Optimizing Affine Control with Semantic Factorizations

Christophe Alias, Alexandru Plesco
Optimizing Affine Control with Semantic Factorizations

Christophe Alias∗, Alexandru Plesco†

Project-Team Roma

Research Report n° 9034 — version 3 — initial version February 2017 — revised version December 2017 — 27 pages

Abstract: Hardware accelerators generated by polyhedral synthesis 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 report, we propose an algorithm 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.

Key-words: High-level synthesis, polyhedral compilation, affine control, FPGA

∗ CNRS/ENS-Lyon/Inria/UCBL/Université de Lyon
† XtremLogic SAS
Optimisation du contrôle affine avec des factorisations sémantiques

Résumé : Les accélérateurs matériels compilés par les algorithmes de synthèse polyédrique utilisent intensivement des expressions affines (fonctions affines par morceaux, polyèdres convexes) dans leur contrôle. Comme le contrôle est pipeliné, ces objets affines doivent être évalués en même temps pour différentes valeurs d’entrée, ce qui interdit une réutilisation aggressive des opérateurs. Dans ce rapport, nous proposons un algorithme pour factoriser une collection d’expressions affines sans empêcher le pipeline. Nos contributions sont (i) l’utilisation de factorisations exploitant les propriétés arithmétiques de l’addition et de la multiplication et (ii) une fonction de coût dont la minimisation assure une utilisation efficace des ressources FPGA. Notre algorithme est totalement paramtré par la fonction de coût, qui peut être adaptée à un FPGA cible donné. Les résultats expérimentaux montrent que notre algorithme complète avantageusement les optimisations RTL bas-niveau implantées dans les outils de synthèse industriels. En particulier, nous montrons comment notre algorithme réduit la taille du circuit en révélant des réductions de force cachées.

Mots-clés : Synthèse de circuit haut-niveau, compilation polyédrique, contrôle affine, FPGA
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 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 report, 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 [23] and control algorithms can be simplified [29] [6]. In our case, this would not apply: control has to be exact, no approximation is allowed.

In this report, 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 report is structured as follows. Section 2 gives a short introduction to polyhedral synthesis and introduces the concepts used in the remaining of the report. Section 3 presents our compaction algorithm. Section 4 presents the experimental results. Section 5 reviews the related work. Finally, Section 6 concludes this report 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 \( a \in \mathbb{R} \), the set \( H_{\geq}(a^*, a) = \{ x \in \mathbb{R}^n, a^*(x) \geq a \} \) is said to be a closed half-space. A convex polyhedron \( P \) is a finite intersection of closed half spaces: \( P = \cap_{i=1}^q H_{\geq}(a^*_i, a_i) \). If \( a^*_i(x) = \tau_i \cdot x \), \( A \) is the matrix whose rows are \( \tau_1, \ldots, \tau_q \) and \( b = (\alpha_1, \ldots, \alpha_q)^T \), the matrix representation of \( P \) is:

\[
P = \{ x \in \mathbb{R}^n, Ax \geq b \}
\]

The interior of \( P \) is the biggest open set int \( P \) included in \( P \). With the matrix representation, \( \text{int } P = \{ x \in \mathbb{R}^n, Ax > b \} \).

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

2.2 Piecewise affine functions

Given a domain \( D \subset \mathbb{R}^n \), a mapping \( \phi : D \to \mathbb{R}^p \) is said to be piecewise affine if there exists a subdivision of \( D \) in convex polyhedra \( D = P_1 \cup \ldots \cup P_q \) such that \( \text{int } P_i \cap \text{int } P_j = \emptyset \) for \( i \neq j \) and a collection of affine mappings \( u_i : \mathbb{R}^n \to \mathbb{R}^p \) for \( i = 1, \ldots, q \):

\[
\phi(x) = u_i(x) \quad \text{if } x \in P_i \quad \text{for } i = 1, \ldots, q
\]

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

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

\[
\hat{\phi}(x) = \hat{u}_i(x) \quad \text{if } x \in \hat{P}_i \quad \text{for } i = 1, \ldots, q
\]

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
for $i := 0$ to $N$
    init($u_0[i]$); // ●

// Heat-1D kernel
for $t := 0$ to $M - 1$
    for $i := 1$ to $N - 1$
        $u_1[i] := \alpha u_0[i - 1] + \beta u_0[i] + \alpha u_0[i + 1]$; // *
    for $i := 0$ to $N$
        if $i = 0$ then $u_0[i] := 0$; // ⊗
        if $1 \leq i \leq N - 1$ then $u_0[i] := u_1[i]$; // ×
        if $i = N$ then $u_0[i] := 0$; // ⊙
    for $i := 0$ to $N$
        use($u_0[i]$); // ◦

(a) Heat-1D kernel    (b) Data dependences    (c) Scheduling

Figure 1: Heat-1D kernel, execution trace (iteration domains) and data dependences

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 $u_0$. $\alpha$ 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 $(S, \vec{i})$. The set $D_S$ of iteration vectors is called the iteration domain of $S$. By construction, the iteration domain $D_S$ is always an integer polyhedron, hence the name of the framework. The original execution order is given by the lexicographic order $\ll$ over $D_S$, 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), 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 [15, 16, 11, 1, 2]. 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 \( (S, \vec{i}) \) to the assignment execution defining the read value \( h_{S_r}(\vec{i}) \). On the running example:

\[
h_{s,u0[i-1]}(t, i) = \begin{cases} 
  t = 0 : & (\bullet, i - 1) \\
  t \geq 1 \land i = 1 : & (\otimes, t - 1, 0) \\
  t \geq 1 \land i \geq 2 : & (\times, t - 1, i - 1)
\end{cases}
\]

Source functions are always integer piece-wise affine modulo the encoding of assignments \( \bullet, \otimes, \times \) 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 [1, 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 \( D_S \). 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 \( \theta_S \) which maps each execution \( (S, \vec{i}) \) 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 \( \theta_S(D_S) \) being vectors ordered by their lexicographic order. In a way, \( \theta_S : \mathbb{R}^n \rightarrow \mathbb{R} \) 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 \((S, i_1, \ldots, i_n) \) 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 \( \text{First}_S \), which issues the first iteration of \( S \) w.r.t \( \theta_S \) (initial state) and a function \( \text{Next}_S \) which maps each iteration of \( S \) to the next iteration of \( S \) to be executed w.r.t. \( \theta_S \) (transition function). On the running example, we would have:

\[
\text{First}_S(N, M) = \begin{cases} 
  N \geq 0 \land M \geq 0 : & (0, 0) \\
  i \leq N - 2 : & (t, i + 1)
\end{cases}
\]

\[
\text{Next}_S(t, i) = \begin{cases} 
  i = N - 1 \land t \leq M - 2 : & (t + 1, 0) \\
  i = N - 1 \land t = M - 1 : & \text{stop}
\end{cases}
\]

To improve reuse, affine scheduling is usually combined with affine partitioning (or tiling) [11]. Each relevant iteration domain \( D_S \) is partitioned into parallelepipeds by translating a collection of cutting hyperplanes \( \Phi_1, \ldots, \Phi_n \). Then, the new iteration domain \( D_S^P \) is indexed with vectors \((\Phi_1, i_1, \ldots, i_n), (\Phi_2, i_1, \ldots, i_n)\) being the coordinates of the partition containing the original iteration vector \((i_1, \ldots, i_n)\). Again, the resulting domain \( D_S^{P} \) is an integer polyhedron which can be scheduled thanks to affine scheduling. However, affine partitioning highly complicates the control on the generated program. Figure[1](c) gives an example of affine partitioning. \( D_s, D_x, D_\otimes \) and \( D_\times \) are partitioned with cutting hyperplanes \( \Phi^1 : t + i = 2\Phi_1 \) and \( \Phi^2 : t = 2\Phi_2 \) with partition coordinates \( \Phi_1 = 0, 3 \) and \( \Phi_2 = 0, 1 \). The affine schedule found is \( \theta_s(i) = (1, i), \theta_x(\Phi_1, \Phi_2, t, i) = \)

Inria
\[ (2, \Phi_1, \Phi_2, t+i, t, 0) \theta_\Box (\Phi_1, \Phi_2, t, i) = \theta_\Box (\Phi_1, \Phi_2, t, i) = \theta_\times (\Phi_1, \Phi_2, t, i) = (2, \Phi_1, \Phi_2, t+i, t, 1) \text{ and } \theta_\circ (i) = (3, i). \]

The final execution order is depicted with grey arrows. For the assignment \( \ast \), the functions First_\ast () and Next_\ast are:

First_\ast (N, M) = \{ N \geq 0 \land M \geq 0 : (0, 0, 0, 1) \}

Next_\ast (\Phi_1, \Phi_2, t, i) =

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 report 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 \( D = (N, E) \) can be modeled as a simple weighted sum:

\[ |D| = \sum_{n \in N} w(n) \cdot bw(n) \]

Where \( bw(n) \) denotes the bitwidth of the result computed by the operator \( n \) of the DAG and \( w(n) \) denotes, roughly, the number of building blocks required by \( n \) to compute 1 bit of result. \( bw \) is simply computed for each node of the DAG by a bottom-up application of the rules \( bw(x + y) = 1 + \max(bw(x), bw(y)) \) and \( bw(x \ast y) = bw(x) + bw(y) \) starting from the bitwidth.
of the input variables. Furthermore, \( w \) can be customized at will to fit a target FPGA. In the following, we will assume \( w(+) = 1 \) and \( w(*) = 100 \). 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) \( |a \cdot x| = 0 \) 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 \( a > 0 \), \( -a \cdot x = a \cdot \overline{x} + a \), where \( \overline{x} \) denotes the logic complement of \( x \). The cost is:

\[
| -a \cdot x| = bw(x) + (bw(a) + bw(x)) \cdot w(*) + (bw(a) + bw(x) + 1) \cdot w(+)
\]

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

\[
u = \sum_{i=0}^{n} a_i \cdot x_i + b \text{ where the } x_i \text{ are integer variables and the coefficients } a_i \text{ and } b \text{ are integer constants. In the worst case, the term } a_i \cdot x_i \text{ (or } b) \text{ with the largest bitwidth is evaluated first, each addition increasing the size of the result of 1 bit. Hence, the worst possible bitwidth for the result of } u \text{ is } bw_{\text{worst}}(u) = n - 1 + \max \{b \cup \{w(a_i) + w(x_i), i \in [1,n]\}\}. \text{ Therefore, an upper bound for } |u| \text{ is:}
\]

\[
|u| = n \cdot bw_{\text{worst}}(u) \cdot w(+) + \sum_{i=1}^{n} |a_i \cdot x_i|
\]

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 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 report 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 $G_r$. Basically, the nodes of $G_r$ are affine expressions and constraints, and an edge $u \xrightarrow{\Delta} v$ means that $v$ can be realized from $u$ with a cost of $\Delta$. Intuitively, a rooted path in $G_r$ 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 $v - u$: $\Delta = (1 + \max\{bw_{\text{worst}}(u), bw_{\text{worst}}(v - u)\}) \cdot w(+) + \lceil v - u \rceil$. We register this possible design choice to a realization graph $G_r$, 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 $\Delta$. 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 $v$.  

RR n° 9034
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 edge taking the output nodes of $u$ and $-1 - u - v$.
- The result of the edge 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 $-1 - u - v$: $\Delta = (1 + \max\{bw_{\text{worst}}(u), bw_{\text{worst}}(-1 - u - v)\}) \cdot w(+) + [-1 - u - v]$. We add a negation edge $u < 0 \xrightarrow{\Delta} v < 0$ to the realization graph.

Final Algorithm  Figure 3 depicts our algorithm to build the realization graph $G_r$ from a pool of expressions $E$ and constraints $C$. Expressions and constraints are inserted incrementally in the graph $G_r$ (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 $G_r$ (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 $\Delta$.

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 $G_r$ is inserted. This ensures a maximal subexpression factorization between the expressions of $E$ 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 $(1 + \max\{bw_{\text{worst}}(u), bw_{\text{worst}}(-1 - u - v)\}) \cdot w(+)$. 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 $G_r$ (lines 5–7). The expression $e$ is inserted as described above (line 6). Then, negation edges between $e < 0$ and constraint nodes of $G_r$ are added (line 7) by using procedure insert_neg_edge (lines 24–32). For each constraint node $u < 0$ of $G_r$ (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 $\phi(u, v)$ for $(1 + \max\{bw_{\text{worst}}(u), bw_{\text{worst}}(-1 - u - v)\}) \cdot w(+)$. 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.

Inria
Example  Consider the affine constraints \( \mathcal{C} \) depicted in the following table with the input bitwidths \( \text{bw}(i) = 2 \) and \( \text{bw}(j) = \text{bw}(k) = 8 \). \text{BUILD-REALIZATION}_G(\emptyset, \mathcal{C}) \) produces the graph depicted in figure 4(a) with node 2j depicted in white. Constraint factorization edges are dashed. Each edge is labeled by its cost \( \Delta \) 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.

<table>
<thead>
<tr>
<th>Id</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( i + 2j + k &lt; 0 )</td>
</tr>
<tr>
<td>2</td>
<td>( 5i + 2j + 3k &lt; 0 )</td>
</tr>
<tr>
<td>3</td>
<td>( 4i + 3j &lt; 0 )</td>
</tr>
<tr>
<td>4</td>
<td>( -5i - 3j - 1 &lt; 0 )</td>
</tr>
</tbody>
</table>

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 \( \Delta \). 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 state). \( u \) and \( v \) might also be realized from \( w = 2j \). In that case, we choose edges initial state \( \xrightarrow{0} w \), \( w \xrightarrow{15} u \) 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 state and including the nodes \( u \) and \( v \) to be realized. More generally, a valid realization of \( v \) is a path:

\[
\text{initial state} \xrightarrow{\Delta_1} u_1 \ldots \xrightarrow{\Delta_n} u_n \xrightarrow{\Delta} v
\]

Each \( u_j \) being realized from \( u_{j-1} \) at cost \( \Delta_j \). 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 \( u_n \):

\[
\text{initial state} \xrightarrow{\Delta_1} u_1 \ldots \xrightarrow{\Delta_n} u_n \xrightarrow{\Delta} \neg v
\]

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.

**Definition 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 state, and \( \mathcal{T} \) spans \( \mathcal{E} \cup \mathcal{C} \).
2. No useless common subexpression is computed: the leaves of \( \mathcal{T} \) belong to \( \mathcal{E} \cup \mathcal{C} \).
BUILD_REALIZATION_GRAPH(ℰ, ℋ)

\[ \mathcal{G}_r := \text{empty} \cdot \text{graph}(); \]

\textbf{for each} expression \( e \in \mathcal{E} \)

\[ \text{INSERT}(e); \]

\textbf{for each} constraint \( (e < 0) \in \mathcal{H} \)

\[ \text{INSERT}(e); \text{mark}(e); // e < 0 \]

\text{INSERT_NEG_EDGE}(e); \]

\textbf{Add node initial_node to} \( \mathcal{G}_r \); \]

\textbf{for each} node \( u \in \mathcal{G}_r - \{ \text{initial_node} \} \)

\[ \text{Add edge} \text{ initial_node} \to u \to \mathcal{G}_r; \]

\text{INSERT}(e)

\textbf{for each} node \( n \in \mathcal{G}_r \)

\[ \text{INSERT_EDGE}(\text{common_sub_expr}(e, n)); \]

\[ \text{INSERT_EDGE}(e); \]

\text{INSERT_EDGE(v)}

\textbf{if} \( v \in \mathcal{G}_r \), return;

\[ \text{Add node} \ v \to \mathcal{G}_r; \]

\textbf{for each} node \( u \in \mathcal{G}_r \)

\[ \text{Add edge} \ u \to \mathcal{G}_r; \]

\textbf{if} \( \phi(u, v) + |v - u| < |v| // v \text{ from } u \)

\[ \text{Add edge} \ u \xrightarrow{\phi(u, v)+|v-u|} v \to \mathcal{G}_r; \]

\text{//Symmetric case}

\textbf{if} \( \phi(v, u) + |u - v| < |u| \)

\[ \text{Add edge} \ v \xrightarrow{\phi(v, u)+|u-v|} u \to \mathcal{G}_r; \]

\text{INSERT_NEG_EDGE(v)}

\textbf{for each} marked node \( u \in \mathcal{G}_r // u < 0 \)

\[ \text{if} \exists u \xrightarrow{\Delta} v \in \mathcal{G}_r \text{ continue}; \]

\[ \text{if} \psi(u, v) + |1 - u - v| < |v| \]

\[ \text{Add edge} \ u \xrightarrow{\psi(u, v)+|1-u-v|} v \to \mathcal{G}_r; \]

\text{//Symmetric case}

\[ \text{if} \exists u \xrightarrow{\Delta} v \in \mathcal{G}_r \text{ continue}; \]

\[ \text{if} \psi(v, u) + |1 - v - u| < |u| \]

\[ \text{Add edge} \ v \xrightarrow{\psi(v, u)+|1-v-u|} u \to \mathcal{G}_r; \]

\text{Figure 3: Algorithm for constructing the realization graph} \( \mathcal{G}_r \)

\[ \text{Inria} \]
Figure 4: (a) Realization graph $G_r$ obtained from $C$, (b) Resource-efficient realization tree $T$ in $G_r$ (in red), (c) Resource-efficient DAG from realization tree $T$ in $G_r$, (d) Recursive compaction of fresh expressions $E_{\text{new}}$
3. Negation edges are terminal.

In other words, a realization is a particular spanning tree of $G_r$. 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 the weights $\Delta$ on its edges. Hence, finding an efficient realization amounts to compute a minimal spanning tree of $G_r$, under the constraints specified in definition 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 $G_r$ 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. 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 $T$ obtained from the realization graph of figure 3(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 $T$ (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 $G_r$. The inputs are: the realization tree $T$ and the set of expressions $E$ 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 $E$ and $C$ to there implementation in the DAG. The algorithm is a recursive depth traversal of $T$. Each time an edge of $T$ is traversed, the DAG is updated accordingly. Two additional inputs are used to traverse $T$ ($t$) 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 $E$, 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_REALIZATION_TREE(G_r,E,C)

1. BUILD_DAG is called with $t = \text{initial\_node}$ and $d = \text{null}$. A DAG $\text{dag}(u)$ 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 $v$ should be updated in that place (lines 16 and 20).

Rules for expression and constraint factorization produces a pool of new expressions $E_{\text{new}}$ 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 $E_{\text{new}}$. The output of the new DAG are used in the current DAG in place of the expressions of $E_{\text{new}}$. 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 $T$. Prior to building the DAG, the expressions $E_{\text{new}}$ 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 $4i + 3j \xrightarrow{12} -5i - 3j - 1$. The recursive call on $E_{\text{new}}$ 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
Figure 6: Algorithm for building a DAG from a realization $\mathcal{T}$

```
BUILD_DAG(t,d)
//Link DAG nodes to inputs $\mathcal{E}$ and $\mathcal{C}$
if ($t = e \in \mathcal{E}$) node[e] := d;
if ($t < 0 = c \in \mathcal{C}$)
    Add edges for ineq_node := d < 0;
    node[c] := ineq_node;

//Base case
if $t = \text{initial}_\text{node}$
    for each edge $t \xrightarrow{\Delta} u \in \mathcal{T}$
        BUILD_DAG(u,dag(u))
    return;

//Expression factorization
for each edge $t \xrightarrow{\Delta} u \in \mathcal{T}$
    BUILD_DAG(u, +(d, $\mathcal{E}(V(u) - V(t))$))

//Constraint factorization
for each edge $t \xrightarrow{\Delta} \neg u \in \mathcal{T}$
    if ($V(t) + V(u) = -1$) //direct negation?
        Add edges for neg_node := $\neg(d < 0)$;
        node[u < 0] := neg_node;
    else
        Add edges for:
        neg_node := $\neg(+d, \mathcal{E}(-1 - V(t) - V(u)) < 0)$;
        node[u < 0] := neg_node;
```
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 total number of affine constraints. #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. 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.

RR n° 9034
<table>
<thead>
<tr>
<th>Kernel</th>
<th>#dags</th>
<th>#C</th>
<th>#E</th>
<th>SEM+Quartus ALM</th>
<th>Quartus ALM</th>
<th>Regs</th>
<th>Regs</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>2mm</td>
<td>15</td>
<td>250</td>
<td>161</td>
<td>1011 612</td>
<td>1430 596</td>
<td>29%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3mm</td>
<td>18</td>
<td>369</td>
<td>206</td>
<td>1775 946</td>
<td>2528 922</td>
<td>30%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>atax</td>
<td>12</td>
<td>134</td>
<td>83</td>
<td>628 328</td>
<td>900 321</td>
<td>30%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bicg</td>
<td>11</td>
<td>112</td>
<td>69</td>
<td>500 278</td>
<td>715 278</td>
<td>30%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>correlation</td>
<td>27</td>
<td>356</td>
<td>205</td>
<td>1609 1129</td>
<td>2567 909</td>
<td>37%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>covariance</td>
<td>16</td>
<td>243</td>
<td>143</td>
<td>1221 708</td>
<td>1872 618</td>
<td>35%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>doitgen</td>
<td>9</td>
<td>145</td>
<td>124</td>
<td>393 280</td>
<td>607 268</td>
<td>35%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ftdt-2d</td>
<td>13</td>
<td>502</td>
<td>167</td>
<td>2339 1713</td>
<td>3293 1603</td>
<td>29%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gemm</td>
<td>10</td>
<td>125</td>
<td>93</td>
<td>672 270</td>
<td>851 270</td>
<td>21%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gemver</td>
<td>20</td>
<td>187</td>
<td>137</td>
<td>847 459</td>
<td>1102 438</td>
<td>23%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gesummv</td>
<td>14</td>
<td>95</td>
<td>84</td>
<td>456 245</td>
<td>549 227</td>
<td>17%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>heat-3d</td>
<td>8</td>
<td>734</td>
<td>175</td>
<td>3545 1194</td>
<td>5667 2559</td>
<td>37%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jacobi-1d</td>
<td>8</td>
<td>134</td>
<td>64</td>
<td>628 556</td>
<td>912 520</td>
<td>31%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jacobi-2d</td>
<td>8</td>
<td>370</td>
<td>111</td>
<td>1660 1204</td>
<td>2547 1144</td>
<td>35%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lu</td>
<td>7</td>
<td>213</td>
<td>87</td>
<td>1116 666</td>
<td>1469 628</td>
<td>24%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mvt</td>
<td>11</td>
<td>118</td>
<td>70</td>
<td>550 290</td>
<td>758 290</td>
<td>27%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>seidel-2d</td>
<td>5</td>
<td>226</td>
<td>63</td>
<td>1161 1464</td>
<td>1758 1291</td>
<td>34%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>symm</td>
<td>13</td>
<td>213</td>
<td>116</td>
<td>1011 471</td>
<td>1540 465</td>
<td>34%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>syrk</td>
<td>10</td>
<td>135</td>
<td>90</td>
<td>721 290</td>
<td>944 281</td>
<td>24%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trisolv</td>
<td>9</td>
<td>118</td>
<td>81</td>
<td>636 246</td>
<td>828 246</td>
<td>23%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trmm</td>
<td>8</td>
<td>118</td>
<td>79</td>
<td>549 262</td>
<td>806 253</td>
<td>32%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Inria
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 ∧ 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 \( u \rightarrow v \) 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 \( u \rightarrow v \) used by several terms \( w_i \). Factorizations \( v \rightarrow w_i \) 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. \(-1 \times c, c - 1, t - 1\) 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 $v \rightarrow w_i = v + \lambda$ 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 $E_{\text{new}}$ 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 applied, producing fresh expressions $E_{\text{new}}$, in turn optimized with our algorithm. Quarter of the kernels even require two recursive calls: the semantic factorizations required by $E_{\text{new}}$ produce fresh expressions $E_{\text{new,new}}$ 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.
Table 2: Semantic factorizations: detailed results

<table>
<thead>
<tr>
<th>Kernel</th>
<th>#dags</th>
<th>#C</th>
<th>#E</th>
<th>#arcs</th>
<th>#c-arcs</th>
<th>#sem</th>
<th>#pow2</th>
<th>#c-arcs</th>
<th>#sem</th>
<th>#pow2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2mm</td>
<td>15</td>
<td>250</td>
<td>161</td>
<td>164</td>
<td>138</td>
<td>51</td>
<td>15</td>
<td>26</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>3mm</td>
<td>18</td>
<td>369</td>
<td>206</td>
<td>293</td>
<td>270</td>
<td>67</td>
<td>21</td>
<td>23</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>atax</td>
<td>12</td>
<td>134</td>
<td>83</td>
<td>83</td>
<td>68</td>
<td>27</td>
<td>7</td>
<td>15</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>bicg</td>
<td>11</td>
<td>112</td>
<td>69</td>
<td>63</td>
<td>52</td>
<td>23</td>
<td>6</td>
<td>11</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>correlation</td>
<td>27</td>
<td>356</td>
<td>205</td>
<td>208</td>
<td>187</td>
<td>65</td>
<td>21</td>
<td>21</td>
<td>18</td>
<td>12</td>
</tr>
<tr>
<td>covariance</td>
<td>16</td>
<td>243</td>
<td>143</td>
<td>153</td>
<td>142</td>
<td>46</td>
<td>14</td>
<td>11</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>doitgen</td>
<td>9</td>
<td>145</td>
<td>124</td>
<td>73</td>
<td>57</td>
<td>34</td>
<td>5</td>
<td>16</td>
<td>15</td>
<td>7</td>
</tr>
<tr>
<td>fdtd-2d</td>
<td>13</td>
<td>502</td>
<td>167</td>
<td>530</td>
<td>507</td>
<td>94</td>
<td>62</td>
<td>23</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>gemm</td>
<td>10</td>
<td>125</td>
<td>93</td>
<td>68</td>
<td>52</td>
<td>34</td>
<td>3</td>
<td>16</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>gemver</td>
<td>20</td>
<td>187</td>
<td>137</td>
<td>91</td>
<td>77</td>
<td>44</td>
<td>13</td>
<td>14</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>gesummv</td>
<td>14</td>
<td>95</td>
<td>84</td>
<td>40</td>
<td>28</td>
<td>22</td>
<td>5</td>
<td>12</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>heat-3d</td>
<td>8</td>
<td>734</td>
<td>175</td>
<td>1025</td>
<td>996</td>
<td>91</td>
<td>54</td>
<td>29</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>jacobi-1d</td>
<td>8</td>
<td>134</td>
<td>64</td>
<td>104</td>
<td>94</td>
<td>25</td>
<td>13</td>
<td>10</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>jacobi-2d</td>
<td>8</td>
<td>370</td>
<td>111</td>
<td>407</td>
<td>386</td>
<td>53</td>
<td>35</td>
<td>21</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>lu</td>
<td>7</td>
<td>213</td>
<td>87</td>
<td>177</td>
<td>167</td>
<td>28</td>
<td>15</td>
<td>10</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>mvt</td>
<td>11</td>
<td>118</td>
<td>70</td>
<td>69</td>
<td>60</td>
<td>25</td>
<td>10</td>
<td>9</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>seidel-2d</td>
<td>5</td>
<td>226</td>
<td>63</td>
<td>277</td>
<td>270</td>
<td>39</td>
<td>18</td>
<td>7</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>symm</td>
<td>13</td>
<td>213</td>
<td>116</td>
<td>129</td>
<td>115</td>
<td>43</td>
<td>19</td>
<td>14</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>syrk2</td>
<td>10</td>
<td>135</td>
<td>90</td>
<td>74</td>
<td>69</td>
<td>27</td>
<td>12</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>syrk</td>
<td>9</td>
<td>118</td>
<td>81</td>
<td>66</td>
<td>61</td>
<td>24</td>
<td>9</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>trisolv</td>
<td>9</td>
<td>93</td>
<td>56</td>
<td>58</td>
<td>51</td>
<td>19</td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>trmm</td>
<td>8</td>
<td>118</td>
<td>79</td>
<td>62</td>
<td>57</td>
<td>26</td>
<td>12</td>
<td>5</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

compared to the usual requirements of circuit synthesis. Table 3. provides the execution time in seconds for the construction of the realization graph $G_r$ (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 $G_r$, 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.

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
Table 3: Recursive calls and execution time

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

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 modulus (emulated by integer divisions). Among the recipes, strength 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 [30, 25], lattice-based representation [28, 24], mix thereof [8] or hash functions [7]. Explicit solutions to MPC can be approximated to reduce the complexity of piece-wise affine controllers [25, 19]. Also, search algorithms can be simplified and give an approximate solution [29, 6]. 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 [28, 20, 33] is an alternative representation of piecewise affine functions as a min of max of elementary affine functions \( f(x) = \min_{1 \leq i \leq k} \max_{j \in \mathcal{I}_i} a_{ij}(x) \), 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 report, 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 strength 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


Contents

1 Introduction 3

2 Preliminaries 4
   2.1 Convex polyhedra 4
   2.2 Piecewise affine functions 4
   2.3 Polyhedral synthesis 4

3 Our Algorithm 7
   3.1 Cost Model 7
   3.2 Motivating Examples 8
   3.3 Realization Graph 9
   3.4 Finding an Efficient Realization 11
   3.5 Building the DAG 14

4 Experimental Evaluation 15
   4.1 Experimental setup 17
   4.2 Experimental results 17
      4.2.1 Synthesis results 17
      4.2.2 Distribution of semantic factorizations 19
      4.2.3 Recursive calls and execution time 20

5 Related Work 21

6 Conclusion 23