Optimizing Affine Control with Semantic Factorizations

Christophe Alias, Alexandru Plesco

▶ **To cite this version:**

Christophe Alias, Alexandru Plesco. Optimizing Affine Control with Semantic Factorizations. [Research Report] RR-9034, INRIA Grenoble - Rhone-Alpes. 2017, pp.24. hal-01470873v1

HAL Id: hal-01470873

https://inria.hal.science/hal-01470873v1

Submitted on 17 Feb 2017 (v1), last revised 24 Nov 2017 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

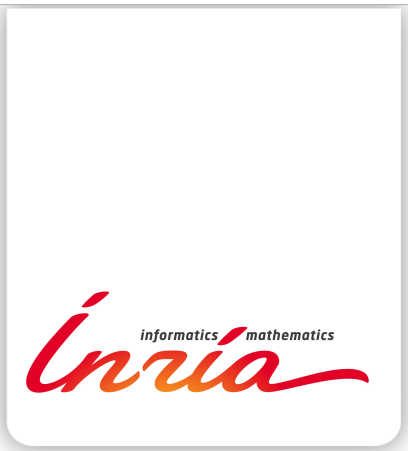
L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Optimizing Affine Control with Semantic Factorizations

Christophe Alias, Alexandru Plesco

RESEARCH REPORT
N° 9034
February 2017
Project-Team Roma

ISSN 0249-6399 ISRN INRIA/RR--9034--FR+ENG



Optimizing Affine Control with Semantic Factorizations

Christophe Alias*, Alexandru Plesco†

Project-Team Roma

Research Report n° 9034 — February 2017 — 24 pages

Abstract: 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 to be evaluated at the same time for different values, which forbids aggressive reuse of operators. In this report, 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 applications show a significant improvement compared to traditionnal common subexpression factorization. As a bonus, the optimization gain is statistically more regular compared to common subexpression elimination.

Key-words: High-level synthesis, FPGA, Polyhedral Synthesis

* Inria/ENS-Lyon/UCBL/CNRS
† XtremLogic SAS

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Optimizing Affine Control with Semantic Factorizations

Résumé : Les accélérateurs matériels générés par synthèse de circuit polyédrique utilisent intensivement des expressions affines (functions affines et polyèdres convexes) dans leur circuit de contrôle. Pour garantir la bande passante, ces expressions doivent être évaluées en même temps pour différentes valeurs, ce qui interdit toute réutilisation aggressive des opérateurs. Dans ce rapport, nous proposons un algorithme pour factoriser efficacement une collection d'expressions affines sans interdire le pipeline. Nos contribution sont (i) l'utilisation de factorisations sémantiques exploitant les propriétés arithmétiques de l'addition et de la multiplication et (ii) l'utilisation d'une fonction de coût générique dont la minimisation assure une utilisation réduite des blocs reconfigurables d'un FPGA. Notre algorithme est totalement paramétré par la fonction de coût, qui peut être modifiée selon l'architecture reconfigurable ciblée. Les résultats expérimentaux sur un nombre important d'applications montrent une amélioration significative de coût comparé à la traditionnelle factorisation de sous-expressions communes. Comme bonus, le gain en coût obtenu avec notre algorithme est statistiquement plus régulier que celui obtenu par factorisation de sous-expressions communes.

Mots-clés : Synthèse de circuits haut-niveau, FPGA, synthèse polyédrique

1 Introduction

Since the end of Dennard scaling, computer architects are striving to build energy efficient computers. The trend is to trade genericity for energy efficiency by using specialized hardware accelerators such as GP-GPU or Xeon-Phi [17] to quote a few. Recently, reconfigurable FPGA circuits [9] have appear to be a competitive alternative [28]. With FPGAs, the program is the circuit: genericity is ultimately trade 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 [16]. 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 [10], scheduling pipelined circuits [3], sizing optimally the buffers [1] or tuning I/O requirements to fit memory bandwidth [2] to quote a few. Polyhedral analysis are used successfully in high-level circuit synthesis [4, 25]. 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 lock in polyhedral synthesis.

Pretty few approaches address affine control synthesis in the context of polyhedral circuit synthesis. With Compaan/Laura, the control frequently executed is synthesized as a DAG with common subexpression factorization [13]. Then, the control less frequently executed is left to a sequential controller. This generates bubbles at each start of the innermost loop. Since loops are usually restructured in such a way that innermost loops have often a few iterations [10], this slightly limits the throughput of the controller. Also, the sequential controller requires storage resources to implement the ROM. As storage resources are limited on FPGA, this approach limits the control synthesizable, hence the parallelism of the circuit. In contrast, piecewise affine function mapping has received a lot of attention in control theory since the solution of linear and quadratic constrained finite time optimal control (CFTOC) problems was shown to be a piecewise affine function [8]. Many approaches exists to map PWA functions into FPGA using binary search trees [24, 21], lattice-based representation [23, 20], mix thereof [7] or hash functions [6]. However, most of these approaches rely on a sequential controller which is not directly pipelineable. This leads to throughputs of several cycles per iteration, which is not desirable for our purpose. Also, (rare) storage resources are often used, which limits the duplication of these units, hence the parallelism of the final circuit. Consequently, these approaches cannot be used profitably for our purpose.

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 always performs better than common subexpression elimination. As a bonus, the compaction gain is statistically much more regular.

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 polyehdra, 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 halfspaces: $\mathcal{P} = \cap_{i=1}^{q} H_{\geq}(a_i^*, \alpha_i)$. If $a_i^*(x) = \tau_i \cdot x$, $\mathbf{A}$ is the matrix whose lines 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 continous. 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 [23], may no longer apply.

2.3 Polyhedral synthesis

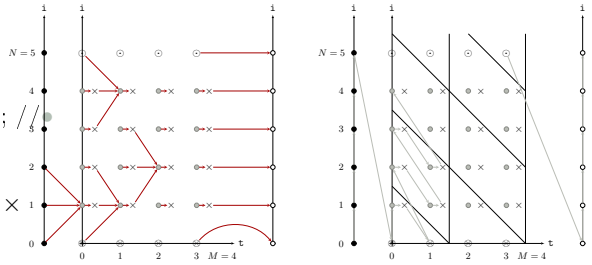
A parallelizing compiler analyzes the input program and maps the computation to a parallel architecture. The new execution order must reproduce the original computation: each operation

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 [16]. 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 [19] 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 assignement 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 *lexigraphic 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), 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 [14, 15, 10, 1, 2]. For instance, array dataflow analysis [14] 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:

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

Source functions are always *integer piece-wise affine* modulo the encoding of assignements $\bullet, \otimes, \times$ with integers and the completion 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 assignement and to handle synchronizations and communications between parallel units [4, 25]. The complexity of the source function $h_{S,r}$ (number of clauses, number of affine contraints 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 with the lexicographic order. In a way, $\theta_S : \mathbb{R}^n \to \mathbb{R}^p$ 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 criteria to maximise parallelism is to minimise p, so a maximum number of operations will share the same date [15] (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 developped [5, 11]. The best approach for HLS is to produce a control automaton per assignement S which issues a new iteration vector i of S at each clock cycle [11]. Two integer piecewise affine functions are required. A function First_S, which issue 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:

$$\text{First}_\bullet(N,M) = \left\{ \begin{array}{rl} N \geq 0 \wedge M \geq 0: & (0,0) \end{array} \right.$$

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

To improve reuse, affine scheduling is usually combined with affine partitionning (or tiling) [10]. Each relevant iteration domain $\mathcal{D}_S$ is partitionned into parallelepipeds by translating a collection of cutting hyperplans $\phi_S^1, \ldots, \phi_S^n$. 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 polyhedon which can be scheduled thanks to affine scheduling. However, affine partitionning highly complexifies the

control on the generated program. Figure 1.(c) gives an example of affine partitionning. $\mathcal{D}_\bullet$, $\mathcal{D}_\times$, $\mathcal{D}_\otimes$ and $\mathcal{D}_\odot$ are partitionned 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:

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

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

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 function, 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

We want to compile affine expressions/constraints to a design DAG such that resource usage is minimized. Resource usage can be assessed by counting the operators involved in the DAG: adders, multipliers (by an integer constant) and shifters (multiplication by a power of 2). We get a resource vector $(\#k \times ., \#. + ., \#2^n \times .)$. Then, the final cost is a weighted sum of the coordinates. Here, we chose $\mathbf{w}_{.+.} = 1$, $\mathbf{w}_{k\times.} = 10$, $\mathbf{w}_{2^n\times.} = 1$. With that choice, our algorithm will tend to decompose affine expressions with multiplications by a power of 2. Notice that the cost function is a parameter of our algorithm. It could perfectly be refined/redefined to fit a specific target. In the following, $|u|$ denotes the cost of an affine expression u.

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

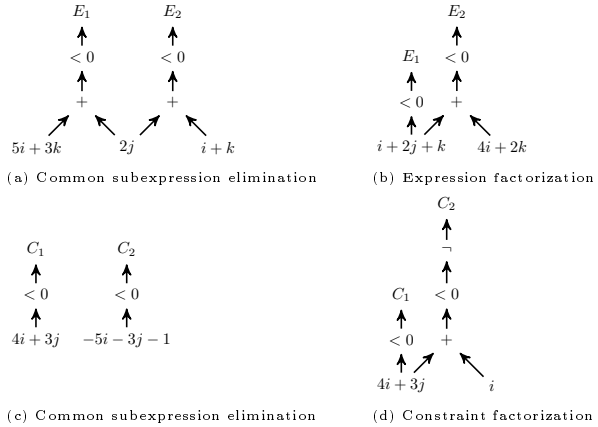


Figure 2: Affine expression and constraint factorization

$E_2 = E_1 + 4i + 2k$. 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 prefered to constraint factorizations.

This report proposes a unified way to represent the possible sequence of factorization 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 which 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: $\mathbf{w}_+ + |v - u|$. 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 (here $\mathbf{w}_+ + |v - u|$). 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 prefered to design v.

Constraint Factorization Given a DAG node computing $u < 0$, 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: $\mathbf{w}_+ + | - 1 - u - v|$. We add a *negation edge* $u < 0 \xrightarrow{\Delta}_{\neg} v < 0$ to the realization graph, with $\Delta = \mathbf{w}_+ + | - 1 - u - v|$.

Final Algorithm Figure 3 depicts our algorithm to build the realization graph $\mathcal{G}_r$ from a pool of expressions $\mathcal{E}$ and constraints $\mathcal{C}$. 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 labelled 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 $\mathcal{C}$. 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 $u = 3i + j + 4k$, $v = 3i + j + k$ 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). As seen above, the additional cost of computing expression v from u is $\Delta = \mathbf{w}_+ + |v - u| < |v|$. 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. Cases with expression factorization edge are prefered, 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 $\mathcal{C}$ depicted in the following table. BUILD_REALIZATION_GRAPH($\emptyset$,$\mathcal{C}$) 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 labelled 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 3 (edge from initial_state), or from subexpression $2j$ with a cost 2. In turn, $2j$ can serve as a basis to realize other expressions as constraint 3 with cost 3 (edge from $2j$ to $4i + 3j$). Constraint 4 can be used to realize constraint 3 thanks to a negation factorization of cost 1. If so, expression of constraint 3 could not be used to realize constraint 2 as expression factorization would no longer be possible. 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.

$\mathcal{C}$:

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 Optimal 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. Hence, 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 will not be available. 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  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 w_+ + |v − u| < |v| //v from u
20       Add edge u --w_+ + |v−u|--> v to G_r;
21     //Symmetric case
22     if w_+ + |u − v| < |u|
23       Add edge v --w_+ + |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 w_+ + | − 1 − u − v| < |v|
28       Add edge u --w_+ + |−1−u−v|-->¬ v to G_r;
29     //Symmetric case
30     if ∃v --Δ--> u ∈ G_r continue;
31     if w_+ + | − 1 − v − u| < |u|
32       Add edge v --w_+ + |−1−v−u|-->¬ u to G_r;
```

Figure 3: Algorithm for constructing the realization graph $\mathcal{G}_r$

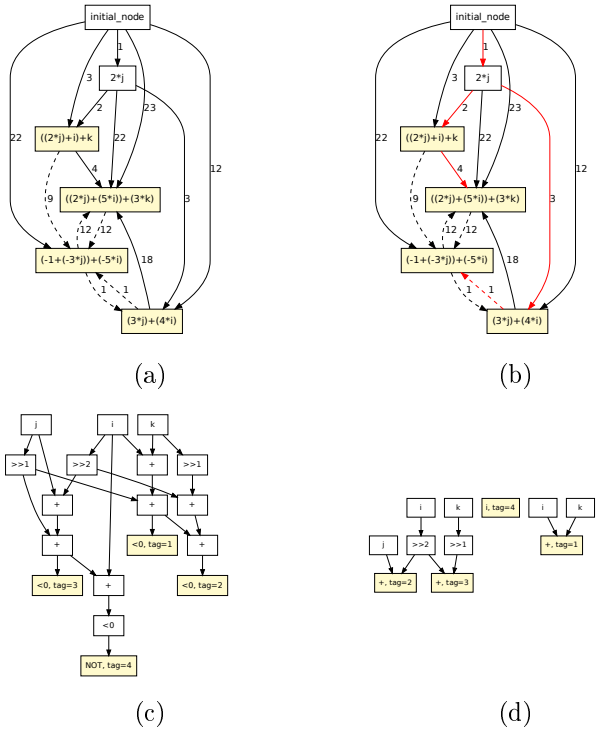


Figure 4: (a) Realization graph $\mathcal{G}_r$ obtained from $\mathcal{C}$, (b) Resource-efficient realization tree $\mathcal{T}$ in $\mathcal{G}_r$ (in red), (c) Resource-efficient DAG from realization tree $\mathcal{T}$ in $\mathcal{G}_r$, (d) Recursive compaction of fresh expressions $\mathcal{E}_{\text{new}}$

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 the weights Δ on its edges. Hence, finding an optimal realization amounts to compute a minimal spanning tree of $\mathcal{G}_r$, 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 $\mathcal{G}_r$ 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 mentionned 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 $\mathcal{T}$ 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 (11) is greatly improved compared to direct realization (60) (only arcs from initial_node, no factorization at all) and compared to common subexpression factorization (50) (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 2 expression factorizations (labelled by cost 2 and 3), and 1 constraint factorization (dashed edge labelled by 1). Among the two expression factorization 1 is a subexpression factorization (labelled by 2), 1 is not (labelled by 3). The latter (labelled by 3) is said to be *semantic*: it is obtained by playing on semantic properties of addition and multiplication and could not be found by subexpression factorization. Constraint factorization (dashed edge labelled by 1) 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 $\mathcal{C}$ 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 $\mathcal{C}$ 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

```
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;
```

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

accordingly. Two additionnal 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 $c \in \mathcal{C}$, 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 mope 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 build the DAG, the expressions $\mathcal{E}_{\text{new}}$ are collected and compacted with a recursive call. They are: $i + k$ from edge $2j \xrightarrow{2} i + 2j + k$, $4i + j$ from edge $2j \xrightarrow{3} 4i + 3j$, $4i + 2k$ from edge $i + 2j + k \xrightarrow{4} 5i + 2j + 3k$ and i from edge $4i + 3j \xrightarrow{1}_{\neg} -5i - 3j - 1$. 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.(b). Again, the nodes has been tagged for the sake of clarity. Here, the tags are the ranks of constraints $\mathcal{C}$ given in section 3.3.

```
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;
```

Figure 6: Algorithm for building a DAG from a realization $\mathcal{T}$

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 discusses the results obtained compared to an extended version of common subexpression factorization.

4.1 Experimental setup

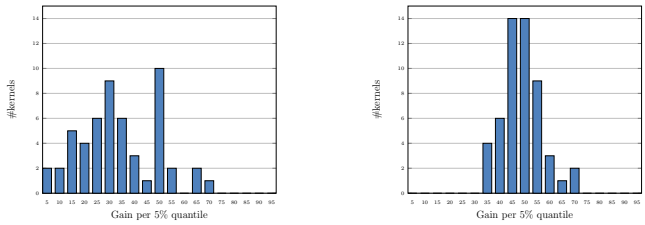
We have applied our algorithm to simplify the affine control generated for the kernels of the benchmark suite PolyBench/C v3.2 [22]. Figure 9 depicts the kernels and the results obtained. Each kernel comes with two versions: a naive version (base) and an optimized version (tiled). Some kernels could not be optimized, they come only with the base version. The optimized version has been obtained with the pluto parallelizer tool [10]. Mainly, the loops are tiled as much as possible, while pushing as much data dependence as possible in the innermost loops. As a result, outermost loops are likely to be parallel, while innermost loops carry a good locality. This is typically the kind of schedule used to reduce I/O downto the available bandwidth [2].

For each kernel, a DPN process network is generated using the Dcc tool [4]. Then, for each process, we apply our algorithm to simplify the affine control (control automaton, mux/demux). Figure 9 presents the sum of the criteria collected for each process. $\#\mathcal{C}$ is the total number of affine constraints. $\#\mathcal{E}$ is the total number of affine expressions the cost. NON-OPT is the cost of a direct implementation, no factorization at all. OPT is the cost after applying our algorithm.

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. We want to make sure that (i) our approach performs better than purely non-semantic factorizations and that (ii) among the factorizations applied, a substantial number are semantic. To do so, we run a modified version of our algorithm where the realization graph $\mathcal{G}_r$ contains only non-semantic arcs. Expression factorizations $u \xrightarrow{\Delta} v$ are such that u is a subterm of v. Constraint factorization are clearly semantic and should be excluded. However, we add the recognition of negations. This way, we emulate an (extended) common-subexpression factorization algorithm which always find the best common-subexpression factorization and which check negations. The cost of the resulting non-semantic optimization is given in column OPT-NO-SEM. To assess point (ii), we count the semantic arcs chosen by BUILD_REALIZATION_TREE in our original algorithm. For expression factorization, column #e-arc gives the total number of arcs chosen. Among these arcs, column #sem-e-arcs gives the total number of semantic arcs. Similarly, we count constraint factorizations (#c-arcs) and constraints factorization which are not a pure negation (#sem-c-arcs). The latter will be refered as semantic constraint factorization. Figure 10.(a) compare graphically the results of common-subexpression factorization (black bar) and our algorithm (white bar) normalized w.r.t. NON-OPT-COST. The ratio cost(opt)/cost(non-opt) will be refered as gain. (b) shows graphically the ratio (%) #sem-e-arcs/#e-arcs and (c) shows graphically the ratio (%) #sem-c-arcs/#c-arcs.

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 shows that our approach always perform better than



(a) Common subexpression factorization (b) Our algorithm

Figure 7: Distribution of gains (%)

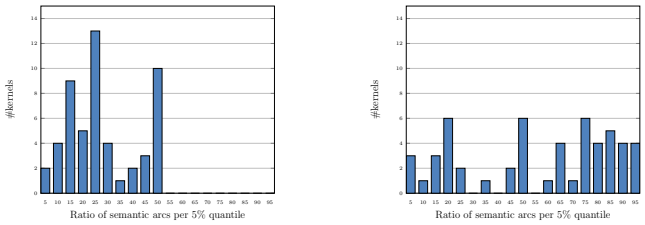
our extended version of common subexpression factorization. Then, Section 4.2.2 analyse the occurence of semantic factorizations over all the factorizations applied. In particular, we show that semantic factorizations are a significant phenomenon. Finally, Section 4.2.3 show how often our algorithm is applied recursively and Section 4.2.4 presents the execution times.

4.2.1 Comparison with common subexpression factorization

Experimental results show that our approach always performs better than common subexpression factorization, except for 3 kernels (lu-base, cholesky-base, nussinov-base) where gains are identical. This is a quite natural since our algorithm subsumes common-subexpression eliminatation as explained in the previous section. Figure 7 depicts the distribution of the gains w.r.t the naive version, per quantile of 5%, for common-subexpression factorization (a) and our approach (b). For instance, the first bar of (a) means that 2 kernels have a gain in $[0\%, 5\%[$. Also, the first bar of (b) means that 4 kernels have a gain in $[30\%, 35\%[$. For common-subexpression elimination (a), the minimum gain is 4%, the average gain is 32%, the maximum gain is 67% and the standart deviation is 13%. For our approach, the dispersion is tighter and the average gain is higher: the minimum gain is 32%, the average gain is 47%, the maximum main is 69% and the standart deviation is 6%. The average difference gain(OPT) - gain(OPT-NO-SEM) for all the kernels is 15%. As a consequence, not only our algorithm is always better than common subexpression factorization, but its behaviour is much more regular.

4.2.2 Distribution of semantic factorizations

Figure 10 provides for each kernel the ratio of semantic arcs among the arcs chosen for expression factorization (a) and constraint factorization (b). When no semantic arcs are chosen, only common subexpression factorizations are applied. In that case, there is no gain compared to common subexpression factorization. This case occurs for lu-base, cholesky-base, nussinov-base. Figure 8 depicts the distribution of the ratio of semantic factorizations (arcs of the realization graph $\mathcal{G}_r$) among the factorizations chosen, for (a) expression factorizations and (b) constraint factorization. As for figure 7, the ratio are grouped per quantile of 5%. For instance, the second bar of (a) means that 4 kernels have a ratio in $[5\%, 10\%[$. It appears that semantic factorizations represents an average of 26% of expression factorization and 54% of constraint factorization. This said, constraint factorization will have less impact on the final cost, since they are terminal. They do not allow further factorization. Thus, semantic factorization is a significant phenomenon. Also, the distribution of semantic expression factorization appears to be tighter than for constraint factorization. That phenomenon appears to be more regular than semantic constraint factorization.



(a) Expressions factorization (b) Constraints factorization

Figure 8: Distribution of semantic factorizations (%)

4.2.3 Recursive calls

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 Figure 9 provides the maximum number of nested recursive calls for each kernel. Many kernels show one recursive call: semantic factorizations were applied, producing fresh expressions $\mathcal{E}_{\text{new}}$. But no opportunities of semantic factorizations were found in $\mathcal{E}_{\text{new}}$. However, half of the optimized (tiled) kernels (12 of 24) requires two recursive calls which means that fresh expression can in turn be optimized with semantic factorization.

4.2.4 Execution time

We have run our algorithm on a Intel Core i5 CPU M 540 @ 2.53GHz with 3072 KB L2 cache and 3GB RAM. Figure 9 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). The overall execution time tends to be small with a median execution time of 1 second and a maximum of 3 minutes and 43 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 [13], 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 [10], 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 parallism and finally the performances of the circuit. The authors also propose a runtime distance approach, which split the iteration domain into phases where the multiplexing is constant (variant domains). The iterations spend in each phase are counted thanks to polyhedral analysis [12], 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 limitated 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 [14], it is then said to be quasi-affine. Zissulescu et al. [29] 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 work could nicely complement our approach.

Piecewise affine functions received a lot of attention in the control community since Bemporad et al [8] 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 [24, 21], lattice-based representation [23, 20], mix thereof [7] or hash functions [6]. Tondel et al [24] 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 [21]. 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 [23, 18, 26] 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 [27] provides an algorithm for generating the lattice based representation of an affine function. Then, the lattice-based represention can be mapped directly on the FPGA [20] or mixed with an improved binary search tree [7]. The direct mapping leads to a througput of 5 cycles per point on a Xilinx 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 [6] 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 mentionned 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. Experimental results on a large benchmark show that not only our method is statistically better than the classical common subexpression factorization but its behavior is much more regular.

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.

References

[1] Christophe Alias, Fabrice Baray, and Alain Darte. 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)*, 2007.

[2] Christophe Alias, Alain Darte, and Alexandru Plesco. 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, 2013.

[3] Christophe Alias, Bogdan Pasca, and Alexandru Plesco. FPGA-specific synthesis of loop-nests with pipeline computational cores. *Microprocessors and Microsystems*, 36(8):606–619, November 2012.

[4] Christophe Alias and Alexandru Plesco. Data-aware Process Networks. Research Report RR-8735, Inria - Research Centre Grenoble – Rhône-Alpes, June 2015.

[5] Cédric Bastoul. 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*, pages 23–30, 2003.

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

[7] Farhad Bayat, Tor Arne Johansen, and Ali Akbar Jalali. 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):632–640, 2012.

[8] Alberto Bemporad, Manfred Morari, Vivek Dua, and Efstratios N. Pistikopoulos. The explicit linear quadratic regulator for constrained systems. *Automatica*, 38(1):3 – 20, 2002.

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

[10] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 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*, pages 101–113, 2008.

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

[12] Philippe Clauss. 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*, pages 278–285, 1996.

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

[14] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[15] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International journal of parallel programming*, 21(6):389–420, 1992.

[16] Paul Feautrier and Christian Lengauer. Polyhedron model. In *Encyclopedia of Parallel Computing*, pages 1581–1592. 2011.

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

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

[19] R.L. Herman. Numerical solution of 1d heat equation. Applied Analytical Methods, course notes, November 2014.

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

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

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

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

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

[25] Alexandru Turjan. *Compiling Nested Loop Programs to Process Networks.* PhD thesis, Universiteit Leiden, 2007.

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

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

[28] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-efficient cnn implementation on a deeply pipelined fpga cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pages 326–331. ACM, 2016.

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

A Detailed experimental results

Kernel	$\#\mathcal{C}$	$\#\mathcal{E}$	NON-OPT	OPT-NO-SEM	OPT	#sem-e-arcs	#e-arcs	#sem-c-arcs	#c-arcs	rec-depth	insert-time	dag-time
trmm (base)	38	27	121	90	75	2	8	3	7	1	0.7	0.1
trmm (tiled)	116	79	548	392	336	11	47	15	24	1	1.6	1.6
gemm (base)	43	22	111	106	56	1	2	10	11	1	0.7	0.2
gemm (tiled)	135	82	521	422	325	7	33	19	21	1	1.2	1.9
syrk (base)	38	22	124	95	71	1	7	5	6	1	0.6	0.1
syrk (tiled)	128	78	579	436	377	8	51	12	13	1	0.9	1.7
symm (base)	76	35	244	165	135	5	19	6	14	1	0.2	0.3
symm (tiled)	237	113	1045	694	559	20	98	25	41	1	4.0	5.3
gemver (base)	71	45	174	167	92	2	4	15	18	1	0.1	0.2
gemver (tiled)	198	137	887	661	592	18	69	23	27	1	1.6	3.9
gesummv (base)	49	29	122	109	64	2	4	9	12	1	0.5	0.1
gesummv (tiled)	113	84	493	395	311	8	31	16	18	1	0.5	1.1
syr2k (base)	44	25	141	112	78	1	7	7	9	1	0.8	0.2
syr2k (tiled)	149	86	668	501	437	9	59	13	14	1	1.4	2.3
gramschmidt (base)	87	52	257	182	143	4	14	8	17	1	0.3	0.5
lu (base)	68	29	268	137	137	0	30	0	5	1	0.3	0.4
lu (tiled)	187	85	984	527	485	12	134	6	13	2	4.1	4.5
durbin (base)	79	39	248	127	123	3	29	1	14	1	0.2	0.2
trisolv (base)	35	18	106	72	67	1	7	1	3	1	0.3	0.6
trisolv (tiled)	103	52	445	306	258	6	43	10	13	1	0.4	0.7
cholesky (base)	69	27	251	128	128	0	27	0	8	1	0.3	0.3
cholesky (tiled)	214	85	1090	554	498	32	150	5	10	2	6.2	4.7
ludcmp (base)	130	74	492	300	290	7	50	2	14	1	0.4	0.7
atax (base)	40	24	104	89	54	2	4	7	10	1	0.6	0.1
atax (tiled)	139	83	721	439	403	17	75	8	16	2	1.3	1.5
doitgen (base)	57	36	146	130	85	3	6	9	12	1	0.2	0.3
doitgen (tiled)	152	124	688	539	467	12	52	18	22	1	1.8	3.6
bicg (base)	42	25	111	97	57	2	4	8	11	1	0.5	0.1
bicg (tiled)	114	64	456	344	267	8	36	13	16	1	0.6	1.7
mvt (base)	43	26	112	98	58	2	4	8	11	1	0.6	0.1
mvt (tiled)	123	65	503	364	308	10	44	12	16	1	0.7	1.3
3mm (base)	95	58	250	228	128	3	6	20	25	1	0.3	0.6
3mm (tiled)	414	253	2706	1363	1276	119	359	21	38	2	19.1	21.1
2mm (base)	73	42	189	176	96	2	4	16	19	1	0.2	0.4
2mm (tiled)	243	161	1225	822	713	29	117	26	34	2	4.9	6.1
covariance (base)	74	48	220	178	114	2	8	10	14	1	0.2	0.3
covariance (tiled)	306	148	1494	1089	719	17	117	38	43	2	12.7	10.4
correlation (base)	104	70	284	232	162	3	15	14	22	1	0.3	0.6
correlation (tiled)	432	212	2038	1499	996	27	152	58	67	2	31.7	22.9
fdtd-2d (base)	100	53	293	193	165	5	22	6	24	1	0.5	0.6
fdtd-2d (tiled)	564	158	3863	1488	1438	205	556	4	37	2	63.2	39.1
jacobi-2d (base)	76	35	250	135	119	5	22	4	22	1	0.5	0.5
jacobi-2d (tiled)	455	109	3294	1158	1072	187	467	3	28	2	62.8	31.0
seidel-2d (base)	41	16	133	67	59	5	12	2	12	1	0.4	0.2
seidel-2d (tiled)	367	62	2901	942	878	204	429	4	17	2	174.7	48.9
adi (base)	185	121	534	380	292	1	31	22	46	1	1.1	1.5
adi (tiled)	563	304	2189	1633	1208	22	159	80	133	2	51.6	40.3
jacobi-1d (base)	44	23	136	78	70	5	13	2	11	1	0.9	0.1
jacobi-1d (tiled)	130	63	672	407	391	31	74	4	20	2	1.7	1.7
heat-3d (base)	109	47	360	193	169	0	30	6	32	1	1.7	1.1
nussinov (base)	104	63	546	252	252	0	70	0	11	1	0.9	0.7
floyd-warshall (base)	35	13	126	64	54	2	15	2	4	1	0.2	0.2
floyd-warshall (tiled)	118	33	540	248	238	13	78	2	12	1	2.3	1.8

Figure 9: Experimental results obtained on Polybench/C v3.2

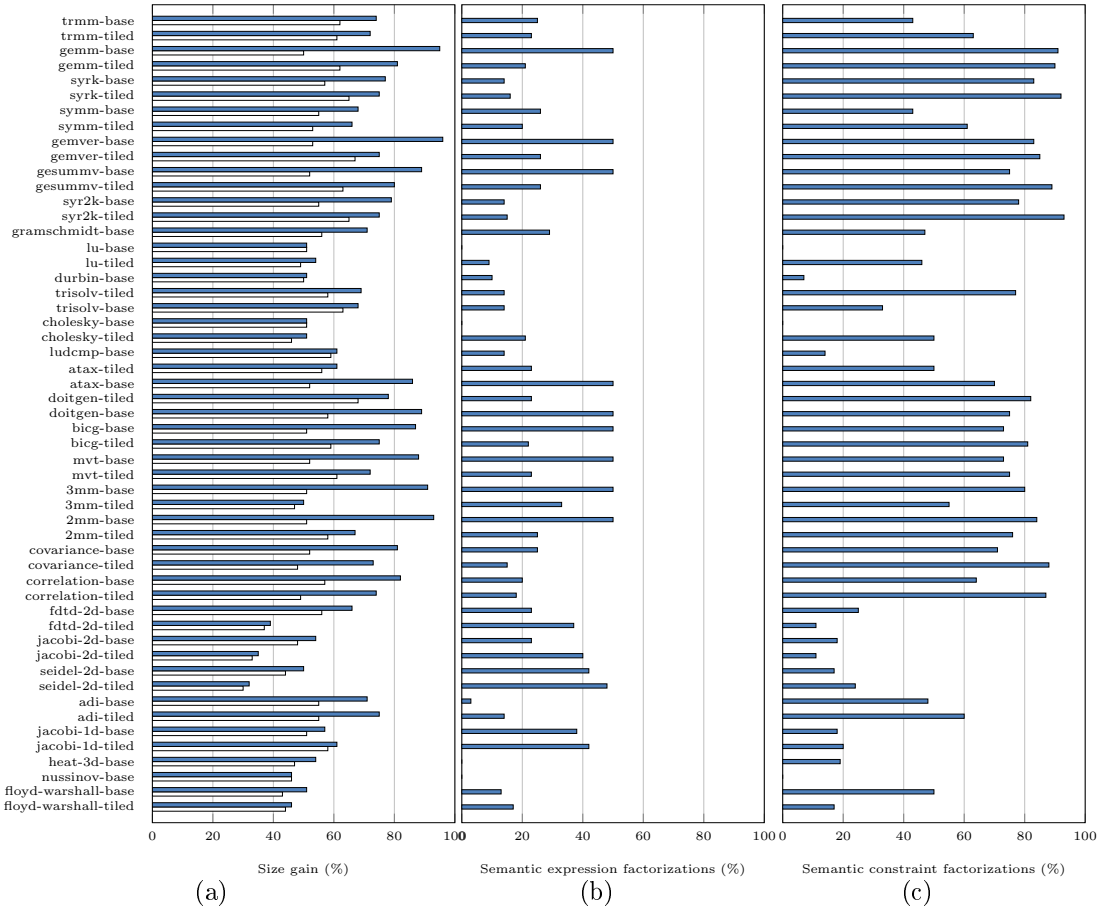
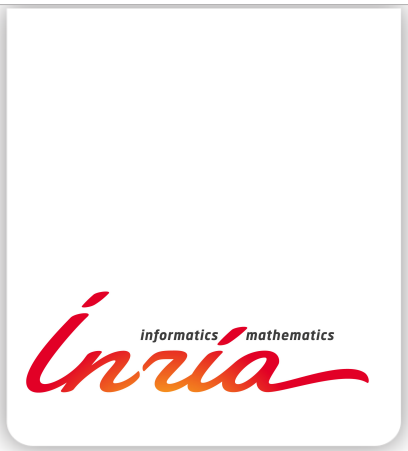


Figure 10: Detailed results

Contents



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399