Code Transformations Based on Speculative SDC Scheduling

Marco Lattuada and Fabrizio Ferrandi
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano - Italy
Email: marco.lattuada@polimi.it fabrizio.ferrandi@polimi.it

***Abstract*—Code motion and speculations are usually exploited in the High Level Synthesis of control dominated applications to improve the performances of the synthesized designs. Selecting the transformations to be applied is not a trivial task: their effects can indeed indirectly spread across the whole design, potentially worsening the quality of the results.**

In this paper we propose a code transformation flow, based on a new extension of the System of Difference Constraints (SDC) scheduling algorithm, which introduces a large number of transformations, whose profitability is guaranteed by SDC formulation. Experimental results show that the proposed technique in average reduces the execution time of control dominated applications by 37% with respect to a commercial tool without increasing the area usage.

I. Introduction

Programmable devices such as FPGAs can potentially offer very significant computational power, but implementing efficient solutions on them can be a hard task. One of the main obstacles is the usage of hardware description language, whose knowledge is usually a rare expertise. To overcome or at least to mitigate this issue, *High Level Synthesis (HLS)* [1] has been introduced. Because of the characteristics of FPGAs, HLS can quite easily produce efficient implementations of data dominated applications which are the most suitable to be implemented in hardware. On the contrary, the implementations of control dominated applications can be very inefficient, especially when compared to software implementations because of the higher frequency of general purpose processors. Nevertheless, there can still be the needing to execute these specifications in hardware. For example, they can be part of larger applications or system full hardware implementations have to be preferred to heterogeneous solutions because of the reduced data transfers. Since the data dominated parts of the applications can be implemented in a very efficient way, the control dominated portions can become the bottleneck, so they have to be optimized as well.

One of the most critical problems of optimizing a control dominated specification is the scheduling, i.e., deciding in which control step each operation is executed. Most of the scheduling techniques (e.g., [2], [3]) compute the scheduling starting from the *Control-DataFlow Graph (CDFG)* [4], a graph based description which represents the control and the data dependences of the analyzed specification. The structure of the *CDFG* can heavily determine the scheduling results, even if exact methods (e.g., [2]) are used. Indeed, scheduling algorithms can produce very different results (better or worse) if they start from a modified version of the *CDFG*, obtained by applying code motion or speculation. Applying this type of transformation is not a trivial task since evaluating the overall effects of a single or of a set of transformations is not easy. State of the art methodologies (e.g., [3], [5]) typically exploit heuristics and consider only a limited set of possible transformations, potentially limiting the benefits of this type of approach.

In this paper a methodology flow based on an exact method for explicitly transforming a *CDFG* to improve the scheduling solution is proposed. The main contributions of this work are:

- it extends the SDC scheduling algorithm formulation [2] to allow implicit global code motion;
- it exploits the results of the modified SDC scheduling algorithm to perform explicit code motion.

The rest of this paper is organized as follows: Section II describes the problem addressed by the proposed methodology which is described in Section III. Section IV presents the experimental evaluation of the proposed flow, then Section V compares the proposed methodology with state of the art techniques and finally Section VI draws the conclusion of this work and presents the possible future extensions.

II. Problem Definition

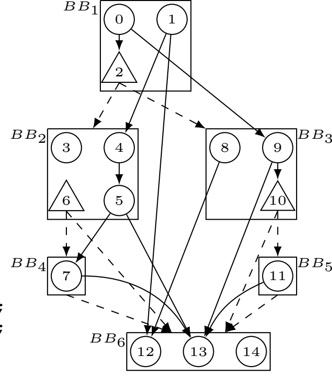
The proposed methodology aims at modifying the *CDFG* to improve the scheduling solution in terms of overall latency of the application. *CDFG* is a directed graph $G_{CDFG} = (V_{op} \cup V_{bb}, E_{op} \cup E_{bb})$ where each vertex $v_{op} \in V_{op}$ corresponds to an operation, each vertex $v_{bb} \in V_{bb}$ corresponds to a basic block, each edge $e_{op} \in E_{op}$ represents a data dependences between two operations and each edge $e_{bb} \in E_{bb}$ represents the control dependences between two basic blocks. Figure 1b shows an example of *CDFG*, extracted from the code of Figure 1a. In the following it is assumed that there are only two available multipliers, the division v_3 takes 3 cycles, all the other operations take 1 cycle. For the sake of simplicity operations chaining is not considered in the presented example, even if it is fully supported in the proposed methodology.

CDFG is modified by means of explicit code motions: operations are moved from their current basic block to other basic block preserving the semantic of the application. The code motion can be applied directly on the *CDFG* or on the code from which it has been generated. Not all the possible code motions are considered: operations cannot be moved from a loop to another one, nor can be moved from after a loop to before a loop. The proposed methodology assumes that the

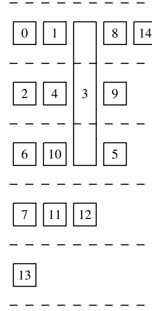
```
 0: a_0 = in1 * in2;
 1: b_1 = in3 * in4;
 2: if(a_0)
 3:   c_3 = in5 / 5;
 4:   d_4 = b_1 * 3;
 5:   f_5 = d_4 - in1;
 6:   if(in1)
 7:     f_7 = f_5 + in1;
      else
 8:   b_8 = in1 + in2;
 9:   f_9 = a_0 + 2;
10:   if(f_9)
11:     f_11 = foo();

12: b_12 = Φ(b_1,b_1,b_8,b_8);
13: f_13 = Φ(f_7,f_5,f_9,f_9);
14: h_14 = in1 + 3;
```

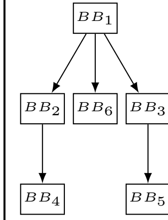
(a) Starting Code



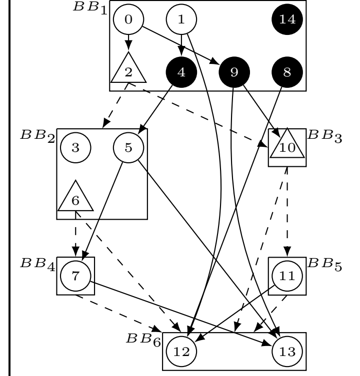
(b) Starting CDFG



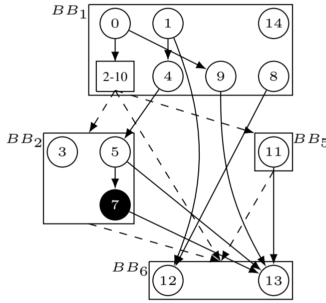
(c) Scheduling



(d) Dominators



(e) CDFG After Code Motion



(f) Final CDFG

```
 0: a_0 = in1 * in2;
 1: b_1 = in3 * in4;
 4: d_4 = b_1 * 3;
 8: b_8 = in1 + in2;
 9: f_9 = a_0 + 2;
14: h_14 = in1 + 3;
 2: if(a_0)
 3:   c_3 = in5 / 5;
 5:   e_5 = d_4 - in1;
 7:   f_7 = e_5 + in1;
10:  else if(f_9)
11:     f_11 = foo();

12: b_12 = Φ(b_1,b_8, b_8;
13: f_13 = Φ(in1 ? f_7 : f_5,
      f_9,f_9);
```

(g) Final Code

Fig. 1: An example of application of the proposed methodology.

analyzed specification is already in *Static Single Assignment* form [6], condition which is already satisfied in most of the compilers and high level synthesis tools. In this form, each variable must be assigned exactly once, so a new version of a variable is created for each its assignment (e.g., `f_5`, `f_7`, `f_9`, `f_11`). Variables used in right side of statement are renamed so that the most recent definition of a variable is used. If multiple definitions of a same variable reach a basic block (e.g., `f_5`, `f_7`, `f_9`, `f_11` reach BB_6), they are merged by means of a new artificial definition Φ, which creates a new version of the variable (e.g., `f_13 = Φ(f_7,f_5,f_9,f_9)`).

Code motions can be classified in:

- *Non-speculative*: the operation is moved between two basic blocks that are executed in exactly the same traces. This transformation is always safe, but it still has to be applied in a conservative way since it can increase the overall latency and the area of the solution. An example of *Non-speculative* code motion in example of Figure 1 is v_{14} to BB_1
- *Speculative*: an operation v_i of basic block BB_a is moved in a BB_b that is executed in a superset of the traces containing BB_a, i.e., it is anticipated with respect to the conditional construct which controls its execution. These transformations can be applied only if the operation does not modify memory status. An example of *Speculative* code motion in example of Figure 1 is v_4 in BB_1. Use of speculated operations instead of operations controlled by guards like in [2] can improve performance results (they have not to wait for the guard), but they can increase the area usage.

In the next section how to solve the problem of identifying only profitable transformations will be presented.

III. Proposed Methodology

The proposed methodology flow is composed of three steps:

1) *Speculative SDC scheduling*: operations are scheduled with SDC scheduling algorithm extended to allow code motion.
2) *Code Motion*: code motions suggested by SDC scheduling results are applied to the analyzed *CDFG*.
3) *Transformations*: *CDFG* is transformed by means of heuristics to further improve overall performances.

In the following each step of the flow will be detailed.

A. Speculative SDC Scheduling

This section describes Speculative SDC scheduling, an extension of SDC scheduling algorithm [2] aimed at supporting code motion. Since code motion has to be allowed only intra loops, Speculative SDC scheduling is independently applied on each loop (the whole function is the most external loop). Its formulation uses the following variables and functions:

- OP_k: the set of operations contained in loop l_k and not contained in any loop nested in l_k.
- SE: the set of operations which cannot be speculated because of side effects (e.g., store operations) or because they are conditional constructs (e.g., `if`).
- $Cond(v_i)$: the conditional construct which controls the execution of an operation [7].
- Lv_i: the cycle latency of v_i.
- T: the clock period.
- $D(v_i, v_j)$: estimation of the delay of the longest critical combinational path between v_i and v_j. The estimations are obtained by applying the same approach adopted in

[2] (i.e., adding the estimated delay of all the operations and of all the interconnections of the path).

$\forall l_k, \forall v_i \in OP_k, \forall t \in [0, Lv_i] : sv_t(v_i) \in \mathbb{N} \cup \{0\}$ is the variable specifying the control step in which the stage of an operation is scheduled. $sv_{beg}(v)$ and $sv_{end}(v)$ are defined as follows: $sv_{beg}(v) \equiv sv_0(v)$, $sv_{end}(v) \equiv sv_{Lv}(v)$. Moreover, for each basic block BB_b an artificial node $ssnk(BB_b)$ is added representing its ending. The constraints are:

1) $\forall v_i \in OP_k | Lv_i \geq 1, \forall t \in [1, Lv_i] : sv_t(v_i) = sv_{t-1}(v_i)+1$: consecutive stages of a multicycle operation must be executed in consecutive control steps (e.g., the three stages of v_3 have to be executed in consecutive control steps).
2) $\forall v_i \in OP_k | v_i \in SE \wedge Cond(v_i) \neq \varnothing : sv_{end}(Cond(v_i)) - sv_{begin}(v) \leq -1$: operations which cannot be speculated have to be scheduled after the conditional construct which controls them (e.g., v_{11} must be scheduled after v_{10}).
3) $\forall v_i \in OP_k | v_i \ is \ \Phi, \forall v_j \in OP_k | v_j \ determines \ output \ of \ v_i : sv_{end}(v_j) - sv_{beg}(v_i) \leq 0$: a Φ operation cannot be executed before the conditional construct which controls which variable version has to be selected (e.g., v_{12} and v_{13} cannot be scheduled before v_6 and v_{10}).
4) $\forall v_i \in OP_k : sv_{end}(v_i) - sv_{end}(BB_b) \leq 0$ (where BB_b is the basic block to which v_i belongs): a basic block does not end before the ending of any its operation (e.g., BB_3 does not end before v_8, v_9 and v_{10} end).
5) $\forall e(v_i, v_j) \in E_d : sv_{end}(v_i) - sv_{beg}(v_j) \leq 0$: data dependencies between operations must be guaranteed (e.g., v_4 cannot start before the end of v_5).
6) $\forall v_i, v_j | exists \ a \ data \ dependence \ path \ (v_i \rightarrow v_j) in \ CDFG : sv_{beg}(v_i) - sv_{beg}(v_j) \leq -(\lceil D(v_i, v_j)/T \rceil - 1)$: a pair of operations whose longest critical combinational path is larger than clock period cannot be chained (e.g., v_2 cannot start in the same control step of v_0).
7) *resource sharing*: for each execution trace inside the *CDFG* and for each limited resource r, a linear order between operations V_r mapped on resources of type r is built. If N_r is the number of resources r, the ith and the i^{th+N_r} operations of the linear order cannot be executed in the same control step: $sv_{beg}(v_i) - sv_{beg}(v_{i+N_r}) \leq -1$ is added to the formulation (e.g., given the linear order of operations mapped on multiplier v_0-v_1-v_4, since there are only two available multipliers, v_4 cannot start in the same cycle of v_0).

The last type of constraints do not avoid the simultaneous scheduling of multiple speculated operations which can require more resources than the available. Instead of adding a set of linear orders between these operations, which introduce a set of unnecessary constraints when these operations are actually not speculated, it is admitted that the solutions produced by SDC are actually not implementable. The following steps of the methodology will transform these solutions so that they will respect resource constraints. The first significant difference with respect to [2] is the absence of $ssrc$ nodes, i.e., nodes that represent beginning of basic blocks, and of the control dependence constraints, since they prevent speculation of operations. Since speculation of SE operations has still to be avoided, the constraints of type 2 are added. The second main difference is the introduction of constraints of type 3 to model the implicit dependences between a Φ operation and the conditional constructs which determine its outcome. Finally, the last significant difference is the objective function used to minimize the overall latency of the solution. In [2] the overall latency is approximated by an expression based on the beginning and the ending of each basic block. Since the beginning of basic blocks cannot be specified in Speculative SDC formulation, a different objective function is adopted:

$$min \sum_{BB_i \in l_k} sv_{end}(ssnk(BB_i)) \quad (1)$$

Algorithm 1: Pseudo-code of Code Motion.

```
1: for all BB_b ∈ CDFG do
2:     for all v_i ∈ BB_b do
3:         if v_i is mapped on a limited resource r then
4:             UsedResources[BB_b][sv_beg(v_i)]++
5:         end if
6:     end for
7: end for
8: for all BB_b ∈ TopologicalOrder(CDFG) do
9:     for all v_i ∈ TopologicalOrder(BB_b) do
10:        end = sv_end(v_i)
11:        curr = dest = BB_b
12:        while sv_end(ssnk(DOM(curr)) ≥ end do
13:            curr = DOM(curr)
14:            if v_i is mapped on a limited resource r then
15:                if UsedResources[curr][sv_beg(v_i)] ≥ N_r then
16:                    continue
17:                end if
18:            end if
19:            dest = curr
20:        end while
21:        if dest ≠ BB_b then
22:            Move(v_i, dest)
23:            if v_i is mapped on a limited resource r then
24:                UsedResources[destination][sv_beg(v_i)]++
25:                UsedResources[BB_b][sv_beg(v_i)]--
26:            end if
27:        end if
28:    end for
29: end for
```

By minimizing this objective function, all the longest paths from the header of the loop (its first basic block) to all its basic blocks are minimized at the same time. On the contrary other paths can be only partially optimized (e.g., BB_1-BB_3-BB_6), so also the proposed objective function is an approximation, but differently from the one proposed in [2] does not require path profiling information.

The Speculative SDC scheduling solution of *CDFG* of Figure 1b is shown in Figure 1c.

B. Code Motion

In this step, the results of the Speculative SDC scheduling algorithm are used to identify which operations can be moved from a basic block to another improving the overall performance of the scheduling solution. The basic block to which each operation has to be moved is identified by analyzing the dominator tree [8] built starting from the *CDFG*. A basic block BB_d dominates basic block BB_i if every path from the entry of *CDFG* to BB_i go through BB_d. A basic block BB_d immediately dominates BB_i if dominates BB_i and there is not any BB_h such that BB_d dominates BB_h and BB_h dominates BB_i. The immediate dominator relationship is described by dominator tree. Dominator tree of example of Figure 1b is shown in Figure 1d. Algorithm 1 describes in details how code motion is performed. All the operations of the *CDFG* are analyzed in topological order (loops of lines 8 and 9). An operation can be moved in a dominator if it does not end after the end of the dominator (line 12). The

algorithm goes up through the immediate dominator chain as long as it encounters basic blocks where it can move the operation (line 13). Figure 1e shows the *CDFG* after application of code motion: black nodes are moved operations. For example v_{14} is moved in BB_1 (*Non-speculative* code motion) since $sv_{end}(v_{14}) \leq sv_{end}(ssnk(BB_1))$ $(1 \leq 2)$, v_4 is moved in BB_1 (*Speculative* code motion) since $sv_{end}(v_4) \leq sv_{end}(ssnk(BB_1))$ $(2 \leq 3)$, v_3 cannot be moved in BB_1 since $sv_{end}(v_3) \nleq sv_{end}(ssnk(BB_1)$ $(3 \nleq 2)$. By considering only dominators as possible candidate destination of code motion, it is guaranteed that the execution of the moved operation is not removed by any execution trace, but it can eventually be added to further execution traces (*Speculative* code motion). The check of the ending time of the destination basic block guarantees that input data of the operation will be ready in that basic block and that the ending of this will be not postponed by the moved operation.

In case of operations mapped on limited resources (line 14), it has to be checked that the ending of the destination basic block is not postponed because of resource contention in a particular control step (line 15). To perform this check, it is necessary to have information about the utilization of limited resources of operations of each basic block in each control step (lines 1-7). When the actual destination has been finally identified, code motion can be actually performed. Since the methodology assumes that the intermediate representation is in SSA form, no further change (e.g., fixing of variables or of Φ instructions) is required.

It is worth noting that SDC scheduling algorithm applied on the transformed *CDFG* can produce a solution different from Speculative SDC solution because not all the code motions suggested by Speculative SDC can actually be applied. There are two main reasons for which the code motion of an operation is not performed: it would violate a resource constraint or it would require to divide a multi-cycle operation among different basic blocks. For example v_3 should be distributed between BB_1, and BB_2.

C. Code Transformations

Since the *CDFG* produced in the previous step does not correspond exactly to the Speculative SDC solution, there can still be the possibility of optimizing it by applying transformations not included in the optimal solution computed by Speculative SDC scheduling. Some of these transformations can be applied directly on the modified *CDFG* by means of heuristics, others can be applied only after that the structure of the *CDFG* has been cleaned.

Algorithm 2 shows the order in which the different transformations are applied on each basic block:

Algorithm 2: Pseudo-code of Code Transformations.

1: **for all** $BB_b \in TopologicalOrder(CDFG)$ **do**
2:     **for all** $v_i \in BB_b$ **do**
3:         $dest = BB_b$
4:         **while** $CanBeMoved(v_i, DOM(dest))$ **do**
5:             $dest = DOM(dest)$
6:         **end while**
7:         **if** $BB_b \neq dest$ **then**
8:             $LocalCodeMotion(v_i, dest)$
9:         **else**
10:             $UpdateTiming(v_i)$
11:         **end if**
12:     **end for**
13:     **if** $IsEmpty(BB_b)$ **then**
14:         $BB_p = Predecessor(BB_b)$
15:         $BB_s = Successor(BB_b)$
16:         $BasicBlockRemoval(BB_b)$
17:         **if** $NumberSuccessors(BB_p) == 1$ **then**
18:             $RemoveConditionalConstruct(pred)$
19:         **end if**
20:     **end if**
21:     **if** $SingleCond(BB_b) and NumberPredecessors(BB_b) == 1$ **then**
22:         $MergeConditionalConstructs(Predecessor(BB_b), BB_b)$
23:     **end if**
24: **end for**

1) *Local Code Motion* (lines 2-12): each operation is analyzed checking if it can be moved in one of the dominator of the current basic block. Profitability of a code motion is not evaluated anymore by considering Speculative SDC solution, but analyzing the relative scheduling of the operations in the destination basic block: an operation v_i can be moved in dominator BB_{dom} if its input operands are all available during execution of BB_{dom} and if it does not increase the overall latency of BB_{dom}. Indeed, this type of increment has to be avoided since it would increase the overall delay of the solution. In the *CDFG* of Figure 1e v_7 can be moved in BB_2 since it does not increase its latency (3). Note that this code motion is not suggested by Speculative SDC because of the ending time of BB_2: according to Speculative SDC solution v_7 should be scheduled in 4 while the ending time of BB_2 is 3, but this can be obtained only by splitting execution of v_3 on BB_1 and BB_2.
2) *Basic Block Removal* (line 16): empty basic blocks are removed. In example of Figure 1e, BB_4 can be removed since it becomes empty after the moving of v_7.
3) *Remove Conditional Construct* (line 18): after removing an empty basic block, it is possible that all the targets of a conditional construct become the same: in this case, the corresponding Φ instructions are fixed and then the conditional construct is removed. After the removal of BB_4, v_6 targets only BB_6, so v_6 is removed and v_{13} is transformed in `f_13 = Φ(in1 ? f_7 : f_5, f_9, f_9)`.
4) *Merge Conditional Constructs* (line 22): if a basic block BB_b is composed only of a conditional construct and it has only one predecessor BB_p, Φ instructions of successors of BB_b are fixed, the conditional construct of BB_b is merged with the conditional construct at the end of BB_p and BB_b is removed. In example of Figure 1e, BB_3 is composed only of v_{10}, so v_2 and v_{10} can be merged and BB_3 can be removed.

Figure 1f shows the *CDFG* after that all the presented transformations have been applied while the corresponding code is presented in Figure 1g. In *CDFG* of Figure 1b four execution paths can be identified whose delays in terms of clock cycles are: 7,6,5,6. After that the proposed methodology is applied, in *CDFG* of Figure 1f their delays become: 6,6,3,4.

D. Complexity of the proposed methodology

The complexity of solving a System of Difference Constraints is $\Theta(n \cdot m)$ where n is the number of variables and m is the number of constraints. Since the Speculative SDC and the original formulation of SDC share all the variables and most of the constraints (types 1,4,6,7), their complexity is comparable. In Speculative SDC formulation there are also constraints of type 2,3, but there are not constraints for *ssrc* and the number of added constraints of type 5 is smaller (since inter loops dependences have not to be considered). In

the worst case, both for original formulation of SDC and for Speculative SDC, the number of variables is $O(|OP_k|)$ and the number of constraints is $O(|OP_k|^2)$, so the complexity of the first step of the proposed methodology is $O(|OP_k|^3)$. The complexity of the second step is $O(|OP_k|)$ (loops of lines 8 and 9 of Algorithm 1) while the complexity of the *Transformations* step corresponds to the complexity of Algorithm 2 which is $O(|BB_b|)$. Since the two last steps have smaller complexity than the Speculative SDC, the overall complexity of the methodology is $O(|OP_k|^3)$.

IV. Experimental Results

To verify the effectiveness of the proposed methodology, the results of four High Level Synthesis flows have been compared:

① *Commercial Tool*, which implements SDC scheduling algorithm and which supports Xilinx FPGAs; this tool supports code motion but only for code of pipelined loops which are not contained in the considered benchmark suite.

② *LegUp 3.0* [9], an open source publicly available High Level Synthesis Tool developed at University of Toronto which already implements SDC scheduling algorithm and which supports Altera FPGAs.

③ *PandA 0.9.3* [10], an open source, publicly available framework for High Level Synthesis developed at Politecnico di Milano, extended with the SDC scheduling algorithm presented in [2]. In the following it will be shown how the results obtained with this implementation are in average quite similar to the ones produced by ①, so that this implementation can be considered a good golden reference.

④ *PandA 0.9.3* [10] extended with the proposed methodology. Intermediate representation adopted in *PandA* is already in SSA form, so implementation of the proposed methodology only requires to implement Speculative SDC scheduling and the proposed code transformations. This tool has been chosen as starting point since it supports both Altera and Xilinx FPGAs allowing to verify the effective generality of the proposed methodology.

For the sake of fairness, the comparison with [5] has not been performed since it does not support operations chaining.

Two platforms have been considered in the experimental evaluation:

- the Xilinx Zynq-7000 xc7z0 (not supported by *LegUp*) with a target frequency of 66.66 MHz (same default experimental setup of ①). Final synthesis is performed by means of Xilinx Vivado [11].
- the Altera Cyclone II EP2C70F896C6 (not supported by commercial tool) with the target frequency of 66.66 MHz (same default experimental setup of ②). Final synthesis is performed by means of Altera Quartus II [12].

The benchmarks adopted to perform the experimental evaluation are the CHStone suite [13]. CHStone suite is a set of 12 benchmarks, explicitly collected for the evaluation of High Level Synthesis flows, which aim at representing all the possible scenarios which have to be addressed by an High Level Synthesis tool. In particular, this suite contains both data dominated applications (*aes*, *blowfish*, *jpeg*, *mpeg2*, *sha*) and control oriented applications (*adpcm*, *dfadd*, *dfdiv*, *dfmul*, *dfsin*, *gsm*, *mips*). For the sake of brevity, their detailed characteristics (e.g., the number of loops, the number of conditional constructs, the number of arithmetic operations, etc.) have not been reported, but they can been found in [13].

PandA can generate two different hardware versions of divisor according to which division algorithm is selected. The proposed methodology has been applied on both the implementations, so that two different sets of results are reported for ③ and ④ on the benchmarks which contain division operations (i.e., *dfdiv* and *dfsin*).

Left part of Table I reports the synthesis time results after place and route obtained by ①, ③, and ④ for Zynq-7000. The results of ① on *blowfish* and *mpeg2* have not been reported (N/A) since the cosimulations fail. SDC scheduling implementations of ① and ③ can be considered quite equivalent, even if there are significant differences in the results on some benchmarks, since these are not caused by SDC scheduling implementation. In particular ① obtains better results on *mips* thanks to the implementation of *Rotation Scheduling*. On the contrary, ③ obtains better results on *dfdiv* and *dfsin* thanks to the implementation of a better division algorithm. Speculative SDC scheduling does not introduce significant benefits on data dominated applications (i.e., *aes, blowfish, jpeg, mpeg2, sha*) as expected. On the contrary, on most of the control dominated applications (i.e., *adpcm, dfadd, dfdiv, dfmul, dfsin, gsm, mips*) the gain is very significant (up to 39% on dfadd when compared to ③ and up to 46% on dfmul when compared to ①). The average gain of ④ with respect to ① on control dominated application is instead 37%.

Table II reports the synthesis area results after place and route obtained by ①, ③, and ④ targeting Zynq-7000. The produced solutions have similar resource usage in terms of Slices, LUTs, and BRAMs while the number of registers used in ③ and ④ because register allocation has not been fully optimized as in ①. The only significant difference in terms of number of BRAMs is in the synthesis of *adpcm* benchmark and is caused by too conservative alias analysis which prevents the application of some memory optimizations. The number of used DSPs is almost the same for all the benchmarks but *dfdiv(2)* and *dfsin(2)* because of the different implemented division algorithm. The solutions produced by ③ and ④ have instead very similar results for all the resource usage metrics: the obtained benefits in terms of clock cycles by exploiting Speculative SDC scheduling have not to be paid in terms of increment of area.

Right part of Table I reports the synthesis time results after place and route obtained by ② (results have been extracted from [9]), ③, and ④ for Cyclone II. Results in terms of cycles obtained by ③ cannot be directly compared with other tool since ② uses a different memory model and implements different optimizations with respect to ③. The differences in terms of clock cycles in the solutions produced by ③ and ④ are similar to the ones already analyzed for Zynq-7000. ③ and ④ produce solutions with similar clock cycles latencies for data dominated applications (*aes, blowfish, jpeg, mpeg2, sha*), while there are significant advantages in exploiting Speculative SDC scheduling on control dominated applications (*adpcm, dfadd, dfdiv, dfmul, dfsin, gsm, mips*). With respect to results

TABLE I: Synthesis time results after place and route. ① are the results obtained with *Commercial Tool*, ② are the results obtained with *LegUp*, ③ and ④ are the results obtained with SDC scheduling and Speculative SDC scheduling implemented in *PandA*.

	Zynq-7000							Cyclone II						
	Clock Frequency			Cycles				Clock Frequency			Cycles			
	①	③	④	①	③	④	③-④	②	③	④	②	③	④	③-④
adpcm	34.3	67.3	67.9	23,075	17,627	15,297	13%	45.7	71.1	66.6	36,795	19,727	17.347	12%
dfadd	72.0	83.9	67.2	383	355	219	39%	124.0	77.6	70.7	2,330	380	289	24%
dfdiv(1)	81.7	69.7	67.6	1,917	1,039	967	7%	74.7	73.5	73.6	2,144	1,995	1,821	9%
dfdiv(2)		68.1	64.8		762	547	16%		70.6	69.1		910	810	11%
dfmul	60.7	74.7	71.8	196	149	105	28%	85.6	80.6	76.1	347	173	141	12%
dfsin(1)	52.5	68.1	66.0	48,226	30,161	26,466	12%	62.6	71.9	70.5	67,466	56,789	47,643	16%
dfsin(2)		68.0	65.5		21,182	17,121	19%		71.0	68.5		34,658	26,079	25%
gsm	66.4	67.0	66.2	3,397	2,775	2,361	15%	58.9	69.6	69.0	6,656	2,935	2,521	14%
mips	77.6	73.0	74.7	2,489	3,413	3,040	11%	90.0	68.4	67.7	6,443	3,413	3,040	11%
aes	89.2	80.2	74.1	2,973	3,188	3,112	3%	60.7	93.1	86.4	14,022	3,332	3,256	3%
blowfish	N/A	83.8	83.5	N/A	89,679	89,549	1%	65.4	73.0	77.6	209,866	108,415	108,285	0%
jpeg	69,0	67.0	66.9	468,011	471,173	465,261	1%	47.0	72.9	72.5	5,861,516	490,996	485,084	1%
mpeg2	N/A	93.8	70.4	N/A	4,221	4,214	0%	91.7	88.0	78.9	8,578	4,228	4,225	0%
sha	97.9	83.1	87.2	111,043	113,329	113,324	0%	86.9	77.2	75.8	247,738	123,611	123,606	0%

TABLE II: Synthesis area results after place and route for Zynq-7000. ① are the results obtained with *Commercial Tool*, ③ and ④ are the results obtained with SDC scheduling and Speculative SDC scheduling implemented in *PandA*.

	Slice			LUTs			Registers			DSPs			BRAMs		
	①	③	④	①	③	④	①	③	④	①	③	④	①	③	④
adpcm	2,773	1,820	1,810	10,358	5,399	5,268	3,488	3,920	3,533	46	44	42	13	29	29
dfadd	913	962	805	2,790	2,411	2,254	1,544	1,730	1,018	0	0	0	0	0	0
dfdiv(1)	1,288	2,372	2,334	4,074	6,858	6,779	2,726	4,815	4,526	24	18	18	0	0	0
dfdiv(2)		1,756	1,661		4,957	4,914		3,352	2,918		66	66		1	1
dfmul	729	541	466	2,433	1,540	1,341	1,143	882	632	16	10	10	0	0	0
dfsin(1)	3,428	4,680	4,785	11,491	14,782	15,545	6,041	10,801	8,549	43	41	41	4	4	4
dfsin(2)		4,249	4,229		13,044	13,813		9,352	6,863		89	89		5	5
gsm	1,307	1,441	1,416	4,569	4,004	4,099	1,391	3,221	3,226	40	31	31	9	5	5
mips	429	535	497	1,359	1,795	1,728	511	644	669	8	8	8	4	4	4
aes	1,188	1,355	1,440	3,625	4,051	4,209	1,888	2,418	2,367	6	0	0	13	10	10
blowfish	N/A	876	838	N/A	2,101	2,217	N/A	2,155	2,144	N/A	0	0	N/A	22	22
jpeg	5,870	4,521	4,593	18,347	14,291	13,904	9,683	10,072	9,756	14	15	9	55	59	59
mpeg2	N/A	645	621	N/A	1,986	1,887	N/A	1,432	1,194	N/A	0	0	N/A	2	2
sha	647	735	720	2,180	2,140	2,086	1,356	2,216	2,214	0	0	0	20	12	12

TABLE III: Synthesis area results after place and route for Cyclone II. ② are the results obtained with *LegUp*, ③ and ④ are the results obtained with SDC scheduling and Speculative SDC scheduling implemented in *PandA*.

	LEs			Multibits			Mults		
	②	③	④	②	③	④	②	③	④
adpcm	22,605	9,819	10,288	29,120	13,202	13,302	300	120	88
dfadd	8,881	6,432	5,544	17,120	10,112	10,112	0	0	0
dfdiv(1)	20,159	11,298	10,356	12,416	5,054	5,054	62	48	48
dfdiv(2)		8,096	7,858		6,352	6,352		102	102
dfmul	4,861	3,133	3,011	12,032	5,120	5,120	32	31	31
dfsin(1)	39,933	21,484	20,477	12,864	6,602	6,602	100	70	85
dfsin(2)		18,415	17,375		7,900	7,900		124	139
gsm	19,131	6,164	6,216	11,168	8,992	8,992	70	50	48
mips	4,479	2,559	2,559	4,480	4,480	4,480	8	16	16
aes	28,490	3,066	3,117	38,336	32,526	32,526	0	0	0
blowfish	15,064	3,802	3,933	150,816	150,528	150,528	0	0	0
jpeg	46,224	16,121	15,593	253,936	438,674	437,394	172	134	138
mpeg2	13,238	2,351	2,328	34,752	32,768	32,768	0	0	0
sha	12,483	3,248	3,182	134,368	117,760	117,760	0	0	0

obtained on Zynq-7000, the gain provided by Speculative SDC scheduling is less significant. The smaller speed of Cyclone II limits the depth of the chains of operations which can be scheduleded in a single clock cycle and so limits the number of profitful code motion. Moreover, results about maximum frequency obtained by ④ show that there is always a significant positive slack. This is caused by a conservative timing model adopted by PandA during High Level Synthesis for Cyclone II which prevents some feasible advantageous code motions and does not allow to obtain the best possible results. Finally, as in case of Zynq-7000, the area results obtained when targeting Cyclone II, which are reported in Table III, show that the gain in terms of cycles of SDC scheduling solutions does not imply an increase of resource usage.

V. Related Work

The algorithms which have been proposed to solve the problem of scheduling in High Level Synthesis can be roughly classified in two categories. The former are oriented at optimizing data dominated applications and in particular the execution of loops, the latter, as the one presented in this paper, are aimed at optimizing control dominated specifications. Most of the algorithms of the second type work on *CDFG*, but without performing explicit code motion. For example, SDC scheduling algorithm, which has been initially proposed by Cong et al. [2], produces the optimal scheduling of the starting *CDFG*, but it only allows limited implicit code motion (execution of consecutive basic blocks can be only partially overlapped). Explicit code motion is instead considered in [5] where a set of possible code transformations is proposed. Differently from the methodology presented in this paper, the decisions about which are the transformations to be performed are taken on the basis of the results of an heuristic and not of an exact method. The second main difference is that their transformations are aimed at improving only the longest path of the specification and not all the paths at the same time.

The combination of SDC scheduling and code motion has been proposed in [3]. The authors proposed a scheduling framework which performs scheduling of *CDFG* in two steps. In the former SDC scheduling is used to schedule critical operations after refinement of the *CDFG*. In the latter, after that the scheduling of critical operations has been guaranteed by the insertion of delay operations, a fast heuristic is exploited to schedule the remaining operations allowing their implicit code motion by relaxing conditional constraints. Usage of an heuristic instead of SDC scheduling potentially prevents production of the optimal solution. A quantitative comparison with the approach proposed in this paper has not been possible: the details about experimental setup are not reported in [3] nor it has been possible to compile the Shang framework to generate the data. However, since the authors of Shang framework state that their framework is 30% faster than LegUp on CHStone benchmarks and since the gain of the proposed methodology with respect to LegUp is much larger (Table I), the results of [3] are expected to be worse than the results of Speculative SDC scheduling. Moreover, the selection of the critical operations is demanded to the designer. Differently from this approach, the methodology proposed in this paper adopts SDC scheduling to compute the schedule of all the operations, potentially allowing code motion of all of them. Moreover, it does not require to introduce delay operations to guarantee the constraints. Finally, SDC algorithm has been extended also to optimize execution of loops. Modulo scheduling and SDC scheduling have been integrated by extending SDC formulation with pipeline dependent constraints and by combining it with greedy heuristic [14] and with backtracking algorithm [15]. In this way it is possible to optimize in an exact way the initiation interval of the loop pipelines and so the overall performance of the applications.

VI. Conclusions and Future Works

This paper presents a methodology flow aimed at transforming the *CDFG* representation of an application to reduce its overall clock cycle latency. The methodology combines Speculative SDC scheduling, global code motion and local heuristics to apply all the profitable transformations. Experimental results show how it is effectively able to improve the performances of hardware implementations of control dominated applications without increasing resource usage. Future works will focus on including profiling information in the proposed Speculative SDC formulation and in allowing code motion inter loops.

References

[1] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *TCAD*, 30(4):473–491, April 2011.

[2] Jason Cong and Zhiru Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. DAC ’06, pages 433–438, New York, NY, USA, 2006. ACM.

[3] Hongbin Zheng, Qingrui Liu, Junyi Li, Dihu Chen, and Zixin Wang. A gradual scheduling framework for problem size reduction and cross basic block parallelism exploitation in high-level synthesis. In *ASP-DAC 2013, Yokohama, Japan, January 22-25, 2013*, pages 780–786, 2013.

[4] Daniel D. Gajski and Loganath Ramachandran. Introduction to high-level synthesis. *IEEE Des. Test*, 11(4):44–54, 1994.

[5] Sumit Gupta, Nick Savoiu, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Conditional speculation and its effects on performance and area for high-level snthesis. ISSS ’01, pages 171–176, New York, NY, USA, 2001. ACM.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, Oct 1991.

[7] R. Cytron, J. Ferrante, and V. Sarkar. Compact representations for control dependence. In *PLDI*, pages 337–351, 1990.

[8] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *(TOPLAS)*, 1(1):121–141, 1979.

[9] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J.H. Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27, 2013.

[10] Politecnico di Milano. PandA framework. http://panda.dei.polimi.it, 2014.

[11] Xilinx. Vivado Design Suite. http://www.xilinx.com, 2013.

[12] Altera. Quartus II. http://www.altera.com, 2013.

[13] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis. *JIP*, 17:242–254, 2009.

[14] Zhiru Zhang and Bin Liu. Sdc-based modulo scheduling for pipeline synthesis. In *ICCAD*, pages 211–218, Nov 2013.

[15] A. Canis, S. Brown, and J. Anderson. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *FPL*, pages 1–8, 2014.