Springer Nature, Journal of signal processing systems for signal image and video technology
Volume: 92, Issue: 10, Pages: 1091-1099, DOI: 10.1007/s11265-020-01568-5

Pipeline Synthesis and Optimization from Branched Feedback Dataflow Programs

Anatoly Prihozhy[1] Simone Casale-Brunet[3] Endri Bezati[2] Marco Mattavelli[3]

Received: 18 April 2019 / Revised: / Accepted:

Abstract
Large dataflow designs are a result of behavioral specification of modern complex digital systems and/or a result of unfolding and transforming looped and branched programs. Since deep-submicron silicon technology provides large amounts of available resources, pipelining optimization without (or with minimal) resource sharing can give significant advantages in performance. High-level synthesis of CAL-programs is particularly popular in computation intensive applications (e.g., image and video processing, cryptography, wireless communication, etc.) where feedback actors with data flows at input and output ports represent loop-like behavior. In this work, we propose techniques for transforming, analysis, speculatively pipelining and optimizing large branched feedback dataflow programs. We develop an accurate algorithm and introduce fast dynamic and mixed static / dynamic heuristics that firstly minimize the number of pipeline stages for a given pipeline-stage time-period, and secondly minimize the overall pipeline registers size by means of appropriate assignment of feedbacks and instructions to pipeline stages. We also propose a genetic algorithm for tuning the heuristics for a particular design. The experimental results show the algorithms we propose give quickly solutions that are very close to accurate solutions and overcomes the earlier developed algorithms regarding computing time and pipeline parameters.

Keywords Dataflow Feedback Branching Pipeline High level synthesis Optimization

1 Introduction

Pipelining is a certain type of transformation of a digital system behavioral specification into a set of partitions that represent pipeline stages and execute in time-sliced fashion on the input data flow [1-5]. Pipelining increases the operating frequency and throughput of data-intensive digital systems with long critical paths. The optimization of pipeline implementations is a hard-combinatorial problem in general case, and its exhaustive solution is not feasible in acceptable CPU time. For early silicon technologies, pipeline architectures intensively explored the sharing of computational resources Most of the known optimization algorithms generate pipelines executing several clock cycles per stage cycle and solve the problem of how to share functional units among operators and to share pipeline registers among variables. Works [6 - 9] propose techniques that optimize highly parallelized pipelines without sharing resources.

Pipelining is a natural technique for dataflow designs, which organize data as flows over all parts of the designs. CAL is a key language for writing dataflow programs and modelling dataflow designs [10, 11]. It was developed and standardized to address the goal of high-level system specification and design, particularly addressing the wide field of streaming applications. A CAL-program is a network of actors, and an actor consists of one or more actions. The actors and actions accept tokens at input ports and produce tokens at output ports. Actions of an actor operate sequentially using a firing mechanism, and actors within a network operate concurrently. The most time-consuming actions determine the clock cycle period and slow down the hardware implementations synthesized from the dataflow CAL-programs when the cycle period is large. Our approach for increasing the throughput of the whole system is to break up the critical paths of such actions into pipelined partitions, which reduces the clock cycle period of the system. Pipelines without sharing resources are a most promising and efficient solution in this case. The algorithms of analyzing and optimizing such dataflow pipelines are the main value of this paper.

Pipelining the loop-like dataflow programs [10 - 11] represented in CAL differs pipelining the loops [12]. CAL models a loop as an actor with data tokens in input ports, one or more actions, one or more state variables, and feedback fragments that process the tokens and variables in the actions. As a result, the pipelining techniques and optimization algorithms are different for the loops and loop-like actors. Therefore, we should revise such known pipeline optimization techniques as modulo scheduling [12] and extend them for dataflow large-size CAL-programs.

Works [7 - 9] have already proposed some techniques for pipelining optimization of CAL-programs: the ASAP and ALAP algorithms minimize the number of pipeline stages; the optimal least cost search branch and bound algorithm (LCSBB) and the algorithm based on pipeline optimization dynamic heuristics (HADD) minimize the overall pipeline registers size. This article is an extended version of the paper [9]. We can summarize the novel contribution of the present work with respect to [9] in the following points:

Anatoly Prihozhy prihozhy@yahoo.com

[1] Computer and System Software Dpt., Belarusian National Technical University, Minsk, Belarus

[2] EPFL IC IINFCOM VLSC, École Polytechnique Fédérale de Lausanne, Switzerland

[3] EPFL SCI STI MM, École Polytechnique Fédérale de Lausanne, Switzerland

- modelling of loops using branched feedback dataflow programs represented in CAL
- transformation of actions of a CAL-program to a single basic block model
- analysis of a transformed CAL-program with respect to mutually exclusive instructions, data dependences, critical paths and feedback fragments
- speculative pipelining of branched feedback dataflow CAL-programs by means of extended ASAP, ALAP and LCSBB algorithms
- generalization and extension of the dynamic heuristic optimization algorithm HADD to speculative pipelining of branched feedback dataflow CAL-programs
- development of a genetic algorithm for tuning heuristic factors to efficiently optimize a particular design.

This paper has the structure as follows. Section II analyzes related work. Section III describes modelling, transformation and analysis of branched feedback dataflow programs in CAL. Section IV formulates a problem of speculative pipelining of branched feedback dataflow programs. Sections V and VI present pipelining optimization algorithms on dynamic and mixed static/dynamic heuristics, and on a genetic algorithm. Section VII reports experimental results, and the last section concludes the paper.

2 Related work

Numerous languages aim at describing pipelines: C, System-C and VHDL languages [13 - 15], data flow graphs [16], signal flow graphs [17], transactional specifications [18], binaries [19], CAPH [20], and other notations. Sehwa [1] is one of the first pipeline synthesis program. It minimizes the latency using a modified list-scheduling algorithm and a resource allocation table. The force directed scheduling that has been proposed in [17, 21] performs a time-constrained functional pipelining. Retiming moves registers in a circuit to decrease the length of longest paths while preserving the circuit behavior [22]. The ASAP (As Soon As Possible) and ALAP (As Late As Possible) algorithms that are developed in [8] on an operator conflict graph are similar to the downward and upward direction traversal algorithms [6].

Pipelining is an effective method for optimizing loops. Work [3] proposes the loop winding method that performs pipelining of data flow graphs without recurrences. The percolation-based scheduling [23] deals with the loop winding by starting with an optimal schedule that is obtained without considering resource constraints. The PLS pipelining is another effective method [4] to optimize loops for DSP. Work [16] introduces the rotation scheduling for loop pipelining by means of retiming formulation. Work [13] proposes the vectorization method based on pipelining of innermost loops by removing vector dependences in a loop nest. The speculative loop pipelining [19] generates a pipeline netlist at compile time and modifies it according to the result of runtime analysis. An integer linear programming formulation of the pipeline optimization problem is presented in [24] as an efficient approach for the design space exploration.

Modulo scheduling [25 - 27] is one of the most popular techniques to perform loop pipelining since it can achieve high-quality solutions with relatively low overhead in resources. Since finding an optimal modulo schedule is NP-hard in general, various heuristics have been proposed and implemented. Iterative modulo scheduling [28] schedules operations with backtracking. In work [29], the list scheduling and iterative modulo scheduling are used for the design space exploration based on slow, but area efficient modules, and fast, but area consuming modules. Swing modulo scheduling [30] reduces the register requirements by placing each operation close to either its predecessors or successors. Slack modulo scheduling [31] orders operations on a priority function and performs bidirectional choosing of time slots to minimize the conflicting variable lifetimes.

Work [12] proposes a method of modulo scheduling based on a system of difference constraints. Its authors have given a linear programming formulation to particularly solve a min-lifetime problem by an integer linear polynomial program and have proved that the constraint matrix is totally unimodular. Their assumption is each variable in the loop has only one producer, which significantly restricts the set of loop descriptions their method can optimally pipeline. Therefore, only heuristic algorithms are capable of solving the problem of pipelining optimization of branched feedback large-size nested loops, given constraints on resources.

3 Modelling of branched feedback dataflow programs

3.1 Dataflow modelling in CAL

Work [32] introduces the concept of actors as means of modeling distributed knowledge-based algorithms. Nowadays, actors are widely used in embedded systems, where actor-oriented design is a natural match to the heterogeneous and concurrent nature of such systems. In this paper, we utilize the CAL dataflow language [10, 11] that supports this concept to specify parallelism explicitly. CAL is suitable to model a variety of applications [33-44] from a wide range of domains (e.g., cryptography, multimedia processing, network processing, control systems, reconfigurable systems, power optimization, monitoring of HW and SW, and others).

An actor consists of input and output ports, state variables, actions, and a scheduler. Actors run in parallel. Actions execute over firing mechanism. Only one action of the actor fires at any moment in time. An actor scheduler specifies the sequence of actions firing. The scheduler operates accounting for the data tokens in input port buffers, the state of guards, priority conditions, and the presence of a finite state machine.

In this paper, in favor of high flexibility of pipeline synthesis and optimization, we focus on pipelining of one action, which fires depending on the tokens flow at input ports of an actor that may have many actions. Figure 1 depicts such sample CAL actor that will help us to illustrate techniques we propose. Although this actor has no scheduler, it represents a loop with branched body due to modeling feedbacks over two state variables $s1$ and $s2$.

```
actor sample ()
  int(size=16) i1_in, int(size=16) i2_in,  int(size=12) i3_in,
  int(size=16) i4_in                  ==>
  int(size=8) o1_out, int(size=8) o2_out, int(size=12) o3_out:
    int(size=16) s1 := 1;   int(size=12) s2 := 2;
    beh: action   i1_in:[i1], i2_in:[i2], i3_in:[i3], i4_in:[i4]
              ==>    o1_out:[o1], o2_out:[o2], o3_out:[o3]
    var
      int(size=12) a, int(size=12) b, int(size=12) c,
      int(size=12) d, int(size=12) e, int(size=16) f,
      int(size=1) t0, int(size=1) t1,
      int(size=1) t2, int(size=1) t3,
      int(size=8) o1, int(size=8) o2, int(size=12) o3
    do
      s1 := s1 ^ i1;   o1 := s1 >> i2;   a := i2 >> 4;
      b := i4 - i3;   s2 := s2 + b;   c := s2 * i4;
      d := 77;   if b < 7 then d := a | c; end
      e := d >> 5;  f := b << 4;
      o2 := c << 3;   if i2 > i4 then o2 := e * f; end
      o3 = i1 * i2;
    end
end
```

Figure 1 A CAL actor *sample* with action *beh* that uses state variables, and consumes and produces multiple tokens from its input and output ports.

3.2 Transformation of CAL-programs

Before pipelining optimizations of a CAL program, preliminary transformations convert it to an appropriate form. To illustrate these transformations, we perform transition from the sample CAL-actor (Figure 1) to the equivalent transformed actor shown in Figure 2.

The first transformation replaces all conditional expressions with Boolean variables in branching statements, and splits expressions so as no more than one operator would occur in the right part of any assignment. For instance, variable *t*0 replaces expression "*b*<7", and variable *t*1 replaces expression "*i*2>*i*4" in two conditional instructions.

Afterwards, it moves assignments over the action code to place all producers of one variable in mutually exclusive branches of nested conditional statements. For instance, assignment "d:=77;" moves into the *else*-part of the first conditional statement, and assignment "o2:=c<<3;" moves into the *else*-part of the second conditional statement. Additional variables may need to accomplish such a transformation in case of sophisticated data dependences. For example, in sequence "*x*:=*m*+*n*; *y*:=*x*/2; **if** *c* **then** *x*:=*m*-*n*; **end**" moving of "*x*:=*m*+*n*;" into the *else*-part of the conditional statement would force the assignment "*y*:=*x*/2;" to move into both conditional branches. This is a costly solution. It is better to introduce an additional variable *z* and to transform the code: "z:=*m*+*n*; *y*:=*z*/2; **if** *c* **then** *x*:=*m*-*n*; **else** *x*:=*z*; **end**".

Next transformation concerns state variables. An action that fires reads the value of a state variable, processes it, and finally writes a new value into the variable. The transformation introduces single *load* instruction before the first consumer, and introduces single *store* instruction after the last producer of each state variable *s*. It adds a new local temporal variable *st* in the action. Variable *st* propagates the value of *s* to all consumers.

The speculative (eager) execution is an optimization technique where a computer system performs some task that may not be needed [19]. Speculative execution refers to branching statements. We need this technique to speed up pipelined dataflow implementations. Speculative execution is very expensive, as amount of resources grows exponentially, and the overall pipeline registers size increases rapidly. Therefore, we solve the problem of finding out, what branches should execute eagerly, in parallel with solving the pipeline optimization problem. To do this, we have developed a recursive procedure that split all nested conditional statements into a sequence of simple *if-then* instructions with only one assignment inside. As a result, this transformation merges many basic blocks that are associated with various branches, to a single basic block [15]. For example, the procedure transforms two nested conditional statements

if *z*4 **then** *a*:=0; **else** **if** *z*5 **then** *a*:=1; **else** *a*:=2; **end end**

to the single basic block as follows:

*b*6:=**not** *z*4; *b*7:=**not** *z*5; *t*8:=*b*6 **and** *z*5; *t*9:=*b*6 **and** *b*7;
if *z*4 **then** *a*:=0; **end** **if** *t*8 **then** *a*:=1; **end** **if** *t*9 **then** *a*:=2; **end**

It has introduced four additional Boolean variables *b*6, *b*7, *t*8, and *t*9. In the single basic block model, simple instructions have larger mobility over pipeline stages, thus increasing capabilities for the pipeline optimizations.

```
beht: action i1_in:[i1], i2_in:[i2], i3_in:[i3], i4_in:[i4]
          ==>   o1_out:[o1], o2_out:[o2], o3_out:[o3]
  var
    int(size=16) s1l,  int(size=16) s1t,
    int(size=12) s2l,  int(size=12) s2t,
    int(size=12) a, int(size=12) b, int(size=12) c,
    int(size=12) d, int(size=12) e, int(size=16) f,
    int(size=1) t0, int(size=1) t1,
    int(size=1) t2, int(size=1) t3,
    int(size=8) o1, int(size=8) o2, int(size=12) o3
  do
    load(s1l, s1);  s1t := s1l ^ i1;            // 0-1
    store(s1, s1t);  o1 := s1t >> i2;           // 2-3
    a := i2 >> 4;  b := i4 - i3;                // 4-5
    load(s2l, s2);  s2t := s2l + b;             // 6-7
    c := s2t * i4;  store(s2, s2t);             // 8-9
    t0 := b < 7;  t2 := not t0;                 // 10-11
    if t0 then d := a | c; end                  // 12
    if t2 then d := 77; end                     // 13
    e := d >> 5;  f := b << 4;                  // 14-15
    t1 := i2 > i4;  t3 := not t1;               // 16-17
    if t1 then o2 := e * f; end                 // 18
    if t3 then o2 := c << 3; end                // 19
    o3 := i1 * i2;                              // 20
end
```

Figure 2 CAL action *beht* that is transformed to single basic block model.

In general terms, the transformed CAL-program specifies a set V of variables and a set P of operators (statements or instructions). Set V includes input tokens, local variables and output tokens. For each operator p it specifies variable subsets $inputs(p) \subseteq V$ and $outputs(p) \subseteq V$. For each variable v of bit-size $size(v)$, it specifies subsets $prod(v) \subseteq P$ of operators-producers and $cons(v) \subseteq P$ of operators-consumers.

It is also a source for computing a direct precedence relation R_{direct} on the set P of operators. Very often, it is difficult to find out the precedence between two *if-then* statements, for instance, between "**if** *t*0 **then** *a*:=0; **end**" and "**if** *t*1 **then** *b*:=*a*; **end**". In case, conditional variables *t*0 and *t*1

take value *true* simultaneously, the first statement precedes the second one. At the same time, in case, $t0$ and $t1$ never take value *true* simultaneously, they do not precede each other.

3.3 Analysis of CAL-programs

Let $T = \{t_1,\ldots t_n\}$ be a set of conditional Boolean variables. Let $Z = \{z_1,\ldots,z_k\}$ be a set of primary Boolean variables, which are not expressed over other Boolean variables using Boolean operators. Sets T and Z may intersect, as a primary variable can be at the same time a conditional one. Let $H = \{h_1(r),\ldots,h_n(r)\}$ be a set of Boolean functions that evaluate the conditional variables of T over vector z of primary variables. Let $F = \{ f(z_i, z_j) \mid i, j = 1,\ldots,k,\ i < j \}$ be a set of feasible Boolean functions for values of pairs of primary variables.

We define two Boolean variables t_i and t_j of T as orthogonal if they never take value 1 simultaneously. We define a subset $E \subseteq T$ of variables as orthogonal if all pairs of variables of E are orthogonal. Boolean equations as follows describe the orthogonal condition for two conditional variables t_i and t_j:

$$\forall z \ \left(\lambda(z) \to \mu(z)\right) \tag{1}$$

where

$$\lambda(z) = \underset{\substack{i,j\in\{1..k\}\\ i<j}}{AND} f(z_i, z_j) \tag{2}$$

and

$$\mu(z) = \neg h_i(z) \vee \neg h_j(z) \tag{3}$$

In (1) – (3), $\forall$ is universal quantifier, $\to$ is Boolean implication, $\vee$ is Boolean disjunction, and $\neg$ is Boolean negation. Boolean function $\lambda(z)$ characterizes the set of vector values of primary variables, which are feasible during program execution. Boolean function $\mu(z)$ takes value 1 when t_i and t_j are orthogonal and takes value 0 otherwise.

Equation (1) describes a partial tautology. We have to prove $\mu(z) = 1$ when $\lambda(z) = 1$, and we do not need a proof of $\mu(z) = 1$ when $\lambda(z) = 0$. The procedure of traversing all vector values of z and checking out the satisfiability of $\lambda(z)\to\mu(z)$ at each value has high computational complexity, therefore we reformulate this tautology problem to a satisfiability (SAT) one and solve it automatically with a contradiction tool.

We define an orthogonal subset E of variables as complete if the equation as follows holds:

$$\forall z\left(\lambda(z) \to \bigvee_{e_i \in E} h_{e_i} \right) \tag{4}$$

At any moment of program execution, exactly one variable of complete subset E takes value 1, and others take value 0.

For example, the CAL-code shown in Figure 2 uses set $Z = \{t0, t2\}$ of primary variables and set $T = \{t0, t1, t2, t3\}$ of conditional variables. Sets R and T intersect. Vector $r = (t0, t1)$, and function $\lambda(r)$ is Boolean constant 1. Functions $h0 = t0$, $h1 = t1$, $h2 = \neg t0$ and $h3 = \neg t1$ evaluate the variables of T. Let us prove the variables $t0$ and $t2$ be orthogonal: $1 \to \mu(r) = 1 \to \neg t0 \vee \neg(\neg t0) = \neg t0 \vee t0 = 1$. Moreover, according to (4), $1\to t0 \vee \neg t0 = 1$. Therefore, orthogonal variables $t0$ and $t2$ constitute the complete subset. Similarly, subset $\{t1, t3\}$ is also completely orthogonal.

The second example code is the result of transform of two nested conditional instructions to the single basic block from Section 3.2. The set of primary variables is $Z = \{z4, z5\}$, and the set of conditional variables is $T = \{z4, t8, t9\}$. Sets Z and T intersect. In vector $z = (z4, z5)$, variables $z4$ and $z5$ are independent, therefore function $\lambda = 1$. According to (1) – (3), variables $z4$ and $t8$ are orthogonal: $1 \to \mu = 1 \to \neg z4 \vee \neg(\neg z4 \wedge z5) = 1$. Similarly, $z4$ and $t9$ are orthogonal: $1 \to \mu = 1 \to \neg z4 \vee \neg(\neg z4 \wedge \neg z5) = 1$. Variables $t8$ and $t9$ are also orthogonal: $1 \to \mu = 1 \to \neg(\neg z4 \wedge z5) \vee \neg(\neg z4 \wedge \neg z5) = 1$. Moreover, according to (4), $1 \to z4 \vee t8 \vee t9 = 1 \to z4 \vee (\neg z4 \wedge z5) \vee(\neg z4 \wedge \neg z5) = 1$. Therefore, $z4$, $t8$ and $t9$ constitute the complete orthogonal subset.

The third example code contains relational operators:

*t*10:=*x*>*y*; *t*11:=*x*<=*y*; **if** *t*10 **then** *a*:=0; **end if** *t*11 **then** *a*:=1; **end**

The set $Z = T = \{t10, t11\}$ of conditional variables is the same as the set of primary variables. Boolean function $\lambda = f(t10, t11) = t10 \oplus t11$ (exclusive or) determines feasible values of pairs of primary variables $t10$ and $t11$. On values 00, 01, 10, 11 of the vector arguments, it takes values 0, 1, 1, 0 respectively. According to (1) -(3), the orthogonal condition is $(t10 \oplus t11) \to (\neg t10 \vee \neg t11) = 1$. Moreover, according to (4), $(t10 \oplus t11) \to t10 \vee t11 = 1$. Therefore, $t10$ and $t11$ constitute the complete orthogonal subset. The same procedure proves, that replacing "*t*10:=*x*>*y*;" with "*t*10:=*x*>=*y*;" makes variables $t10$ and $t11$ to be non-orthogonal.

In our work, we investigate the orthogonal condition for various CAL operators (including relational operators) and various operands of these operators.

The inputs and outputs of operators and the orthogonal relation between conditional variables provide evaluation of the operators direct precedence relation: R_{direct} = { (0,1), (1,2), (1,3), (2,0), (4,12), (5,7), (5,10), (5,15), (6,7), (7,8), (7,9), (8,12), (8,19), (9,6), (10,11), (10,12), (11,13), (12,14), (13,14), (14,18), (15,18), (16,17), (16,18), (17,19) }. It describes a directed cyclic direct precedence graph. R_{direct} without the feedback pairs (i.e. pairs (2,0) and (9,6) in our example) is a source of calculating its transitive closure R. Computing an anti-transitive relation from R allows to calculate a set *succ*(p) of all direct successors and a set *pred*(p) of all direct predecessors of each operator $p \in P$.

We use relative delays of operators and an additive model to calculate the delays along the longest paths in the acyclic graph that we derive from relation R. Figure 3 presents a matrix G of all operator pairs longest paths lengths. Its rows and columns correspond to operators. The elements on the principal diagonal are operators' relative delays. During pipeline synthesis, we also take into account delays associated with *if-then* instructions in case at least two of them are producers of the same variable. According to Figure 3, the critical path length equals 9.54.

Another goal of analysis is determining feedback regions of state variables. A feedback region $Fbr(s) \subseteq P$ of variable s is a subset of operators that include the successors of *load* instructions of s and predecessors of *store* instructions of s:

$$Fbr(s) = load_s \cup store_s \cup \left[\left(\bigcup_{p \in load_s} succ(p)\right) \cap \left(\bigcup_{q \in store_s} pred(q)\right)\right] \quad (5)$$

where $load_s \subseteq P$ is a subset of read operators of s, and $store_s \subseteq P$ is a subset of write operators of s. the sample CAL-actor has two state variables, $s1$ and $s2$. For $s1$, $load_{s1} = \{0\}$, $store_{s1} = \{2\}$, $succ(0) = \{1, 2, 3\}$ and $pred(2) = \{0, 1\}$. According to (5), $Fbr(s1) = \{0, 1, 2\}$. For $s2$, $load_{s2} = \{6\}$, $store_{s2} = \{9\}$, $succ(6) = \{7, 8, 9, 12, 14, 18, 19\}$, and $pred(9) = \{5, 6, 7\}$. According to (5), $Fbr(s2) = \{6, 7, 9\}$.

The longest path in region $Fbr(s1)$ connects operators 0, 1 and 2, therefore element (0,2) of matrix G in Figure 3 is the path length of 1.3. Similarly, the length of the longest path in region $Fbr(s2)$ that connects operators 6, 7 and 9 is 2.32. As all operators of each feedback region must belong to one pipeline stage, the pipeline stage time may not be less than the maximum length over all regions. Therefore, inequality $T_{stage} \geq 2.32$ must hold for the example CAL actor.

4 Speculative pipelining of a branched feedback dataflow program

Given a dataflow program describing a system behavior, the objective is to minimize the number of pipeline stages denoted as S, and to find a best assignment of each operator p to a stage denoted as $stage(p)$.

4.1 Conflicts between operators in pipeline

Given the delays of operators, matrix G of longest path lengths between the pairs of operators, and a constraint T_{stage} on the pipeline-stage time-period, we calculate the operator conflict relation C as:

$$C = \left\{(i, j) \mid i < j \text{ and } G_{ij} \geq T_{stage}\right\} \quad (6)$$

In pair $(i, j) \in C$, operators i and j may not belong to the same pipeline stage. An operator nonconflict relation C_n is computed as $C_n = R \setminus C$. In pair $(i, j) \in C_n$, operator j may not be assigned to a stage that precedes the stage which operator i is assigned to. We may use set R instead of set C_n. To speed up the optimization process, we use instead of C and C_n their anti-transitive versions. Thus, for T_{stage}=4.0 and for matrix G shown in Figure 3, the operator conflict relation is C={(4,18), (5,8), (5,12), (5,19), (6,8), (6,12), (6,19), (7,8), (7,12), (7,19), (8,14), (8,18), (10,18), (11,18), (12,18), (13,18)}.

For each pair $(i, j) \in C$, inequality $stage(i) < stage(j)$ holds, and for each pair $(i, j) \in C_n$, inequality $stage(i) \leq stage(j)$ holds. We evaluate the set $cdpred(p) \subseteq P$ of direct predecessors on the relation (graph) C, and the set $ncdpred(p) \subseteq P$ of direct predecessors on relation (graph) C_n for each operator p. We also evaluate the set $cdsucc(p) \subseteq P$ of direct successors on C, and the set $ncdsucc(p) \subseteq P$ of direct successors on C_n.

4.2 ASAP and ALAP branched feedback program scheduling

In a dataflow program without feedbacks, the downward and upward direction dataflow traversal algorithms, ASAP and ALAP [6-8] can generate *asap* and *alap* pipeline schedules on the conflict graph C. The algorithms do not meet the requirements of the dataflow programs that guarantee valid processing of feedbacks. We have extended these algorithms to map all operators of one feedback region to one stage, and called them FASAP and FALAP. Applying the extended algorithms to the sample action at T_{stage}=4.0 yields *fasap* and *falap* pipeline schedules shown in Figures 4 and 5. A function $stage(p)$, $p \in P$ maps the operators onto the set S of pipeline stages, and formally describes the schedule.

Both FASAP and FALAP algorithms process conditional instructions in the same way: they distribute *if-then* instructions on pipeline stages according to relations C and C_n, and according to their operation strategies. They can distribute *if-then* instructions that produce the same variable in different manner. Thus, in Figure 4 (*fasap*), both instructions 12 and 13 produce values of the same variable d, but FASAP assigns instruction 13 to stage 1, and assigns instruction 12 to stage 2.

$$G = \begin{bmatrix}
0.10 & 0.30 & 1.30 & 1.10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 0.20 & 1.20 & 1.00 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & 1.00 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
 & & & 0.80 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & 0.01 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.30 & 0 & 1.21 & 0 & 0 & 0 & 4.14 & 0 & 0 \\
 & & & & & 1.36 & 0 & 2.58 & 5.40 & 3.58 & 2.46 & 2.56 & 5.40 & 2.76 & 6.61 & 1.37 & 0 & 0 & 9.54 & 5.40 & 0 \\
 & & & & & & 0.10 & 1.32 & 4.14 & 2.32 & 0 & 0 & 4.44 & 0 & 5.35 & 0 & 0 & 0 & 8.28 & 4.14 & 0 \\
 & & & & & & & 1.22 & 4.04 & 2.22 & 0 & 0 & 4.34 & 0 & 5.25 & 0 & 0 & 0 & 8.18 & 2.82 & 0 \\
 & & & & & & & & 2.82 & 0 & 0 & 0 & 3.12 & 0 & 4.03 & 0 & 0 & 0 & 6.96 & 0 & 0 \\
 & & & & & & & & & 1.00 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & 1.10 & 1.20 & 1.40 & 1.40 & 2.31 & 0 & 0 & 0 & 5.24 & 0 & 0 \\
 & & & & & & & & & & & 0.10 & 0 & 0.30 & 1.21 & 0 & 0 & 0 & 4.14 & 0 & 0 \\
 & & & & & & & & & & & & 0.30 & 0 & 1.21 & 0 & 0 & 0 & 4.14 & 0 & 0 \\
 & & & & & & & & & & & & & 0.20 & 1.11 & 0 & 0 & 0 & 4.04 & 0 & 0 \\
 & & & & & & & & & & & & & & 0.01 & 0 & 0 & 0 & 2.94 & 0 & 0 \\
 & & & & & & & & & & & & & & & 0.01 & 0 & 0 & 2.94 & 0 & 0 \\
 & & & & & & & & & & & & & & & & 1.00 & 1.10 & 3.94 & 1.11 & 0 \\
 & & & & & & & & & & & & & & & & & 0.10 & 0 & 0.11 & 0 \\
 & & & & & & & & & & & & & & & & & & 2.94 & 0 & 0 \\
 & & & & & & & & & & & & & & & & & & & 0.01 & 0 \\
 & & & & & & & & & & & & & & & & & & & & 2.82
\end{bmatrix}$$

Figure 3 Longest paths lengths matrix G for action in Figure 2.

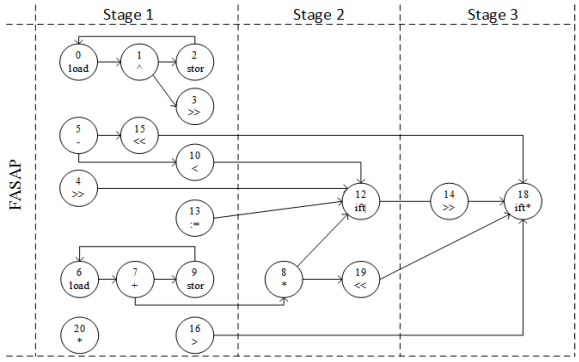

Figure 4 Example 3-stage pipeline *fasap* with overall registers size of 147 bit.

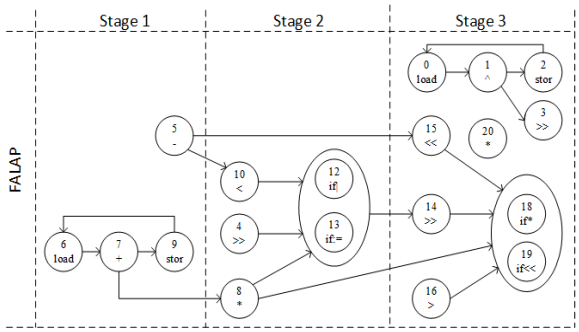

Figure 5 Example 3-stage pipeline *falap* with overall registers size of 156 bit.

Inputs	i1, i2, i3, i4

Stage 1	**load**(s1l,s1); s1t := s1l ^ i1;
	store(s1,s1t); o1 := s1t >> i2;
	a := i2 >> 4; b := i4 - i3;
	load(s2l,s2); s2t := s2l + b;
	store(s2,s2t); t0 := b < 7;
	d_ := 77; f := b << 4;
	t1 := i2 > i4; o3 := i1 * i2;

Registers	i4, o1, o3, s2t, a, d_, f, t0, t1 // 90 bit

Stage 2	c := s2t * i4;
	if t0 **then** d := a | c; **else** d := d_; **end**
	o2_ := c << 3;

Registers	o1, o2_, o3, d, f, t1 // 57 bit
Stage 3	e := d >> 5; **if** t1 **then** o2 := e * f; **else** o2 := o2_; **end**
Outputs	o1, o2, o3

Figure 6 Stage instructions and registers between stages in *fasap* pipeline.

As a result, it releases operator 13 from *if-then* in the form "*d*_:=77;", and replaces instruction 12 with "**if** *t*0 **then** *d*:=*a*|*c*; **else** *d*:=*d*_; **end**". Similarly, FASAP replaces operator 19 with "*o*2_:=*c*<<3;", and replaces instruction 18 with "**if** *t*1 **then** *o*2:=*e***f*; **else** *o*2:=*o*2_; **end**". FALAP assigns instructions 12 and 13, as well as instructions 18 and 19, to the same stage (Figure 5). Our scheduling tool has merged the instructions in these pairs, resulting in *if-then-else* instructions as shown in Figure 1. It has removed instructions 11 and 17 in both *fasap* and *falap*.

The *fasap* and *falap* schedules play key role in the pipeline synthesis and optimization. Firstly, they determine the minimal number of stages in pipeline; it is equal to three in our sample action. Secondly, they allow the evaluation of each operator mobility over pipeline stages and allow the evaluation of the mobility of each feedback fragment. For example, operator 10 has mobility of two, as it can execute in stage 1 at earliest (Figure 4) and can execute in stage 2 at latest (Figure 5). Feedback fragment *Fbr*(*s*1) has the mobility of three; it executes in stage 1 according to *fasap*, and executes in stage 3 according to *falap*. Feedback fragment *Fbr*(*s*2) has the mobility of one.

Figure 6 shows an assignment of the instructions to the stages, and an assignment of variables to registers in schedule *fasap*. Our tool generates a CAL-representation of the pipeline. It represents each stage as a separate actor and represents all registers as ports.

4.3 Overall pipeline registers size

Once the minimal number of stages has been determined, moving operators and feedback fragments of operators from one stage to previous or next stages produces a space of all pipeline schedules for the given stage-time period T_{stage}. As shown in works [7, 8], the space size grows exponentially. For the given number of stages and for the generated *stage*(*p*), $p \in P$, we can calculate the overall registers size *RSize* as

$$RSize(stage) = \sum_{v \in V} size(v) \times lifetime(v) \tag{7}$$

$$lifetime(v) = \max_{q \in cons(v)} stage(q) - \min_{p \in prod(v)} stage(p) \tag{8}$$

Applying (7) and (8) to the pipeline schedules *fasap* and *falap* yields to *RSize* = 147 bit and *RSize*= 156 bit respectively (Figure 4 and 5). Figures 6 shows that in the *fasap* schedule the size of registers between the first and second stage equals 90 bit, and equals 57 bit between the second and third stage.

4.4 Optimal scheduling of branched feedback programs

In this paper, we extend the FLCSBB algorithm [8] for the branched feedback dataflow programs and call it FLCSBB. The key step in the extension is evaluating the mobility of feedback regions over pipeline stages in addition to the mobility of separate instructions. We determine the earliest and latest stages of each region, and construct a conflict relation (graph) on the set of regions, that is similar to the conflict relation (graph) *C*. The new graph provides a generation of valid assignments of feedback regions to pipeline stages. This is a master procedure, for which former LCSBB is a slave procedure.

For the sample actor, FLCSBB has produced three combinations of assignments of feedback regions *Fbr*(*s*1) and *Fbr*(*s*2) to three pipeline stages: (1, 1), (2, 1), (3, 1). For each combination, it has generated a best pipeline, and obtained using (7) and (8) three values of *Rsize*: 126, 125, and 137 bit. Among them, the 3-stage pipeline *flcsbb* that is shown in Figure 7 has given a minimum of *RSize* = 125 bit.

At the same assignment of the feedback regions, FASAP has given larger *RSize* of 148 bit against FLCSBB that has given 126 bit, and FALAP has given 156 bit against 137 bit by FLCSBB. This example proves that FLCSBB is capable of significant reduction of *RSize*. At the same time, FLCSBB can handle only small designs.

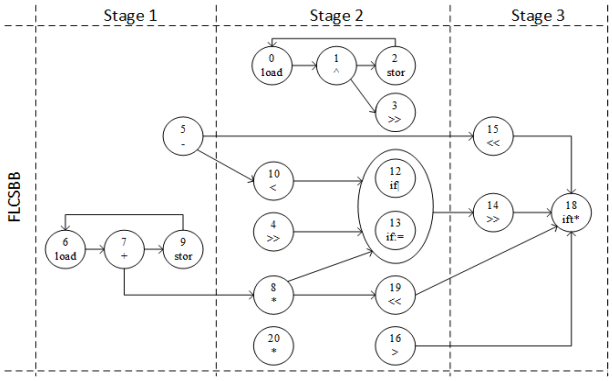

Figure 7 Example 3-stage pipeline *flcsbb* with overall registers size of 125 bit.

5 Pipeline optimization based on heuristics

The key task of a pipeline optimization technique without sharing resources is to choose for each instruction an appropriate stage. To obtain an optimal solution, we need a backtracking mechanism. To obtain a heuristic solution, we take iteratively two decisions: what operator (feedback fragment) is to be scheduled next, and what is the pipeline stage the operator has to be assigned to. In this Section, we propose two Feedback dataflow optimization Heuristic Algorithms: FHADD uses Dynamic heuristics for choosing the operator and Dynamic heuristics for choosing the stage; FHASD uses Static heuristics for choosing the operator and uses Dynamic heuristics for choosing the stage. These algorithms explores a concept of partially generated pipeline schedule.

5.1 Optimization flow

This Section gives an overview of how FHADD works. Firstly, it recognizes the feedback fragments of instructions, computes a precedence relation on the set of fragments, and determines their lifetime intervals over the pipeline stages. In the sample CAL-action, there are two feedback fragments, *Fbr*(*s*1) and *Fbr*(*s*2). In 3-stage pipelines, *Fbr*(*s*1) lives over the stages from 1 to 3, and *Fbr*(*s*2) lives within stage 1 (see Figures 4 and 5). FHADD performs a traversal of all combinations of stages for fragments, assigns the instructions of these fragments to corresponding stages, and searches for an optimal assignment of other instructions (that are out of the fragments) to the stages. Bellow, we consider the assignment of *Fbr*(*s*1) to stage 2 and consider the assignment of *Fbr*(*s*2) to stage 1.

On our sample CAL-actor, Figures 8 and 9 depict a systematic pipeline heuristic optimization flow that FHADD implements. In the initial state (Figure 8a), FHADD assigns operators 0, 1, 2 of *Fbr*(*s*1) to stage 2, and assigns operators 6, 7, 9 of *Fbr*(*s*2) to stage 1. Then it associates a range of stages with each operator *p* that is out of the feedback fragments. We determine this range by running the FASAP and FALAP algorithms.

We denote a lower bound of the operator's range as *early*(*p*) and denote an upper bound as *late*(*p*). When the range includes one stage, the corresponding operator *p* has mobility one, is assigned to this stage, and equality *stage*(*p*) = *early*(*p*) = *late*(*p*) holds. When FHADD has assigned only a part of operators on stages, it cannot calculate the accurate value of *Rsize*. Instead, it estimates a lower bound of the overall pipeline registers size (*rslb*). In this Section, we report only values of *rslb*, and describe details of the estimation in the next Section.

In Figure 8a there are five operators, i.e. 5, 8, 12, 14 and 18, that have mobility one and are initially assigned to stages 1, 2, 2, 3 and 3 respectively. Other operators have a mobility larger than one. Such operators (i.e. 3, 4, 10, 13, 15, 16, 19 and 20) are assigned to stages depending on *rslb*, as the size of operator's inputs is not equal to the size of operator's outputs. Figure 8a reports the value of *rslb* for each such operator and each available stage. Operators 11 and 17 that have the same size of inputs and outputs may move over stages without changing *rslb*. FHADD assigns such operators to stages at the end of scheduling process.

For each operator *p* whose range of available stages satisfies inequality *early*(*p*) < *late*(*p*), FHADD evaluates a weight $\chi(p)$, the larger value of which indicates its prominence for good scheduling. It selects an operator *p* with a maximum of a heuristic parameter $\chi(p)$ and assigns it to a *stage*(*p*), which provides a minimum of *rslb*. In this Section, we only report values of $\chi(p)$, and introduce a method of their computation in Section 5.3.

Thus, at the scheduling steps from a) to h) (Figure 8), FHADD chooses operators *p* = 3, 16, 10, 4, 13, 19, 20 and 15 with maximum weights of $\chi(p)$ = 0.2495, 0.2267, 0.2244, 0.1939, 0.1794, 0.1939, 0.1568 and 0.1648, and assigns them to stages 2, 2, 1, 2, 2, 2, 2 and 3 respectively. The assignment of an operator to a stage can make tighter a stage range of other operators. The lower bound of the overall pipeline registers size can only grow during systematic assignment of operators to stages (Figure 9): *rslb* = 80, 81, 82, 82, 82, 90, 102 and 126. This growth is due to the increase of the lower bound of variables lifetimes. Thus, variables *t*1, *t*0, *o*2, *o*3 and *b* has increased their lifetime at scheduling steps b), c), f), g) and h) (Figure 9).

5.2 Lower and upper bounds of registers size

Assume that the lower bound *early*(*p*) of operator *p* has been updated to *early'*(*p*). Then we can calculate a new lower bound *early'*(*q*) of successor *q* of operator *p* applying (9) to the current *early*(*q*) and the new *early'*(*p*).

$$early'(q) = \begin{cases} early'(p)+1, & if\ q \in cdsucc(p)\ and \\ & early(q) < early'(p)+1, \\ early'(p), & if\ q \in ncdsucc(p)\ and \\ & early(q) < early'(p), \\ early(q), & otherwise. \end{cases} \quad (9)$$

Assume that the upper bound *late*(*p*) of operator *p* has been updated to *late'*(*p*). Then we can calculate a new upper bound *late'*(*q*) of a predecessor *q* of the operator *p* applying (10) to the current *late*(*q*) and the new *late'*(*p*).

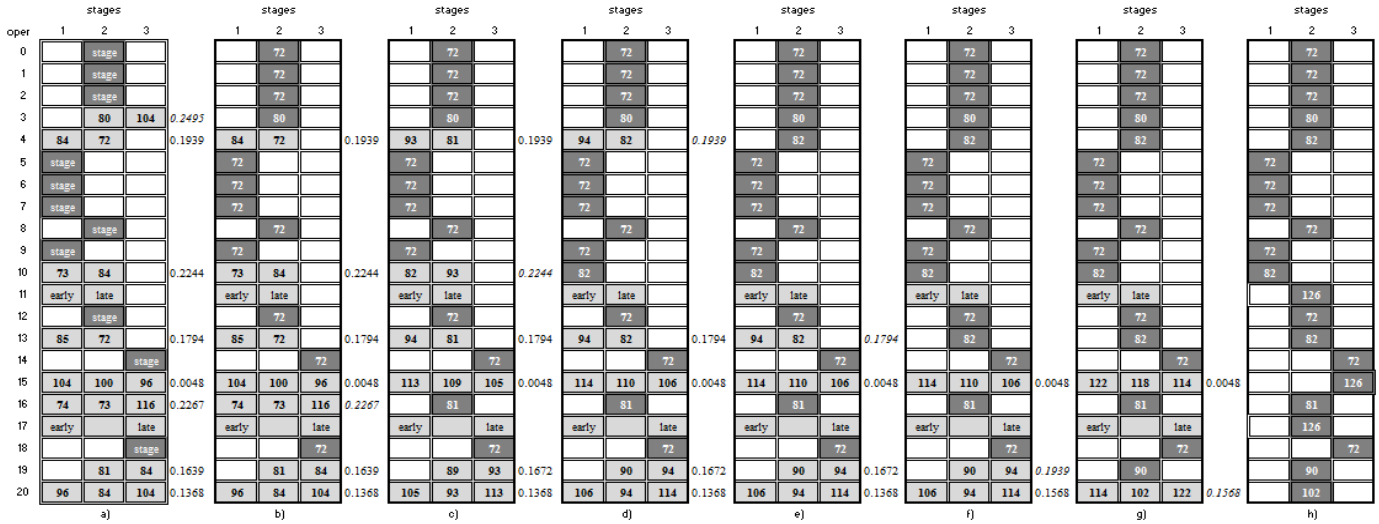

Figure 8 Assignment of operators to stages in example 3-stage pipeline; operator 11 and 17 are assigned to stage 2 at the end of optimization process and are removed after reconstruction of conditional instructions

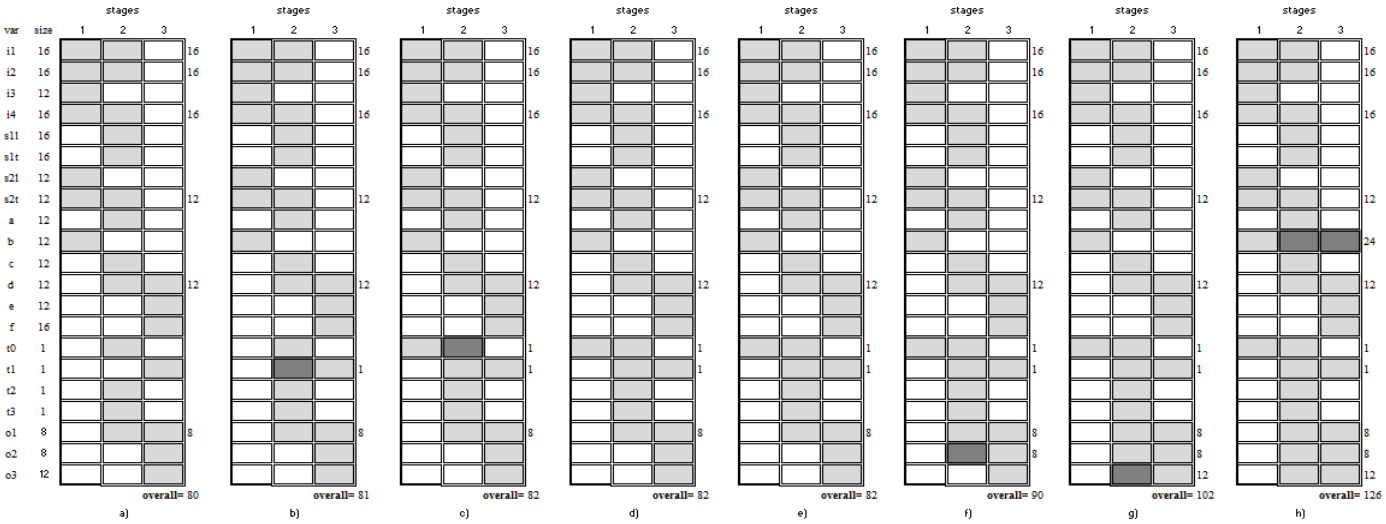

Figure 9 Lower bound of variables lifetimes over pipeline stages in example 3-stage pipeline

$$late'(q) = \begin{cases} late'(p) - 1, & \text{if } q \in cdpred(p) \text{ and} \\ & late(q) > late'(p) - 1, \\ late'(p), & \text{if } q \in ncdpred(p) \text{ and} \\ & late(q) > late'(p), \\ late(q), & \text{otherwise.} \end{cases} \quad (10)$$

The lower bound *rslb* of the overall pipeline registers size is a key heuristic parameter for pipeline optimization. We evaluate it using (11) and (12).

$$rslb = \sum_{v \in V} size(v) \times lifestim(v) \quad (11)$$

$$lifestim(v) = pos\big(late(v) - early(v)\big) \quad (12)$$

In (12), $pos(x) = 0$ if $x \le 0$ and $pos(x) = x$ otherwise. The upper bound *early*(*v*) is the earliest stage of producers of variable *v*, and lower bound *late*(*v*) is the latest stage of consumers of *v*. The initialization of FHADD assigns a minimum of *falap*(*p*) on all $p \in prod(v)$ to *early*(*v*), and assigns a maximum of *fasap*(*p*) on all $p \in cons(v)$ to *late*(*v*). Whenever the upper bound *late*(*p*) has been updated to *late*'(*p*) for any $p \in prod(v)$, a new upper bound *early*'(*v*) is calculated applying (13) to the current *early*(*v*) and the new *late*'(*p*).

$$early'(v) = \begin{cases} late'(p), & \text{if } v \in outputs(p) \text{ and} \\ & early(v) > late'(p), \\ early(v), & \text{otherwise} \end{cases} \quad (13)$$

Similarly, whenever the lower bound *early*(*p*) has been updated to *early*'(*p*) for any $p \in cons(v)$, a new lower bound *late*'(*v*) is calculated applying (14) to the current *late*(*v*) and the new *early*'(*p*).

$$late'(v) = \begin{cases} early'(p), & if\ v \in inputs(p)\ and \\ & late(v) < early'(p), \\ late(v), & otherwise \end{cases} \quad (14)$$

FHADD explores (9) – (14) at each step of pipeline optimization flow.

5.3 Heuristics of choosing operators and stages

Now our focus is on the operators (instructions) that have mobility larger than one. At each step of optimization flow, FHADD uses heuristics for choosing a next-scheduled operator (instruction) and uses heuristics for choosing an available stage for the operator assignment. As FHADD does not perform backtracking, these two decisions significantly influence the final pipeline parameters.

To take the first decision, we introduce for operator p a heuristic weight $\chi(p)$ whose maximal value indicates a preferable operator for scheduling:

$$\chi(p) = \sum_{i=1}^{\kappa} \omega_i \cdot \rho_i(p) \quad (15)$$

where $\rho_i(p)$ is a heuristic parameter; ω_i is a heuristic factor such that $\sum_{i=1...k} \omega_i = 1$; k is the number of parameters. Each parameter $\rho_i(p)$ varies its value from 0 to 1. A higher value of the parameter implies better properties of pipeline. In this paper, we explore four heuristic parameters $\rho_1 - \rho_4$. Parameter ρ_1 evaluates the operator mobility, and parameter ρ_4 evaluates the difference between the size of the operator inputs and the size of its outputs. Parameter ρ_2 evaluates *rslb*-difference over operator's available stages, and parameter ρ_3 evaluates *rslb*-difference over non-scheduled operators.

To take the second decision, we use *rslb* as heuristics whose minimal value indicates a preferable stage to assign the operator.

5.4 Heuristic algorithms FHADD and FHASD for feedback dataflow programs

Algorithm 1 assigns the feedback fragments to the pipeline stages, and maps other instructions onto the stages by means of calling algorithm HADD or algorithm HASD depending on the value of *mode*. It generates various valid combinations of fragments assignment by means of using a precedence relation on the set of feedback fragments, available early and late stages for each fragment, and a mobility of each fragment over the stages. HADD explores dynamic heuristics, and HASD explores both static and dynamic heuristics. They use the mapping of the fragments onto the stages as input and generate the mapping *stage* of other operators onto the stages.

5.5 Heuristic algorithm HADD

Algorithm 2 summarizes HADD as follows. Firstly, it initializes the optimization state by calculating the initial values of *early* and *late* for all operators and calculating initial values of *early*, *late* and *rslb* for all variables. Secondly, in a loop, it iteratively assigns nonscheduled operators of Q to the most preferable pipeline stages. In order to choose the best operator for scheduling, HADD computes a value of *rslb*(q, s),

---

Algorithm 1: FHADD-FHASD

Input: A *mode*∈{*FHADD*, *FHASD*} of algorithm operation
Input: A set F of feedback fragments
Input: A set S of pipeline stages
Input: A precedence relation P_F on set F
Input: An earliest stage *earlyf*(f)∈S of fragment f∈F
Input: A latest stage *latef*(f)∈S of fragment f∈F
Output: A best function *stageFB*(f) of mapping F onto S
Output: A best function *stageB*(p) of mapping operators P onto S
Output: A global optimum *GRSize* of overall pipeline registers size
Intermediate: Current mapping *stagef* of F onto S
Intermediate: Current mapping *stage* of P onto S
GRSize ← ∞
while *New assignment of fragments to stages exists* **do**
  stagef ← *GenerateNewAssignment*(F, S, P_F, *earlyf*, *latef*)
  if *mode=FHADD* **then**
    RSize ← HADD (F, *stagef*, *stage*)
  else
    RSize ← HASD (F, *stagef*, *stage*)
  if *GRSize* > *RSize* **then**
    GRSize ← *Rsize* *stageFB* ← *stagef* *stageB* ← *stage*
return *stageFB*, *stageB*, *GRSize*

---

which is evaluated by function *UpdateOptimizationState* in case the nonscheduled operator q was assigned to the available pipeline stage s in the current optimization state.

Then it estimates the vector $\rho(q)$ of dynamic heuristic parameters executing function *heuristicsEvaluateD*, and calculates the operator's weight $\chi(q)$. After that, it selects the best operator *operBest* yielding a maximum of χ and the best stage *stageBest* yielding a minimum of *rslb* and recalculates the optimization state after the assignment of *operBest* to *stageBest* by executing function *UpdateOptimizationState*. The actual value *true* of the first parameter means that the function computes and returns a new value of the arrays *stage*, *early* and *late* for the operators of Q, a new value of the arrays *earlyv*, *latev* and *rslb* for the variables of V, and finally returns a value of *RSize*. The arrays *earlyv* and *latev* contain *early*(v) and *late*(v) for all v∈V. When the first parameter is equal to *false*, the function only computes and returns the value of *rslb*.

5.6 Updating the optimization state

Algorithm 3 describes the *OptimizationStateUpdate* procedure that uses two lists. It works when HADD assigns operator q to stage s. List P_{upd} includes operator q or other operator p whose bounding stages *early*(p) and/or *late*(p) are updated due to updating the state of q. List V_{upd} includes variable v whose bounding stages *early*(v) and/or *late*(v) are updated due to updating the state of q.

Firstly, the algorithm sorts list P_{upd} on ascending and then iteratively extracts the first operator f from the list. It computes a new value of *early'*(p) for each successor p of operator f. If *early'*(p) is unequal to *early*(p), the algorithm considers p as updated and includes it in P_{upd}. After that it computes a new value of *late'*(v) for each v∈*input*(f). If *late'*(v) is unequal to *late*(v), v is included in V_{upd}. Secondly, the algorithm sorts list P_{upd} on descending and iteratively extracts the first operator f from the list. It computes a new value of *late'*(p) for each predecessor p of operator f. If *late'*(p) is unequal to *late*(p), it considers p as updated and includes it in P_{upd}. Then it computes a new value of *early'*(v) for each v∈*output*(f).

Algorithm 2: HADD

Input: A set F of feedback fragments
Input: A function *stagef*(f) which maps fragments F onto stages S
Input: A set V of variables
Input: A set P of operators
Input: A set S of pipeline stages
Input: A set *inputs*(p)$\subseteq V$ of variables for operator $p \in P$
Input: A set *outputs*(p)$\subseteq V$ of variables for operator $p \in P$
Input: A set *producers*(v)$\subseteq P$ of operators for variable $v \in V$
Input: A set *consumers*(v)$\subseteq P$ of operators for variable $v \in V$
Input: A set *cdpred*(p)$\subseteq P$ of predecessors of operator $p \in P$ on conflict graph C
Input: A set *cdsucc*(p)$\subseteq P$ of successors of operator $p \in P$ on conflict graph C
Input: A set *ncdpred*(p)$\subseteq P$ of predecessors of operator $p \in P$ on nonconflict graph C_n
Input: A set *ncdsucc*(p)$\subseteq P$ of successors of operator $p \in P$ on nonconflict graph C_n
Input: An initially earliest stage *fasap*(p)$\in S$ of operator $p \in P$
Input: An initially latest stage *falap*(p)$\in S$ of operator $p \in P$
Input: A vector ω of heuristic factors
Output: A function *stage*(p) which maps operators P onto stages S
Output: An overall pipeline registers size *RSize*

```
RSize ← 0
for p∈P do
    early(p) ← fasap(p)    late(p) ← falap(p)
for v∈V do
    early(v) ← |S|
    for p∈producers(v) do if early(v) > falap(p) then early(v)← falap(p)
    late(v) ← 0
    for p∈consumers(v) do if late(v) < fasap(p) then late(v) ← fasap(p)
    rslb(v) ← size(v) × pos(late(v) – early(v))
    RSize ← RSize + rslb(v)
Q ← P
while Q ≠ ∅ do
    for q∈Q do
        if early(q) = late(q) then
            stage(q) ← early(q)    Q ← Q / {q}    continue
        for s ← early(q) to late(q) do
            rslb(q, s) ← OptimizationStateUpdate(false, Q, q, s, V, S,
                         inputs, outputs, cdpred, cdsucc, ncdpred,
                         ncdsucc, stage, early, late, earlyv, latev, rslb)
    if Q = ∅ then break
    weightMax ← –1
    for q∈Q do
        if q∈f and f∈F then operBest ← q  break
        else
            ρ(q) ← heuristicsEvaluateD(Q, early, late, rslb,
                   inputs, outputs)
            χ(q) ← 0
            for k ← 1 to |ρ| do
                χ(q) ← χ(q) + ωk × ρk(q)
            if weightMax < χ(q) then
                operBest ← q    weightMax ← χ(q)
    if q∈f and f∈F then stageBest ← stagef(f) else
        rslbMin ← ∞
        for s ← early(operBest) to late(operBest) do
            if rslbMin > rslb(operBest, s) then
                stageBest ← s    rslbMin ← rslb(operBest, s)
    stage(operBest) ← stageBest
    RSize ← OptimizationStateUpdate(true, Q, operBest, stageBest, V, S,
                inputs, outputs, cdpred, cdsucc, ncdpred,
                ncdsucc, stage, early, late, earlyv, latev, rslb)
    Q ← Q \ {operBest}
return stage, RSize
```

Algorithm 3: OptimizationStateUpdate

Input: *Mode* of algorithm operation
Input: A set Q of nonscheduled operators
Input: An operator q that is being scheduled
Input: A stage s that q is assigned to
Input: A set V of variables
Input: A set S of pipeline stages
Input: An overall pipeline registers size *RSize*
Input: An array *inputs* of sets *inputs*(p)$\subseteq V$ of variables for $p \in P$
Input: An array *outputs* of sets *outputs*(p)$\subseteq V$ of variables for $p \in P$
Input: An array *cdpred* of sets *cdpred*(p)$\subseteq P$ of predecessors of $p \in P$ on conflict graph C
Input: An array *cdsucc* of sets *cdsucc*(p)$\subseteq P$ of successors of $p \in P$ on conflict graph C
Input: An array *ncdpred* of sets *ncdpred*(p)$\subseteq P$ of predecessors of $p \in P$ on nonconflict graph C_n
Input: An array *ncdsucc* of sets *ncdsucc*(p)$\subseteq P$ of successors of $p \in P$ on nonconflict graph C_n
InOut: A function *stage*(p) which maps operators of $\neg Q$ onto stages of S
InOut: An array *early* of stages *early*(p)$\in S$ for $p \in Q$
InOut: An array *late* of stages *late*(p)$\in S$ for $p \in Q$
InOut: An array *earlyv* of stages *early*(v)$\in S$ for $v \in V$
InOut: An array *latev* of stages *late*(v)$\in S$ for $v \in V$
InOut: An array *rslb* of pipeline registers sizes *rslb*(v) for $v \in V$
Output: An updated overall pipeline registers size *RSize*'

```
P_upd ← ∅    V_upd ← ∅    RSize' ← RSize    early'(q) ← late'(q) ← s
if early(q) ≠ early'(q) then
    P_upd ← {q}
    while P_upd ≠ ∅ do
        f ← getFirstOperator(P_upd)
        if Mode then
            early(f) ← early'(f)
            if early(f) = late(f) then stage(f) ← early(f)
        for p ∈ cdsucc(f) ∪ ncdsucc(f) do
            if p ∈ cdsucc(f) and early(p) < early'(f) + 1 then
                early'(p) ← early'(f) + 1
            if p ∈ ncdsucc(f) and early(p) < early'(f) then
                early'(p) ← early'(f)
            if p ∉ P_upd and early'(p) ≠ early(p) then
            InsertAscending(p, P_upd)
        for v∈input(f) do
            if late(v) < early'(f) then
                late'(v) ← early'(f)    early'(v) ← early(v)
                V_upd ← V_upd ∪ {v}
if late(q) ≠ late'(q) then
    P_upd ← {q}
    while P_upd ≠ ∅ do
        f ← getFirstOperator(P_upd)
        if Mode then
            late(f) ← late'(f)
            if early(f) = late(f) then stage(f) ← early(f)
        for p ∈ cdpred(f) ∪ ncdpred(f) do
            if p ∈ cdpred(f) and late(p) > late'(f) – 1 then
                late'(p) ← late'(f) – 1
            if p ∈ ncdpred(f) and late(p) > late'(f) then
                late'(p) ← late'(f)
            if p ∉ P_upd and late'(p) ≠ late(p) then
                InsertDescending(p, P_upd)
        for v∈output(f) do
            if early(v) > late'(f) then
                early'(v) ← late'(f)    late'(v) ← late(v)
                V_upd ← V_upd ∪ {v}
for v ∈ V_upd do
    rslb'(v) ← size(v) × pos(late'(v) – early'(v))
    RSize' ← RSize' + rslb'(v) – rslb(v)
    if Mode then
        early(v) ← early'(v)  late(v) ← late'(v)  rslb(v) ← rslb'(v)
return RSize'
```

If *early'*(*v*) is unequal to *early*(*v*), *v* is included in V_{upd}. Finally, the algorithm computes new values of *rslb'*(*v*), $v \in V_{upd}$ and *RSize*'. It returns a value of *RSize*' and updates the values of *early'*(*p*), *late'*(*p*), *early'*(*v*), *late'*(*v*) and *rslb'*(*v*) when *Mode* equals *true*.

5.7 Heuristic algorithm HASD

To speed up the optimization process, we have developed an algorithm HASD (alternative to HADD), which uses static heuristics for ordering operators before their assignment to pipeline stages and uses dynamic heuristics for choosing a stage at each step of optimization process. Algorithm 4 summarizes HASD as follows. Firstly, it initializes the optimization state. Secondly, it computes vector $\rho(p)$ of static heuristic parameters by means of calling function *heuristicsEvaluateS*, and estimates a static weight $\chi(p)$ of each operator $p \in P$. HASD sorts operators of *P* on descending of χ, and generates an *order* on *P* by means of calling function *operatorsOrdering*. It puts the operators of feedback fragments *F* in the beginning of the sorted list. Function *orderInverse* generates an inverse $order^{-1}$. Then in a loop on variable *i*, HASD iteratively chooses an operator $operBest = order^{-1}(i)$ and assigns it to *stagef*(*f*) in case the operator is in a feedback fragment $f \in F$. Otherwise, it computes a value of *rslb*(*operBest*, *s*) using function *UpdateOptimizationState*, assuming the nonscheduled operator *operBest* were allocated on an available pipeline stage *s*. The *stageBest* whose *rslb*(*operBest*, *stageBest*) is minimal is selected for the allocation of *operBest*. Function *UpdateOptimizationState* whose first parameter takes value *true* recalculates the optimization state after assignment of *operBest* to *stageBest*.

6 Genetic algorithm for tuning heuristics

The heuristic weight $\chi(p)$ determines the dynamic ordering of operators during pipeline optimization. The generated order significantly influences the overall pipeline registers size. The next operator choice depends not only on the heuristic parameters ρ, but also on the heuristic factors ω. A high value of ω_i shows importance of the corresponding parameter ρ_i in the weight $\chi(p)$. A low value of ω_i is taken when the corresponding parameter ρ_i poorly recognizes best solutions. Finding an optimal value of vector ω is a complex problem, as the pipeline registers size has many local minima in the multidimensional solution space. In this paper, we develop a genetic algorithm (GA) for efficiently solving this problem.

6.1 Basics of the genetic algorithm

An individual $\omega = (\omega_1,\ldots,\omega_k)$ is a vector of heuristic factors. A gen ω_i is a heuristic factor. A population is a set of individuals that exist during the genetic algorithm operation. A generation is a set of individuals that exist during one iteration of the genetic algorithm. A fitness function $F(\omega)$ of individual ω represents quality of the corresponding pipeline solution. In the pipeline optimization problem, this function is determined over the objective function that is a minimum of *RSize*(ω) obtained by HADD or HASD. Fitness function $F(\omega)$ is a difference between a maximum of $RSize(\omega^{worst})$ of the worst individual and *RSize*(ω) of individual ω.

Algorithm 4: HASD

Input: A set *F* of feedback fragments
Input: A function *stagef*(*f*) which maps fragments *F* onto stages *S*
Input: A set *V* of variables
Input: A set *P* of operators
Input: A set *S* of pipeline stages
Input: A set $inputs(p) \subseteq V$ of variables for operator $p \in P$
Input: A set $outputs(p) \subseteq V$ of variables for operator $p \in P$
Input: A set $producers(v) \subseteq P$ of operators for variable $v \in V$
Input: A set $consumers(v) \subseteq P$ of operators for variable $v \in V$
Input: A set $cdpred(p) \subseteq P$ of predecessors of operator $p \in P$ on conflict graph *C*
Input: A set $cdsucc(p) \subseteq P$ of successors of operator $p \in P$ on conflict graph *C*
Input: A set $ncdpred(p) \subseteq P$ of predecessors of operator $p \in P$ on nonconflict graph C_n
Input: A set $ncdsucc(p) \subseteq P$ of successors of operator $p \in P$ on nonconflict graph C_n
Input: An initially earliest stage $fasap(p) \in S$ of operator $p \in P$
Input: An initially latest stage $falap(p) \in S$ of operator $p \in P$
Input: A vector ω of heuristic factors
Output: A function *stage*(*p*) which maps operators *P* onto stages *S*
Output: An overall pipeline registers size *RSize*

```
RSize ← 0
for p∈P do   early(p) ← fasap(p)    late(p) ← falap(p)
for v∈V do
    early(v) ← |S|
    for p∈producers(v) do if early(v) > falap(p) then early(v)← falap(p)
    late(v) ← 0
    for p∈consumers(v) do if late(v) < fasap(p) then late(v) ← fasap(p)
    rslb(v) ← size(v) × pos(late(v) – early(v))
    RSize ← RSize + rslb(v)
for p ← 1 to |P| do
    ρ(p) ← heuristicsEvaluateS(fasap, falap, inputs, outputs)
    χ(p) ← 0
    for k ← 1 to |ρ| do   χ(p) ← χ(p) + ωk × ρk(p)
order ← operatorsOrdering(P, F, stagef, χ)
order^1 ← orderInverse(order)
Q ← P
for i ← 1 to |P| do
    operBest ← order^-1(i)
    if early(operBest) = late(operBest) then
        stage(operBest) ← early(operBest)
    else
        if q∈f and f∈F then  stageBest ← stagef(f) else
            rslbMin ← ∞
            for s ← early(operBest) to late(operBest) do
                rslb ← optimizationStateUpdate(false, Q, operBest, s,
                        V, inputs, outputs, cdpred, cdsucc, ncdpred,
                        ncdsucc, stage, early, late, earlyv, latev, rslb)
                if rslbMin > rslb then   stageBest ← s    rslbMin ← rslb
        stage(p) ← stageBest
        RSize ← optimizationStateUpdate(true, Q, operBest, stageBest,
                V, inputs, outputs, cdpred, cdsucc, ncdpred, ncdsucc,
                stage, early, late, earlyv, latev, rslb)
    Q ← Q / {operBest}
return stage, RSize
```

6.2 Genetic operations

The fitness proportionate selection (*FPS*) normalizes each fitness value $F(\omega)$ by dividing it by the sum of all fitness values. The sum of normalized values equals 1, and the values can be considered as probabilities. The worst parent selection

(*WPS*) chooses a parent with the worst fitness value and replaces it in the next generation with the best offspring in case the fitness value of the offspring is larger than the fitness value of a parent. The worst individual selection (*WIS*) chooses the individual with the worst fitness value in the current generation, and replaces it with the best offspring in case the fitness value of this offspring is larger than fitness value of the individual.

The half uniform crossover (*HUX*) randomly chooses a half of gen indices that are represented with a subset K^1 of set $K = 1,\ldots,k$. *HUX* is a partially matched crossover. The simple recombination of parent's gens is not sufficient for obtaining a valid offspring, as for new individual the sum of heuristic factors may appear unequal to 1. We differentiate two cases as follows for two parents.

Case 1. The fitness values of ω^1 and ω^2 are approximately equal. In this case, *HUX* tries to save the genotype of parent ω^1 in the first offspring and the genotype of parent ω^2 in the second offspring. It constructs the first offspring ω^3 of original gens of parent ω^1 that are indexed with $i \in K^1$, and of normalized gens of parent ω^2 that are indexed with $i \in K \backslash K^1$. It constructs the second offspring ω^4 of original gens of parent ω^2 that are indexed with $i \in K \backslash K^1$ and of normalized gens of parent ω^1 that are indexed with $i \in K^1$. *HUX* performs the gen normalization with the ratios as follows:

$$a = \sum_{i \in K^1} \omega_i^1 \tag{16}$$

$$b = \sum_{i \in K^1} \omega_i^2 \tag{17}$$

$$\gamma^1 = (1-a)/(1-b) \tag{18}$$

$$\gamma^2 = b/a \tag{19}$$

Ratio γ^1 aims at normalizing the gens of ω^2 in offspring ω^3: $\omega^3{}_i = \gamma^1 \times \omega^2{}_i$ for $i \in K \backslash K^1$. Ratio γ^2 aims at normalizing the gens of ω^1 in offspring ω^4: $\omega^4{}_i = \gamma^2 \times \omega^1{}_i$ for $i \in K^1$.

Case 2. The fitness value of ω^1 significantly exceeds the fitness value of ω^2, In this case, *HUX* tries to save the genotype of parent ω^1 in both offspring. The first offspring is the same as ω^3. *HUX* constructs the second offspring ω^5 of the original gens of parent ω^1 that are indexed with $i \in K \backslash K^1$, and of the normalized gens of ω^2 that are indexed with $i \in K^1$. The gen normalization is performed as $\omega^5{}_i = \omega^2{}_i / \gamma^2$ for $i \in K^1$.

The single offspring crossover (*SOX*) takes two parents, ω^1 and ω^2 and produces one individual. Firstly, it computes the heuristic factor weights α and β as:

$$\alpha = F(\omega^1)/(F(\omega^1) + F(\omega^2)) \tag{20}$$

$$\beta = 1 - \alpha \tag{21}$$

Secondly, it computes the single offspring ω as a vector of weighted sum of parent gens:

$$\omega_i = \alpha \times \omega_i^1 + \beta \times \omega_i^2 \quad \text{for } i=1\ldots k. \tag{22}$$

The two gens mutation (*TGM*) alters two heuristic factor values in one parent ω^1 from its initial state. It selects heuristic factors $\omega^1{}_i$ and $\omega^1{}_j$ randomly, and calculates corresponding factor values in the single offspring ω using a mutation factor ε whose value satisfies inequality $0<\varepsilon<1$:

$$\delta = \varepsilon \times \omega_i^1 \tag{23}$$

$$\omega_i = \omega_i^1 - \delta \tag{24}$$

$$\omega_j = \omega_j^1 + \delta \tag{25}$$

Algorithm 5 summarizes GA. It consists of an initialization stage and a loop that repeatedly updates the population by means of genetic operations in such a way as to find a pipeline schedule with a minimal overall registers size. The exit condition can be defined over the maximum number of iterations, or over a CPU time constraint. To determine the operation that will be performed next, either crossover or mutation, we use probabilities p_{cross} and p_{mut}. We can exploit GA in two modes: 1) while actually solving the optimization problem in real time, and 2) during accumulation of knowledge on the best heuristics factors. We can apply GA to both HADD and HASD.

Algorithm 5: Genetic algorithm for tuning heuristics

Produce initial population by repeatedly generating k - 1 random numbers μ_i; i = 1,…, k-1 of the range [0,1], order the numbers on ascending, and compute an individual as $\omega = (\mu_1, \mu_2 - \mu_1, \ldots, \mu_{k-1} - \mu_{k-2}, 1 - \mu_{k-1})$, and add it to the population.

Perform HADD(ω) or HASD(ω) for each individual ω, find the worst individual, compute fitness function $F(\omega)$ for all individual, and reorder the individuals on descending of $F(\omega)$.

while not *Exit_Condition* **do**

- Randomly choose crossover or mutation using probabilities p_{cross} and p_{mut}.
- Randomly choose parents using selection operation *FPS*.
- Perform crossover *HUCX* or *SOCX* and mutation *TGM* operations, and obtain offsprings ω_i; $i=1,\ldots,m$.
- Perform HADD(ω_i) or HASD(ω_i) and produce $F(\omega_i)$ for each offspring ω_i.
- Perform selection operation *WPS* or *WIS* to update population.

return HADD(ω^{best}) or HASD(ω^{best})

---

7 Results

We have developed a pipelines synthesis and optimization tool and have integrated it with the CAL HLS flow based on Xronos [45]. We have conducted experiments on several test benches, including Bayer filter, forward discrete cosine transform, inverse discrete cosine transform, random TB1000-TB5000 benchmarks [7 - 9] and others. We report results obtained on Intel® Core™ i3 CPU 550 @ 3.20 GHz 3.19 GHz, 4 GB.

7.1 Results for FHADD

In the design TB1000 of 1000 operators, we use relative delays of operators. The maximum delay equals 3.0 and the sum of all delays equals 906. The delay over the critical path in the data flow graph equals 89.7. Table 1 and Figure 10

report results for FHADD and for various number of stages. Increasing in the stage number from 2 to 14 implies decreasing in the stage-time period from 45.43 to 7.33. Multiplying the number of stages by the stage-time period yields the all-stages-time (columns 1-3) that fluctuates from 90.87 to 109.21. The non-ideal packing of neighbor operators in one stage explains these fluctuations.

Table 1
Results for FHADD on TB1000, from 2 to 14 pipeline stages

Stages	Stage time period	All stages time	Registers size, bit	Run time, s	Size of list P_{upd}	Size of list V_{upd}
2	45.43	90.87	477	0.004	1.46	1.32
3	30.71	92.14	854	0.162	1.04	0.97
4	23.78	95.14	1288	0.701	1.10	1.04
5	19.45	97.27	1861	0.952	1.07	1.01
6	15.99	95.94	2329	1.261	1.16	1.09
7	14.26	99.81	2550	1.818	1.21	1.16
8	12.53	100.21	3090	3.498	1.31	1.27
9	11.66	104.94	3595	12.337	1.87	1.86
10	10.79	107.94	4095	10.976	1.96	1.95
11	9.93	109.21	4474	6.866	1.70	1.70
12	9.06	108.74	5081	3.486	1.56	1.55
13	8.20	106.55	5537	9.248	1.87	1.88
14	7.33	102.62	6328	3.161	1.55	1.55
				On average:	1.45	1.41

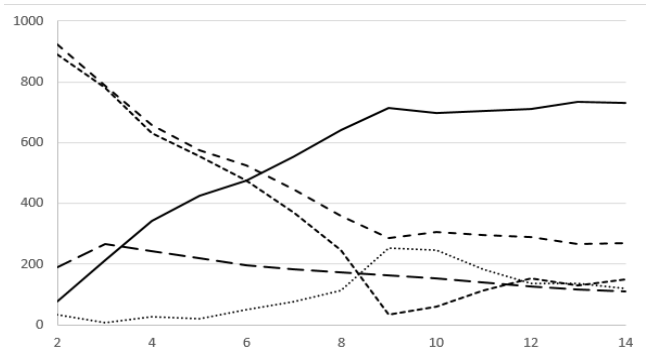

Figure 10 Number of iterations in FHADD (solid), total number of mobility-one-operators (dash), number of static mobility-one-operators (square dot), number of dynamic mobility-one-operators (round dot), and average number of operator's predecessors/successors (wide dash) vs. stage count.

While increasing the number of stages from 2 to 14, the overall pipeline registers size grows from 477 to 6328 bit near linearly. The FHADD runtime grows rapidly for the stages count until 9. Then it fluctuates. We can explain such behavior in the way as follows. Firstly, the FHADD runtime directly depends on the number of iterations in its main loop (Figure 10, solid). This number grows from 76 to 714, and then it wavers near 700. Secondly, the number of iterations is larger when the number of scheduled mobility-one-operators is smaller (Figure 10, dash). There are two sources for such mobility: 1) the equality of the stages that ASAP and ALAP have assigned to the operator (static mobility-one); 2) the reduction of the operator mobility to one due to the assignment of other operators to stages (dynamic mobility-one). While the stage count increases from 2 to 9, the number of static mobility-one-operators falls in TB1000 from 890 down to 32 (square dot), and the number of dynamic mobility-one-operators remains small but increases (round dot). Consequently, the FHADD runtime grows from 0.004 *s* to 12.337 *s*. Thirdly, after 9 stages the number of static mobility-one-operators begins to grow due to the stage time period approaches to the maximal operator delay. In this case, it is difficult to obtain ideal packing of neighbor operators in one stage. The number of dynamic mobility-one-operators is close to that of the static ones. Both do not reduce the number of iterations, therefore the FHADD runtime remains high. Fourthly, the FHADD runtime significantly depends on the runtime of *OptimizationStateUpdate*. In its turn, this runtime crucially depends on the size of lists/sets P_{upd}, V_{upd}, *cdsucc*, *cdpred*, *ncdsucc* and *ncdpred*. According to Table 1, lists P_{upd} and V_{upd} include a very small number of operators and variables, the maximum runtime occurs when P_{upd}, V_{upd} have the maximal size. Fifthly, the heuristic factors that the genetic algorithm finds can significantly vary the overall registers size and vary the number of dynamic mobility-one-operators. The factors that produce the lowest registers size often generate the smaller number of dynamic mobility-one-operators, thereby increasing the FHADD runtime.

Figure 10, wide dash shows the size of *cdsucc* and *cdpred* vs. stage count. The size of 267 is maximal for 3 stages, and then it monotonically falls to 111 at 14 stages. The total size of sets *ncdsucc* and *ncdpred* is small and equals 3.63 on average.

Table 2 shows another source of the reducing the FHADD runtime, i.e. the transition from the operators' conflict relation to its minimal anti-transitive analogue. The analogue reduces the number of operator pairs by 36.69% – 42.74% and decreases the runtime by 21.2% – 29.3%.

Table 3 compares the new FHADD algorithm against the best-known HT technique [8] on five benchmarks of different size. FHADD's gain is 10.63% over HT regarding the quality of pipelines and its gain is 9.5x on average regarding the runtime.

Table 2
Influence of anti-transitivity on FHADD runtime for 4-stage pipelines

Parameter	Test bench size				
	1000	2000	3000	4000	5000
FHADD runtime (sec)	0.94	2.59	8.50	13.43	10.48
Reduction of conflict relation size (%)	42.74	36.69	38.61	38.40	42.51
FHADD runtime after relation size reduction (sec)	0.69	2.04	6.33	9.50	7.62
Decrease in FHADD runtime (%)	26.6	21.2	25.5	29.3	27.3

Table 3
Results for FHADD and HT on TB1000-TB5000, 4 stages

Design Size	Pipeline registers size			CPU time		
	FHADD (bit)	HT (bit)	%	FHADD (sec)	HT (sec)	times
1000	1288	1469	14.05	0.687	12.00	17.5
2000	1529	1712	11.97	2.044	8.00	3.9
3000	2016	2192	8.73	6.326	29.00	4.6
4000	2735	3067	12.14	9.499	62.00	6.5
5000	3123	3318	6.24	7.615	115.00	15.1
		On average	10.63			9.5

7.2 Results for FHASD

Table 4 reports results obtained for FHASD on TB1000 at various stage count. *RSize* grows from 481 to 6393 near linearly with increasing the number of stages from 2 to 14. For each number, FHASD gives *RSize* higher to FHADD. The gain of FHADD is 6.07% on average. At the same time, FHASD is much faster to FHADD, with the gain of 345.3 times on average.

Table 4
Results for FHASD and its comparison against FHADD on TB1000

Stages	Overall registers size (bit)	CPU time (millisecond)	Registers size FHASD over FHADD (%)	CPU time FHADD over FHASD (times)
2	481	1.0	0.84	4.0
3	916	2.6	7.26	62.3
4	1482	4.6	15.06	152.4
5	2080	5.1	11.77	186.7
6	2526	6.3	8.46	200.2
7	2768	7.6	8.55	239.2
8	3396	10.0	9.90	349.8
9	3747	16.6	4.23	743.2
10	4099	13.8	0.10	795.4
11	4758	13.9	6.35	494.0
12	5269	10.4	3.70	335.2
13	5629	15.0	1.66	616.5
14	6393	10.2	1.03	309.9
		On average	6.07	345.3

7.3 Results for genetic algorithm

Our numerous experiments prove that the heuristic algorithm FHADD gives about 70 % of the reduction regarding a minimum of the overall pipelines registers size, and the genetic algorithm that tunes the heuristic factors in FHADD gives the rest 30 % of the reduction.

GA provides generating the best heuristic factor values for various designs and various number of pipeline stages. Figure 11 shows cumulative distribution probability functions (CDFs) generated on the best heuristic factors that result from numerous optimization runs on designs TB1000-TB5000. The best average values of heuristic factors are as follows: ω_1=0.292 (mobility of operators), ω_2=0.466 (*rslb* over stages), ω_3=0.056 (*rslb* over operators) and ω_4=0.186 (difference between inputs size and outputs size of operator). Each factor takes value in a restricted interval. Thus, ω_2 should be between 0.15 and 0.9, ω_1 should be between 0.0 and 0.6, ω_4 should be between 0.0 and 0.55 and ω_3 should be between 0.0 and 0.25. CDFs are an efficient facility for the generation of initial

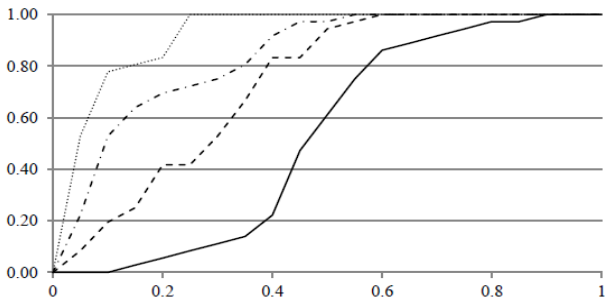

Figure 11 Cumulative probability distribution functions for best heuristic factors: ω_1 (dash), ω_2 (solid), ω_3 (round dot), ω_4 (dash dot).

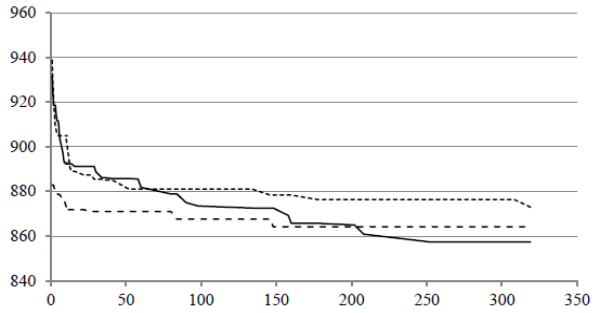

Figure 12 Overall registers size (bit) of 3-stage pipeline optimized by crossover HUX (solid), SOX (round dot) and CDF-HUX (dash) vs. population size (average on 5 run of TB1000 for each crossover).

population in GA. They randomly produce quick solutions in case FHADD can perform few runs in acceptable CPU time.

Figure 12 compares two crossovers *HUX* and *SOX* with respect to high performance of GA. Starting conditions for SOX (884.8) appear preferable over starting conditions for HUX (885.8). But very quickly HUX began to give the registers size much lower than SOX and the difference had been increased. Therefore, SOX is preferable on a restricted population size, and HUX is preferable when GA can generate many individuals in the population. Figure 12 also shows that the replacement of the uniform probability distribution with CDFs significantly speeds up the reduction of registers size in a low-size population but can give a worse result in a high-size population.

7.4 Tuning the pipeline optimization tool

Figure 13 summarizes the preferable usage regions of four configurations of pipeline optimization algorithms: FLCSBB, GA-on-FHADD, CDFs-on-FHADD and FHASD. We have constructed the regions at the requirement that the runtime of the configurations except FLCSBB does not exceed 120 seconds on Intel® Core™ i3 CPU. The ability of optimizing large pipelines with many stages grows from FLCSBB to FHASD, which induces the increase in the algorithm's inaccuracy. We have measured the algorithm inaccuracy by conducting computational experiments and by the comparison of neighbor configurations on several test-benches, whose size is suitable for the left configuration in the pair (Figure 13).

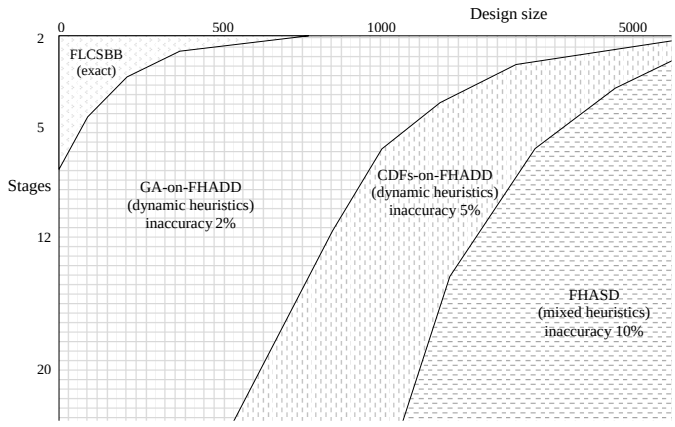

Figure 13 Preferable regions of pipeline optimization algorithms in "design size / pipeline stages" solution space.

Thus, we compared FLCSBB and GA-on-FHADD on designs of up to 100 instructions. The inaccuracy of GA-on-FHADD regarding the registers size is about 2 % on average against the optimal values given by FLCSBB. The inaccuracy of CDFs-on-FHADD is about 3% against GA-on-FHADD and therefore 5 % on average against FLCSBB on designs of up to 1000 statements. The inaccuracy of FHASD is about 8% against GA-on-FHADD and therefore about 10 % on average against FLCSBB on designs of up to 5000 statements and larger.

8 Conclusion

This paper proposes algorithms for transforming, analysis, speculatively pipelining and optimizing large branched feedback dataflow CAL-programs, which represent loop-like behavior. The exhaustive algorithm that optimally assigns feedbacks and instructions to pipeline stages is very slow; therefore, the paper presents fast dynamic and mixed static / dynamic heuristics, and the genetic approach to speed up the optimization process and to produce solutions that are close to optimal ones. We summarize our findings as follows:

1. The proposed techniques of CAL-actor transformation and analysis allow us to perform speculative pipelining and optimizing of branched feedback dataflow programs.
2. Pipelining of time-consuming CAL-actors that lie on critical paths of a dataflow program reduces the clock time-period and increases the throughput of the concurrent system implementations.
3. The extended pipelining algorithms FASAP, FALAP and FLCSBB perform speculative optimization of loop-like branched dataflow programs with feedbacks.
4. Our dynamic heuristics overcome mixed static / dynamic heuristics regarding pipeline quality, but the latter is orders of magnitude faster.
5. Our genetic algorithm performs tuning of heuristic factors to obtain higher quality solutions.
6. The proposed set of algorithm configurations can handle both a small dataflow design with a small number of pipeline stages resulting in optimal solution, and a large design with many stages resulting in increased inaccuracy against optimal solutions.

Overall, we conclude that the developed algorithms and tool show that the pipelining technology can be extended for dataflow programs and successfully used in existing CAL-based design flows.

References

[1] N. Park and A. C. Parker, “Sehwa: A software package for synthesis of pipelines from behavioral specifications,” IEEE Trans. on CAD of ICs, vol. 7, pp. 356–370, March 1988.

[2] K. S. Hwang, A. E. Casavant, C.-T. Chang, and M. A. d’Abreu, “Scheduling and hardware sharing in pipelined data paths,” in Proc. ICCAD-89, November 1989, pp. 24–27.

[3] E. M. Girczyc, “Loop winding – a data flow approach to functional pipelining,” in Proc. of the IEEE ISCAS, May 1987, pp. 382–385.

[4] C.-T. Hwang, Y.-C. Hsu, and Y.-L. Lin, “Pls: A scheduler for pipeline synthesis,” Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 12, no. 9, pp. 1279–1286, September 1993.

[5] H.-S. Jun and S.-Y. Hwang, “Design of a pipelined datapath synthesis system for digital signal processing,” , IEEE Trans. VLSI Syst., vol. 2, no. 3, pp. 292-303. Sep 1994.

[6] S. Bakshi and D. D. Gajski, “Component selection for high-performance Pipelines,” IEEE Trans. VLSI Syst., vol. 4, no. 2, pp. 181-194, June 1996.

[7] Ab Rahman, A.A., Prihozhy, A. & Mattavelli, M. Pipeline synthesis and optimization of FPGA-based video processing applications with CAL. *J Image Video Proc.* **2011,** 19 (2011). https://doi.org/10.1186/1687-5281-2011-19

[8] A. Prihozhy, E. Bezati, A.-H. Ab Rahman, M. Mattavelli. “Synthesis and Optimization of Pipelines for HW Implementations of Dataflow Programs,” IEEE Transactions on CAD, vol. 34, no. 10, pp. 1613–1626, 2015.

[9] A. Prihozhy, S. Casale-Brunet, E. Bezati and M. Mattavelli. “Efficient Dynamic Optimization Heuristics for Dataflow Pipelines,” IEEE International Workshop on Signal Processing Systems, IEEE, pp. 337-342, October 2018.

[10] J. Eker and J. Janneck, CAL Language Report: Specification of the CAL Actor Language. University of California-Berkeley, December 2003.

[11] M. Mattavelli, I. Amer, M. Raulet, “The Reconfigurable Video Coding Standard“ [Standards in a Nutshell], Signal Processing Magazine, IEEE 27 (3) (2010) 159 –167.

[12] Z. Zhang, B. Liu. “SDC-Based Modulo Scheduling for Pipeline Synthesis,” IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 211-218, November 2013.

[13] M. Weinhardt and W. Luk, “Pipeline vectorization,” Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 20, no. 2, pp. 234–248, Feb. 2001.

[14] G. Demicheli, “Hardware synthesis from C/C++ models,” in Design, Automation and Test in Europe Conference and Exhibition 1999, pp. 382–383.

[15] A. Prihozhy. “High-level synthesis through transforming VHDL models,” System-on-Chip Methodologies & Design Languages, Kluwer Academic Publishers, Springer, Boston, MA, 2001, pp. 135-146.

[16] L.-F. Chao, A. LaPaugh, and E.-M. Sha, “Rotation scheduling: a loop pipelining algorithm,” Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 16, no. 3, pp. 229–239, Mar 1997.

[17] W. F. J. Verhaegh, P. E. R. Lippens, E. H. L. Aarts, J. H. M. Korst, J. Van Meerbergen, and A. van der Werf, “Improved force-directed scheduling in high-throughput digital signal processing,” Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 14, no. 8, pp. 945–960, Aug 1995.

[18] E. Nurvitadhi, J. Hoe, T. Kam, and S. Lu, “Automatic pipelining from transactional datapath specifications,” Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 30, no. 3, pp. 441–454, March 2011.

[19] S. Oh, T. G. Kim, J. Cho, and E. Bozorgzadeh, “Speculative loop pipelining in binary translation for hardware acceleration,” Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 27, no. 3, pp. 409–422, March 2008.

[20] J. Serot, F. Berry and S. Ahmed, "Implementing Stream-Processing Applications on FPGAs: A DSL-Based Approach," *2011 21st International Conference on Field Programmable Logic and Applications*, Chania, 2011, pp. 130-137.

[21] P. G. Paulin and J. P. Knight, “Force-directed scheduling for the behavioral synthesis of ASIC’s,” IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 8, No. 6, 1989, pp. 661–679.

[22] N. Shenoy, “Retiming: Theory and practice,” VLSI Journal Integr., vol. 22, no. 1-2, pp. 1–21, 1997.

[23] R. Potasman, J. Lis, A. Aiken, and A. Nicolau, “Percolation based synthesis,” in Proc. 27th Design Automation Conf., 1990, pp. 444–449.

[24] H. Javaid, A. Ignjatovic, and S. Parameswaran, “Rapid design space exploration of application specific heterogeneous pipelined multiprocessor systems,” Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 29, no. 11, pp. 1777–1789, November 2010.

[25] B. R. Rau and C. D. Glaeser. “Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing,” ACM SIGMICRO Newsletter, 12(4):183–198, 1981.

[26] J. Codina, J. Llosa, and A. Gonz´alez. “A Comparative Study of Modulo Scheduling Techniques,” Int’l Conf. on Supercomputing, pp. 97–106, June 2002.

[27] Y. Ben-Asher, D. Meisler, and N. Rotem. Reducing Memory Constraints in Modulo Scheduling Synthesis for FPGAs. ACM Trans. on Reconfigurable Technology and Systems, 3(3), 2010.
[28] B. R. Rau. “Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops,” Int’l Symp. on Microarchitecture, pp. 63–74, November 1994.
[29] W. Sun, M. Wirthlin, and S. Neuendorffer, “FPGA pipeline synthesis design exploration using module selection and resource sharing,” Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 26, no. 2, pp. 254–265, Feb 2007.
[30] J. Llosa, E. Ayguad´e, A. Gonzalez, M. Valero, and J. Eckhardt. “Lifetime-Sensitive Modulo Scheduling in a Production Environment,” IEEE Trans. on Computers, 50(3), March 2001.
[31] R. A. Huff. “Lifetime-Sensitive Modulo Scheduling,” ACM SIGPLAN Conf. on Programming Languages Design and Implementation, pp. 258–267, June 1993.
[32] C. Hewitt. “Viewing control structures as patterns of passing messages. Journal of Artificial Intelligence,” 8(3):323{363, June 1977.
[33] M. Wipliez, G. Roquier, and J. Nezan, “Software Code Generation for the RVC-CAL Language,” Journal of Signal Processing Systems, 63(2),·May 2011, pp. 1–9.
[34] E. Bezati, S. Casale-Brunet, M. Mattavelli, and J. Janneck, “Synthesis and optimization of high-level stream programs,” in The 2013 Electronic System Level Synthesis Conference, 2013, pp. 1–6.
[35] M. Mattavelli, S. Casale-Brunet, A. Elguindy, E. Bezati, R. Thavot, G. Roquier, and J. Janneck, “Methods to explore design space for MPEG RVC codec specifications,” Signal processing Image Communication, Elsevier, 2013.
[36] E. Bezati, S. Casale-Brunet, M. Mattavelli, J. W. Janneck: Clock-Gating of Streaming Applications for Energy Efficient Implementations on FPGAs. In: IEEE Trans. on CAD of Integrated Circuits and Systems 36(4): 699-703, 2017.
[37] Palumbo, F., Sau, C., Fanni, T., Meloni, P., Raffo, L.: Dataflow-based design of coarse-grained reconfigurable platforms. In: 2016 IEEE International Workshop on Signal Processing Systems, SiPS 2016.
[38] Sau, C., Fanni, L., Meloni, P., Raffo, L., Palumbo, F.: Reconfigurable coprocessors synthesis in the MPEG-RVC domain. In:2015 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2015.
[39] Ren, R., Juarez, E., Sanz, C., Raulet, M., Pescador, F.: Energy-aware decoder management: a case study on RVC-CAL specification based on just-in-time adaptive decoder engine. IEEE Trans. Consumer Electronics 60(3), 499-507, 2014.
[40] Palumbo, F., Sau, C., Raffo, L.: DSE and profiling of multi-context coarse-grained reconfigurable systems. In: 8th International Symposium on Image and Signal Processing and Analysis, ISPA 2013.
[41] Gorin, J., Yviquel, H., Preteux, F.J., Raulet, M.: Just-in-time adaptive decoder engine: a universal video decoder based on MPEG RVC. In: Conference on Multimedia, 2011.
[42] Beaumin, C., Sentieys, O., Casseau, E., Carer, A.: A coarse-grain reconfigurable hardware architecture for rvc-cal-based design. In: Design and Architectures for Signal and Image Processing, DASIP 2010.
[43] Amer, I., Lucarz, C., Mattavelli, M., Raulet, M., Nezan, J., Deforges, O.: Reconfigurable video coding on multicore: an overview of its main objectives. In: IEEE signal Processing Magazine, 26(6), 113-123, 2009.
[44] Roquier, G., Wipliez, M., Raulet, M., Janneck, J.W., Miller, I.D., Parlour, D.B.: Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study. In: IEEE Workshop on Signal Processing Systems, SiPS 2008.
[45] M. Canale, S. Casale-Brunet, E. Bezati, M. Mattavelli, J. Janneck: "Dataflow Programs Analysis and Optimization Using Model Predictive Control Techniques", Journal of Signal Processing Systems, 2016, Vol: 84, No. 3, Pages 371—381.

Table of acronyms

No	Acronym	Full name
1	CAL	Concurrent algorithmic language
2	ASAP	As soon as possible pipeline scheduling algorithm
3	ALAP	As late as possible pipeline scheduling algorithm
4	P	A set of operators (statements, instructions).
5	V	A set of variables
6	$inputs(p)$	A set of input variables of operator p
7	$outputs(p)$	A set of output variables of operator p
8	$size(v)$	A bit-size of variable v
9	$prod(v)$	Operators-producers of variable v
10	$cons(v)$	Operators-consumers of variable v
11	T	A set of conditional Boolean variables
12	Z	A set of primary Boolean variables
13	H	A set of Boolean functions that evaluate the conditional variables over the primary variables
14	F	A set of feasible Boolean functions for values of pairs of primary variables
15	λ	A Boolean function that characterizes the feasible set of vector values of primary variables
16	μ	A Boolean function that takes value 1 when two conditional variables are orthogonal
17	R_{direct}	Operators direct precedence relation
18	R	Transitive closure of R_{direct}
19	$succ(p)$	Operators-successors of operator p
20	$pred(p)$	Operators-predecessors of operator p
21	G	A matrix of all operator pairs longest paths lengths
22	$Fbr(s)$	A subset of operators in a feedback region of variable s
23	S	A set of pipeline stages
24	$stage(p)$	An assignment of operator p to a stage
25	T_{stage}	A constraint on the pipeline-stage time-period

26	C	An operator conflict relation (graph)
27	C_n	An operator nonconflict relation (graph)
28	*cdpred*(p)	A set of direct predecessors of operator p on graph C
29	*ncdpred*(p)	A set of direct predecessors of operator p on graph C_n
30	*cdsucc* (p)	A set of direct successors of operator p on graph C
31	*ncdsucc* (p)	A set of direct successors of operator p on graph C_n
32	*asap*	A pipeline schedule that ASAP algorithm generates on conflict graph C
33	*alap*	A pipeline schedule that ALAP algorithm generates on conflict graph C
34	*Rsize*(*stage*)	An overall pipeline registers size of the schedule described by the operators vector assignment *stage*
35	*lifetime*(v)	Lifetime of variable v over pipeline stages
36	FASAP	Extension of ASAP for branched feedback programs
37	FALAP	Extension of ALAP for branched feedback programs
38	*fasap*	A pipeline schedule that FASAP algorithm generates on conflict graph C
39	*falap*	A pipeline schedule that FALAP algorithm generates on conflict graph C
40	LCSBB	Least cost search branch and bound algorithm of pipeline optimization
41	FLCSBB	Extension of LCSBB for branched feedback dataflow programs
42	*flcsbb*	A pipeline schedule that FLCSBB algorithm generates on conflict graph C
43	FHADD	Feedback dataflow optimization Heuristic Algorithm using dynamic heuristics for operators and stages
44	FHASD	Feedback dataflow optimization Heuristic Algorithm using static heuristics for operators and dynamic heuristics for stages
45	*early*(p)	A lower bound of a range of available stages for operator p
46	*late*(p)	A upper bound of a range of available stages for operator p
47	*lifestim*(v)	A lower bound of lifetime of variable v over pipeline stages
48	*early*(v)	An upper bound of the earliest stage of producers of variable v
49	*late*(v)	A lower bound of the latest stage of consumers of variable v
50	*pos*(x)	A function whose value equals 0 if $x \leq 0$, and equals x otherwise
51	$\chi(p)$	A heuristic weight of operator p whose maximal value indicates that p is preferable for scheduling
52	ω	A vector of heuristic factors
53	$\rho(p)$	A vector of heuristic parameters of operator p
54	GA	Genetic algorithm for tuning heuristics
55	$F(\omega)$	A fitness function of individual ω that represents quality of the corresponding pipeline solution
56	*FPS*	A fitness proportionate selection operation
57	*WPS*	A worst parent selection operation
58	*WIS*	A worst individual selection in the current generation
59	*HUX*	A half uniform crossover operation
60	*SOX*	A single offspring crossover operation
61	α, β	Heuristic factor weights
62	ε	A mutation factor
63	p_{cross}	Probability of choosing the crossover
64	p_{mut}	Probability of choosing the mutation

Anatoly Prihozhy received his Diploma of Electrical Engineering from the State Polytechnic, Minsk, Belarus in 1975, his PhD degree in computer-aided design from the National Academy of Sciences Minsk, Belarus in 1984, and his Doctor Habilitation degree in computer sciences from Ukraine, Kyiv and Belarus, Minsk in 1999. He was Visiting Professor at the Swiss Federal Institute of Technology, Lausanne, Switzerland in 2001, 2004, 2010 and 2013, 2015 and 2016 and at the Freiburg University, Germany in 2000. He is currently full professor at Computer and System Software Department of the Belarusian National Technical University. He has several books and more than 250 publications in Eastern and Western Europe, USA and Canada His research interests include programming, hardware and system description languages, compilers and tools, system-, high- and logic-level computer aided design and optimization of parallel and incompletely specified digital systems.

Simone Casale-Brunet is a software/hardware scientist in the Vital-IT group of the Swiss Institute of Bioinformatics (SIB). He received a B.S. degree in Electrical Engineering and an M.S. degree in Mechatronics Engineering, both with honors, from the Politecnico di Torino, Italy, in 2008 and 2010, respectively, and the Ph.D. degree in Electrical Engineering from the École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, in 2015. His research interests include FPGA-accelerated systems for bioinformatics algorithms and genomic data compression algorithms.

Endri Bezati is a software/hardware scientist in the VLSC laboratory of the Swiss Federal Institute of Technology (EPFL). He received his master's degree in electrical engineering and computer science from Institut National des Sciences Appliquées of Rennes (INSA) in 2010, France and his Ph.D. Degree in Microsystems and Microelectronics at EPFL in 2015. He is a member of the Institute of Electrical and Electronics Engineers (IEEE), and of European Network on High Performance and Embedded Architecture and Compilation (HIPEAC). His research interests include high-level synthesis of dataflow programs, hardware-software co-design and massively parallel processing for bioinformatics algorithms.

Marco Mattavelli started his research activity at the "Philips Research Laboratories" of Eindhoven in the framework of EUREKA-95 HDMAC project. Since 1991 he joined EPFL where he got his PhD in 1996. Since then, he has been involved in several research projects and didactic activities. He has been Chairman of the Implementation Study Group of ISO/IEC MPEG for more than 10 years. For his work he received the ISO/IEC Award in 1997, 2003, 2011, 2013 and 2015. He is at the origin of the introduction of dataflow programs as specifications for MPEG video compression and graphic standards, ISO/IEC 23001-4 and 23002-4. Since 2006 he is leading the "Multimedia Architectures Research Group" of EPFL. His major research activities include: signal processing system implementations, methodologies for high level specification and modeling of complex systems, architectures and systems for video coding, high speed image acquisition and video processing, applications of combinatorial optimization to signal processing. He holds patents in the multimedia and video processing fields. He is the author of more than 200 publications and has served as invited editor for several conferences and associated editor to IEEE publications. Since 2014 he is leading the MPEG-G ISO/IEC 23092 standardization project applying digital media technologies to the emerging field of genomic data compression and processing.