Abstract—Computers across the board, from embedded to future exascale computers, are consistently designed with deeper memory hierarchies. While this opens up exciting opportunities for improving software performance and energy efficiency, it also makes it increasingly difficult to efficiently exploit the hardware. Advanced compilation techniques are a possible solution to this difficult problem and, among them, the polyhedral compilation technology provides a pathway for performing advanced automatic parallelization and code transformations. However, the polyhedral model is also known for its poor scalability with respect to the number of dimensions in the polyhedra that are used for representing programs. Although current compilers can cope with such limitation when targeting shallow hierarchies, polyhedral optimizations often become intractable as soon as deeper hardware hierarchies are considered.

We address this problem by introducing two new operators for polyhedral compilers: focalisation and defocalisation. When applied in the compilation flow, the new operators reduce the dimensionality of polyhedra, which drastically simplifies the mathematical problems solved during the compilation. We prove that the presented operators preserve the original program semantics, allowing them to be safely used in compilers. We implemented the operators in a production compiler, which drastically improved its performance and scalability when targeting deep hierarchies.

I. INTRODUCTION

The ever-increasing need for higher performance and the recent concern about energy consumption has put a lot of pressure on hardware manufacturers to increase parallelism. A direct consequence of the increased level of parallelism in computers is the creation of hierarchies of processing units and memories. It seems indeed simpler to build efficient parallel computers when they are designed as a flexible hierarchy. The various problems due to the system scale can then be solved independently from each other in the different levels. Interestingly, the trend holds for several scales: the expected computer design for exascale machines leans more and more towards deep hierarchies while, in the general-purpose computing environment, Intel recently introduced a fourth level of cache in some of its Broadwell chips. Deeper hierarchies are then becoming more and more prominent, which must be accounted for in compilers.

The polyhedral model provides powerful tools and representations to optimize statically analyzable programs for hierarchical platforms. Within the polyhedral model, programs can be automatically parallelized, specific memory levels can be targeted through tiling, and explicit communications between the various levels of the hierarchies can be automatically generated, among other powerful optimizations. The extreme performance improvements provided by polyhedral compilation have been reported numerous times in a large variety of contexts [1], [2], [3], [4], [5]. However, the model and its associated techniques are also known to be exponentially slower as the dimensionality of the considered polyhedra increases [6], [7]. This is a major concern for deep hierarchies because multi-level tiling is required to properly exploit every memory level while each level of tiling adds new dimensions to polyhedra. This issue is well known and, when deep hierarchies are considered, it is commonly dealt with using other optimization techniques that tend to be much less powerful than the general polyhedral model. For instance, syntactic tiling allows loop tiling to be performed with a low computational complexity but cannot be combined with several other desirable optimizations available in the polyhedral model. Thus, in order to benefit from the powerful optimizations of the polyhedral model when targeting deep hierarchies, its scalability limitations have to be addressed.

We propose two new operators to improve the scalability of the polyhedral compilation framework: a focalisation operator and an associated defocalisation operator. The operators effectively control the number of dimensions in the most important polyhedra used in the polyhedral representation. The operators are integrated in a general compilation flow for hierarchical hardware where the levels are optimized one after the other. Within that flow, the focalisation and defocalisation operators are applied on the program representation every time a new level of the hardware hierarchy has been targeted. As a result, the complexity of the mathematical problems solved during the compilation is drastically reduced and depends much less on the number of levels in the hardware hierarchy. In practice, the two operators are crucial for polyhedral compilers when targeting deeply hierarchical platforms.

The main contributions presented in this paper are:

• The specification of focalisation and defocalisation operators in the polyhedral model. The operators allow the simplification of the internal compiler representations. As a result, the computational complexity of the program optimization is drastically reduced.
• A proof ensuring that the program semantics is preserved when transforming programs using the focalised representation.
• An evaluation of focalisation in the context of a production polyhedral compiler. Our experiments illustrate how the focalisation drastically reduce the compilation time when hierarchical machines are targeted.
A theoretical generalization of the focalisation operator for program regions made of complex memory references.

A brief introduction to the classical compilation flow, the polyhedral model, and the associated notations are presented in Section II. The focalisation and defocalisation operators are presented in Section III. Their ability to preserve the program semantics through program transformations is also demonstrated. The operators are evaluated in a production compiler in Section IV and a theoretical generalization is presented in Section V, along with a proof of its semantic preservation properties. The method is compared to other related compiler enhancements in Section VI, before concluding in Section VII.

II. BACKGROUND AND NOTATIONS

A. General Notations

We denote vectors using lowercase letters and the arrow notation, such as with \( \vec{v} \). Vectors can also be decomposed into their different components using parenthesis: \( \vec{v} = (v_1, ..., v_n) \). Moreover, vectors and values can be concatenated using \( \langle \cdot, \cdot \rangle \). For instance, \( \langle 1, (2, 3) \rangle = (1, 2, 3) \) and \( \langle \vec{u}, \vec{v} \rangle = (u_1, ..., u_n, v_1, ..., v_m) \). The context determines whether vectors are in row or column format.

Matrices are named using an uppercase letter. \( M_{i,j} \) designates the element at row \( i \) and column \( j \) in the matrix \( M \). Polyhedra are defined as the intersection of linear inequalities and are named using stylized capital letters, such as \( \mathcal{P} \). Accolades designate an intersection of constraints and we interchangeably consider a polyhedron or the set of constraints that defines it.

We define a projection operation over polyhedra using the classical definition. We call \( \Pi_1 \) the operator projecting away the outermost dimension of a polyhedron. The operator is defined as follows.

\[ \Pi_1(\mathcal{P}) = \{ \vec{y} \mid \exists \vec{x}, \langle \vec{x}, \vec{y} \rangle \in \mathcal{P} \} \]

We also define an extension operator \( E_1 \), that extends a polyhedron by an unbounded outermost dimension. In some sense, the extension operation is the opposite of the projection operation defined earlier. The extension is defined as follows.

\[ E_1(\mathcal{P}) = \{ \langle \vec{x}, \vec{y} \rangle \mid \forall \vec{x} \in \mathbb{Z}, \vec{y} \in \mathcal{P} \} \]

One can notice that \( \Pi_1(E_1(\mathcal{P})) = \mathcal{P} \), but \( E_1(\Pi_1(\mathcal{P})) \subseteq \mathcal{P} \). Moreover, note that an extended empty polyhedron remains empty: \( E_1(\emptyset) = \emptyset \).

B. Polyhedral Model

The polyhedral model provides a coherent framework to efficiently represent loop nests in programs. Specifically, it enables the representation of program regions with statically analyzable control flow and memory accesses. As such, it is mostly targeted at loop nests where loop bounds, tests, and memory access functions are affine functions of the surrounding loop indices and constant parameters. Advanced loop transformations and optimizations are performed within the polyhedral model in a consistent fashion. In particular, every combination of loop fission, fusion, skewing, and interchange can be performed in the polyhedral model, among other optimizations.

In the polyhedral model, the program instructions define polyhedral statements. The statements in a program can be executed multiple times when they are enclosed in loops. In that case, the polyhedral model allows the various instances of the statement to be distinguished. The set of instances of a statement define its iteration domain, which is a polyhedron defined by the set of all the constraints that apply to the statement. In particular, the bounds of the loops enclosing the statement are intersected to form its iteration domain. The instances of the statement are then the points at integer coordinates in the polyhedron. For example, a statement \( S \) enclosed in two nested loops iterating from 0 to a constant \( N \), has an iteration domain \( D_S \) defined as

\[ D_S = \{ (i, j) \in \mathbb{Z}^2 \mid 0 \leq i \leq N \land 0 \leq j \leq N \} \]

In iteration domains, the points at integer coordinates represent the statement instances. The coordinates of instances can be expressed as integer vectors and represent the value of the loop indices when the statement instance is run. Such coordinate vectors are called iteration vectors. In the previous example, the valid iteration vectors are defined by \( \vec{s} \in D_S \).

The programs described in the polyhedral model usually expose data dependences. Two statements \( S \) and \( T \) are dependent on each other if they both access the same memory location during the program execution and if at least one of the two accesses is a write. The dependence between the two statements is classically noted \( \delta_{S,T} \) where \( S \) is considered to be the source of the dependence and \( T \) the destination if \( S \) is executed before \( T \) in the original program. As a special case, \( S \) and \( T \) can refer to the same statement in which case the dependence source and destination are two instances of a unique statement.

Dependences between two statements are also represented by convex polyhedra in the polyhedral model. When considering a dependence \( \delta_{S,T} \), the associated dependence polyhedron precisely characterizes the instances of \( S \) and \( T \) that are accessing the same memory location, provoking the dependence. Dependence polyhedra are defined as a subset of the Cartesian product of all the statement instances. In general, dependence polyhedra combine at least three types of constraints:

- **Validity**: the dependent source and target statement instances must be within their respective iteration domains;
- **Precedence**: the dependence source must run before the target;
- **Intersection**: the represented instances only participate in the dependence if they are accessing the same memory location.

As an example, a typical dependence polyhedron for a
dependence $\delta_{S,T}$, can be defined by the following constraints.

$$
\Delta_{S,T} = \begin{cases}
\vec{s} \in D_S \\
\vec{t} \in D_T \\
\vec{s} \prec \vec{t} \\
f_S(\vec{s}) = f_T(\vec{t})
\end{cases}
$$

where $\prec$ is the lexicographic precedence, defined by $(a_1, ..., a_n) \prec (b_1, ..., b_m) \iff \exists i, 1 \leq i \leq \min(n, m) \land (a_1, ..., a_{i-1}) = (b_1, ..., b_{i-1}) \land a_i < b_i$. In our notation, $f_S$ and $f_T$ are the affine functions used respectively in the conflicting memory references of $S$ and $T$.

In the polyhedral model, the loop transformations are represented as multidimensional affine schedules. A schedule is a transformation of the order in which the statement instances must be executed. When considering a statement $S$ enclosed in $d$ loops within a program expressed in terms of the parameters $\vec{p}$, the statement’s $m$-dimensional schedule is an affine function $\Theta_S$ defined such that

$$
\Theta_S(\vec{s}) = \begin{pmatrix}
\theta_{1,1} & \cdots & \theta_{1,d+p+1} \\
\vdots & \ddots & \vdots \\
\theta_{m,1} & \cdots & \theta_{m,d+p+1}
\end{pmatrix}
\begin{pmatrix}
\vec{s} \\
\vec{p}
\end{pmatrix}
$$

with $p = |\vec{p}|$, i.e. the number of parameters. The scheduling function $\Theta_S$ defines a new ordering for the instances of $S$: in the transformed space, the instances of $S$ are expected to be executed in the lexicographic order of the iteration vectors in $\{\Theta_S(\vec{s}) | \vec{s} \in D_S\}$. In this context, a program transformation can be expressed as a schedule for every statement in the program.

Not all the schedules lead to an execution order that preserves the original program semantics. A schedule, or transformation, is said legal if it maintains the relative ordering of all the dependent statement instances. In particular, for every dependence $\delta_{S,T}$, a schedule is valid if it enforces the following condition:

$$
\Theta_S(\vec{s}) \prec \Theta_T(\vec{t})
$$

In this paper, we do not focus on the method used to compute a legal and efficient program transformation. Instead, whenever it is required, we consider that a legal transformation can be provided by any of the existing scheduling methods [1], [8], [9], [10], [11].

C. Polyhedral Compilation Flow

A typical polyhedral compilation flow is structured as follows. At first, the program source code is parsed into an equivalent polyhedral representation. Then, the dependence analysis phase detects all the dependent statements pairs in the program and builds all the corresponding dependence polyhedra. Once the all the dependences in the program are computed, a legal transformation, or schedule, is computed for the program. Usually, an ILP solver is used to determine the optimal transformation to apply on the statements in order to maximize various metrics such as the memory locality or the parallelism [1], [10], [11]. Once the scheduling is done, additional transformations may be applied on the program. It is for instance typical to tile the program loop nests, determine an efficient memory assignment for the variables, and generate explicit communications at this phase. Finally, the polyhedral representation is transformed back into a lower representation and the transformed program can be generated.

The typical compilation flow can be adapted to target deep hierarchies in different ways. A possible approach would be to optimize the programs for all the memory levels at once. Because the overall architecture is considered, this would allow the most accurate optimizations to be performed. On the other hand, this approach cannot scale beyond one or two memory levels in polyhedral compilers, as explained in the introduction. We propose instead to use an iterative approach where the architecture level is successively targeted, as illustrated in Figure 1. In our approach, a different optimization problem is solved specifically for every considered level, according to the hardware characteristics of the considered level. On one hand, not considering the whole architecture in a single optimization step may prevent the compiler from choosing the best optimizations for the program but, on the other hand, such iterative compilation flow can be combined with the focalisation and defocalisation operators introduced in the next section. As a consequence, the iterative compilation flow, combined with the focalisation and defocalisation operators, makes it feasible to target deeply hierarchical architectures, which were previously out of reach for a polyhedral compiler.

III. FOCALISATION AND DEFOCALISATION

A. Overview

We propose a new method able to simplify the polyhedra used to represent and transform programs in compilers. The general strategy employed consists in focusing the representation on the dimensions identified as more important in polyhedra, ignoring the other ones. The usual sequence of compiler optimizations is then applied on the simplified polyhedra in order to transform and optimize the program more efficiently. Finally, a defocalisation phase recovers the initial program semantics while carrying over the transformations previously applied on the program. As a consequence, some of the most expensive passes of polyhedral compilation are performed using simpler polyhedra, leading to faster compilation and improved compiler scalability.

We integrate the focalisation and defocalisation phases within the general compilation flow, presented in Figure 1.
First, we decompose the general compilation flow for a hierarchical hardware target into a sequence of optimizations. At each step, a different level of the hardware hierarchy is considered: the optimizations applied and the parameters used are specialized for that specific level. Then, the focalisation is performed as soon as a hierarchy level has been fully handled, as shown in Figure 2. A direct benefit from such compilation flow is its scalability. Indeed, if the focalisation allows all the newly introduced dimensions to be ignored in the next hierarchy level, then the compilation flow can scale to arbitrarily deep hardware hierarchies.

The improved compiler scalability does not come for free. In fact, because some polyhedral dimensions are hidden to compiler optimizations, they cannot be involved in the transformations applied after the focalisation. For instance, it is not possible to perform a loop interchange with one of the hidden dimensions. Thus, the dimensions involved in the focalisation have to be carefully chosen to maintain a good balance between the polyhedra complexity and the relevance of the remaining optimizations.

Different heuristics and criteria can be used to determine which dimensions in polyhedra are of lesser importance and should be removed in the focalised representation. For instance, one could consider that parallel dimensions should be targeted in order to preserve them from being further transformed and potentially loose some parallelism. We suggest to target the inter-tile dimensions when considering hierarchical memory models. When tiling is applied, the inter-tile loops are then semantically associated to the higher memory level, while intra-tile loops are associated to lower memory levels. Then, when the compiler focuses on lower memory levels, it should mostly consider the intra-tile loops and is less likely to involve the inter-tile dimensions in new transformations. Thus, the inter-tile dimensions are, in this case, good candidates for being hidden by the focalisation.

B. Focalisation and Defocalisation

The computation of a new schedule, or scheduling, determines the transformations and optimizations applied on a program. It is then one of the core operations performed during the program compilation. The scheduling is also among the most computationally expensive compilation phases. It is then crucial that the focalisation performed benefits to the scheduling pass. The scheduling pass depends mostly on dependence polyhedra, which define the legality of the generated schedule. Thus, we define the focalisation operator \( \alpha \) as a transformation of dependence polyhedra. Its input is a dependence polyhedron \( \Delta_{S,T} \), resulting from dependence analysis, and its output is a simpler dependence polyhedron \( \Delta_{\tilde{S},T} \) of lower dimensionality:

\[
\alpha(\Delta_{S,T}) = \Delta_{\tilde{S},T}
\]

In the polyhedral model, most of the transformations applied on a program can be encoded in a program schedule. However, after focalisation has been applied, the new program schedule \( \Theta_{\tilde{S}} \) is computed using the focalised program representation and is then not relevant to the original program. The goal of the defocalisation operator is then to transform the schedule so that it can be applied on the original program representation with the additional constraint that all the transformations defined by the schedule \( \Theta_{\tilde{S}} \) must be carried through the defocalisation. Thus, we define a defocalisation operator \( \rho \) as a transformation performed on schedules. The operator takes the schedule \( \Theta_{\tilde{S}} \) as an input and converts it into a schedule \( \Theta_S \). Finally, the resulting schedules \( \Theta_S \) are applied on the original program representation. The defocalisation operator is then defined by

\[
\rho(\Theta_{\tilde{S}}) = \Theta_S
\]

The focalisation process is fully defined as a tuple \( (\alpha, \rho) \) where \( \alpha \) is the focalisation operator and \( \rho \) is the defocalisation operator applied on the program. Several operators can potentially be used to perform the two operations. We present two focalisation operators with different properties and a common defocalisation operator in the following sections. We also demonstrate that the focalisation processes they define preserve the original program semantics.

C. Focalisation Operator

We introduce a focalisation operator whose goal is to remove the inter-tile dimensions from the dependence polyhedra. The intuition behind selecting the inter-tile dimensions is that they realize an optimization related to a memory level which has already been optimized by the compiler, on the contrary...
to the intra-tile dimensions. For this focalisation operator, we assume that the inter-tile dimensions can be identified in the program, which is not an issue when the compiler generated them. In order to simplify the presentation, we define a sub-operator ˇα which removes the outermost dimension of the iteration domains used in dependence polyhedra. The sub-operator can be extended to any dimension with no difficulty and ˇα can be repeatedly applied, eliminating the inter-tile dimensions one at a time. Such repeated application define the full focalisation operator α.

Based on the dependence polyhedron formulation, we define the focalisation operator ˇα as

$$\hat{\alpha}(\Delta_{S,T}) = \begin{cases} \vec{s}' \in \Pi_1(D_S) \\ \vec{t}' \in \Pi_1(D_T) \\ \phi_1(f_S)(\vec{s}') = \phi_1(f_T)(\vec{t}') \end{cases}$$

where φ1 removes any reference to the outermost dimension in a memory access function.

The focalisation operator exploits the projection operation, which is known to be computationally expensive. The complexity of the projection can be an issue in some pathological cases and may prevent the focalisation from being applied. In practice, we never encountered such cases and the benefit of the focalisation always prevailed over the cost of the operator.

### D. Defocalisation Operator

In order to exploit the schedules computed using the focalised program representation, we associate the focalisation operator to a defocalisation operator ˇρ. As previously, we simplify the notations by defining a sub-operator ˇρ, which can be repeatedly applied on the schedule computed from the focalised program representation. Every application of the operator ˇρ re-introduces the outermost dimension on a schedule, assuming it has been projected away by the last application of ˇα. The operator ˇρ can be extended with no difficulty to any arbitrary dimension. The operator is defined as follows.

$$\Theta_S = ˇ\rho(\Theta_S^-) = \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 0 & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & \rho(\Theta_S^-) \\ 0 & \ldots & 0 \end{pmatrix}$$

Intuitively, the defocalisation re-introduces an outermost loop and leaves it in its original state: it is not transformed nor combined with any other dimension. The defocalisation process is purely syntactic and has a negligible overhead if the matrix representation of the schedule is used internally. It is repeated as many times as the program has been focalised in order to recover a schedule of the same dimension as the original program semantics. Finally, the defocalised schedule is applied on the original program representation to replay the transformation computed based on the focalised program representation. The computational cost of the schedule application is also limited and causes no specific trouble. As a consequence, the computational cost of the overall defocalisation remains limited.

### E. Legality

We defined earlier the focalisation and defocalisation operators as the repetition of two sub-operators ˇα and ˇρ. It is crucial that the focalisation and defocalisation, achieved by the repetitive application of the sub-operators, preserve the original program semantics. In other words, a schedule computed from the focalised program representation and defocalised afterwards must be legal with regard to all the dependences in the original program.

A schedule Θ is said legal if it is legal with regard to every dependence in the program. A schedule is legal with regard to a dependence δ_{S,T} if it never leads to execute the source statement S after the target T. Thus, when considering any dependence δ_{S,T}, the schedule Θ is legal if there is no solution to the following set of constraints.

$$\begin{cases} \vec{s} \in D_S \\ \vec{t} \in D_T \\ \vec{s} \prec \vec{t} \\ f_S(\vec{s}) = f_T(\vec{t}) \\ \Theta_S(\vec{s}) \succ \Theta_T(\vec{t}) \end{cases}$$

The previous constraints define all the couples of iteration vectors such that S runs after T in the transformed program, despite the order imposed by δ_{S,T}. When the schedule Θ results from the defocalisation of another schedule, Θ^−, computed on the focalised program representation, the constraints can then be rewritten into the following equivalent set.

$$\begin{cases} \vec{s} \in D_S \\ \vec{t} \in D_T \\ \vec{s} \prec \vec{t} \\ f_S(\vec{s}) = f_T(\vec{t}) \\ \rho(\Theta_S^-)(\vec{s}) \succ \rho(\Theta_T^-)(\vec{t}) \end{cases}$$

This constraint set must be proved empty for the schedule Θ to be valid with regard to δ_{S,T}.

We expand the defocalisation operator ˇρ using its definition:

$$\begin{cases} \vec{s} \in D_S \\ \vec{t} \in D_T \\ \vec{s} \prec \vec{t} \\ f_S(\vec{s}) = f_T(\vec{t}) \\ \langle s_1, \Theta_S^-((s_2, \ldots, s_n)) \rangle \succ \langle t_1, \Theta_T^-((t_2, \ldots, t_m)) \rangle \end{cases}$$

We also expand the two lexicographic precedence relations in the constraints according to the definition of the relation. The relation \( \vec{s} \prec \vec{t} \) is expanded into either

$$\begin{cases} s_1 = t_1 \\ \vec{s} \prec \vec{t} \end{cases}$$

or

$$s_1 < t_1$$

Similarly, the relation

$$\langle s_1, \Theta_S^-((s_2, \ldots, s_n)) \rangle \succ \langle t_1, \Theta_T^-((t_2, \ldots, t_m)) \rangle$$
is expanded into either
\[
\begin{align*}
\{ & s_1 = t_1 \\
& \Theta_\bar{S}^-((s_2, ..., s_n)) > \Theta_\bar{T}^-((t_2, ..., t_m)) \}
\end{align*}
\]
or
\[
\begin{align*}
& s_1 > t_1
\end{align*}
\]

One can notice that the ordering of \( s_1 \) and \( t_1 \) is always contradicting when the lexicographic relations are expanded, except for the case where \( s_1 = t_1 \) in both relations. Thus, the schedule \( \Theta \) is legal with regard to a dependence \( \delta_{8,T} \) if the following set of constraints is empty.

\[
A = \begin{cases}
\begin{aligned}
\bar{s} & \in D_S \\
\bar{t} & \in D_T \\
\bar{s} & < \bar{t} \\
f_{S}(\bar{s}) & = f_{T}(\bar{t}) \\
s_1 & = t_1 \\
\Theta_\bar{S}^-((s_2, ..., s_n)) & > \Theta_\bar{T}^-((t_2, ..., t_m))
\end{aligned}
\end{cases}
\]

We now restrict the set of considered memory references to Piecewise Uniformly Generated References (PUGRs). We define PUGRs as a natural generalization of the Uniformly Generated References (UGRs) [12]: while two memory references are said to be uniformly generated if they only differ by a constant, we call them piecewise uniformly generated along a dimension \( i \) if they refer to the dimension \( i \) using the same coefficient. Notice that, if two references are UGRs, they are PUGRs along all the dimensions. For instance \( A[i] \) and \( A[i+2] \) are uniformly generated and PUGRs along \( i \), but \( A[i+j] \) and \( A[i+2j+1] \) are only piecewise uniformly generated along \( i \). Although they obviously do not cover all the valid memory references, PUGRs are common among the memory accesses existing in programs. Moreover, we present a more general focalisation operator, valid for every kind of memory references in Section V.

We assume that the schedules \( \Theta_\bar{S}^- \) and \( \Theta_\bar{T}^- \) are legal with regard to the program representation focalised using \( \bar{\alpha} \). In other words, we assume that the scheduling method used in compiler produces legal schedules. Then, the following set of constraints is necessarily empty.

\[
\begin{cases}
\begin{aligned}
\bar{s} & \in \Pi_1(D_S) \\
\bar{t} & \in \Pi_1(D_T) \\
\bar{s} & < \bar{t} \\
\phi_1(f_{S}(\bar{s})) & = \phi_1(f_{T}(\bar{t})) \\
\Theta_\bar{S}^- (\bar{s}) & > \Theta_\bar{T}^- (\bar{t})
\end{aligned}
\end{cases}
\]

The preceding constraints can be extended as follows, while preserving the vacuity of the represented polyhedron.

\[
\begin{cases}
\begin{aligned}
\bar{s} & \in E_1(\Pi_1(D_S)) \\
\bar{t} & \in E_1(\Pi_1(D_T)) \\
(s_2, ..., s_n) & \prec (t_2, ..., t_m) \\
\phi_1(f_{S}(s_2, ..., s_n)) & = \phi_1(f_{T}(t_2, ..., t_m)) \\
\Theta_\bar{S}^- ((s_2, ..., s_n)) & > \Theta_\bar{T}^- ((t_2, ..., t_m))
\end{aligned}
\end{cases}
\]

Additional constraints cannot transform an empty polyhedron into a non-empty one. Thus, the following set of constraints also defines an empty polyhedron:

\[
\begin{cases}
\begin{aligned}
\bar{s} & \in E_1(\Pi_1(D_S)) \\
\bar{t} & \in E_1(\Pi_1(D_T)) \\
\bar{s} & < \bar{t} \\
s_1 & = t_1 \\
\phi_1(f_{S}(s_2, ..., s_n)) & = \phi_1(f_{T}(t_2, ..., t_m)) \\
\Theta_\bar{S}^- ((s_2, ..., s_n)) & > \Theta_\bar{T}^- ((t_2, ..., t_m))
\end{aligned}
\end{cases}
\]

Finally, because the previous constraints set is empty and a superset of \( A \), \( A \) is also guaranteed to be empty. Thus, when applied on PUGRs, the focalisation and defocalisation process is guaranteed to result in legal schedules \( \Theta \), provided that the schedules \( \Theta^{-} \) are legal with regard to the focalised program representation.

\[
\square
\]

IV. EXPERIMENTS

A. EXPERIMENTAL SETUP

In order to evaluate the presented focalised compilation process, we implemented it in the R-Stream compiler, developed by Reservoir Labs. R-Stream is a source-to-source polyhedral compiler implementing many modern program optimizations including automatic parallelization, memory management, and explicit communication generation. The complexity and the scope of the optimizations implemented in R-Stream make of the compiler a good candidate for our experiments.

In all the presented experiments, we compare two versions of R-Stream. Both of them perform hierarchical mapping of the programs over successive levels of the targeted memory and processing hierarchies. As such, a sequence of polyhedral transformations is computed and applied on the programs, successively optimizing the programs for every hierarchy level. The two evaluated versions of the compiler differ in that one of them applies the focalisation operator on the parts of the program that comply with the operator requirements every time a new hierarchy level is targeted. The version using the focalisation operator is performing “focalised compilation”, while the other version is simply referred to as the “reference”.

In order to evaluate a large variety of target platforms, we designed a set of hypothetical hardware targets for the compiler. A first platform has a single processing level made of 64 x86-like processing units with a single 32KB memory level. Every other machine model is deduced by replicating
the highest processing level 4 times and adding a new memory level twice as big as the highest memory level in the reference model. As a result, we designed 5 hierarchical machine models exposing from 1 to 5 processing and memory levels. We believe that such machine models are made of quantities in the same order of magnitude as those expected for exascale machines.

The evaluation has been performed over various programs including the PolyBench/C benchmark suite, manually edited to fix its known issues, and representative kernels extracted from complex programs of interest. We included the Chebyshev smoother from the High Performance Geometric Multigrid (HPGMG) benchmark, a sparse matrix-matrix multiplication as expressed in the CoSP2 proxy application, two sub-kernels found in space-time adaptive processing (STAP), and a 3D Reverse Time Migration algorithm (RTM-3d). The two kernels extracted from STAP are a covariance estimation using complex numbers, and an Adaptive Matched Filter (AMF) kernel.

All the experiments have been performed on a standard x86 server made of 4 Intel® Xeon® E5-4620 processors exposing 32 hyper-threaded processor cores. The server runs Linux 3.13 as provided in Ubuntu 14.04 and exploits gcc 4.8 as a low-level C compiler.

B. Performance

As a first step in our experiments, we evaluated the performance gains provided by the focalisation and defocalisation operators using two metrics: the compilation time and execution time of the generated program. For the focalisation process to be said efficient, the focalisation and defocalisation operator must provide measurable speedups during the compilation while not significantly impacting the resulting program performance. Finally, in order to perform a comparison as fair as possible, we considered for this study a shallow hierarchy of 2 levels only. Such shallow hierarchy limits the number of dimensions in polyhedra, which still allows a classical polyhedral compilation flow to work reasonably well. In such configuration, the benefits of the focalisation process are expected to be limited because the operators can only be applied once on the program, typically removing or restoring two or three dimensions on polyhedra.

The effects of the focalisation and defocalisation operators depend on the considered program, as shown in Figure 3. First, some programs do not allow the operator to be applied because they cannot be decomposed into loop nests only made of PUGRs. A typical example of such program is the lu kernel which contains several non-uniform dependences. For several other programs, the compilation time can be drastically reduced thanks to the focalisation process. It appears that the programs that benefit the most from the focalisation are made of deep loop nests and are mostly made of PUGRs. The kernels jacobi-2d-imper and STAP/amf are extreme cases for which the reference compiler is not able to optimize the program in less than 5 minutes and is then killed. For those two programs, the focalised compilation flow ends in a few seconds. The kernel seidel is another similar case, although the reference compiler ends just before the 5 minutes timeout. From the experiments, it is then clear that the focalisation operator, when it can be applied, significantly reduces the time required to optimize a program.

Although we observed reduced compilation times, it is also worth checking that the resulting programs still provide decent performance compared to the reference compilation flow, when it can produce a program. The programs generated by the exact and focalised compilation flows are compared in Figure 4. No data is provided for jacobi-2d-imper and STAP/amf with the reference compilation flow because no program can be generated in a realistic time if the focalisation operator is not used. The figure clearly shows that the focalised compiler manages to produce efficient code in all the cases.

The execution time fluctuations for lu must be ignored and are only due to measurement noise. However, for ludcmp, and seidel the focalised compilation flow generates better programs. Such surprising result can be explained from two
reasons. First, compilers use many heuristics and models to determine the transformation for a program. It may then happen that a sub-optimal transformation is chosen in the reference compilation flow but is not anymore valid in the focalised compilation flow, which unexpectedly results in a better program. Second, R-Stream relies on solvers to determine which optimization to apply on a program. However, the solver may perform worse with the larger program representation because of the time constraints imposed by the compiler. As a result, the focalised representation creates simpler problems for the solvers which can reach better solutions within the time constraints.

C. Scalability

Additionally to the raw performance, we also evaluate the impact of the focalisation operator on the compiler scalability. The compiler scalability is evaluated by recompiling a given program using increasingly deeper hardware hierarchies as a target. In our experiments, we evaluated processor and memory hierarchies with 1 to 5 levels. Among the benchmark programs, we selected 3 representative cases leading to different scalability patterns. The 3 cases are detailed hereafter.

The first program whose scalability is studied is covariance. In this program, only a small subset of the program is made of PUGRs. Thus, the focalisation operator is not expected to perform well. In Figure 5, one can see a slight improvement in compilation time when considering hierarchies of 2 and 3 levels. However, the improvement is insufficient to target 4 or more levels. As a result, both the reference compilation flow and the focalised one reach the 5 minutes timeout without producing any optimized program.

A second case is presented in Figure 6 with the gemm kernel. For this kernel, both compilers manage to optimize the program in less than 5 minutes. However, the focalised compilation flow provides a clear improvement over the reference as soon as at least 3 hierarchy levels are targeted. gemm is a perfect candidate for the operator because it is entirely made of UGRs.

The third studied kernel, doitgen, has a similar scalability trend as gemm, as shown in Figure 7. However, with doitgen, not only the focalisation operator improves the compilation time but it also allows the compiler to produce an optimized program in all the studied cases whereas the hierarchies with 4 or more levels are out of reach when the focalisation operator is not used. This example illustrates the importance of the focalisation operator when targeting deep hardware hierarchies: it effectively provides the significant scalability improvement required to target the next hardware generation.

V. GENERALIZATION

We introduced in Section III a focalisation operator able to reduce the polyhedra dimensionality while preserving the semantics of programs made of PUGRs. However, as illustrated in Section IV, some memory references in programs are not PUGRs and cannot benefit from the focalisation operator. We present in this section an additional focalisation operator that does not require memory references to be PUGRs.
We define a new, more general, focalisation operator $\alpha_g$ performing the focalisation by projecting away the outermost dimension in $\Delta_{S,T}$:

$$\alpha_g(\Delta_{S,T}) = \Pi_1(\Delta_{S,T})$$

In the polyhedral model and its usage in compilers has been known for several decades [8], [9]. The model provides a way of such a practically-viable general focalisation operator. We assume that they are generated legal with regard to the focalised program dependences. In particular, $\Theta^g_S$ and $\Theta_T^g$ must be valid with regard to the dependence $\delta_{S,T}$, after it has been focalised. As a consequence, the following constraint set is guaranteed to be empty, by definition of the schedule legality.

$$\left\{ \begin{array}{l}
\alpha_g(\Delta_{S,T}) \\
\Theta^g_S((s_2, \ldots, s_n)) > \Theta_T^g((t_1, \ldots, t_m))
\end{array} \right.$$ 

We choose to extend the previous constraints set using the extension operator $E_1$. Because the polyhedron extension does not impact the polyhedron vacuity, the following extended constraint set is also empty.

$$E_1 \left( \left\{ \begin{array}{l}
\alpha_g(\Delta_{S,T}) \\
\Theta^g_S((s_2, \ldots, s_n)) > \Theta_T^g((t_1, \ldots, t_m))
\end{array} \right. \right)$$

Interestingly, the previous set of constraint is both empty and a superset of $A$. As a result, the constraint set $A$ is also empty. Thus, the restored schedules $\Theta_S$ and $\Theta_T$ are legal with regard to the dependence $\delta_{S,T}$ if the schedules $\Theta^g_S$ and $\Theta_T^g$ are also legal in the focalised program representation.

The generalized focalisation operator $\alpha_g$ is correct even if the considered memory references are not PUGRs. However, it suffers from several limitations that prevent its use in a realistict polyhedral compilation toolchain. Indeed, $\alpha_g$ is defined over dependence polyhedra and not over access functions and iteration domains, like $\alpha$ is. There are significantly more dependence polyhedra than iteration domains in the polyhedral model representation of programs. Thus, the operator must be applied much more frequently. It also requires full dependence polyhedra to be built before being focalised, which can be computationally expensive. Moreover, in a modern polyhedral compilation flow, numerous complex optimizations are applied after scheduling, usually based on iteration domains more than dependence polyhedra. All those compiler passes cannot benefit from the focalisation performed on dependence polyhedra, which greatly reduces the benefits offered by $\alpha_g$.

Despite its practical limitations, $\alpha_g$ still demonstrates how the focalisation process can theoretically be safely applied to any program within the polyhedral model. It also illustrates the flexibility of our framework based on generic operators. Finally, although we are not aware of any feasible general focalisation operator, we obviously do not preclude the existence of such a practically-viable general focalisation operator.

VI. RELATED WORK

The polyhedral model and its usage in compilers has been known for several decades [8], [9]. The model provides a coherent framework in which programs can be efficiently represented, analyzed, and optimized. It has then naturally been implemented in several production compilers such as GCC [13], LLVM [14], or the R-Stream compiler [15]. The polyhedral model is then mature and its efficiency is an important matter for existing compilers.

The computational complexity is one of the major challenges induced by the polyhedral compilation. Various directions have been investigated and different methods have been proposed to enhance the scalability of polyhedral compilers. The various proposed solutions focus on different aspects of the polyhedral representation and are often complementary.

First, several techniques restrict the set of constraints allowed to define polyhedra. Several variants of the same techniques exist, each one restricting differently the form of the constraints that can be handled. For instance, Difference Bound Matrices (DBM) only allows constraints in the form $x_i - x_j \leq k$, $x_i \geq 0$, $x_j \geq 0$ [16], [17]. Other representations allow more complex constraints such as Unit Two Variables Per Inequality (UTVPI) [18], [19] or Two Variables Per Inequality (TVPI) [20], [21], [22] for instance. The general idea is to restrict the form of the constraints in order to use specialized algorithms to handle the polyhedra, usually with strong worst-case complexity guarantees.

Second, Mehta and Yew recently proposed to overapproximate a sequence of statements as a single element called O-molecule [23]. Their approach reduces the number of statements considered in a program, which drastically improves the complexity of several polyhedral operations performed during compilation.

Finally, Feautrier suggests directions to improve the implementation of the scheduling step [24]. In the same paper, a modular approach to scheduling is presented. The modularity is defined in the same spirit as the O-molecules but prevents some global optimizations to be performed on the program.

All the presented improvements target different sources of computational inefficiencies and never tackle the difficulties induced by the polyhedra dimensionality, addressed by the focalisation and defocalisation operators. Thus, all the previous approaches are orthogonal to ours and could be used on top of it to reduce further the computational complexity. Moreover, although the focus of our approach is on the number of dimensions in polyhedra, the presented operators also have positive side effects on the number of constraints in the internal representation.

Deep hardware hierarchies are the target of several compiler optimizations [25], [26], [27]. Among them, multi-level tiling represents a significant part of the optimizations for deep hierarchies. Interestingly, it has been reported to suffer from similar scalability issues as the polyhedral model [28], even though the considered optimizations are limited to tiling. The proposed scalability improvements target code generation and are restricted to tiling [28]. The focalisation and defocalisation operators on the other hand are more general because they are not related to any particular optimization. Moreover, the operators specialized to PUGRs provide a way to reduce the complexity of the overall program representation in a
authors and should not be interpreted as representing the compiler performance and scalability. The focalisation process is presented focalisation operator significantly improves the methodology for significantly reducing the computational cost operators. As a result, we have provided a safe and sound process for both the general and specialized version of the program semantics during the focalisation and defocalisation references. We have demonstrated the preservation of the gram regions made of piecewise uniformly generated memory polyhedral model: a focalisation operator and a defocalisation performed more efficiently.

We have presented two new operators in the context of the polyhedral model: a focalisation operator and a defocalisation operator. The focalisation operator is also specialized for program regions made of piecewise uniformly generated memory references. We have demonstrated the preservation of the program semantics during the focalisation and defocalisation process for both the general and specialized version of the operators. As a result, we have provided a safe and sound methodological for significantly reducing the computational cost of program optimization.

Through our experiments, we have illustrated how the presented focalisation operator significantly improves the compiler performance and scalability. The focalisation process is a major asset for optimizing compilers that enables them to realistically target the next-generation hardware.

ACKNOWLEDGMENT

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

REFERENCES


