Row 9, it is remarkable that the area is very large, but the throughput cycle is the best and it seems not to be achieved by the method of Catapult C. Row 8 shows the trade-off between the area and the throughput cycle. In this solution, the % operation inside the loop operation J2 is replicated, but the whole J2 is not. Compared with the Catapult C result in Row 2 the area is less, and the throughput is with 20% better.

7 Conclusion

A novel method for pipelining of nested loops with constant trip count, has been presented. The main advantage of the method is that the pipeline optimization can be performed in each level of the loop hierarchy separately. In this way shorter restart time (initiation interval) can be achieved contrary to the flattening based approaches. The method presented in this paper is based on the novel dataflow graph model (MR-HSDFG) for represented the nested loops by applying the multi-rate dataflow property. On two experimental algorithm the method has been evaluated and compared with the results provided by the commercial HLS tool Catapult C.

```
#include "ac_fixed.h"
#include "ac_channel.h"
#include "type.h"

#pragma hls_design top

void test (
ac_channel<int> &data_in,
ac_channel<int> &data_out)
{
  int buf[4][64];

  I: for (int i=0; i<64; i++)
  {
    int is = i % 4;
    int ia = (i-1) % 4;
    int ib = (i-3) % 4;

    J1: for (int j=0; j<64; j++)
    {
      buf[is][j] = data_in.read();
    }

    int a1=0, b1=0, a2=0, b2=0;

    J2: for (int j=0; j<64; j++)
    {
      int a = buf[ia][j];
      int b = buf[ib][j];

      int atlo1 = a-b2;
      int atlo2 = b-a2;
      if (atlo1<0) atlo1 = -atlo1;
      if (atlo2<0) atlo2 = -atlo2;

      a2 = a1; a1 = a;
      b2 = b1; b1 = b;

      int el = atlo1 > atlo2
               ? atlo1 : atlo2;

      data_out.write(el);
    }
  }
}
```

Fig. 8. Catapult C source code of the task

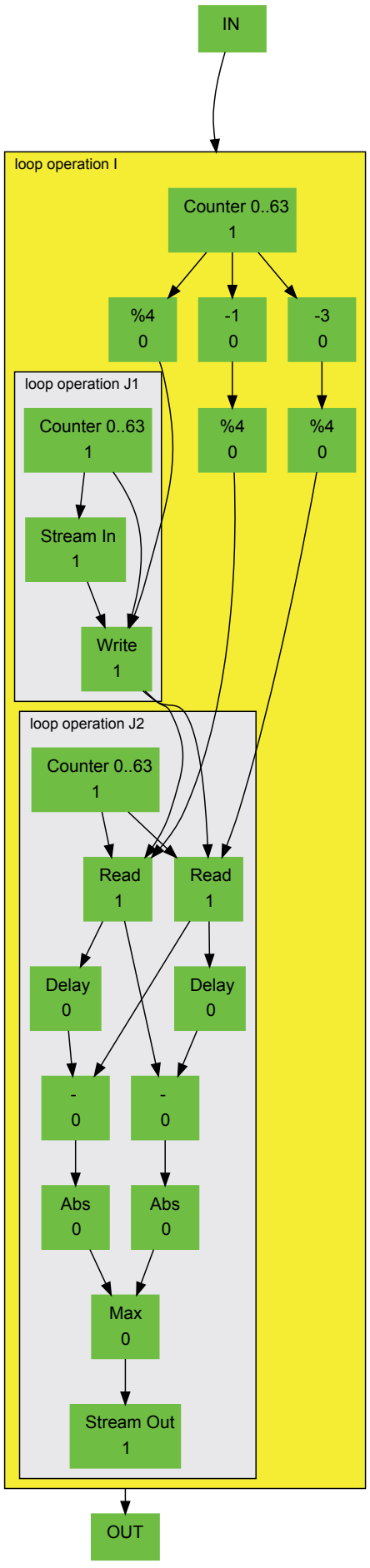


Fig. 9. MR-HSDFG of the edge detection algorithm

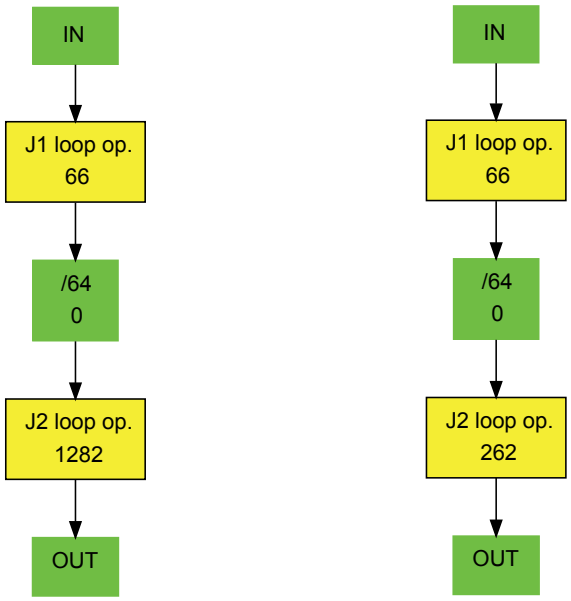
```
#include "ac_fixed.h"
#include "ac_channel.h"
#include "type.h"

#pragma hls_design top

int test(ac_channel<char> &data_in)
{

  int sum=0, count=0;
  for (int j=0; j<64; j++) {
    sum += data_in.read();
  }
  int number = sum / 64;
  for (int i=1; i<256; i++) {
    if (number%i==0 && i<=number)
      count++;
  }
  return count;
}
```

Fig. 10. Catapult C source code of the divisor algorithm



(a) R_d=R_{prn}=1282 (b) R_d=R_{min}=66

Fig. 11. MR-HSDFG of the divisor algorithm

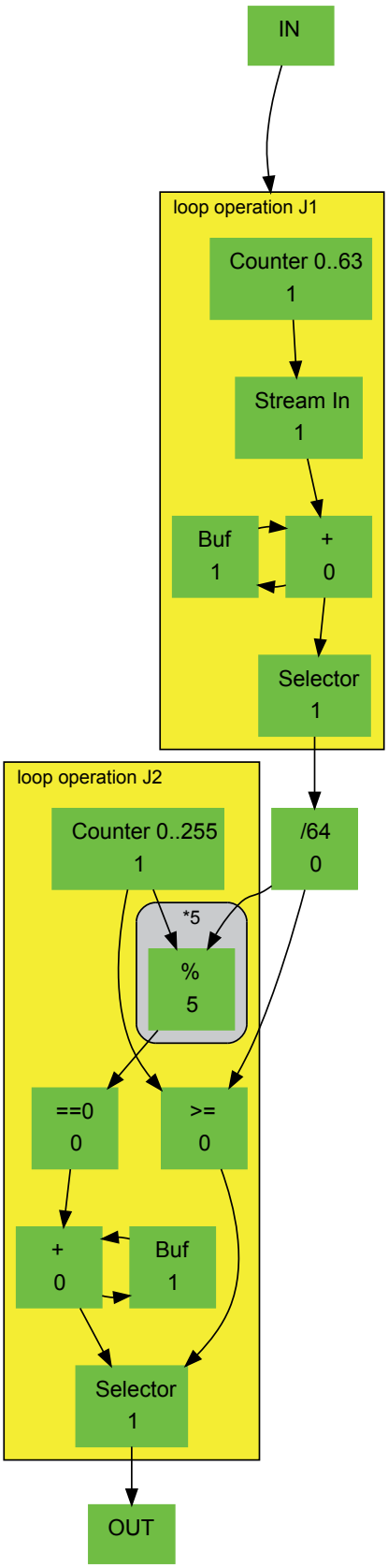


Fig. 12. MR-HSDFG of the divisor algorithm

Acknowledgement

The research work presented in this paper has been supported by the Hungarian Scientific Research Fund (OTKA 72611 and by the "Research University Project" TAMOP IKT T5 P3.

The author thanks to his supervisor, Prof. Dr. Péter Arató for the support and help in applying and extending the algorithms in [2].

References

1 Vivado design suite user guide - high-level synthesis (v2012.4), dec. 2012.

2 **Arato P., Visegrady T., Jankovits I.,** *High Level Synthesis of Pipelined Datapaths*. John Wiley & Sons, Inc., New York, USA, (2001).

3 **Bakshi S., Gajski D. D.,** P*erformance-constrained hierarchical pipelining for behaviors, loops, and operations*. ACM Transactions on Design Automation of Electronic Systems, 6 (1), pp. 1-25, (2001). DOI: 10.1145/371254.371256

4 **Chandrachoodan N., Bhattacharyaa S. S., Liu K. J. R.,** *An efficient timing model for hardware implementation of multirate data flow graphs*. In Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP '01). 2001 IEEE International Conference on, Vol. 2., pp. 1153-1156, (2001). DOI: 10.1109/ICASSP.2001.941126

5 **Fingeroff M.,** *High-Level Synthesis Blue Book*. Xlibris Corporation, (2010).

6 **Ghuloum A. M., Fisher A. L.,** *Flattening and parallelizing irregular, recurrent loop nests*. In Wexelblat R. L. (ed.), Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, (PPOPP '95), pp. 58-67, ACM, New York, NY, USA, (1995). DOI: 10.1145/209936.209944

7 **Gill G., Hansen J., Singh M.,** *Loop Pipelining for High-Throughput Stream Computation Using Self-Timed Rings*. In Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on, pp. 289- 296, 5-9 Nov. 2006. DOI: 10.1109/ICCAD.2006.320135

8 **Hennessy J. L., Patterson D. A.,** *Computer Architecture.* Fourth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

9 **Huang J. C., Leng T.,** *Generalized loop-unrolling: a method for program speedup*. In Application-Specific Systems and Software Engineering and Technology, 1999. ASSET '99. Proceedings. 1999 IEEE Symposium on, pp. 244-248, (1999). DOI: 10.1109/ASSET.1999.756775

10 **Ito K., Parhi K. K.,** *Determining the iteration bounds of single-rate and multi-rate dataflow graphs*. In Circuits and Systems, 1994. APCCAS '94., 1994 IEEE Asia-Pacific Conference on, pp. 163-168, (1994). DOI: 10.1109/APCCAS.1994.514543

11 **Jones R. B., Allan V. H.,** Software pipelining: a comparison and improvement. In Microprogramming and Microarchitecture. Micro 23. Proceedings of the 23rd Annual Workshop and Symposium., Workshop on, pp. 46-56, nov 1990. DOI: 10.1109/MICRO.1990.151426

12 **Lam M.,** *Software pipelining: an effective scheduling technique for vliw machines*. In Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, PLDI '88, pp. 318-328, ACM, New York, NY, USA, (1988). DOI: 10.1145/960116.54022

13 **Lee E. A., Messerschmitt D. G.,** *Synchronous data flow*. Proceedings of the IEEE, 75 (9), pp. 1235-1245, (1987). DOI: 10.1109/PROC.1987.13876

14 **Parhi K. K.,** *Algorithm transformation techniques for concurrent processors*. Proceedings of the IEEE, 77 (12), pp. 1879-1895, (1989). DOI: 10.1109/5.48830

15 **Roberts L. G.,** *Machine perception of three-dimensional solids*. PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering, (1963).