Optimizing Affine Control with Semantic Factorizations

CHRISTOPHE ALIAS*, CNRS/ENS-Lyon/Inria/UCBL/Université de Lyon, France
ALEXANDRU PLESCO, XtremLogic SAS, France

Hardware accelerators generated by polyhedral synthesis techniques make an extensive use of affine expressions (affine functions and convex polyhedra) in control and steering logic. Since the control is pipelined, these affine objects must be evaluated at the same time for different values, which forbids aggressive reuse of operators. In this paper, we propose a method to factorize a collection of affine expressions without preventing pipelining. Our key contributions are (i) to use semantic factorizations exploiting arithmetic properties of addition and multiplication and (ii) to rely on a cost function whose minimization ensures a correct usage of FPGA resources. Our algorithm is totally parametrized by the cost function, which can be customized to fit a target FPGA. Experimental results on a large pool of linear algebra kernels show a significant improvement compared to traditional low-level RTL optimizations. In particular, we show how our method reduces resource consumption by revealing hidden strength reductions.

CCS Concepts: • **Hardware → High-level and register-transfer level synthesis**; • **Computer systems organization** → *Reconfigurable computing*;

Additional Key Words and Phrases: High-level synthesis, FPGA, Polyhedral Synthesis

ACM Reference Format:
Christophe Alias and Alexandru Plesco. 2017. Optimizing Affine Control with Semantic Factorizations. *ACM Transactions on Architecture and Code Optimization* 1, 1 (December 2017), 22 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Since the end of Dennard scaling, computer architects are striving to build energy efficient computers. The trend is to trade generality for energy efficiency by using specialized hardware accelerators such as GP-GPU or Xeon-Phi [18] to quote a few. Recently, reconfigurable FPGA circuits [10] have appear to be a competitive alternative [35]. With FPGAs, the program is the circuit: genericity is ultimately traded for energy efficiency. However, designing a circuit is far more complex than writing a C program. Disruptive compiler technologies are required to generate automatically a circuit configuration from an algorithmic description, while finding an appropriate trade-off between parallelism and I/O bandwidth. Polyhedral compilation techniques have a long term history of success in automatic parallelization for HPC [17]. Roughly, loop iterations are represented with polyhedra (hence the name), then code optimizations are specified with geometric operations and integer linear programming. Polyhedral analysis enables reasoning about massively parallel computations with a compact representation. Powerful analysis were designed for extracting parallelism [11], scheduling pipelined circuits [3], resizing optimally the buffers [1] or tuning I/O requirements to fit

*The corresponding author

Authors' addresses: Christophe Alias, Christophe.Alias@ens-lyon.fr, CNRS/ENS-Lyon/Inria/UCBL/Université de Lyon, 46, Allée d'Italie, Lyon, 69364, France; Alexandru Plesco, XtremLogic SAS, Lyon, France.

XXXX-XXXX/2017/12-ART $15.00
https://doi.org/0000001.0000001

memory bandwidth [2, 27] to quote a few. Polyhedral analysis are used successfully in high-level circuit synthesis [4, 31]. The result is a high-level description of the circuit whose control logic involves a large collection of piecewise affine (PWA) functions. Minimizing the resource usage of affine control while guaranteeing the throughput is a major challenge in polyhedral synthesis.

Pretty few approaches address low-level affine control synthesis in the context of polyhedral circuit synthesis. Actually, most of the research effort in the polyhedral model has focused on source-level transformations. For instance, Alias et al. [2] propose a source-level approach at C level before high-level synthesis to produce an optimized I/O system for a circuit. Zuo et al. [38] optimize the control structure at source-level on a C program before using VivadoHLS. In this paper, we will not follow the same guidelines. Rather, we show how, for a given control, and without any attempt to optimize its structure, we can produce a dedicated hardware machinery which outperforms by 30% the generic optimizations applied on RTL by a state-of-the-art synthesis tool. To do so, we rely on semantic factorizations, a generalization of common subexpression elimination, which proves to be particularly effective on affine control. Semantic factorizations take profit of associativity and commutativity of addition and multiplication to simplify the control. Note that semantic transformations are not new in the polyhedral model. To quote a few, semantic properties of operators are already exploited to recognize algorithms [22] or to extract instruction patterns at source-level [37]. As far as we know, this is the first time that semantic factorizations are used to optimize affine control, while keeping it exact. However, when approximation is allowed, the complexity of the controller can be reduced [19, 23] and control algorithms can be simplified [6, 29]. In our case, this would not apply: control has to be exact, no approximation is allowed.

In this paper, we propose a technique to compact a collection of affine objects (affine expressions and affine constraints) by exploiting semantic properties of addition and multiplication. More specifically:

- The compaction is driven by a cost function whose minimization ensures a proper usage of FPGA resources. The cost can be customized to target a given FPGA.
- The result is a DAG pipelinable at will and ready to be mapped on the FPGA, whose resource usage minimize the cost function.
- Experimental results show that our algorithm outperforms significantly the low-level optimizations applied on RTL by a state-of-the-art synthesis tool.

This paper is structured as follows. Section 2 gives a short introduction to polyhedral synthesis and introduces the concepts used in the remaining of the paper. Section 3 presents our compaction algorithm. Section 4 presents the experimental results. Section 5 reviews the related work. Finally, Section 6 concludes this paper and draws perspectives.

2 PRELIMINARIES

This section presents the basic math concepts required to understand the notion of affine control (convex polyhedra, piecewise affine functions). Then, polyhedral control synthesis is briefly introduced. In the remaining of this section, n, p and q are positive natural integers: $n, p, q \in \mathbb{N} - \{0\}$.

2.1 Convex polyhedra

Given a linear form $a^* : \mathbb{R}^n \to \mathbb{R}$ and a scalar $\alpha \in \mathbb{R}$, the set $H_{\geq}(a^*, \alpha) = \{x \in \mathbb{R}^n,\ a^*(x) \geq \alpha\}$ is said to be a *closed half-space*. A *convex polyhedron* $\mathcal{P}$ is a finite intersection of closed half spaces: $\mathcal{P} = \cap_{i=1}^{q} H_{\geq}(a_i^*, \alpha_i)$. If $a_i^*(x) = \tau_i \cdot x$, $\mathbf{A}$ is the matrix whose rows are $\tau_1, \ldots, \tau_q$ and $\mathbf{b} = (\alpha_1, \ldots, \alpha_q)^T$, the *matrix representation* of $\mathcal{P}$ is:

$$\mathcal{P} = \{x \in \mathbb{R}^n, \mathbf{A}x \geq \mathbf{b}\}$$

The *interior of* $\mathcal{P}$ is the biggest open set int $\mathcal{P}$ included in $\mathcal{P}$. With the matrix representation, int $\mathcal{P} = \{x \in \mathbb{R}^n, \mathbf{A}x > \mathbf{b}\}$.

An *integer polyhedron* is the set of integral points lying in a convex polyhedron $\mathcal{P}$, $\hat{\mathcal{P}} = \mathcal{P} \cap \mathbb{Z}^n$. More generally, a $\mathbb{Z}$-polyhedron is the set of points from a lattice $\mathcal{L} \subset \mathbb{Z}^n$ lying in a convex polyhedron $\mathcal{P}$, $\mathcal{L} \cap \mathcal{P}$. In polyhedral synthesis, we often use integer polyhedra and sometimes $\mathbb{Z}$-polyhedra.

2.2 Piecewise affine functions

Given $\mathcal{D} \subset \mathbb{R}^n$, a mapping $\phi : \mathcal{D} \to \mathbb{R}^p$ is said to be *piecewise affine* if there exists a subdivision of $\mathcal{D}$ in convex polyhedra $\mathcal{D} = \mathcal{P}_1 \cup \ldots \cup \mathcal{P}_q$ such that int $\mathcal{P}_i \cap$ int $\mathcal{P}_j = \emptyset$ for $i \neq j$ and a collection of affine mappings $u_i : \mathbb{R}^n \to \mathbb{R}^p$ for $i = 1, \ldots, q$ such that:

$$\phi(x) = u_i(x) \quad \text{if } x \in \mathcal{P}_i \quad \text{for } i = 1, \ldots, q$$

Since the pieces are closed, a piecewise affine mapping ϕ is always continuous. Indeed, ϕ should share the same value on the common facets of two adjacent convex polyhedra.

An *integer piecewise affine mapping* $\hat{\phi} : \hat{\mathcal{D}} \to \mathbb{R}^p$ is defined over a partition of $\hat{\mathcal{D}}$ into integer polyhedra: $\hat{\mathcal{D}} = \hat{\mathcal{P}}_1 \uplus \ldots \uplus \hat{\mathcal{P}}_q$, each piece being provided with an affine mapping $u_i : \mathbb{R}^n \to \mathbb{R}^p$ for $i = 1, \ldots, q$:

$$\hat{\phi}(x) = u_i(x) \quad \text{if } x \in \hat{\mathcal{P}}_i \quad \text{for } i = 1, \ldots, q$$

Remark that an integer piecewise affine mapping $\hat{\phi}$ is not necessarily continuous. Some results on piecewise affine mappings, for instance lattice-based representation [28], may no longer apply.

2.3 Polyhedral synthesis

A parallelizing compiler analyzes the input program and maps the computation to a parallel architecture. The new execution order must reproduce the original computation: each operation must be fed with the same data, the original data-dependences must be respected. However, checking data dependence between two operations is undecidable. Even the sequence of operations executed on a given input – the execution trace – is undecidable. Usually, compiler analysis over-approximates the execution trace as well as the data dependences. However, the approximation made is usually rough and the compiler may miss many opportunities of parallelization. Another approach is to restrict the compiler analysis to programs whose execution trace and data dependences are input invariant and can be expressed with decidable sets.

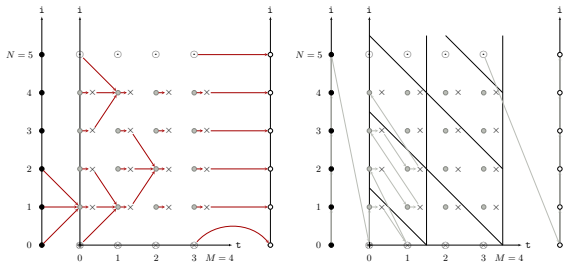
Affine control loops. The polyhedral model focuses on kernels with affine control loops manipulating arrays [17]. The control is exclusively made of `for` loops, `if` and sequence. Data types allowed are arrays, structures and scalar variables (seen as dimension 0 arrays), there are no pointers. Also, loop bounds, conditions and array indices must be affine functions of surrounding loop counters and structure parameters (*e.g.* array size). This ensures that execution trace may always be expressed as a union of integer polyhedra. Most linear algebra and signal processing kernels can fit into this model. Figure 1.(a) depicts such a kernel computing iteratively the heat equation on a 1D mesh [21] stored in the array u_0. α is a rational constant depending on discretization parameters and $\beta = 1 - 2\alpha$. With this program model, the execution of a single assignment S is always controlled by a nest of affine `for` loops guarded by affine conditions. Such an iteration is uniquely represented by the vector of surrounding loop counters $\vec{i}$. The execution of S at iteration $\vec{i}$ is denoted by $\langle S, \vec{i} \rangle$. The set $\mathcal{D}_S$ of iteration vectors is called the *iteration domain* of S. By construction, the iteration domain $\mathcal{D}_S$ is always an *integer polyhedron*, hence the name of the framework. The original execution order is given by the *lexicographic order* $\ll$ over $\mathcal{D}_S$, which is also computable. Figure 1.(b) depicts the iteration domains for the different assignments of the heat-1D kernel. As mentioned on the code (a),

```
for i := 0 to N
  init(u0[i]); //•

//Heat-1D kernel
for t := 0 to M − 1
  for i := 1 to N − 1
    u1[i] := αu0[i − 1] + βu0[i] + αu0[i + 1]; //•
  for i := 0 to N
    if i = 0 then u0[i] := 0; //⊗
    if 1 ≤ i ≤ N − 1 then u0[i] := u1[i]; //×
    if i = N then u0[i] := 0; //⊙

for i := 0 to N
  use(u0[i]); //∘
```

(a) Heat-1D kernel



(b) Data dependences (c) Scheduling

Fig. 1. Heat-1D kernel, execution trace (iteration domains) and data dependences

the initialization iterations are represented with •; the kernel iterations with •, ⊗ and ⊙; and the use iterations with ∘. Red arrows represent data dependences, as discussed in the next paragraph.

Data dependences. With the polyhedral model, execution traces can be summarized exactly with integer polyhedra. This make possible to build precise compiler analysis (data dependences, scheduling, data/computation allocation, etc) thanks to integer linear programming and geometric operations [1, 2, 11, 15, 16]. For instance, array dataflow analysis [15] computes exact data dependences. That is, a function $h_{S,r}$, called source function, which maps each read r of each assignment execution $\langle S, \vec{i} \rangle$ to the assignment execution defining the read value $h_{S,r}(\vec{i})$. On the running example:

$$h_{\bullet,u_0[i-1]}(t,i) = \begin{cases} t = 0: & \langle \bullet, i-1 \rangle \\ t \geq 1 \wedge i = 1: & \langle \otimes, t-1, 0 \rangle \\ t \geq 1 \wedge i \geq 2: & \langle \times, t-1, i-1 \rangle \end{cases}$$

Source functions are always *integer piece-wise affine* modulo the encoding of assignments •, ⊗, × with integers and the padding of iteration vectors so they have the same dimension. In polyhedral HLS, source functions are often used to multiplex the data for each read of each assignment and to handle synchronizations and communications between parallel units [4, 31]. The complexity of the source function $h_{S,r}$ (number of clauses, number of affine constraints per clause) may increase exponentially with the dimension of the iteration domain $\mathcal{D}_S$. Hence, efficient compaction techniques are required.

Scheduling and code generation. Provided the data dependences, the next step is to change the execution order to improve quality criteria (parallelism, data reuse, etc). This is done by computing a scheduling function θ_S which maps each execution $\langle S, \vec{i} \rangle$ to a timestamp $\theta_S(\vec{i})$. In the polyhedral model, we seek for affine schedules $\theta_S(\vec{i}) = A\vec{i} + b$, the timestamps $\theta_S(\mathcal{D}_S)$ being vectors ordered by their lexicographic order. In a way, $\theta_S : \mathbb{R}^n \rightarrow \mathbb{R}^p$ translates a nest of n loops to a target nest of p loops, each component of $(t_1, \ldots, t_p) = \theta_S(i_1, \ldots, i_n)$ being the iteration of the operation $\langle S, i_1, \ldots, i_n \rangle$ in the transformed loop nest. A simple criterion to maximize parallelism is to minimize p, so a maximum number of operations will share the same date [16] (and thus will be scheduled to be executed in parallel). Once the schedule is found, it remains to generate the control which executes the assignments in the order prescribed by the schedule. Many approaches were developed

[5, 12]. The best approach for HLS is to produce a control automaton per assignment S which issues a new iteration vector i of S at each clock cycle [12]. Two integer piecewise affine functions are required. A function First_S, which issues the first iteration of S w.r.t θ_S (initial state) and a function Next_S which maps each iteration of S to the next iteration of S to be executed w.r.t. θ_S (transition function). On the running example, we would have:

$$\text{First}_\bullet(N,M) = \left\{ \; N \geq 0 \wedge M \geq 0 : \quad (0,0) \right.$$

$$\text{Next}_\bullet(t,i) = \left\{ \begin{array}{rl} i \leq N-2 : & (t,i+1) \\ i = N-1 \wedge t \leq M-2 : & (t+1,0) \\ i = N-1 \wedge t = M-1 : & \text{stop} \end{array} \right.$$

To improve reuse, affine scheduling is usually combined with affine partitioning (or tiling) [11]. Each relevant iteration domain $\mathcal{D}_S$ is partitioned into parallelepipeds by translating a collection of cutting hyperplanes $\phi_S^1, \ldots, \phi_S^n$. Then, the new iteration domain $\mathcal{D}_S^T$ is indexed with vectors $(\Phi_1, \ldots \Phi_n, i_1, \ldots i_n)$, $(\Phi_1, \ldots \Phi_n)$ being the coordinates of the partition containing the original iteration vector $(i_1, \ldots i_n)$. Again, the resulting domain $\mathcal{D}_S^T$ is an integer polyhedron which can be scheduled thanks to affine scheduling. However, affine partitioning highly complexifies the control on the generated program. Figure 1.(c) gives an example of affine partitioning. $\mathcal{D}_\bullet$, $\mathcal{D}_\times$, $\mathcal{D}_\otimes$ and $\mathcal{D}_\odot$ are partitioned with cutting hyperplans $\phi^1 : t + i = 2\Phi_1$ and $\phi^2 : t = 2\Phi_2$ with partition coordinates $\Phi_1 = 0, 3$ and $\Phi_2 = 0, 1$. The affine schedule found is $\theta_\bullet(i) = (1, i)$, $\theta_\bullet(\Phi_1, \Phi_2, t, i) = (2, \Phi_1, \Phi_2, t+i, t, 0)$ $\theta_\otimes(\Phi_1, \Phi_2, t, i) = \theta_\odot(\Phi_1, \Phi_2, t, i) = \theta_\times(\Phi_1, \Phi_2, t, i) = (2, \Phi_1, \Phi_2, t + i, t, 1)$ and $\theta_\circ(i) = (3, i)$. The final execution order is depicted with grey arrows. For the assignment $\bullet$, the functions $\text{First}_\bullet()$ and $\text{Next}_\bullet$ are:

$$\text{First}_\bullet(N,M) = \{ \; N \geq 0 \wedge M \geq 0 : \quad (0,0,0,1)$$

$$\text{Next}_\bullet(\Phi_1,\Phi_2,t,i) = \left\{ \begin{array}{l} -t + 2\Phi_1 \geq 0 \wedge -1 - t + 2\Phi_2 \geq 0 \wedge \\ 126 - t \geq 0 : \\ \quad (t - \Phi_1, \Phi_2, 1 + t, -1 + i) \\ -t - i + 2\Phi_2 \geq 0 \wedge \\ 126 - t - i + 2\Phi_1 \geq 0 : \\ \quad (\Phi_1, t + i - \Phi_2, 2\Phi_1, 1 + t + i - 2\Phi_1 \\ 126 - \Phi_2 \geq 0 \wedge 63 + \Phi_1 - \Phi_2 \geq 0 \wedge \\ 62 + \Phi_1 - \Phi_2 < 0 : \\ \quad (-63 + \Phi_2, 1 + \Phi_2, -125 + 2\Phi_2, 127) \\ 62 + \Phi_1 - \Phi_2 \geq 0 : \\ \quad (\Phi_1, 1 + \Phi_2, 2\Phi_1, 2 - 2\Phi_1 + 2\Phi_2) \\ 62 - \Phi_1 \geq 0 : \\ \quad (1 + \Phi_1, 1 + \Phi_1, 2 + 2\Phi_1, 1) \end{array} \right.$$

Remark that this Next function is simplified to avoid the exponential blow-up of clauses. When several domains overlap, the first clause is chosen. It is another reason why techniques to simplify generic piecewise affine functions do not apply here. All in all, the multiplexing and the control involved in this example have a total of 669 affine constraints and 137 affine expressions. Clearly, they should be compacted before being mapped to an FPGA. This paper provides an efficient algorithm to compact several integer piecewise affine functions, provided as a pool of affine constraints and expressions, as a DAG using efficiently FPGA resources.

3 OUR ALGORITHM

In this section, we present our algorithm to turn a collection of affine expressions and affine constraints to a compact DAG. Section 3.1 discusses the cost function to be minimized. Then, Section 3.2 defines the semantic factorizations considered to optimize the control: expression factorization and constraint factorization. Section 3.3 explains how all possible combinations of semantic factorizations (expression and constraint) can be summarized with a graph. Finally, Section 3.4 shows how to select the best composition with respect to the cost function.

3.1 Cost Model

Our algorithm leverages a cost function to derive a resource efficient DAG from a pool of affine control functions. Our algorithm is fully parametrized by the cost function, which could be customized at will to fit a given target. In this section, we provide an example of cost function, which happens to be relevant for FPGA targets.

Cost of a DAG. An FPGA consists of reconfigurable building blocks with lookup tables, 1 bit adders and 1 bit registers (ALM with Altera, CLB with Xilinx). In addition, RAM blocks and DSP blocks are usually provided. Our DAGs use only integer operators (integer addition, integer multiplication by an integer constant) which require an amount of building blocks proportional to the bitwidth of the result. Hence, the resource usage of a DAG $\mathcal{D} = (N, E)$ can be modeled as a simple weighted sum:

$$|\mathcal{D}| = \sum_{n \in N} \mathbf{w}(n) \cdot \mathbf{bw}(n)$$

Where $\mathbf{bw}(n)$ denotes the bitwidth of the result computed by the operator n of the DAG and $\mathbf{w}(n)$ denotes, roughly, the number of building blocks required by n to compute 1 bit of result. $\mathbf{bw}$ is simply computed for each node of the DAG by a bottom-up application of the rules $\mathbf{bw}(x + y) = 1 + \max(\mathbf{bw}(x), \mathbf{bw}(y))$ and $\mathbf{bw}(x * y) = \mathbf{bw}(x) + \mathbf{bw}(y)$ starting from the bitwidth of the input variables. Furthermore, $\mathbf{w}$ can be customized at will to fit a target FPGA. In the following, we will assume $\mathbf{w}(+) = 1$ and $\mathbf{w}(*) = 100$. With that choice, our algorithm will tend to decompose affine expressions with multiplications by a power of 2. Note that the cost model is not intended to reflect the actual resources requirement. Rather, it should be viewed as an objective function whose minimization leads to desired properties.

This model is refined to handle two special cases: (i) $|a * x| = 0$ if a is 0, 1 or a power of 2. (ii) when n is a multiplication by a negative constant. In the latter case, the extra cost of complement-by-2 must be taken into account. With $a > 0$, $-a * x = a * \overline{x} + a$, where $\overline{x}$ denotes the logic complement of x. The cost is:

$$|-a * x| = \mathbf{bw}(x) + (\mathbf{bw}(a) + \mathbf{bw}(x)) \cdot \mathbf{w}(*) + (\mathbf{bw}(a) + \mathbf{bw}(x) + 1) \cdot \mathbf{w}(+)$$

The first term is the cost of the logic complement $\overline{x}$, the second term is the cost of the multiplication (when a is 1 or a power of 2, this term is removed), and the third term is the cost of the addition.

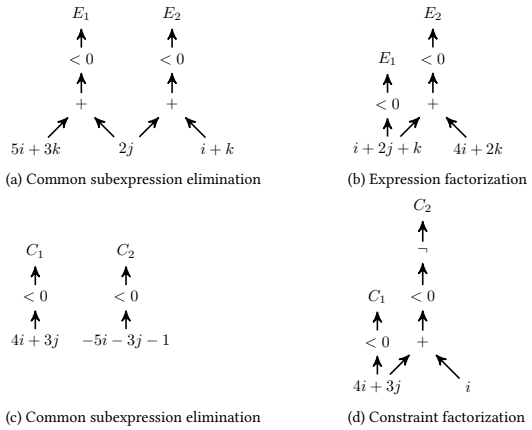
Affine Forms. Our algorithm builds the DAG by adding affine forms incrementally, and needs an upper bound on the cost of the sub-DAG computing an affine form. Consider an affine form $u = \sum_{i=1}^{n} a_i x_i + b$ where the x_i are integer variables and the coefficients a_i and b are integer constants. In the worst case, the term $a_i x_i$ (or b) with the largest bitwidth is evaluated first, each addition increasing the size of the result of 1 bit. Hence, the worst possible bitwidth for the result of u is $\mathbf{bw}_{\text{worst}}(u) = n - 1 + \max\{b\} \cup \{\mathbf{w}(a_i) + \mathbf{w}(x_i),\ i \in [\![1, n]\!]\}$. Therefore, an upper bound for $|u|$ is:

$$\lceil u \rceil = n \cdot \mathbf{bw}_{\text{worst}}(u) \cdot \mathbf{w}(+) + \sum_{i=1}^{n} |a_i * x_i|$$

Again, the cost function $|.|$ is a parameter of our algorithm. It could perfectly be refined/redefined to fit a different target.

3.2 Motivating Examples

Consider affine expressions $E_1 = i + 2j + k$ and $E_2 = 5i + 2j + 3k$ where i, j and k are input variables. Common subexpression elimination would produce the DAG sketched in Figure 2.(a). The resources used are 4 adders, 2 multipliers by a constant and 1 shifter. Now remark that $E_2 = E_1 + 4i + 2k$. This



(a) Common subexpression elimination

(b) Expression factorization

(c) Common subexpression elimination

(d) Constraint factorization

Fig. 2. Affine expression and constraint factorization

leads to the DAG in Figure 2.(b). With that *expression factorization*, the resources required are now 4 adders and 3 shifters, which is better than the first solution.

A similar factorization scheme can be applied to affine constraints. Consider the normalized affine constraints $C_1 : 4i + 3j < 0$ and $C_2 : -4i - 3j + 1 < 0$. Writing $C_2 : 4i + 3j \geq 0$, it is easy to detect that $C_2 = \neg C_1$. Now, consider the affine constraint $C_2 : -5i - 3j - 1 < 0$. There is no direct connexion with C_1. But if we write $C_2 : 5i + 3j \geq 0$, the affine expression of C_2 $(5i + 3j)$ can be obtained from the affine expression of C_1 $(4i + 3j)$, giving the improved DAG depicted in Figure 2.(d). With that *constraint factorization*, the resources used are reduced to 2 adders, 1 multiplier by a constant and 1 shifter.

Expression and constraint factorization rise several issues which must be handle carefully:

- Expression factorization is not always beneficial. If one tries to derive E_1 from E_2, with $E_1 = E_2 + (-4i - 2k)$, the resource usage would be worse than the direct solution (4 adders and 5 multipliers by a constant). A best combination of factorizations must be found among all the possible combinations.
- Constraint factorization (C_2 from C_1) is a terminal transformation. Indeed, the expression of C_2 (e_2 s.t $C_2 : e_2 < 0$) is never computed. Hence, subsequent factorizations involving the expression e_2 are not possible. For this reason, expression factorizations will be preferred over constraint factorizations.

This paper proposes a unified way to represent the possible sequence of factorizations of affine expressions and constraints and to select combination of factorizations minimizing the resource consumption.

3.3 Realization Graph

All the possible combination of semantic factorizations will be summarized in a *realization graph* $\mathcal{G}_r$. Basically, the nodes of $\mathcal{G}_r$ are affine expressions and constraints, and an edge $u \xrightarrow{\Delta} v$ means that v can be realized from u with a cost of Δ. Intuitively, a rooted path in $\mathcal{G}_r$ would give a realization of the reached nodes. Depending on the factorization (expression or constraint) a specific edge is issued, as explained in the following sections.

Expression Factorization. Given a DAG node computing an expression u, an expression v can be computed by applying the factorization rule $v = u + (v - u)$. In that case, we would add to the DAG the following components:

- A sub-DAG computing $v - u$ (to be optimized as well)
- An adder taking the output nodes of u and $v - u$.

The additional resource cost would then be the cost of the operator + plus the cost of the affine form $v - u$: $\Delta = (1 + \max\{\mathbf{bw}_{\text{worst}}(u), \mathbf{bw}_{\text{worst}}(v-u)\}) \cdot \mathbf{w}(+) + \lceil v - u \rceil$. We register this possible design choice to a *realization graph* $\mathcal{G}_r$, whose nodes are expressions and constraints to be computed and whose edges $u \xrightarrow{\Delta} v$ express that v can be computed from u with an additional resource cost of Δ. When the target node is a constraint $v < 0$, the edge has the same meaning. In general, the incoming edge with the smallest cost $u \xrightarrow{\Delta} v$ will be preferred to design v.

Constraint Factorization. Given a DAG node computing an affine constraint normalized as $u < 0$, the constraint $v < 0$ can be derived from $u < 0$ with a simple logic negation when $v < 0 \equiv \neg(u < 0)$, which means: $v < 0 \equiv -u - 1 < 0$ or more simply: $u + v = -1$. This gives a first simple test to detect negations. Otherwise, remark that $(u + (-1 - u - v)) + v = -1$. This means that $v < 0 \equiv \neg(u + (-1 - u - v) < 0)$. Hence $v < 0$ can be computed from u by adding the following components to the DAG:

- A sub-DAG computing $-1 - u - v$ (to be optimized)
- An adder taking the output nodes of u and $-1 - u - v$.
- The result of the adder is checked by connecting the most significant bit (to have < 0) to a negation.

The additional resource cost would then be the cost of the operator +, plus the cost of the affine form $-1 - u - v$: $\Delta = (1 + \max\{\mathbf{bw}_{\text{worst}}(u), \mathbf{bw}_{\text{worst}}(-1-u-v)\}) \cdot \mathbf{w}(+) + \lceil -1 - u - v \rceil$. We add a *negation edge* $u < 0 \xrightarrow{\Delta}_{\neg} v < 0$ to the realization graph.

Final Algorithm. Figure 3 depicts our algorithm to build the realization graph $\mathcal{G}_r$ from a pool of expressions $\mathcal{E}$ and constraints C. Expressions and constraints are inserted incrementally in the graph $\mathcal{G}_r$ (lines 3–7), constraint nodes are marked to be distinguished from expression nodes (line 6). Finally, a special node initial_node is added to the graph $\mathcal{G}_r$ (line 8) and connected to each node u with an expression factorization edge labeled by $|u|$ (lines 9–10). initial_node will serve as a starting point to select the best realization as explained in the next section. Indeed, edges initial_node $\xrightarrow{|u|} u$ suggest a *direct* realization of u whereas edges $u \xrightarrow{\Delta} v$ suggest that v can be realized *from* u with a cost Δ.

Each expression is inserted with procedure INSERT (lines 11–14). Prior to inserting the expression e, the maximal strict subexpression with each node n of $\mathcal{G}_r$ is inserted. This ensures a maximal subexpression factorization beetween the expressions of $\mathcal{E}$ and C. Indeed, expression factorization is not able to factor strict subexpressions, only cases where u is a subexpression of v are detected: if $u = 3i+j$ and $v = 3i+j+5k$ then v is naturally expressed as $u+5k$. However, if $u = 3i+j+4k$, $v = 3i+j+k$ then the best solution is a factorization by the strict subexpression $3i+j$ which does not appear with pure expression factorization. Then, expression factorization edges are inserted between each pair of nodes whenever it is beneficial (lines 15–23) using the rule described above. For presentation reason, we use the notation $\phi(u, v)$ for $(1 + \max\{\mathbf{bw}_{\text{worst}}(u), \mathbf{bw}_{\text{worst}}(-1-u-v)\}) \cdot \mathbf{w}(+)$. Expression factorization is beneficial when the circuitry added for v is strictly less expensive than computing v directly (line 19). Then, the symmetric case (computing u from v) is considered for completeness.

Each constraint $e < 0$ is inserted in the graph $\mathcal{G}_r$ (lines 5–7). The expression e is inserted as described above (line 6). Then, negation edges between $e < 0$ and constraint nodes of $\mathcal{G}_r$ are added

(line 7) by using procedure INSERT_NEG_EDGE (lines 24–32). For each constraint node $u < 0$ of $\mathcal{G}_r$ (line 25) without expression factorization edge to $e = v < 0$ (line 26), the negation edge is added whenever constraint factorization is beneficial. For presentation reason, we use the notation $\psi(u, v)$ for $(1 + \max\{\mathbf{bw}_{\text{worst}}(u), \mathbf{bw}_{\text{worst}}(-1 - u - v)\}) \cdot \mathbf{w}(+)$. Cases with expression factorization edge are preferred, as expression factorization is always more beneficial than constraint factorization (see discussion in section 3.2). As for expressions, constraint factorization is beneficial when the circuitry added for $v < 0$ is strictly less expensive than computing $v < 0$ directly (line 27). Similarly, the symmetric case (computing $u < 0$ from $v < 0$) is considered for completeness.

Example. Consider the affine constraints C depicted in the following table with the input bitwidths $\mathbf{bw}(i) = 2$ and $\mathbf{bw}(j) = \mathbf{bw}(k) = 8$. BUILD_REALIZATION_GRAPH($\emptyset$,C) produces the graph depicted in figure 4.(a). Common subexpression between constraints 1 and 2 (inserted at line 13) produces the node $2j$ depicted in white. Constraint factorization edges are dashed. Each edge is labeled by its cost Δ computed according to the rules given in section 3.1. Again, these rules can be parametrized to fit the target. Consider constraints 1 and 2 and there nodes in $\mathcal{G}_r$. $\mathcal{G}_r$ suggests that constraint 1 can be realized either directly with cost 22 (edge from initial_state), or from subexpression $2j$ with a cost 19. In turn, $2j$ can serve as a basis to realize other expressions as constraint 3 with cost 19 (edge from $2j$ to $4i + 3j$). Constraint 3 can be used to realize constraint 4 thanks to a negation factorization of cost 12. This shows that choices need to be done on $\mathcal{G}_r$ to find the best combination of factorization. This is the purpose of the next section.

C :

Id	Constraint
1	$i + 2j + k < 0$
2	$5i + 2j + 3k < 0$
3	$4i + 3j < 0$
4	$-5i - 3j - 1 < 0$

3.4 Finding an Efficient Realization

An expression factorization edge $u \xrightarrow{\Delta} v$ of the realization graph $\mathcal{G}_r$ means that expression of v (if v is a constraint $e < 0$, the expression is e) may be realized from the expression of u with a cost Δ. For instance, the edge $i + 2j + k \xrightarrow{18} 5i + 2j + 3k$ in Figure 4.(a) means that $v = 5i + 2j + 3k$ might be realized from $u = i + 2j + k$ with an expression of cost 18. Note that u and v might be realized directly with cost 22 and 122 respectively (see edges from *initial_node*). u and v might also be realized from $w = 2j$. In that case, we choose edges $initial\_node \xrightarrow{0} w$, $w \xrightarrow{15} u$ and $w \xrightarrow{117} v$ for a total cost of $0 + 15 + 117 = 132$. From this observation, we may conclude that a realization of u and v is a subtree of $\mathcal{G}_r$ rooted at *initial_node* and including the nodes u and v to be realized. More generally, a valid realization of v is a path:

$$initial\_node \xrightarrow{\Delta_1} u_1 \ldots \xrightarrow{\Delta_n} u_n \xrightarrow{\Delta} v$$

Each u_i being realized from u_{i-1} at cost Δ_i. The total cost is $\Delta_1 + \ldots + \Delta_n + \Delta$. If v and u_n are constraints, then v may be realized with a constraint factorization edge from u_n:

$$initial\_node \xrightarrow{\Delta_1} u_1 \ldots \xrightarrow{\Delta_n} u_n \xrightarrow{\Delta}_{\neg} v$$

In that case, the expression of v would not be available. Indeed, $v < 0$ would be evaluated without computing v. Then, no realization could start from v: the negation edges are terminal. Also, nodes along the path can be used to compute others nodes of $\mathcal{G}_r$. However, each node must have a single predecessor in the obtained subgraph, which is then a tree. These remarks lead to the following definition.

```
1  BUILD_REALIZATION_GRAPH(E,C)
2   G_r := empty_graph();
3   for each expression e ∈ E
4    INSERT(e);
5   for each constraint (e < 0) ∈ C
6    INSERT(e); mark(e); //e < 0
7    INSERT_NEG_EDGE(e);
8   Add node initial_node to G_r;
9   for each node u ∈ G_r − {initial_node}
10   Add edge initial_node --|u|--> u to G_r;

11 INSERT(e)
12  for each node n ∈ G_r
13   INSERT_EDGE(common_sub_expr(e,n));
14  INSERT_EDGE(e);

15 INSERT_EDGE(v)
16  if v ∈ G_r return;
17  Add node v to G_r;
18  for each node u ∈ G_r
19   if φ(u,v) + |v − u| < |v| //v from u
20    Add edge u --φ(u,v)+|v−u|--> v to G_r;
21   //Symmetric case
22   if φ(v,u) + |u − v| < |u|
23    Add edge v --φ(v,u)+|u−v|--> u to G_r;

24 INSERT_NEG_EDGE(v)
25  for each marked node u ∈ G_r //u < 0
26   if ∃u --Δ--> v ∈ G_r continue;
27   if ψ(u,v) + | − 1 − u − v| < |v|
28    Add edge u --ψ(u,v)+|−1−u−v|--¬--> v to G_r;
29   //Symmetric case
30   if ∃v --Δ--> u ∈ G_r continue;
31   if ψ(v,u) + | − 1 − v − u| < |u|
32    Add edge v --ψ(v,u)+|−1−v−u|--¬--> u to G_r;
```

Fig. 3. Algorithm for constructing the realization graph $\mathcal{G}_r$

Definition 3.1 (Realization). Let $\mathcal{G}_r$ be the realization graph of expressions $\mathcal{E}$ and constraints $\mathcal{C}$. A *realization* is a subgraph $\mathcal{T} \subseteq \mathcal{G}_r$ which satisfies the following conditions:

(1) Each expression/constraint is realized correctly: $\mathcal{T}$ is a tree rooted at initial_node, and $\mathcal{T}$ spans $\mathcal{E} \cup \mathcal{C}$.
(2) No useless common subexpression is computed: the leaves of $\mathcal{T}$ belong to $\mathcal{E} \cup \mathcal{C}$.
(3) Negation edges are terminal.

In other words, a realization is a particular spanning tree of $\mathcal{G}_r$. The condition 2) avoid useless computation of common subexpressions (see white nodes in figure 4.(a)): common subexpressions are forced to be intermediate results in the final realization. The cost of a realization is the sum of

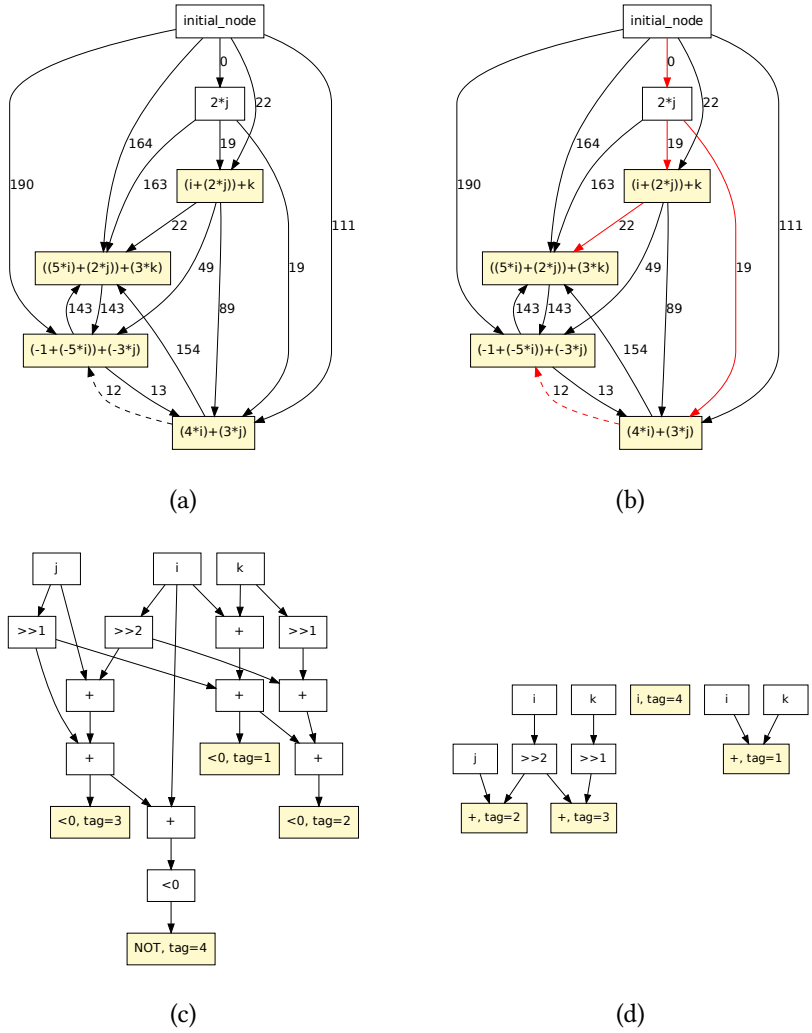


Fig. 4. (a) Realization graph $\mathcal{G}_r$ obtained from C, (b) Resource-efficient realization tree $\mathcal{T}$ in $\mathcal{G}_r$ (in red), (c) Resource-efficient DAG from realization tree $\mathcal{T}$ in $\mathcal{G}_r$, (d) Recursive compaction of fresh expressions $\mathcal{E}_{\text{new}}$

the weights Δ on its edges. Hence, finding an efficient realization amounts to compute a minimal spanning tree of $\mathcal{G}_r$, under the constraints specified in definition 3.1.

The algorithm for finding a minimum realization is given in figure 5. The algorithm proceeds into two steps. First, a minimum spanning tree rooted on initial_node is found among the expression factorization edges of $\mathcal{G}_r$ by using a variant of Prim's greedy heuristic (lines 4–7). The search is stopped once all expression/constraint nodes are covered. Second, constraint factorization is considered for orphan constraints (lines 8–15). An orphan constraint v is neither factorized (father is initial_node) nor involved in an expression factorization (leaf) (lines 9–10) nor involved in a previous constraint factorization (line 11). A best local constraint factorization is found for v and added to the realization (lines 12–13). As mentioned above, constraint factorizations are terminal.

```
1  BUILD_REALIZATION_TREE(G_r,E,C)
2   T := ({initial_node},∅);
3   O := E ∪ C;
4   while O ≠ ∅
5    Find u --Δ--> v s.t. u ∈ T, v ∉ T with Δ minimum;
6    Add to T;
7    if (v ∈ O) O := O − {v};
8   to_evaluate := ∅;
9   for each initial_node --Δ--> v ∈ T s.t. v is a leaf in T
10   if (v ∉ C) continue;
11   if (v ∈ to_evaluate) continue;
12   Find u --Δ-->¬ v with Δ minimum;
13   Add to T;
14   to_evaluate := to_evaluate ∪ {u};
15   Remove edge initial_node --Δ--> v from T;
```

Fig. 5. Algorithm for finding a minimal realization $\mathcal{T}$ in $\mathcal{G}_r$

This is enforced by excluding the source u from the nodes to be considered for subsequent constraint factorizations (line 14). Edge from initial_node to v is removed from the realization, as v is now realized from u with a constraint factorization (line 15).

We choose to restrict constraint factorization to orphan constraints, since constraint factorization is always less beneficial than expression factorization: it is terminal and the gain for the constraint is likely to be comparable to an expression factorization. Hence constraints involved in expression factorization (thus non-orphan) are excluded.

Example (cont'd). Figure 4.(b) depicts the realization tree $\mathcal{T}$ obtained from the realization graph of figure 4.(a). The edges chosen for the realization tree are in red. The total cost of the design (72) is greatly improved compared to direct realization (487) (only arcs from initial_node, no factorization at all) and compared to common subexpression factorization (391) (edges from node $2j$ and direct realization of $-5i - 3j - 1$). Among the 4 *factorization edges* of $\mathcal{T}$ (edges not coming from initial_node), there are 3 expression factorizations (labeled by costs 19, 19 and 22), and 1 constraint factorization (dashed edge labeled by 12). Among the two expression factorizations 1 is a subexpression factorization (targeting $i + 2j + k$), the two others are not (targeting $4i + 3j$ and $5i + 2j + 3k$). The two latter are said to be *semantic*: they are obtained by playing on semantic properties of addition and multiplication and could not be found by subexpression factorization. Constraint factorization (dashed edge labeled by 12) is also semantic: it could not be found directly on the negation with subexpression factorization. All in all, this example show the important role played by semantic factorization for expression and constraints in reducing the resource cost of set of constraints.

3.5 Building the DAG

Figure 6 depicts our algorithm to build the DAG from the realization tree found in $\mathcal{G}_r$. The inputs are: the realization tree $\mathcal{T}$ and the set of expressions $\mathcal{E}$ and constraints C to be realized. They are not specified to simplify the presentation. The output is the DAG and a mapping node[.] linking expressions/constraints of $\mathcal{E}$ and C to there implementation in the DAG. The algorithm is a recursive depth traversal of $\mathcal{T}$. Each time an edge of $\mathcal{T}$ is traversed, the DAG is updated accordingly. Two

additional inputs are used to traverse $\mathcal{T}$ (t) and to build the DAG (d). The invariant is: when calling BUILD_DAG(t,d), t is already realized in the DAG, and realization root in the DAG is pointed by d. If t is an expression of $\mathcal{E}$, node[.] is updated with d (line 3). If t is the expression of a constraint $c \in C$, the circuitry to check $t < 0$ is added to the DAG, and the root is linked to c (lines 4–6). The recursive traversal is handled in the remaining lines. Initially, BUILD_DAG is called with t = initial_node and d = null. A DAG dag(u) is built for each target node u. Its root serves as starting point for the traversal (lines 7–10). The circuitry is added to the DAG for selected expression factorizations (lines 11-12) and constraint factorizations (lines 13-20) by following the rules described in section 3.3. Since constraint factorization is terminal, no recursive call is required. Consequently, node[.] should be updated in that place (lines 16 and 20).

Rules for expression and constraint factorization produces a pool of new expressions $\mathcal{E}_{\text{new}}$ in the DAG ($u - t$ for expression factorization line 12, $-1 - t - u$ for constraint factorization line 19). In turn, these new expressions may be further optimized. Then, our algorithm is applied recursively on $\mathcal{E}_{\text{new}}$. The output of the new DAG are used in the current DAG in place of the expressions of $\mathcal{E}_{\text{new}}$. Notice that recursive calls are optional. The recursive depth can be used as a tuning knob to customize the degree of reuse in the DAG.

Example (cont'd). Figure 4.(b) depicts the DAG obtained from the realization tree $\mathcal{T}$. Prior to building the DAG, the expressions $\mathcal{E}_{\text{new}}$ are collected and compacted with a recursive call. They are: $i+k$ from edge $2j \xrightarrow{19} i+2j+k$, $4i+j$ from edge $2j \xrightarrow{19} 4i+3j$, $4i+2k$ from edge $i+2j+k \xrightarrow{22} 5i+2j+3k$ and i from edge $4i+3j \xrightarrow{12}_{\neg} -5i-3j-1$. The recursive call on $\mathcal{E}_{\text{new}}$ gives the result depicted in Figure 4.(d). For the sake of clarity, we have tagged realization roots in the DAG. $i+k$ has tag 1, $4i+j$ had tag 2, $4i+2k$ has tag 3 and i has tag 4. Here, the compaction has detected that $4i$ is a common subexpression. Multiplications by a power of 2 are represented by shifts ($\gg 1$ for $\times 2$ and $\gg 2$ for $\times 4$). These realizations serve as building blocks for the final DAG on Figure 4.(c). Again, the nodes has been tagged for the sake of clarity. Here, the tags are the ranks of constraints C given in section 3.3.

4 EXPERIMENTAL EVALUATION

In this section, we present the results obtained by applying our algorithm on a large benchmark of applications, with and without polyhedral optimization. Section 4.1 describes the experimental setup. Then, Section 4.2.1 presents synthesis results on FPGA. Section 4.2.2 discusses the semantic factorizations found in the benchmarks. Finally, Section 4.2.3 presents statistics on the behavior of our algorithm (execution time, recursive depth).

4.1 Experimental setup

We have applied our algorithm to simplify the affine control generated for the kernels of the benchmark suite PolyBench/C v3.2 [26]. Table 1 depicts the kernels and the synthesis results obtained on FPGA.

For each kernel, a DPN process network is generated using the Dcc tool [4]. Dcc optimizes the data transfers and the pipeline execution with an affine schedule based on a loop tiling [2, 3]. The execution order is deeply restructured and the affine control per process (control automaton,mux/demux) can be quite complex. Before deriving the DAGs, we simplify the control polyhedra with various heuristics including gist and integer set coalescing [32]. Then, for each process, we collect the affine control and we apply our algorithm to produce a DAG. Table 1 presents the sum of the criteria collected for each process. #dags is the total number of DAG produced, #C is the

```
1  BUILD_DAG(t,d)
2   //Link DAG nodes to inputs E and C
3   if (t = e ∈ E) node[e] := d;
4   if ((t < 0) = c ∈ C)
5    Add edges for ineq_node := d < 0;
6    node[c] := ineq_node;

    //Base case
7   if t = initial_node
8    for each edge t -Δ-> u ∈ T
9     BUILD_DAG(u,dag(u))
10   return;

    //Expression factorization
11  for each edge t -Δ-> u ∈ T
12   BUILD_DAG(u, +(d, (E(V(u) − V(t)))))

    //Constraint factorization
13  for each edge t -Δ,¬-> u ∈ T
14   if (V(t) + V(u) = −1) //direct negation?
15    Add edges for neg_node := ¬(d < 0);
16    node[u < 0] := neg_node;
17   else
18    Add edges for:
19     neg_node := ¬(+(d, E(−1 − V(t) − V(u))) < 0);
20    node[u < 0] := neg_node;
```

Fig. 6. Algorithm for building a DAG from a realization $\mathcal{T}$

total number of affine constraints. $\#\mathcal{E}$ is the total number of affine expressions. All in all, we have produced and analyzed a total of 261 DAGs from 4990 constraints and 2464 expressions.

The main innovation of this work is to explore *semantic factorizations* for simplification. Indeed the expression $3i + j$ could be factorized by $2i + j$ because of *semantic* properties of addition and multiplication, whereas common-subexpression factorization $3i + j$ would be restricted to *syntactic* subterms $3i$, j and $3i + j$. The latter is also referred as *non-semantic factorization*. The factorizations found by our algorithm are either semantic or not, depending on the arcs chosen by BUILD_REALIZATION_TREE. Thus, we want to make sure that through a FPGA synthesis tool, a DAG optimized with semantic factorization will effectively use less resources. This would ensure that semantic factorizations allow a significant improvement compared to the optimizations applied by an industrial tool.

4.2 Experimental results

This section analyzes the results obtained on Polybench/C according to two criteria defined in the experimental setup. Section 4.2.1 presents synthesis results on a FPGA and show the effectiveness of our approach. Then, Section 4.2.2 discusses the semantic factorizations found in the benchmarks.

Kernel	#dags	#$\mathcal{C}$	#$\mathcal{E}$	SEM+Quartus		Quartus		Gain
				ALM	Regs	ALM	Regs	
2mm	15	250	161	1011	612	1430	596	29%
3mm	18	369	206	1775	946	2528	922	30%
atax	12	134	83	628	328	900	321	30%
bicg	11	112	69	500	278	715	278	30%
correlation	27	356	205	1609	1129	2567	909	37%
covariance	16	243	143	1221	708	1872	618	35%
doitgen	9	145	124	393	280	607	268	35%
fdtd-2d	13	502	167	2339	1713	3293	1603	29%
gemm	10	125	93	672	270	851	270	21%
gemver	20	187	137	847	459	1102	438	23%
gesummv	14	95	84	456	245	549	227	17%
heat-3d	8	734	175	3545	1194	5667	2559	37%
jacobi-1d	8	134	64	628	556	912	520	31%
jacobi-2d	8	370	111	1660	1204	2547	1144	35%
lu	7	213	87	1116	666	1469	628	24%
mvt	11	118	70	550	290	758	290	27%
seidel-2d	5	226	63	1161	1464	1758	1291	34%
symm	13	213	116	1011	471	1540	465	34%
syr2k	10	135	90	721	290	944	281	24%
syrk	9	118	81	636	246	828	246	23%
trisolv	9	93	56	474	218	632	213	25%
trmm	8	118	79	549	262	806	253	32%

Table 1. Synthesis results on Polybench/C v3.2

Finally, Section 4.2.3 shows how often our algorithm is applied recursively and presents the execution times.

4.2.1 Synthesis results. We have implemented a VHDL generator for our DAGs and a direct generator which puts the affine expressions in VHDL and let the synthesis tool do the optimizations – typically common subexpression elimination and boolean optimizations. This way, we can compare our approach to the optimizations applied by the synthesis tool. The DAGs are generated using the hierarchical approach. Both direct and optimized designs are pipelined at ALM level by adding a sufficient number of registers to the outputs. This way, the synthesis tool will perform low level logic optimizations and retiming to redistribute the registers through the design. The synthesis was performed using Quartus Prime TM 16.1.2 from Intel on the platform on the Arria 10 10AX115S2F4I1SG FPGA with default synthesis options (optimization level - balanced). Intel Quartus Prime is capable of applying highly advanced optimizations automatically including common subexpression factorization and many other advanced boolean optimizations. The DAGs were tested using GHDL simulation tool over uniformly distributed random stimuli.

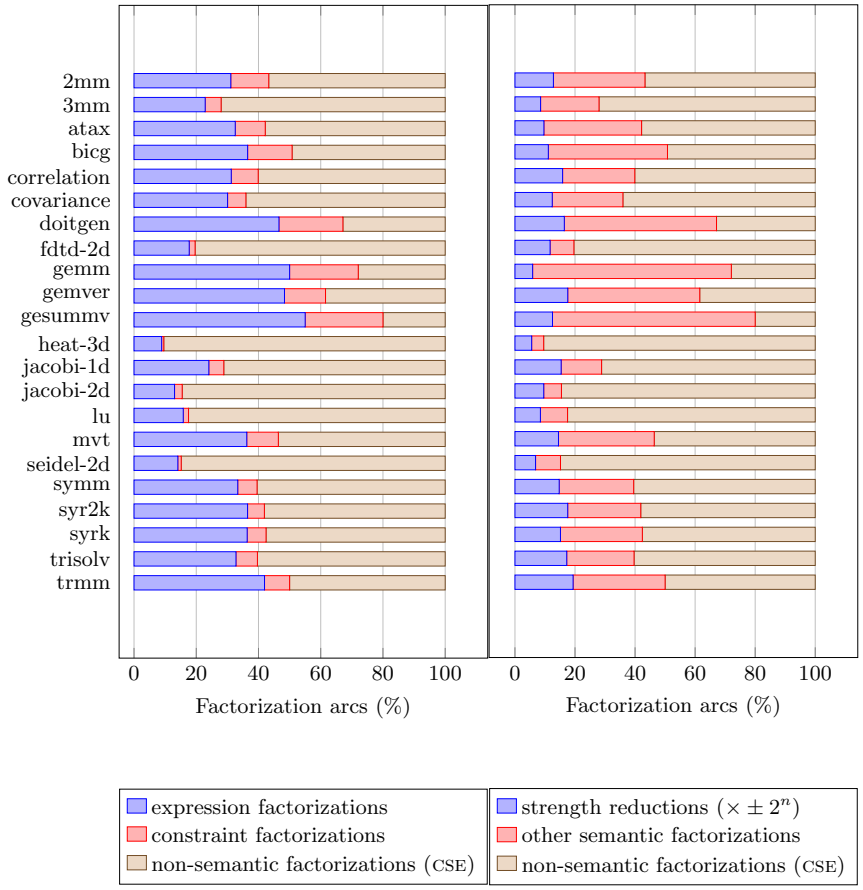
The synthesis results are presented in the table 1. The synthesis results for our DAGs are provided in the column SEM+Quartus (semantic factorizations + quartus). The synthesis results for the direct implementation are given in the column Quartus (quartus only). The gain in ALMs compared to the direct implementation is given in the column Gain. Both implementations run at the maximum FPGA frequency of 645.16 MHz. The frequency is limited by the target MAX delay limited by the signal hold timing and other physical constraints. Both versions use a significant amount of 5-6 inputs ALMs, thus achieving higher compression ratio. There is no ALM overhead because of registers, as

for all the examples they are entirely packed inside the ALMs containing logic. Experimental results show a significant gain in ALM, compared to the direct implementation optimized by Quartus (common subexpression elimination). The average gain is 30%, with a deviation of 5%. Remark that the kernel `gesummv` shows slightly less gain than the other kernels. The kernel `gesummv` has many simple polyhedra with small bitwidths in the computations. In that case, low-level boolean optimizations are more effective than semantic factorizations: arithmetic operations are merged with boolean $\wedge$ operations from the polyhedra using large (up to 7 bit) LUT. For the DAGs parts involved, this results in 3 to 4 times less ALMs than the optimized dags. The optimized dags can benefit less from these optimizations, as the bitwidth of the operations increases through the computations. Nonetheless, even for that kernel the benefit of semantic factorizations is significant. The synthesis results confirms the validity of our models and approach. Semantic factorization appears to complement nicely the optimization applied by quartus, and may be used profitably as an optimizing preprocessing for affine control.

4.2.2 Distribution of semantic factorizations. Figure 7.(a) depicts the ratio of semantic factorizations (blue and red) vs non-semantic – common subexpression – factorizations (brown) for each kernel. Non-semantic expression factorizations $u \xrightarrow{\Delta} v$ are such that u is a subterm of v. Non-semantic constraint factorizations $u \xrightarrow{\Delta}_{\neg} v$ are such that $v < 0$ iff $\neg(u < 0)$: negation recognition is not considered as a semantic factorization. Table 2 provides the actual number of factorizations (arcs) found for each kernel. Column #e-arcs gives the number of expression factorizations. The next column #sem give the number of semantic expression factorizations. The same goes for constraints: column #c-arcs gives the number of constraint factorizations and the next column #sem gives the number of semantic constraint factorizations. Figure 7.(b) depicts the proportion of semantic factorizations $u \to v$ which replace v by an expression with less multiplications and more multiplications with a power of 2. These factorizations produce a strength reduction. With our cost model, our algorithm tends to maximize these factorizations. Table 2 gives the number of such factorizations for each kernel (columns #pow2). Again, we distinguish between expression factorizations and constraints factorizations. We observe that 12% of the factorizations (36% of the semantic factorizations) enable a strength reduction. In general, strength reductions are very effective to reduce hardware resources

Among the numerous combinations of semantic factorizations found in the examples, a frequent pattern is a semantic factorization with a strength reduction $u \to v$ producing a term v used by several terms w_i. Factorizations $v \to w_i$ are often non-semantic. Here is an example from the kernel `gesummv`: $u = -17 + t$, $v = -15t + c$, $w_1 = -14 - 15t + c$ (non-semantic factorization), $w_2 = -15 - 15t + c$ (non-semantic factorization). Kernels `gemm`, `gemver`, `gesummv`, and `doitgen` are dominated by semantic factorizations (see Figure 7.(a)). The two third of the terms are very simple (e.g. $-1 * c$, $c - 1$, $t - 1$ for `gesummv`) and do not exhibit factorization opportunities. They are derived directly, without factorization. The remaining third contains more complex terms with factorization opportunities. For these terms, the interesting factorizations are semantic. In contrast, kernels `heat3d` and `fdtd2d` are dominated by non-semantic factorizations. These are kernels with a large number of constraints. We found many occurrences of our factorization pattern where constraints w_i differ only by a constant shift $v \to w_i = v + \lambda$ and can be produced by a non-semantic factorization.

4.2.3 Recursive calls and execution time. Semantic factorizations applied by our algorithm produces fresh expressions (denoted by $\mathcal{E}_{\text{new}}$ in section 3.5), which are in turn optimized by applying our algorithm recursively. Column rec-depth of Table 3.(a) provides the maximum number of nested recursive calls for each kernel. Many kernels need one recursive call: semantic factorizations were



(a) Distribution of semantic factorizations (b) Distribution of strength reductions ($\times \pm 2^n$)

Fig. 7. Semantic factorizations

applied, producing fresh expressions $\mathcal{E}_{\text{new}}$, in turn optimized with our algorithm. Quarter of the kernels even require two recursive calls: the semantic factorizations required by $\mathcal{E}_{\text{new}}$ produce fresh expressions $\mathcal{E}_{\text{new}_{\text{new}}}$ in turn optimized with our algorithm. We have run our algorithm on a Intel Core i5 CPU M 540 @ 2.53GHz with 3072 KB L2 cache and 3GB RAM, which is a pretty light configuration compared to the usual requirements of circuit synthesis. Table 3.(a) provides the execution time in seconds for the construction of the realization graph $\mathcal{G}_r$ (BUILD_REALIZATION_GRAPH, column insert-time) and for the computation of the best realization, the generation of the DAG and the subsequent recursive calls (BUILD_REALIZATION_TREE; BUILD_DAG, column dag-time). Most of the time is spent in the construction of the realization graph $\mathcal{G}_r$, the significant operation being the insertion of an affine expression (INSERT, fig. 3). Even with this light configuration, the overall execution time tends to be small with a median execution time of 3 seconds.

Kernel	#dags	#$\mathcal{C}$	#$\mathcal{E}$	#arcs	#e-arcs	#sem	#pow2	#c-arcs	#sem	#pow2
2mm	15	250	161	164	138	51	15	26	20	6
3mm	18	369	206	293	270	67	21	23	15	4
atax	12	134	83	83	68	27	7	15	8	1
bicg	11	112	69	63	52	23	6	11	9	1
correlation	27	356	205	208	187	65	21	21	18	12
covariance	16	243	143	153	142	46	14	11	9	5
doitgen	9	145	124	73	57	34	5	16	15	7
fdtd-2d	13	502	167	530	507	94	62	23	10	0
gemm	10	125	93	68	52	34	3	16	15	1
gemver	20	187	137	91	77	44	13	14	12	3
gesummv	14	95	84	40	28	22	5	12	10	0
heat-3d	8	734	175	1025	996	91	54	29	7	3
jacobi-1d	8	134	64	104	94	25	13	10	5	3
jacobi-2d	8	370	111	407	386	53	35	21	10	4
lu	7	213	87	177	167	28	15	10	3	0
mvt	11	118	70	69	60	25	10	9	7	0
seidel-2d	5	226	63	277	270	39	18	7	3	1
symm	13	213	116	129	115	43	19	14	8	0
syr2k	10	135	90	74	69	27	12	5	4	1
syrk	9	118	81	66	61	24	9	5	4	1
trisolv	9	93	56	58	51	19	10	7	4	0
trmm	8	118	79	62	57	26	12	5	5	0

Table 2. Semantic factorizations: detailed results

Kernel	#dags	#$\mathcal{C}$	#$\mathcal{E}$	insert-time (s)	dag-time (s)	Rec-depth
2mm	15	250	161	3.5	3.1	2
3mm	18	369	206	11.4	8.5	3
atax	12	134	83	0.8	0.7	2
bicg	11	112	69	0.5	0.5	2
correlation	27	356	205	6.4	5.4	2
covariance	16	243	143	2.7	2.6	2
doitgen	9	145	124	0.8	1.9	2
fdtd-2d	13	502	167	37.2	20.3	3
gemm	10	125	93	0.6	0.8	2
gemver	20	187	137	1.0	1.1	2
gesummv	14	95	84	0.2	0.3	2
heat-3d	8	734	175	502.6	249.8	3
jacobi-1d	8	134	64	2.2	1.3	2
jacobi-2d	8	370	111	37.3	19.6	3
lu	7	213	87	4.4	3.9	2
mvt	11	118	70	0.6	0.6	2
seidel-2d	5	226	63	38.2	15.4	3
symm	13	213	116	2.2	2.5	2
syr2k	10	135	90	0.8	1.3	2
syrk	9	118	81	0.6	0.8	2
trisolv	9	93	56	0.4	0.4	2
trmm	8	118	79	0.8	0.9	2

Table 3. Recursive calls and execution time

5 RELATED WORK

Pretty few approaches address the mapping of affine control in the context of polyhedral circuit synthesis. With Compaan/Laura, the control frequently executed is synthesized as a DAG with common subexpression factorization [14], the control less frequently executed being left to a sequential controller. This generates bubbles at each start of the innermost loop. When loops are restructured in such a way that innermost loops have often a few iterations [11], this limits the throughput of the controller. For instance, high-degree stencils often used in HPC require very sharp tiles whose corner have a few innermost loop iterations. Also, the sequential controller requires a microprogram to be stored in a ROM. As storage resources are limited on FPGAs, this would limits the control, hence the parallelism and finally the performances of the circuit. The authors also propose a runtime distance approach, which splits the iteration domain into phases where the multiplexing is constant (variant domains). The iterations spent in each phase are counted thanks to polyhedral analysis [13], then the control iterates through the phases with a counter. As far as we know, this approach was not evaluated. However, the amount of clock cycles is usually expressed with a piecewise affine pseudo-polynomial which is usually far more complex than the original control. Also, it requires full multipliers (variable times variable), which are also quite limited on today's FPGA (DSP units). Again, this approach would limit the parallelism of the application. Sometimes, the control can involve integer divisions by a constant [15], it is then said to be quasi-affine. Zissulescu et al. [36] propose a set of recipes to get rid of integer divisions and modulos (emulated by integer divisions). Among the recipes, strenght reduction adds data dependences which may hinder parallelism. Also additional (but light) control is required. However, this is an important optimization which could be profitably used in complement to our approach. Zuo et al. [38] propose several source-level transformations to simplify the control for affine loop nests in front of an HLS tool. This approach is relevant when the outcome of a polyhedral optimization is a single unperfect loop nest with all the program statements. As stated in section 2, our front-end polyhedral optimizations splits the control between processes communicating through channels. This way, the control per process is simpler – a simple perfect loop nest, and does not require such optimizations. This approach complements ours: we are not optimizing the control structure, we derive an optimized hardware structure for a given affine control.

Piecewise affine functions received a lot of attention in the control community since Bemporad et al. [9] show that explicit solutions of Model Predictive Control (MPC) can be expressed with piecewise affine functions. Since then, many approaches were designed to map piecewise affine functions to FPGA using binary search trees [25, 30], lattice-based representation [24, 28], mix thereof [8] or hash functions [7]. Explicit solutions to MPC can be approximated to reduce the complexity of piece-wise affine controllers [19, 23]. Also, search algorithms can be simplified and give an approximate solution [6, 29]. In our case, this would not apply: control has to be exact, no approximation is possible. Tondel et al. [30] relies on a binary search tree to seek the right affine function to apply. The construction minimizes the depth of the tree by grouping in the same branch domains sharing the same affine function. Then, the circuit walks through the tree by using a sequential controller [25]. However, the sequential controller is not directly pipelinable. This leads to throughputs of several cycles per iteration, which is not desirable for our purpose. Also, storage resources are required to store the tree. This limits the duplication of these units, hence the parallelism of the final circuit. Lattice-based representation [20, 28, 33] is an alternative representation of piecewise affine functions as a min of max of elementary affine functions $f(x) = \min_{1 \leq i \leq k} \max_{j \in I_i} a_{ij}(x)$, each min term representing a convex part of f. Wen [34] provides an algorithm for generating the lattice based representation of an affine function. Then, the lattice-based representation can be mapped directly on the FPGA [24] or mixed with

an improved binary search tree [8]. The direct mapping leads to a throughput of 5 cycles per point on a Links Spartan 3 FPGA, which is not sufficient for our purpose (we expect 1 cycle per point). Also, it is not clear that the min/max representation would be more compact than our DAGs. Anyway, lattice-based representation assumes piecewise affine functions (hence continuous), which is generally not the case for affine control as explained in section 2. Bayat et al. [7] uses a hash function to locate the affine function to be applied. Basically, the function domain is subdivided in cells with a grid. The hash function maps each cell to the intersecting function clauses. Then, a few iteration finds out the relevant clauses. The trade-off is: the bigger is the cell, the smaller is the storage requirement, the bigger are the cycles per point (throughput). The throughput can only be increased at the price of a bigger storage. As mentioned previously, using storage resources of the FPGA is not desirable to implement affine control.

6 CONCLUSION

In this paper, we have proposed an efficient algorithm to compact a collection of affine constraints and expressions by exploiting semantic properties of addition and multiplication. The compaction is driven by a customizable cost function whose minimization ensure a proper usage of FPGA resources. The result is a DAG ready to be mapped on the target FPGA. Synthesis results on FPGA show that our approach complements very well the optimizations applied by Quartus, and can be used as a preprocessing step. We also show that, according to our cost model, our method is significantly better than the classical common subexpression factorization.

So far, the technique has been used to optimize the control at the process level, each process running in parallel. If we try to optimize the control involved in all processes as a single DAG, the control would be serialized and we would miss the benefit of parallelization. In the future, we plan to extend this technique to factorize the control common to processes without hindering the parallelism. Our method is bounded to affine control, but polyhedral control may include integer divisions. In addition to a preprocessing, new rules and patterns can be defined. Also, our method deals with constraints with inequalities only. When equalities are explicitly stated, other factorizations may apply. Finally, nothing forces strenght reductions, they occur only when the pool of affine expressions happens to allow it. We believe that special expressions could be systematically added to allow more strength reductions.

REFERENCES

[1] Christophe Alias, Fabrice Baray, and Alain Darte. 2007. Bee+Cl@k: An Implementation of Lattice-Based Array Contraction in the Source-to-Source Translator Rose. In *ACM Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*.

[2] Christophe Alias, Alain Darte, and Alexandru Plesco. 2013. Optimizing Remote Accesses for Offloaded Kernels: Application to High-Level Synthesis for FPGA. In *ACM SIGDA Intl. Conference on Design, Automation and Test in Europe (DATE'13)*. Grenoble, France.

[3] Christophe Alias, Bogdan Pasca, and Alexandru Plesco. 2012. FPGA-Specific Synthesis of Loop-Nests with Pipeline Computational Cores. *Microprocessors and Microsystems* 36, 8 (November 2012), 606–619.

[4] Christophe Alias and Alexandru Plesco. 2015. *Data-aware Process Networks.* Research Report RR-8735. Inria - Research Centre Grenoble – Rhône-Alpes. 32 pages. https://hal.inria.fr/hal-01158726

[5] Cédric Bastoul. 2003. Efficient Code Generation for Automatic Parallelization and Optimization. In *2nd International Symposium on Parallel and Distributed Computing (ISPDC 2003), 13-14 October 2003, Ljubljana, Slovenia.* 23–30.

[6] Farhad Bayat, Tor A Johansen, and Ali A Jalali. 2011. Combining truncated binary search tree and direct search for flexible piecewise function evaluation for explicit MPC in embedded microcontrollers. *IFAC Proceedings Volumes* 44, 1 (2011), 1332–1337.

[7] Farhad Bayat, Tor Arne Johansen, and Ali Akbar Jalali. 2011. Using hash tables to manage time-storage complexity in point location problem: Application to Explicit MPC. *Automatica* 47, 3 (2011), 571–577.

[8] Farhad Bayat, Tor Arne Johansen, and Ali Akbar Jalali. 2012. Flexible piecewise function evaluation methods based on truncated binary search trees and lattice representation in explicit MPC. *IEEE Transactions on Control Systems*

Technology 20, 3 (2012), 632–640.

[9] Alberto Bemporad, Manfred Morari, Vivek Dua, and Efstratios N. Pistikopoulos. 2002. The explicit linear quadratic regulator for constrained systems. *Automatica* 38, 1 (2002), 3 – 20. https://doi.org/10.1016/S0005-1098(01)00174-1

[10] Michaela Blott. 2016. Reconfigurable future for HPC. In *High Performance Computing & Simulation (HPCS), 2016 International Conference on*. IEEE, 130–131.

[11] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 101–113. https://doi.org/10.1145/1375581.1375595

[12] Pierre Boulet and Paul Feautrier. 1998. Scanning Polyhedra without Do-loops. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*. 4–9.

[13] Philippe Clauss. 1996. Counting Solutions to Linear and Nonlinear Constraints Through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs. In *Proceedings of the 10th international conference on Supercomputing, ICS 1996, Philadelphia, PA, USA, May 25-28, 1996*. 278–285.

[14] Steven Derrien, Alexandru Turjan, Claudiu Zissulescu, Bart Kienhuis, and Ed F Deprettere. 2008. Deriving efficient control in process networks with compaan/laura. *International Journal of Embedded Systems* 3, 3 (2008), 170–180.

[15] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53. https://doi.org/10.1007/BF01407931

[16] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International journal of parallel programming* 21, 6 (1992), 389–420.

[17] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*. 1581–1592.

[18] Al Geist and Daniel A Reed. 2015. A survey of high-performance computing scaling challenges. *International Journal of High Performance Computing Applications* (2015), 1094342015597083.

[19] BAG Genuit, Liang Lu, and WPMH Heemels. 2011. Approximation of PWA control laws using regular partitions: An ISS approach. *IFAC Proceedings Volumes* 44, 1 (2011), 4540–4545.

[20] Valentin V. Gorokhovik and Oleg I. Zorko. 1994. Piecewise affine functions and polyhedral sets. *Optimization* 31, 2 (1994), 209–221.

[21] R.L. Herman. 2014. Numerical Solution of 1D Heat Equation. Applied Analytical Methods, course notes. (Nov. 2014).

[22] Guillame Iooss, Sanjay Rajopadhye, and Christophe Alias. 2013. Semantic Tiling. In *Workshop on Leveraging Abstractions and Semantics in High-performance Computing (LASH-C'13)*. Shenzhen, China.

[23] Colin N Jones and Manfred Morari. 2010. Polytopic approximation of explicit model predictive controllers. *IEEE Trans. Automat. Control* 55, 11 (2010), 2542–2553.

[24] Macarena C Martínez-Rodríguez, Iluminada Baturone, and Piedad Brox. 2011. Circuit implementation of piecewise-affine functions based on lattice representation. In *Circuit Theory and Design (ECCTD), 2011 20th European Conference on*. IEEE, 644–647.

[25] Alberto Oliveri, Andrea Oliveri, Tomaso Poggi, and Marco Storace. 2009. Circuit implementation of piecewise-affine functions based on a binary search tree. In *Circuit Theory and Design, 2009. ECCTD 2009. European Conference on*. IEEE, 145–148.

[26] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/[cited July,]* (2012).

[27] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-based Data Reuse Optimization for Configurable Computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '13)*. ACM, New York, NY, USA, 29–38. https://doi.org/10.1145/2435264.2435273

[28] JM Tarela and MV Martinez. 1999. Region configurations for realizability of lattice piecewise-linear models. *Mathematical and Computer Modelling* 30, 11-12 (1999), 17–27.

[29] Petter Tondel, Tor Arne Johansen, and Alberto Bemporad. 2002. Computation and approximation of piecewise affine control laws via binary search trees. In *Decision and Control, 2002, Proceedings of the 41st IEEE Conference on*, Vol. 3. IEEE, 3144–3149.

[30] Petter Tøndel, Tor Arne Johansen, and Alberto Bemporad. 2003. Evaluation of piecewise affine control via binary search tree. *Automatica* 39, 5 (2003), 945–950.

[31] Alexandru Turjan. 2007. *Compiling Nested Loop Programs to Process Networks*. Ph.D. Dissertation. Universiteit Leiden.

[32] Sven Verdoolaege. 2015. Integer set coalescing. In *5th International Workshop on Polyhedral Compilation Techniques (IMPACT'15)*.

[33] Valentin V.Gorokhovik. 2007. Geometrical and analytical characterizations of piecewise affine mappings. *Proceedings of Institute Mathematics (The National Academy of Sciences of Belarus)* 15, 1 (2007), 22–32. in Russian.

[34] Chengtao Wen, Xiaoyan Ma, and B Erik Ydstie. 2009. Analytical expression of explicit MPC solution via lattice piecewise-affine function. *Automatica* 45, 4 (2009), 910–917.

[35] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. 2016. Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ACM, 326–331.

[36] Claudiu Zissulescu, Bart Kienhuis, and Ed Deprettere. 2005. Expression synthesis in process networks generated by LAURA. In *2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*. IEEE, 15–21.

[37] Julien Zory and Fabien Coelho. 1998. Using Algebraic Transformations to Optimize Expression Evaluation in Scientific Codes. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, Paris, France, October 12-18, 1998*. 376–384.

[38] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. 2013. Improving polyhedral code generation for high-level synthesis. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 1–10. https://doi.org/10.1109/CODES-ISSS.2013.6659002

Received May 2017; revised October 2017; accepted November 2017