Visual Program Manipulation in the Polyhedral Model

OLEKSANDR ZINENKO, Inria and University Paris-Saclay
STÉPHANE HUOT, Inria
CÉDRIC BASTOUL, University of Strasbourg and Inria

Parallelism is one of the key performance sources in modern computer systems. When heuristics-based automatic parallelization fails to improve performance, a cumbersome and error-prone manual transformation is often required. As a solution, we propose an interactive visual approach building on the polyhedral model that visualizes exact dependences and parallelism; decomposes and replays a complex automatically-computed transformation step by step; and allows for directly manipulating the visual representation as a means of transforming the program with immediate feedback. User studies suggest that our visualization is understood by experts and non-experts alike, and that it may favor an exploratory approach.

CCS Concepts: •**Human-centered computing** → *Human computer interaction (HCI);* •**Software and its engineering** → *Compilers;*

Additional Key Words and Phrases: Polyhedral model, direct manipulation

ACM Reference format:
Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2018. Visual Program Manipulation in the Polyhedral Model. *ACM Transactions on Architecture and Code Optimization* 15, 1, Article 16 (March 2018), 25 pages.
DOI: https://doi.org/10.1145/3177961

1 INTRODUCTION

Large-scale adoption of heterogeneous parallel architectures requires efficient solutions to exploit the available parallelism from applications. Despite significant effort in simplifying parallel programming through new languages, high-level language extensions, frameworks and libraries, manual parallelization may still be required although often ruled out as time consuming and error-prone. Thus, programmers mostly rely on automatic optimization tools, such as those based on the polyhedral model, to improve program performance. The polyhedral model [21] has been the cornerstone of loop-level program transformation in the last two decades [3, 8, 20]. It features exact iteration-wise dependence analysis and optimization for both parallelism and locality. However, automatic polyhedral compilation is based on imprecise heuristics [8, 11]. Polyhedral compilers give user some (limited) control over the optimization process, which requires understanding their internal operation anyway. Furthermore, they are applicable globally and do not allow for finer-grain control, e.g., affecting only one loop nest. Visual interfaces for configuring the polyhedral compiler [39] partially mitigate these issues by making polyhedral compiler blocks discoverable, but still require a deep understanding of internal operation of a compiler.

Semi-automatic approaches provide the user with a set of predefined program transformations, typically exposed as compiler directives [22, 31, 56]. They shift the expertise requirements from heuristics to loop-level code transformations. They also require program transformation to be performed *from scratch* (as polyhedrally-transformed code is barely readable) while offering little

Conference Extension: Visualization was presented at VL/HCC 2014 [58]; preliminary evaluation at IMPACT workshop in 2015 [57]. In this paper, we introduce (1) the mapping between graphical manipulations and program transformations; (2) the animated replay of transformation sequences; (3) a study of visualization relevance compared to code.

Authors' version. Final version published in ACM Transactions on Architecture and Code Optimization 15, 1, Article 16. https://doi.org/10.1145/3177961

support in identifying illegal or profitable transformations. Recently, we proposed the *Chlore* algorithm that identifies a sequence of transformation directives equivalent to an automatically computed transformation, decoupled from a polyhedral compiler [2]. However, sequences of directives are often long, and effects of individual transformations are unclear.

In this paper, we go further than transformation primitives by presenting an interactive visual program transformation toolsuite, *Clint*. It is a result of interdisciplinary work between optimization and human-computer interaction (HCI). We demonstrate how the geometrical nature of the polyhedral model can be leveraged to design efficient visualization and interactions.

Beyond static visualization, *Clint* features a step-by-step animated replay of *Chlore*-identified transformations with immediate feedback on their effects. It also allows the user to directly manipulate [46] the visual representation to modify these sequences or to transform programs manually while ensuring transformation legality and final code generation. Following a *user-centric* approach, we conducted user studies to asses the visualization and interactions.

While visualization-based interactive program restructuring is applicable to different program representations, *Clint* relies on the polyhedral model in order to operate on individual loop iterations. We extensively detail the design methodology and evaluation process so as to allow its application to different program manipulation tasks as well as replication of our studies.

2 PROGRAM TRANSFORMATION IN THE POLYHEDRAL MODEL

The polyhedral model is an algebraic representation of "sufficiently regular" imperative programs that encodes dynamic executions of statements inside loop nests [21]. It is used within several production compilers such as GCC [41], LLVM [25] and IBM XL [10], as well as in research compilers [37]. Programs parts that can be represented are loop-based kernels with static control and affine memory accesses, referred to as SCoPs. They feature loop bounds, conditions and array subscripts that are affine forms of outer loop counters and runtime constants referred to as *parameters*. SCoPs cover a large range of compute-intensive mathematical programs where loop-level optimization is critical [52]. Moreover, the model can be extended beyond SCoPs [7].

The key aspect of the polyhedral model is to encode individual executions of each statement, called *statement instances*. Geometrically, affine loop bounds define a polyhedron in a multidimensional space, hence the name of the model.

2.1 Workflow in the Polyhedral Model

Contrary to conventional optimizers, polyhedral tools do not operate on syntactic forms. Instead, they transform or *raise* the code into a union of relations, then analyze and alter these relations, and finally generate the restructured code. The following provides more detail on these steps.

2.1.1 Raising. Transforming a program into a polyhedral representation is a matter of defining *iteration domains* of a statement and its *access relations*. Statement instances are identified by a vector of values of surrounding loop counters. An iteration domain of a statement is a set of all such vectors. For the sake of notation generality, sets are considered as degenerate relations with zero-dimensional input space. For example, the polynomial multiply kernel shown in Fig. 1a has one statement S whose domain is written in set-constructor notation as

$$\mathcal{D}_{\mathsf{S}}(N) = \{(i,j)^T \mid 0 \le i < N \wedge 0 \le j < N\}.$$

Access relations map statement instances to accessed array elements. Scalars are treated as zero-dimensional arrays. Union of such relations allows for encoding access to different arrays. For statement S in the same example, this access union is

$$\mathcal{A}_{\mathsf{S}}(N) = \{(i,j)^T \to (\mathtt{arr}, a_1)^T \mid (\mathtt{arr} = \mathsf{z} \wedge a_1 = i + j) \cup (\mathtt{arr} = \mathsf{x} \wedge a_1 = i) \cup (\mathtt{arr} = \mathsf{y} \wedge a_1 = j)\}.$$

```
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
S:    z[i+j] += x[i] * y[j];
```

(a) Original

```
#pragma omp parallel for private(t2)
  for (t1 = 0; t1 <= 2*N-2; t1++)
    for (t2 = max(0, t1-N+1); t2 <= min(t1, N-1); t2++)
S:    z[t1] += x[t2] * y[t1-t2];
```

(b) Transformed and Parallelized

Fig. 1. *Polynomial Multiply* computation kernel.

where `arr` corresponds to the name of accessed array and a_1 corresponds to its only dimension.

2.1.2 Program Transformation and Schedules. Changing the relative execution order of statement instances transforms the program. We can define a *scheduling relation* to map iteration domain points to logical execution dates. If these dates are multidimensional, statement instances are executed following the lexicographical order of their dates. Scheduling relations are expressive enough to encode a complex composition of program transformations including, e.g., loop interchange, fusion, fission, skewing, tiling, index-set-splitting, etc. [22].

For example, loop *tiling* [30] for the polynomial multiply can be expressed by the schedule

$$\theta_S(N) = \{(i,j)^T \rightarrow (t_1,t_2,t_3,t_4)^T \mid (32t_1 \leq t_3 \leq 32t_1{+}31) \wedge (32t_2 \leq t_4 \leq 32t_2{+}31) \wedge t_3 = i \wedge t_4 = j\},$$

where 32 is the tile size. Note that t_3 and t_4 are defined *explicitly* by equalities while t_1 and t_2 are defined *implicitly* by bounding inequalities, which correspond to integer division.

Schedule relations can be constructed manually or using high-level frameworks [2, 22, 31]. Automatic optimizers directly construct a scheduling relation with certain properties, including minimal reuse distances, tilability and inner/outer parallelism [8, 9]. However, they may fail to improve performance when achieving different properties requires contradictory transformations, for example exploiting spatial locality may be detrimental for parallelism [45].

2.1.3 Encoding Lexical Order. Throughout this paper, we use the so called $(2d+1)$ structure of scheduling relations. It introduces $(d+1)$ auxiliary dimensions to the scheduling relation [31] to represent lexical order. They are referred to as β-dimensions [22], as opposed to α dimensions that represent the execution order of the d loops. Zero-based contiguous constant values of β_i enforce the relative order between different objects (loops or statements) at depth i. They express code motion transformations such as loop fusion and fission. For example, the $(2d+1)$ form of the identity scheduling relation for polynomial multiply is

$$\theta_S(N) = \{(i,j)^T \rightarrow (\beta_1,\alpha_1,\beta_2,\alpha_2,\beta_3)^T \mid \beta_1 = 0 \wedge \alpha_1 = i \wedge \beta_2 = 0 \wedge \alpha_2 = j \wedge \beta_3 = 0\}.$$

Given that β-dimensions are constant, they can be concisely rewritten as a vector $\vec{\beta} = (0,0,0)^T$. β-vectors uniquely identify statements since no two statements can have the same lexical position. Prefixes of β-vectors (β-*prefixes*) uniquely identify loops, with their length corresponding to the nesting depth. Statements that share d loops, have identical β-prefixes of length d.

2.1.4 Program Analysis and Parallelism. They key power of the polyhedral model is its ability to compute exact instance-wise dependences [19]. Two statement instances are dependent if they access the same array element and at least one of them writes to it. For a program transformation to preserve original program semantics, it is sufficient that pairs of dependent instances are executed in the same order as before the transformation [32]. A dependence relation maps statement instances (dependence *sources*) to the instances that must be executed after them (dependence *sinks*). If a transformation inverses the execution order of dependent instances or assigns them the same logical execution time, the dependence becomes *violated* and the transformation is *illegal*. The polyhedral model provides means to verify the legality of a scheduling relation [4, 19, 43, 48].

Groups of instances, including loops, that do not transitively depend on each other may be executed in an arbitrary order, including in *parallel*. Loop-level parallelism is expressed by attaching a "parallel" mark to an α dimension, which requires code generator to issue a parallel loop.

2.1.5 Code Generation. After a scheduling relation is defined, code generation is a matter of building a program that scans the iteration domain with respect to the schedule [1]. Modern code generators rely on *generalized change of basis* that combines the iteration domain and the scheduling relation and puts scheduling dimensions in the foremost positions before creating loops from all dimensions. Several efficient algorithms and tools exist for that purpose including `CLooG` [3], `CodeGen+` [13] and `ppcg` [26]. For example, given the schedule $\mathcal{T}_S' = \{(i,j)^T \to (t1,t2)^T \mid t1 = i + j \wedge t2 = j\}$ that implements loop *skewing* for the polynomial multiply kernel and a parallel mark for dimension $t1$, `CLooG` may generate the code in Fig. 1b.

2.2 Transformation Directives

Even though polyhedral and syntactic approaches can be combined in an automatic tool [45], the polyhedral optimizer does not operate in syntactic terms and provides only little control over its parameters through compiler flags. Recently, Bagnères *et.al.* proposed the *Clay* transformation set that expresses a large number of syntactic loop transformations as structured changes to scheduling relations and rely on β-prefixes to identify targets [2]. They also proposed the *Chlore algorithm* that identifies a sequence of *Clay* primitives that would transform any given scheduling relation into another scheduling relation.

For example, the aforementioned loop *skewing* transformation is expressed as a dimension substitution: $\textsc{Skew}(\vec{\rho}, i, k)$: $\forall \theta_S : \vec{\beta}_{S,1..\dim\vec{\rho}} = \vec{\rho}, \alpha_{\dim\vec{\rho}} \mapsto \alpha_{\dim\vec{\rho}} + k \cdot \alpha_i$. Any occurrence of the output dimension $\alpha_{\dim\vec{\rho}}$ is replaced by a linear combination of itself with another output dimension α_i. Thus, the schedule $\mathcal{T}_S'$ from the previous section is obtained from the identity schedule by $\textsc{Skew}((\beta_1)^T = (0)^T, i = 2, k = 1)$ where β_1 identifies the outer `i` loop. Loop Reshape is similar to Skew except that it uses a linear combination of the *input* rather than *output* dimensions.

Transformations of the lexical order are encoded as modifications of β-vectors. For example, fusing two subsequent loops is expressed as $\textsc{FuseNext}(\vec{\rho})$: $\forall \theta_S : \vec{\beta}_{S,1..\dim\vec{\rho}-1} = \vec{\rho}_{1..\dim\vec{\rho}-1} \wedge \vec{\beta}_{S,\dim\vec{\rho}} = \vec{\rho}_{\dim\vec{\rho}} + 1, \vec{\beta}_{S,1..\dim\vec{\rho}-1} \leftarrow \vec{\rho}_{S,1..\dim\vec{\rho}}, \vec{\beta}_{S,\dim\vec{\rho}} \leftarrow \vec{\beta}_{S,\dim\vec{\rho}} + \max_{T:\vec{\beta}_T=\vec{\rho}} \vec{\beta}_{T,\dim\vec{\rho}}$, where $\dim\rho$ encodes fusion depth. This transformations assigns equal β values up to given depth, which corresponds to fusion, and updates the remaining ones to maintain uniqueness and contiguity.

Clay transformations are applicable to unions of scheduling relations such that the entire union (but not necessarily individual relations) is left-total and injective. Internally, *Clay* operates on a matrix representation of systems of linear inequalities and supports arbitrarily complex transformations as long as the properties of an union are preserved. *Chlore* algorithm builds on matrix decompositions to identify sequences of *Clay* primitives that transform one set of matrices into another set. More information and full specification of transformations is available in [2].

Although *Clay* and *Chlore* enable interaction with a polyhedral engine using syntactic terms, they face several challenges in application. (1) *Target selection*—β-prefixes are required for each transformation, yet they are not easily accessible in the source code. (2) *Target consistency*—the generated code may have a different structure than the original code, for example due to loop separation [3], resulting in a mismatch between β-vectors and loop nesting. (3) *Effect separation*—even if *Chlore* produces a sequence of primitive transformations, it is difficult to evaluate (potentially negative) effects of individual transformation by reading the polyhedrally transformed code.

We address these challenges with *Clint*, a new interactive tool based on a graphical representation of SCoPs which: simplifies *target selection* to directly choosing a visualization of a transformation

target; maintains *target consistency* by matching the visualization to the original (often simpler) code; and replays primitive "steps" of transformation to *separate* their *effects*, supporting further interactive modification.

3 DIRECTLY MANIPULATING POLYHEDRAL VISUALIZATIONS

To reduce the burden of code editing and transformation primitive application, we propose *Clint*, an interactive loop-level transformation assistant based on the polyhedral model. It leverages the geometric nature of the model by presenting SCoPs in a directly manipulable [46] visualization that combines scatter plots of iteration domains and node-link diagrams of instance-wise dependences. This approach is similar to the one commonly used in the polyhedral compilation community to illustrate iteration domains. *Clint* goes beyond these static views by allowing program transformation to be initiated directly from the visualization, and provides an animation-based visual *explanation* of an automatically computed program transformation. Animated transitions correspond to program transformations that, when applied, would change the program to obtain the final visualization. The user can replicate the action by directly manipulating the visualization similarly to the transition or in a more elaborate way. The set of interactive manipulations builds on the geometry-related vocabulary of classical loop transformations, such as skewing or shifting, which is expected to give the user supplementary intuition on the transformation effects and to support exploration and learning.

The design of *Clint* is motivated by the need for (1) a single and consistent visual interface to bridge the gap between dependence analysis and subsequent program transformation; (2) an efficient way to explore multiple alternative loop transformations without rewriting the code; (3) explaining the code modifications yielded by an automatic optimization. Although built around the complete *Clay* transformation set [2], it can be extended to support different transformations as long as effects of any transformation can be undone by (a sequence of) other transformations. *Clint* seamlessly combines loop transformations to support reasoning about execution order and dependences rather than loop bounds and branch conditions. The interactive visual approach reduces parallelism extraction to visual pattern recognition [50] and code transformation to geometrical manipulations, giving even non-expert programmers a way to manage the complexity of the underlying model [38]. Finally, it brings insight into the code-level effects of the polyhedral optimization by decomposing a complex program transformation into primitive steps and providing a step-by-step visual replay, independent of how an automatic optimizer operates internally.

3.1 Structure of the Visualization

Clint visualizes *scheduled* iteration domains, e.g., statement instances mapped by the scheduling relation to the new coordinates in logical time space, see Fig. 2, for an example of a simple code and its corresponding visualization. The main graphical elements are as follows.

Points and Polygons. Our visualization consists of *polygons* containing *points* on the integer lattice. Each point represents a statement instance, positioned using values of α dimensions. Points are linked together by arrows that depict instance-wise data flow between them. The polygon delimits the loop bounds in the *iteration space* and is computed as a convex hull of the points it includes. The space itself is displayed as a coordinate system where axes correspond to loop iteration variables.

Color Coding. Statements are color coded to ensure matching between code and visual representations. A transformation, such as peeling or index-set-splitting, may result in sets of instances of the same statement being executed in different loop nests. We refer to this case as multiple *occurrences* of the statement. Different occurrences of the statement share the same color coding.

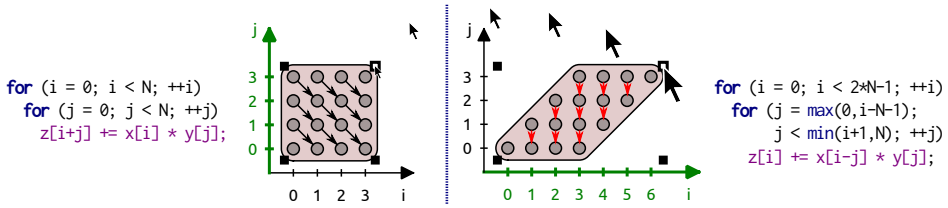


Fig. 2. Performing a skew transformation to parallelize polynomial multiplication loop by deforming the polygon. The code is automatically transformed from its original form (left) to the skewed one (right).

Coordinate Systems. Each coordinate system is at most two-dimensional. The horizontal axis represents the outer loop, and the vertical axis represents the inner loop. Statement occurrences enclosed in both loops are displayed in the same coordinate system, with optional slight displacement to discern them (see Fig. 4). Statement occurrences that share only the outer loop are placed into different coordinate systems, vertically aligned so that they visually share the horizontal axis. We refer to this structure as *pile* (see Fig. 8b). Finally, statement occurrences not sharing loops are displayed as a sequence of piles (see Fig. 8a), arranged to follow the lexical order.

We use β-vectors internally to arrange polygons and coordinate systems. Statements with identical β-prefixes of length d share a coordinate system if d is the depth of the inner loop, and a pile if d is the depth of the outer loop. Consequently, coordinate systems and piles are uniquely identified by a β-prefix.

Execution Order. Statement instances are executed bottom to top, then left to right, crossing the bounds of coordinate systems in both cases. Multiple instances sharing a loop iteration are executed in the order of increasing displacement. Arrows point at the instance executed second.

Tiling. Tiled domains are displayed as polygons with wide lines inside to delimit tile shapes. All dimensions that are *implicitly defined* (see Section 2.1.2) are considered as tile loops and serve to build the tile shapes creating scatterplots with *nested* axes [28]. Tiling makes execution order two-level: entire tiles are executed following the previously described order; instances inside each tile are executed bottom to top, then left to right without crossing tile boundaries.

Multiple Projections. The overall visualization is a set of two-dimensional projections, where loops that are not matched to the axes are ignored. As the goal of *Clint* is program transformation, we only display projections on the schedule α-dimensions, which coincide with iteration domain dimensions before transformation. For a single statement occurrence, they may be ordered in a scatterplot matrix as in Fig. 5a. The points are displayed with different intensity of shade depending on how many multidimensional instances were projected on this point. We motivate the choice of 2D projections vs 3D visualization by easier direct manipulation with a standard 2D input device (e.g., mouse) [5, 14] and consistency of the visualization for even higher dimensionality.

Dependences and Parallelism. Dependences between points in the same coordinate system are shown as arrows pointing from source to sink. By default, only *direct* (i.e., non transitively-covered) dependences are shown. When hovering a point, *all* its dependences are visualized. Dependences between vertically or horizontally adjacent coordinate systems are aggregated into large dots (Fig. 7b). Finally, dependences between points in distant coordinate systems are only visualized when either their source or sink is being manipulated to avoid visual cluttering. Arrows and dots turn red if the dependence is violated. Transformation legality check is performed parametrically. If

legality violation exists for values of parameters other than currently selected, the polygon contour turns red instead of arrows.

Generally, parallel dependence arrows imply some parallelism is present in the loops – e.g., if they are orthogonal to an axis, the loop corresponding to an axis features DoAll parallelism. *Clint* highlights "parallel" axes in green to simplify parallelism identification (see Fig. 2).

Parametric Domains. Domains whose bounds involve parametric expressions are visualized for a fixed value of the parameters. By default, all parameters are assigned identical values computed as follows. *Clint* computes the *dependence distance sets* from dependence relations by subtracting the relation's range from its domain. It then takes the maximum non-parametric absolute value across all dimensions. Finally, it takes a minimum of this value and a predefined constant. We selected this constant as 6 from our preliminary studies, observing that it is sufficient to represent the majority of dependence patterns in our test suite. The user can dynamically modify values of individual parameters and the visualization will be automatically updated.

3.2 Directly Manipulable Visual Objects

Since program transformations in the polyhedral model correspond to changes of the statement instance order, they can be performed on the visual representation of that order. In *Clint*, the execution dates are mapped to point positions. Therefore, moving points corresponds to program transformations. Visual marks such as points and polygons *afford* direct manipulation, i.e., they can be dragged and dropped directly to the desired position.

Because many of the visual elements are mapped from the underlying SCoP properties, manipulation should be structured so as to maintain those properties. For example, point coordinates should remain integer to properly map to counted `for` loops. Furthermore, the polyhedral model represents *parametric* iteration domains—having constant yet unknown sizes—making it technically impossible to schedule each instance separately. Therefore, we only enable *structured* point manipulation that can be mapped to similarly structured program transformations as expressed in, e.g., *Clay* framework. Visually, we use polygons and coordinate systems as manipulation *substrates* [34] that mediate interaction with groups of points while ensuring structure preservation.

We refer to polygons and coordinate system as point *containers*. They can be seen as *persistent selection* of the points manipulable together and sharing a common property: representing instances of the same statement or being enclosed in the same loops. Polygons and coordinate systems also allow to *reify* the conventional target selection and make it a first-class interactive object [6]. The user no longer needs an explicit (and sometimes cumbersome) selection step, by either clicking or lassoing the objects with cursor, before starting the manipulation.

3.3 Mapping Interactions to Loop Transformations

As motivated above, we center the manipulation around polygons. We augment the polygon with *handles* at its corners and borders, similarly to a conventional graphical editor. They appear when the polygon is hovered and support many transformations without using any instruments or modes.

We rely on structured scheduling relation modifications of *Clay* framework, most of which were inspired by well-known "classical" loop transformations [52]. Some of them map directly to *Clint* visualization (e.g., Shift), while others do not (e.g., Interchange) or, even worse, can be mapped in a misleading way (Skew). Therefore, instead of trying to map *Clay* transformations, we rather follow an interaction-centered approach by mapping the *possible* graphical actions to *sequences* of *Clay* transformations. Fig. 3 lists the graphical actions and the corresponding program transformations. The action parameters correspond to the attributes of the object being manipulated or properties of the manipulation.

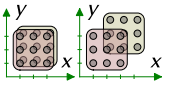
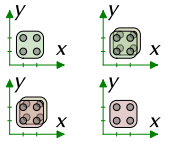
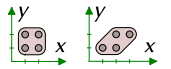
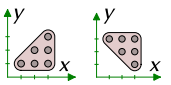
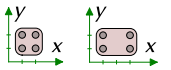
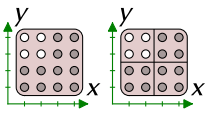
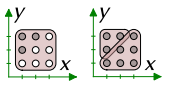
Action	Parameters	Transformations	Before/After
Drag polygon within CS	$\vec{\beta}, x, y, dx, dy$	1. Shift($\vec{\beta}, x, dx$) 2. Shift($\vec{\beta}, y, dy$)	
Drag polygon between CS	$\vec{\beta}, x, y, \vec{\rho}$	1. Reorder($\beta_{1..y}$, put last), $\beta_y \leftarrow \max_S \beta^S_y - 1$ 2. Distribute($\beta_{1..y}$), $\beta_{y-1..y} = (\beta_{y-1} + 1, 0)^T$ 3. *repeat 1,2 until dimension x* 4. Reorder($\beta_{1..x}$, put after ρ_x), $\beta_x \leftarrow \rho_x + 1$ 5. FuseNext($\beta_{1..x}$), $\beta_{x..x+1} \leftarrow (\beta_x - 1, \max_S \beta^S_{x+1} + 1)$ 6. *repeat 4,5 until dimension y* 7. Reorder($\beta_{1..y}$ put last)	
Drag corners from center	$\vec{\beta}, x, y, dx, dy, sx, sy$	1. Reshape($\vec{\beta}, y, x, \lfloor dx/sy \rfloor$) 2. Reshape($\vec{\beta}, x, y, \lfloor dy/sx \rfloor$) *use skew when possible*	
Drag corners towards center	$\vec{\beta}, x, dx, sx$ (y axis used if $dy > dx$)	1. Interchange($\vec{\beta}, x, y$) if $\lfloor dx/sx \rfloor \bmod 2 = 1$ 2. Reverse($\beta_{1..x}$) if $1 \le \lfloor dx/sx \rfloor \bmod 4 \le 2$ 3. Reverse($\beta_{1..y}$) if $\lfloor dx/sx \rfloor \bmod 4 \ge 2$	
Drag border	x, dx, sx	1. Densify($\vec{\beta}$) 2. Reverse($\vec{\beta}$) if $dx < 0$ 3. Grain($\vec{\beta}, \lfloor dx/sx \rfloor$)	
Click on rectangular selection of points	$\vec{\beta}, x, y, tx, ty$	1. Interchange($\beta_{1..y+2}, y, y+1$) if y *implicitly defined* 2. Linearize($\beta_{1..y+1}$) if $y + 1$ *implicitly defined* 3. Linearize($\beta_{1..x}$) if x *implicitly defined* 4. StripMine($\beta_{1..x}, tx$) 5. StripMine($\beta_{1..y+1}, ty$) 6. Interchange($\beta_{1..y_2}, y, y + 1$)	
Select points and move	$\vec{\beta}, \vec{\rho}$, *selection shape* $\{f_i(x, y) \ge 0\}$	1. $\forall i$, IndexSetSplit($\vec{\beta}, f_i$) if $\vec{\rho} = \varnothing$ Collapse($\vec{\rho}$) otherwise.	

Fig. 3. Mapping between interactive polygon manipulations and Clay transformations. $\vec{\beta}$ identifies the statement occurrence corresponding to the polygon; $\vec{\rho}$ identifies the β-prefix of the coordinate system; x and y are loop depths corresponding to the horizontal and vertical axes, respectively; dx and dy are cursor offsets from its position when the manipulation started; sx and sy are sizes of the polygon; tx and ty are sizes of the selection. Offsets and sizes are expressed in coordinate system units, i.e., iterations.

For example, dragging a polygon along one of the axes directly corresponds to the Shift transformation. However, dragging it to a different coordinate system corresponds to a complex sequence of *Clay* directives that perform code motion (see "Drag polygon between CS" in Fig. 3). Transformations that result in an *identical* schedule are omitted, for example, no Reorder is applied before Distribute if the statement occurrence is already the last in the loop.

Polymorphic Actions. The coordinate system can be automatically extended to fit the polygon being dragged. We leverage the *equivalence* property of transformation to stop automatic extension.

Shifting past the largest bound does not change the *relative* execution order. In such cases, the polygon goes outside the coordinate system, which is shrunk to fit only the remaining polygons.

Parametric Transformations. Transforming a parametrically-bounded domain may result in parametric transformations. In particular, we look for a parametric bound closest to the mouse cursor at the end of manipulation. For example, the amount of Shift is computed with respect to the closest bound of the polygon other than the one being shifted. Alternatively, the conditions for IndexSetSplit are (first) computed as affine expressions of the closest bound. If there is no such expression, they are computed without using parameters.

Skew and Reshape. By default, the graphical action of skewing corresponds to the Reshape transformation, and not the Skew transformation. The latter transforms the loop with respect to the *current expression* for the other loop rather than to the *original* iterator. This makes Skew transformation combine badly: if the x loop is skewed by y to become $(x+y)$, it becomes impossible to skew y by x as it does not appear independently of y anymore. The graphical intuition behind loop skewing does not hold for combinations of skews. However, when a Reshape is identical to Skew, *Clint* will perform a Skew since it is one of the well-known classical transformations[1].

Targeting Individual Statements. Many *Clay* transformations operate on β-prefixes, that is loops rather than statements. We circumvent this by distributing away the target statement, applying the desired transformation to a loop nest with only this statement, and then fusing everything back.

Manipulating Multiple Statements. If multiple polygons are selected within a coordinate system, transformations are applied to all of them in inverse lexicographical order of their respective β-vectors. Inversion prevents transformations from modifying β-vectors used to target subsequent transformations. If a user manipulates a pile (or a coordinate system), the action is propagated to all the polygons it contains, making the pile an implicit selector for the polygons it contains.

Manipulating Groups of Points. Individual points or groups thereof can be manipulated by turning them into a polygon first. Selecting a group of points and dragging it away from existing polygon separates it into two parts, mapping to the IndexSetSplit transformation. It creates a new statement occurrence that can be manipulated separately. Dropping this polygon on top of another polygon that represents a different occurrence of the same statement is mapped to the Collapse transformation. In cases of selections that are not adjacent to borders and/or not convex, multiple IndexSetSplit transformations are performed. Each of the two resulting parts may correspond to multiple occurrences of the statement, but is visualized and manipulated as a whole.

Cross-Projection Selections. When multiple projections are used, the selection of statement instance points is combined from different projections. The overall multidimensional selection is an intersection of constraints imposed by each separate two-dimensional selection. Empty selection in a projection is thus equivalent to selecting everything.

Decoupling Visualization from Code. In *Clint*, we keep the visualization consistent with the original program structure unless the user manually modifies the code. This allows for manipulating multiple statement occurrences together, for example in case of shifting one statement with respect to another inside the loop, which may result in loop *separation* as in Fig. 4.

[1]In fact, we created Reshape transformation in *Clay* to address the skew combination problem. It was the last missing transformation that enabled completeness of the set.

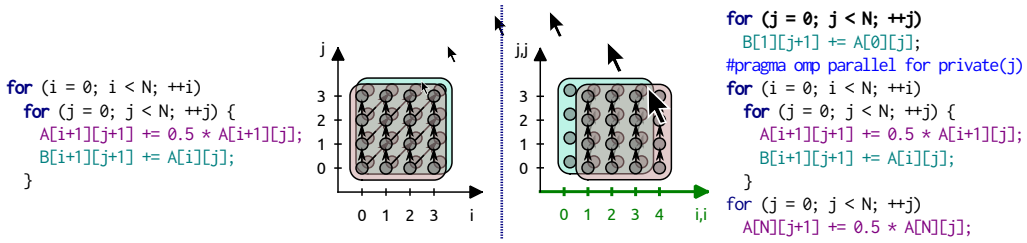


Fig. 4. Manipulation for SHIFT Transformation: the darker polygon is dragged right so that dependence arrows become vertical without spanning between different iterations on `i`. The visualization is then decoupled from the code structure, and both statements can still be manipulated as if they were not split between two loops.

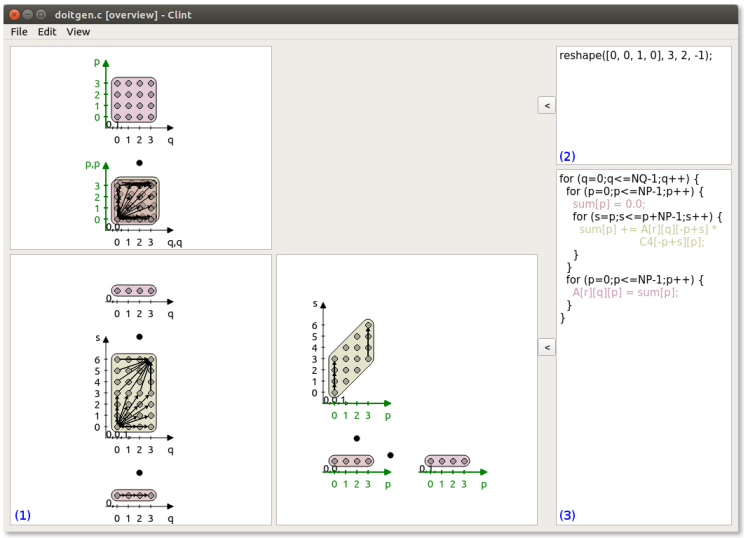
Transformation Legality Feed-Forward. Clint graphical interactions are structured so that it is possible to identify the transformation before it is completed. For example, dragging a corner of a polygon away from its center corresponds to a RESHAPE, the dragging direction and distance define transformation parameters. Since they are typically expressed in units of iteration steps through division, we can use *ceil* instead of normal rounding to obtain the parameters earlier. Hence *Clint* can perform a transformation *before* the end of corresponding user interaction. This allows to provide *feed-forward* about the transformation, i.e., its effects (in particular dependence violation) are visualized *during* the interaction, guiding the user in their choice. In addition, this approach allows *Clint* to hint the user about the state of the visualization if they finish manipulation immediately using a grayed-out *preview* shape (see Fig. 8).

3.4 Mapping Loop Transformations to Animated Transitions

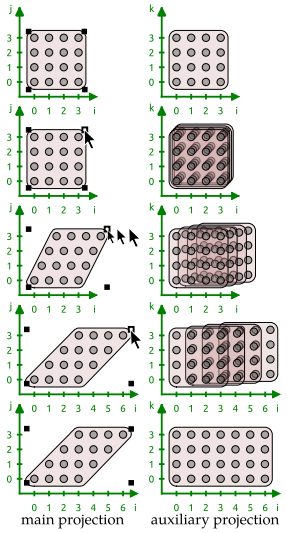
Clint visualization enables the illustration of step-by-step execution of a *Clay* transformation script, either constructed manually or translated from a compiler-computed schedule using *Chlore* [2]. Instead of providing a one-to-one mapping between individual transformations and animated transitions, we take a generalized approach based on the structure of transformations. They can be divided based on the scheduling relation dimensions they affect: (1) only α, (2) only β or (3) both α and β. The first group contains all transformations except FUSENEXT, DISTRIBUTE and REORDER, which belong to the second group, and STRIPMINE, LINEARIZE, INDEXSETSPLIT and COLLAPSE, which belong to the third group. This classification allows us to limit the animation scope. Transformations that do not modify β-dimensions may only affect points inside one container while points cannot be moved between containers. Furthermore, only the projections on iterators involved in the transformation should be updated. Transformations that only modify β-dimensions affect entire containers without modifying the point positioning inside them.

Within-Container Transformations. Transformations of the first group are animated by simultaneously moving individual points to their new positions. During the transition, polygonal shapes are updated to match the convex hull of the respective points. Thus SHIFT transformation moves all points simultaneously in one direction and corresponds to visual displacement, while RESHAPE transformation moves rows (or columns) of points at different lengths and results in shape skewing.

Multiple Projections. Several transformations operate on two dimensions, for example RESHAPE and INTERCHANGE. For these cases, we consider the projection on both of these dimensions as the *main* one, and the projections on one of the dimensions as *auxiliary* ones. In the main projection, the one-to-one point transition remains applicable. On the other hand, in the auxiliary ones, points may be created or deleted. For example, an auxiliary projection retains the rectangular



(a) *Clint* interface includes: (1) interactive visualization with multiple projections, (2) editable history view of transformations, and (3) source code editor; all coordinated with each other.



(b) When main projection is manipulated, auxiliary projections are updated simultaneously.

Fig. 5. *Clint* displays multiple projections for deep loop nests.

shape after a RESHAPE but becomes larger as some points are projected onto new coordinates (see Fig. 5). *Clint* handles this by introducing a temporary third axis, orthogonal to the screen plane. This axis corresponds to the dimension present in the transformation, but not in the projection. Points and arrows are then re-projected on three dimensions. Extra objects become visible only during the animated transition and create a pseudo-3D effect. After the transition, the third axis is deleted while the projected points remain in place (see Fig. 5b). This technique is analogous to ScatterDice [18], but without axis switching.

Between-Container Transformations. As transformations of the second group affect entire polygons only, we can translate them into motion of polygons. If all polygons of a container are moved, the entire container is moved instead. Target containers are identified using β-prefixes.

Container Creation and Deletion. Transformations of the third group may result in containers being created or deleted. However, without points, a polygon would correspond to statement occurrence that has no instances and thus is never executed. Therefore, it must be impossible to create empty containers. The only way to create a container in *Clint* is by *splitting* an existing container into multiple parts. This exactly corresponds to the INDEXSETSPLIT transformation if the container is a polygon. It also maps to the DISTRIBUTE transformation when the container is a coordinate system or a pile. Conversely, COLLAPSE and FUSENEXT transformations correspond to visually *joining* two containers.

3.5 Clint Interface

Clint combines three editable and synchronized representations (see Fig. 5a): (1) the interactive visualization; (2) a navigable and editable transformation history view based on Clay scripts; and (3) the source code editor. A consistent color scheme is used between the views to match code

statements to the visualization. Transformation directives corresponding to graphical actions are immediately appended to the history view. The user can then navigate through the history by selecting an entry, which will update the visualization to the corresponding previous state, or edit it directly using Clay syntax. As the target code tends to become complex and unreadable after several manipulations, the user has the option to keep the original code visible instead of the transformed one. Finally, when the code is edited, the visualization is updated, thus making *Clint* a dynamic visualizer for polyhedral code.

4 USE SCENARIOS

Clint can be used as a stand-alone program transformation tool or in conjunction with an automatic optimizer. In the first case, the user must decide on the transformation to perform. In the second case, *Clint* proposes a sequence of primitive transformations equivalent to the automatically computed one, letting the user complement or modify it independently from the optimizer. In both cases, the user may reason in terms of an instance-wise dependence graph rather than in terms of loop transformations or parameters of the optimization algorithm.

Our approach does not impose a particular transformation heuristic. Instead, we suggest to build intuition by visualizing (optimized) programs that perform well and identifying visual *patterns*. For an optimization expert, these patterns may eventually lead to a novel heuristic. We provide two end-to-end illustrative examples, in which we attempt to make dependence arrows short to improve reuse and orthogonal to axes to exploit parallelism.

4.1 Assisted Semi-Automatic Transformation

Clint can be used as a tool for applying loop-level transformations that provides instant legality feedback and generates transformed code automatically. Let us continue with the polynomial multiplication kernel example, see Fig. 1a, to demonstrate how a long sequence of transformations can be applied. Default representation of the kernel, with parameters set to 4, is shown in Fig. 6a. The loop `j` features parallelism and is marked accordingly. Inner parallelism is often less desirable as it would incur barrier synchronization cost on every iteration of the outer loops. Therefore, observing that dependence arrows are diagonal, the user may decide to make them orthogonal to the `i` loop to make it parallel. They can do so by dragging the top right handle of the polygon right, Fig. 6a. However, such transformation is illegal as indicated by the red arrows that point in the direction opposite to the `j` access. This dependence violation can be removed by switching the direction of arrows, which is achieved by dragging the top right handle left to rotate the polygon around its center, Fig. 6b. The combined transformation sequence is now legal yet potentially inefficient: different iterations of parallel loop `i` execute different numbers of statement instances. Observing the symmetry of the polygon, the user selects a triangular-shaped group of points on the right, Fig. 6c, and drags it to the empty space on the left, Fig. 6d, until the balanced, rectangular shape is reconstructed, Fig. 6e. The final transformation corresponds to loop skewing, followed by two loop reversals and shifts, then by index-set splitting, and finally by shifting. However, at no time during transformation, the user must be aware of particular loop transformations, their legality or the transformed code. They can operate on an instantiation of the instance-wise dependence graph as opposed to directive-based approaches where, even with visualization, they would have to find the transformation directive that would result in a desired visual shape.

4.2 Understanding, Improving and Rectifying Automatic Transformation

Manual program transformation, even with efficient support tools, may require sufficient effort from the programmer. Fully automated program optimizers are designed to yield decent performance in

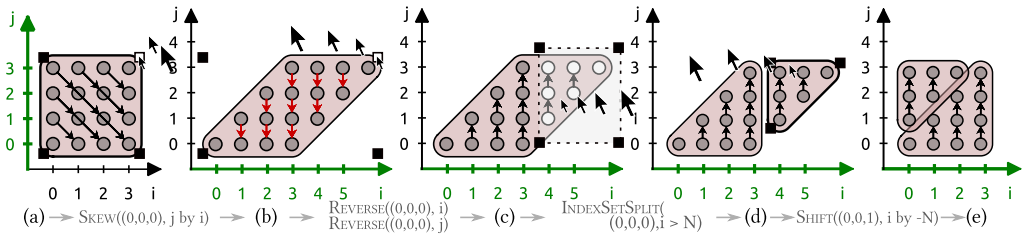


Fig. 6. Users can directly manipulate the visual representation of SCoPs and have the transformed program generated automatically. Dragging the corner from the center performs loop skewing, to the center—reversal; selecting the points and dragging them performs index-set splitting followed by loop shifting. Dependence arrows orthogonal to axes enable parallel execution.

most cases. However, they are based on imprecise heuristics, which may fail to improve performance or even degrade it. Polyhedral optimizers are essentially source-to-source black boxes offering little control over the optimization process. *Clint* relies on *Chlore* [2] to find a sequence of primitive directives equivalent to the automatically computed optimization and let the user replay and modify it, independently of the optimization algorithm. *The user does not have to know or understand the internal operation of the optimizer and its configuration.*

Consider the Multi-Resolution Analysis Kernel code, available in `doitgen` benchmark of the PolyBench/C 4.2 suite [42] and presented in Fig. 7a. A sequential version of this kernel runs in 0.83s on our test machine.[2] We applied Pluto[3] polyhedral compiler [11] to extract parallelism from this code. We also requested Pluto to tile the transformed code, which is likely to improve performance thanks to data locality and expose wavefront parallelism. A simplified version of the resulting code is presented in Fig. 7c. It indeed contains tiled and parallelized loops. Yet this code executes in 0.91s, a 10% slowdown compared to the sequential version (untiled parallel version executes in 50.1s, a 62× slowdown). Without any further suggestion from Pluto, the user may either stick with a non-transformed sequential version or with a non-efficient parallel one. The code was transformed so aggressively that the user is unlikely to attempt code modifications or even understanding the transformation that was applied.

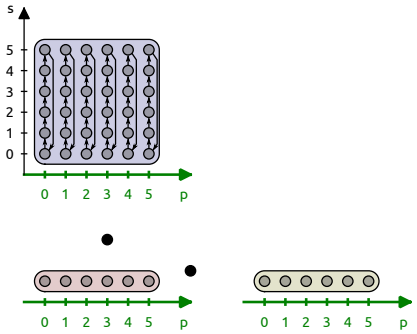
Comparing *Clint* visualizations before, Fig. 7b, and after, Fig. 8a, transformation suggests loop fission took place, which can also be inferred from the generated code. Step-by-step replay confirms this and also demonstrates loop tiling followed by skewing. It also shows that *inner* loops were parallelized, which is known to result in large barrier synchronization overheads. A fat dot between coordinate systems indicates there is some reuse between loops, but it is unclear whether Pluto performed fission to ensure legality of skewing and tiling or because of its fusion heuristic. To discover that, the user may undo the fission by fusing the loops back together, Fig. 8a. While they drag the polygon, legality *feedforward* appears in a shape of gray arrows that indicate that transformation would be legal and would preserve parallelism. Motivated by the success and observing the remaining reuse, the user may decide to fuse the remaining loop as well. This transformation would be illegal as indicated by red arrows appearing as the polygon is being dragged. The users can still finish the manipulation, and then use a conventional "undo" command.

[2] 4× Intel Xeon E5-2630 (Sandy Bridge, 6 cores, 15MB L3 cache), 64 GB RAM, running CentOS Linux 7.2.1511, compiled with GCC 4.9.3 with `-O3 -march=native` flags, benchmark size `LARGE`, `NQ`= 140, `NR`= 150, `NP`= 160. Average of 12 runs is reported, kernel execution time only, using high-resolution CPU timers.

[3] Pluto 0.11.4 with `--parallel --tile`, as available on https://github.com/bondhugula/pluto/releases/tag/0.11.4

```
for (r = 0; r < NR; r++)
  for (q = 0; q < NQ; q++) {
    for (p = 0; p < NP; p++) {
      sum[p] = 0.0;
      for (s = 0; s < NP; s++)
        sum[p] += A[r][q][s] * C4[s][p];
    }
    for (p = 0; p < NP; p++)
      A[r][q][p] = sum[p];
  }
```

(a) Original Kernel



(b) Visual Representation

(c) Pluto-transformed Kernel ⟶

```
for (t1=0;t1<=NR-1;t1++)
  for (t2=0;t2<=NQ-1;t2++) {
    lbp=0;
    ubp=floord(NP-1,32);
#pragma omp parallel for \
 private(lbv,ubv,t5,t6,t7)
    for (t4=lbp;t4<=ubp;t4++) {
      lbv=32*t4;
      ubv=min(NP-1,32*t4+31);
      for (t5=lbv;t5<=ubv;t5++)
        sum[t5] = 0.0;
    }
#pragma omp parallel for \
 private(lbv,ubv,t5,t6,t7)
    for (t4=lbp;t4<=ubp;t4++)
      for (t5=0;t5<=floord(NP-1,32);t5++)
        for (t6=32*t5;
             t6<=min(NP-1,32*t5+31);t6++) {
          lbv=32*t4;
          ubv=min(NP-1,32*t4+31);
          for (t7=lbv;t7<=ubv;t7++)
            sum[t7] += A[t1][t2][t6] *
                C4[t6][t7];
        }
#pragma omp parallel for \
 private(lbv,ubv,t5,t6,t7)
    for (t4=lbp;t4<=ubp;t4++) {
      lbv=32*t4;
      ubv=min(NP-1,32*t4+31);
      for (t5=lbv;t5<=ubv;t5++)
        A[t1][t2][t5] = sum[t5];
    }
  }
```

Fig. 7. Multi-resolution Analysis Kernel adapted from [42].

The final manually retouched version runs in 0.67s with a (modest) 25% speedup. Without step-by-step replay and direct manipulation, it would be hard to experiment with different fusion structures using a general trial-and-error strategy.

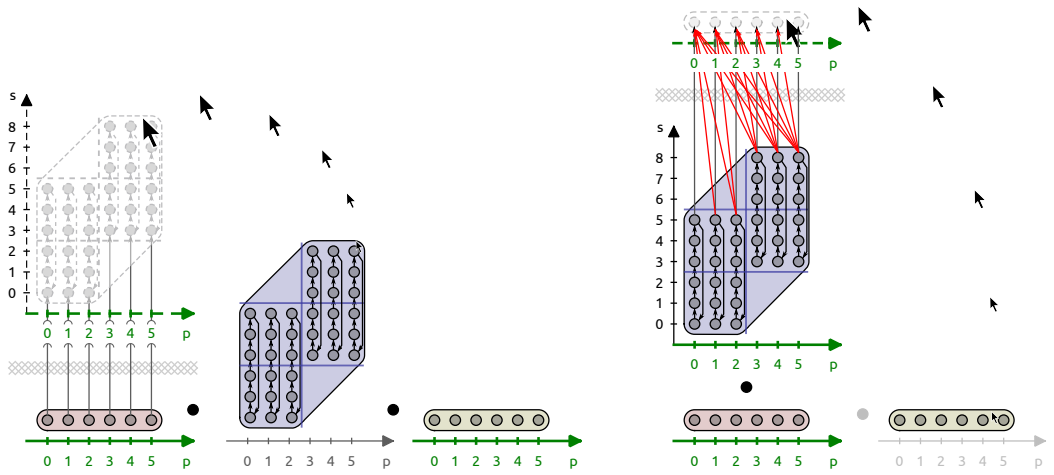
Although loop fusion is often implemented as a separate optimization problem in polyhedral optimizers, it is no easier to control externally. Clint *allows users to understand and directly modify the fusion/fission structure, instead of reasoning about how a particular heuristic would behave.*

5 ASSESSING THE USABILITY OF CLINT

Particular use cases of the previous section illustrate well the potential benefits of the tool in specific cases, but they do not help evaluating and understanding its overall usability in more general cases and with different users. Therefore, as it is commonly done in Human-Computer Interaction, we conducted a series of user studies considering more abstract tasks that assess the usability of *Clint*.

5.1 Understanding the Visualization

Although similar visualizations have been already used for descriptive or pedagogical purposes, there is no empirical evidence of their appropriateness for conveying program structures. We designed an experiment to assess the suitability of our visual representation. In particular, we test whether both experts in the polyhedral model and non-expert programmers can establish a bidirectional mapping between *Clint* visualization and code.



(a) Undoing distribution (b) Trying more fusion

Fig. 8. Using visual representation to re-adjust automatically computed transformation with immediate feed-forward on semantics preservation. Dependences violated by the intended transformation turn red, lines within shapes depict tiles. Shaded shapes are positions before manipulation.

5.1.1 *Protocol.*

Participants. We recruited 16 participants (aged 18-53) from our organizations. All of them had experience in programming using imperative languages with C-like syntax and basic understanding of the polyhedral model and its limitations. Six participants reported to have *manually constructed* similar visualizations from scratch and were therefore considered *Experts*. Because participants were asked to construct visualizations following given rules, previous exposure to these rules is a more *relevant* criterion of expertise than familiarity with the polyhedral model.

Procedure. Our experiment is a $[3 \times 2]$ mixed design having two factors:

- Task: mapping direction (between participants)
  - Visualization to Code (*VC*) — writing a code snippet corresponding to a given visualization using a C-like language featuring loops and branches with affine conditions;
  - Code to Visualization (*CV*) — drawing an iteration domain visualization given the corresponding code.
- Difficulty: problems may be (within participants)
  - *Simple* — two-dimensional with constant bounds;
  - *Medium* — multi-dimensional with constant bounds;
  - *Hard* — two-dimensional with mutually-dependent bounds and branches.

We divided participants in two groups with equal number of experts. Group 1 performed the *VC* task, group 2 performed the *CV* task. This *between participant* factor allowed us to present the same problems to all participants while avoiding learning effect. Both tasks were performed on paper with squared graph support for the *CV* task. Participants were instructed about visualization and performed two practice tasks before the session. They were asked to work as accurately as possible without time limit and were allowed to withdraw from a task. Expected solutions were shown at the end of the experiment. Each session lasted about 20 minutes.

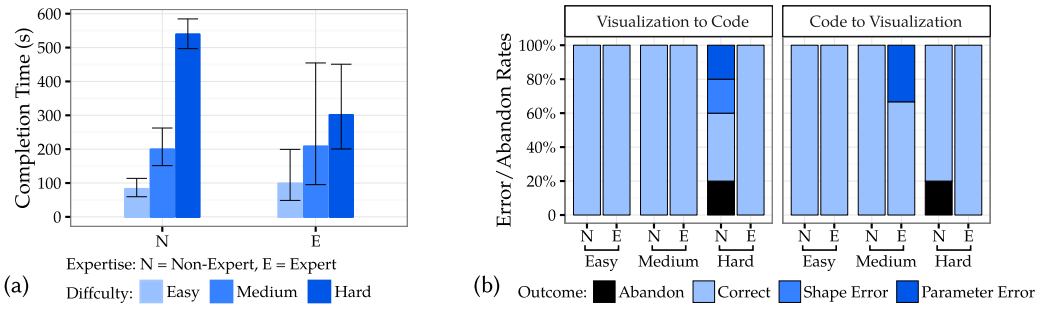


Fig. 9. (a) *Completion Times* increase with task difficulty but less so for *Experts*. Results are similar between *Experts* and *Non-Experts*. Error bars are 95% confidence intervals. (b) overall *Error Rate* is low. *Experts* are more successful but fail at simpler tasks; *Non-Experts* may abandon.

Data Collection. For each trial, we measured *Completion Time*, *Error* and *Abandon* rates. The errors were split in two categories: *Parameter Errors*, when the shape of the resulting polyhedron was drawn correctly, but linear sizes or position were wrong; *Shape Errors*, when the shape of the polyhedron was incorrect. Codes describing the same iteration domain were considered equivalent (e.g., `i <= 4` and `i < 5`). Upon completion, participants filled out a demographics questionnaire.

Data Processing and Analysis. We performed *log-transformation* of the *Completion Time* to compensate for the positive skew of its distribution, resulting in asymmetric confidence intervals. Due to concerns over the limits of null hypothesis significance testing in various research fields [15, 17], our analyses are based on *estimation* [16]. We report symmetric effect sizes on means $-es = 2(m_1 - m_2)/(m_1 + m_2)$ where m_1, m_2 are means– and 95% confidence intervals (CIs).

5.1.2 Results. We did not observe significant order effect on the *Error Rate* or *Completion Time*, meaning that there were neither learning nor fatigue effect along the experiment.

Completion Time. We discarded 7 trials in which participants produced erroneous code. Task did not strongly affect the *Completion Time*: *VC* took 182s ($95\%CI = [127s, 262s]$) on average while *CV* took 215s ($95\%CI = [156s, 296s]$) on average, resulting in an effect size of 16.3% ($95\%CI = [-39.2, 50.9]$). Despite Experts being familiar with similar representations, we observed no interaction between expertise and Task. Experts performed 56.7% ($95\%CI = [26.8, 98.8]$) faster than Non-Experts for *Hard* tasks. Both performed similarly on *Easy* and *Medium* tasks. In general, *Completion Time* is more consistent across Non-Expert participants than across Expert participants (Fig. 9a). These results suggest that our representation is suitable for both Experts and Non-Experts if the complexity of the task remains limited. They also confirm our assessment of task difficulty.

Errors and Abandons. Participants performed the tasks with very low error rates, 8.3% ($95\%CI = [-3.6\%, 20.3\%]$) for *VC* tasks and 4.2% ($95\%CI = [-4.5\%, 12.8\%]$) for *CV*. Non-Experts proposed wrong code for *Hard VC* tasks, equally split between *Parameter* and *Shape Errors*. Experts made *Parameter Errors* for some *Medium* tasks. We observed only two withdrawals during a trial, both from non-experts on a *Hard* task, one in *VC* and *CV*, and after more than 500s (Fig. 9b). Overall, such low error rates make it difficult to conclude on the causes of the errors, but suggest that both experts and non-experts users can reliably map *Clint* visual representation to the code and vice versa.

5.2 Interactive Manipulation

After assessing the visualization approach, we focused on interactive program transformation with *Clint*. We conducted a preliminary usability study with users already familiar with the visualization. In order to separate the effect of direct manipulation from individual differences in expertise, participants were not allowed to use any automatic parallelizing compiler that would help experts to achieve better performance. We also decided not to use *Clay* syntax directly as it is little-known and was designed as an intermediate representation for graphical manipulation. Noone attempted to use other diretive-based tools.

5.2.1 Protocol.

Participants. We recruited 8 participants (aged 23-47) by direct email to the participants of the previous study. Since they all were familiar with *Clint*, our expertise criterion does not apply.

Apparatus. The study was conducted with a prototype of *Clint* running on a 15" MacBook Pro. Participants were interacting with the laptop keyboard and a standard Apple mouse.

Procedure. The task consisted in transforming a program part so that the maximum number of loops becomes parallelizable. Participants had to transform the program, but not to include parallelism-specific constructs, e.g., OpenMP pragmas, in order to avoid bias from individual expertise differences. The experiment has a $[3 \times 3]$ within-subject design with two factors:

- Technique used in the trial: *Code* — writing code in an editor of user's choice, no visualization available; *Viz* — direct manipulation, no code visible; *Choice* — full interface, with direct manipulation and source code editing.
- Difficulty of the task: *Easy* — two-dimensional case with at most two transformations; *Medium* — two- or three-dimensional case with rectangular bounds and at most three transformations; *Hard* — two- or three-dimensional case with mutually-dependent bounds and at least two transformations.

Trials were grouped in three blocks by Technique. The *Code* and *Viz* blocks were presented first. Their order was counterbalanced across participants. *Choice* was always presented last in order to assess participants' preference in using code editing or direct manipulation after having used both. In each block, participants were presented with one task of each difficulty level in random order. Tasks were randomly picked into different blocks across participants. They were drawn from real-world program examples and polyhedral benchmarks. Trials were not limited in time and participants were asked to explicitly end the trial by pushing an on-screen button. Prior to the experiment, participants were instructed about source code transformations and the corresponding direct manipulation techniques. They also practiced 4 trials of medium difficulty for each technique before the experiment and were allowed to perform two "recall" practice trials before each Technique block. Each session lasted about 60 minutes. The study was completed by a demographics questionnaire.

Data Collection. For each trial, we measured:

- the overall trial *Completion Time*;
- *First Change Time*, the amount of time from the start to the first change in the program structure (code edited or visualization manipulated);
- *Success Rate*, the ratio between the number of loops made parallel by transformation and the total number of possibly parallel loops.

We recorded both the final state and all intermediary transformations to the program. During the analysis, we performed a log-transform of the *Completion Time* and *First Change Time*.

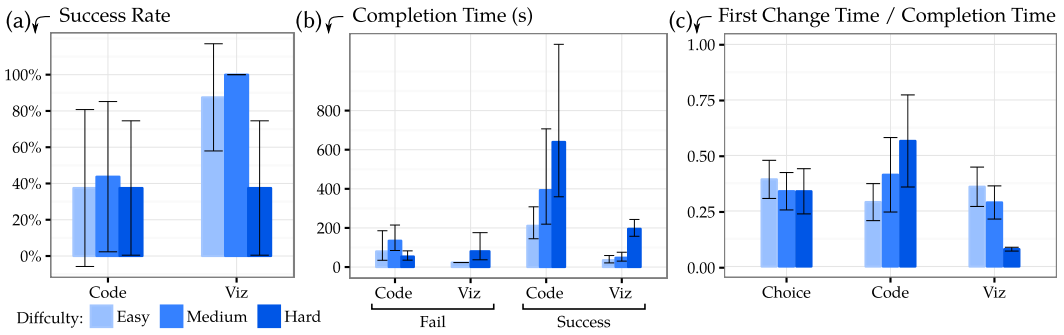


Fig. 10. (a) *Success Rate* is higher with *Viz*, except for *Hard* tasks. (b) *Completion Time* is lower with *Viz*, especially in successful trials. (c) Ratio *First Change Time* / *Completion Time*; the change in trend between *Code* and *Viz* may be due to users adopting an exploratory strategy. Error bars are 95% CIs.

5.2.2 Results and Discussion. Because this experiment was conducted with a small sample, we mostly report results graphically in order to illustrate general trends. We did not observe any ordering effect of Technique or Difficulty on *Completion Time* and *Success Rate*.

Accuracy and Efficiency. Fig. 10a suggests, despite large variability, that participants were in general more successful in transforming the program with direct manipulation than with code editing. Effect sizes reach 40% and 44% for *Easy* and *Medium* tasks. However, for *Hard* tasks, the success rates are identical. This suggests that finding a multi-step transformation is a key difficulty.

Fig 10b suggests that, for successful trials, participants performed the transformation consistently faster in *Viz* condition. The difference in variability between *Code* and *Viz* suggests that direct manipulation compensates for individual expertise differences. Similar *Completion Times* for failed trials can be explained, after analyzing the transformations, by participants "abandoning" the trial if their first attempt did not expose parallelism and submitting a non-parallelizable version.

Strategy and Exploration. Participants at least tried to perform a transformation in 76% cases with *Code* and 94% with *Viz*, suggesting that visualization engages participants by changing the perception of task difficulty. We computed the ratio *First Change Time*/*Completion Time* as a measure of "engagement" (Fig. 10c). It increases with difficulty for *Code*, but drastically decreases for *Viz*, suggesting that participants were more likely to adopt an exploratory trial-and-error strategy supported by the interactive visualization as opposed to code. In *Choice* condition, the ratio remains stable, as participants spent time choosing which representation to use.

Choice between Code Editing and Direct Manipulation. In the *Choice* condition, only 3 participants interacted with the code. They made edits during the first 30s and then switched to the visualization. After the experiment, they explained to have modified the code for the sake of analysis, e.g., to see whether a dependence was triggered by a particular access they temporarily removed.

We observed that most participants were examining the code, but not selecting it. This observation suggests that, although they see the limitations of code representation, participants may need it to relate to the conventional program editing that better corresponds to their expertise.

5.3 Preference for Code or Visualization

Our last experiment investigates the use of textual and visual representations for SCoPs. We relied on *eye tracking* technology in order to precisely measure visual attention between code and

visualization when both were available. We expect that, given sufficient training, users will prefer visualization to code analysis if there is a meaningful task-relevant mapping between the two.

This experiment required a pair of small program analysis tasks such that either code or visualization support each of them better, but never both. Participants had to answer a binary question, with positive or negative formulation to avoid bias. The study was structured as the previous one.

5.3.1 Protocol.

Participants. We recruited 12 participants (aged 21-34, mean=27) through mailing lists. They did not participate in previous studies and had a self-reported experience in programming of 5 to 15 years. All had normal uncorrected vision.

Apparatus. The experimental setup consisted of a 15" MacBook Pro with 2880×1800 screen at 220 ppi connected to the SMI-ETG v1 eye-tracking system[4]. The participant was seated 70 cm away from the screen, which resulted in gaze position accuracy of 27.7px in screen space. The tracking system outputs a 30 FPS video stream from its frontal camera. We placed bright-colored tokens on the screen corners to locate it in the video and compensate for perspective distortion. These tokens were tracked by a custom OpenCV-based script that generated gaze position in screen coordinates through linear interpolation with perspective correction.

We ensured that the sizes of both representations are identical across conditions, with the content centered in each of them. Unused space was filled with neutral gray to avoid distraction. When visible, multiple representations were 60 px away (2× resolution) to identify gaze into one of them.

Procedure. The study is a $[3 \times 3 \times 2]$ within-participants experiment with 4 repetitions per participant and the following factors:

- Representation used in the trial, one of visual representation (*Viz*), source code (*Code*) or both simultaneously (*Choice*);
- Difficulty of the question, one of *Easy*, a loop nest with constant conditions, *Medium*, a loop nest with at least 3 non-constant conditions, or *Hard*, a loop nest with a branch inside and at least 5 non-constant conditions;
- a binary Question asked to the user, either concerns the textual form of loop bounds (*Bounds*) or a statement instance being executed or not inside a loop (*Execution*).

Bounds questions were targeted at *Code*, where the answer is immediately visible, while *Execution* questions were targeted at *Viz*. We refer to these conditions as *matching questions*, and to other conditions as *mismatching questions*. In total, we collected data for $12 \cdot 3 \cdot 3 \cdot 2 \cdot 4 = 864$ trials.

Trials were first blocked by Representation and then by repetition. Representation blocks are ordered identically to the previous study. Each of them comprises 4 repetition blocks, each of which has 6 trials with different Questions and Difficulties in a randomized order. Representation blocks were preceded by a practice session with 4 trials of *Medium* difficulty. After each trial in *Choice* condition, participants were asked about their preferred representation for this question.

Blocks featuring only *Code* or *Viz* were conducted without eye tracking. Participants were wearing the eye-tracking glasses for the third block, after we performed a 3-point calibration with 30px tokens and checked if the glasses did not affect their vision by performing a read-aloud test.

Participants started the trial by clicking the "start" button and ended it by clicking the answer button. They could abandon the trial after at least 15s to avoid immediate abandons for *Hard* tasks with *mismatching* questions. Software provided the correct answer after each trial. One session lasted 50 minutes on average and was complemented by a demographic questionnaire.

[4]http://www.eyetracking-glasses.com/

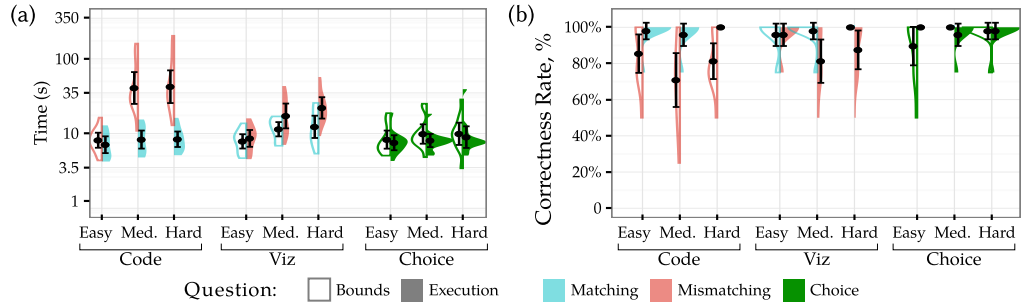


Fig. 11. (a) mismatching questions required up to 4× more time. (b) *Medium* and *Hard* questions with *mismatching* representation result in more incorrect answers. *Completion Times* and *Correctness Ratios* for Choice are close to those for *matching* representation. Dots are means, error bars are 95% CIs, vertical density plots show underlying distributions.

Data Collection and Processing. We collected the following data:

- *Completion Time* of the trial;
- *Correctness* of the answer;
- *Preference* between Representations for the last block;
- *Gaze* from the eye-tracking glasses for the last block.

Given gaze position in screen coordinates, we identified the widget in the focus of attention as one out of three: *Code Widget*, *Viz Widget* or *Question Widget*. Outside any of the widget areas, the gaze was considered *Off Screen*. We randomly sampled 10 frames from each video and verified manually that the script provides exact classification.

Completion Time was log-transformed to compensate the positive skew of its distribution.

5.3.2 *Results and Discussion.*

Ordering effects. We observed a slight decrease in *Completion Time* between first blocks, effect size −13.6% ($95\%CI = [-37.7, 6.1]$), but large variability does not allow to conclude on the presence of a learning effect. *Correctness* did not vary substantially between blocks.

Completion Time. Mismatching questions required substantially more time to complete the trial than *matching* questions, except for *Easy* tasks as shown in Fig. 11a. With *Code*, participants spent 14% ($95\%CI = [-22, 40]$) more time on *Easy Execution* questions, and respectively 132% ($95\%CI = [108, 146]$) and 134% ($95\%CI = [111, 147]$) more time on *Medium* and *Hard Execution* questions than on the *Bounds* questions of the same difficulty. Similarly, with *Viz* representation, they answered *Execution* questions 9% ($95\%CI = [-20, 49]$), 40% ($95\%CI = [1, 102]$) , and 57% ($95\%CI = [10, 129]$) faster for increasing Difficulties. This result supports the definition of *mismatching* question suggesting that a representation not adapted for the question slows participants down. The smaller increase of *Completion Time* with *Viz* compared to *Code* suggests that *Viz* representations allows to reason about mismatching questions easier than *Code*.

Choice condition shows *Completion Times* close to those for *matching* representation. For *Bounds* questions, it took on average 6% ($95\%CI = [-27, 56]$), −3% ($95\%CI = [-40, 56]$), and 7% ($95\%CI = [-33, 70]$) more time compared to *Code* for increasing Difficulties. For *Execution* questions, it took 5% ($95\%CI = [-27, 53]$), −14% ($95\%CI = [-54, 54]$) and −21% ($95\%CI = [-63, 58]$) more time than *Viz* for increasing Difficulties. These results suggest that, given two representations, participants are likely to chose the *matching* one. Although they do not spend more time on average, the variability

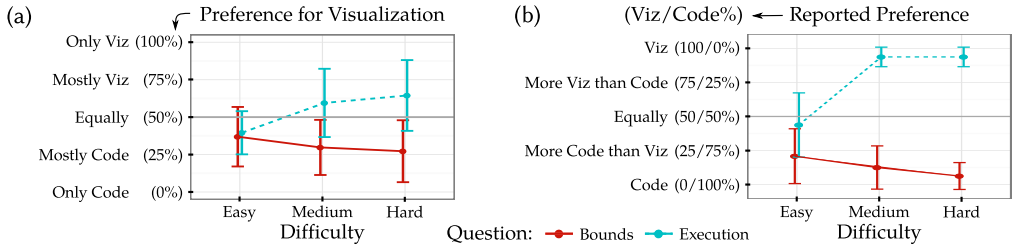


Fig. 12. (a) *matching* representations are more used for *Medium* and *Hard* tasks, but *Code* for *Easy* tasks. (b) reported preference demonstrates similar trend. Dots are means, error bars are 95% CIs.

is larger for *Choice* condition. It suggests that participants only effectively use one representation, but consider both. We illustrate this later with eye tracking data.

Correctness. The participants succeeded to answer the majority of the questions with 93% ($95\%CI = [90, 95]$) of correct results on average as shown in Fig. 11b. Abandoned trials were considered as incorrect answers. Overall trends are similar to *Completion Time*.

Given *Choice*, participants had a high success rate overall, except *Easy Execution* questions with mean *Success Rate* 89.5% ($95\%CI = [79\%, 100\%]$). This may be explained by choosing the *mismatching Code* representation due to visible task simplicity. Due to low error rates, we did not perform any further analyses. Only 4 trials were abandoned, all featuring *mismatching* questions, 3 of which with *Code*. Abandons took place after 91s on average whereas the mean trial duration is 13.7s.

Representation Choice. Our analyses are built on the following metrics, defined prior to the study.

Visual Preference, VP — total duration of gaze on the *Viz Widget* divided by the total duration of gaze on *Viz* or *Code Widget*. Values close to 1 indicate participant looking more at the visualization.

Representation Uncertainty, RU — the measure of attention distribution computed as $RU = 2 \cdot \text{abs}(VP - 0.5)$. High values mean attention was distributed evenly between representations, low values – that only one representation was used.

We expect *Completion Time* to increase with *Representation Uncertainty* as the participant uses two representations where one would suffice. At the same time, it may increase even more for lower values of *Representation Uncertainty* and high *Visual Preference* for the unadapted representation.

Fig. 12a shows the *Visual Preference* for different conditions, the center line corresponding to the equal distribution of visual attention. For *Medium* and *Hard* tasks, participants spent more time on *matching* representations, Question effect sizes reach 66.6% ($95\%CI = [4.8, 128.5]$) and 81.2% ($95\%CI = [16.7, 145.7]$), respectively. For *Easy* tasks they relied on the *Code* independent of Question.

The reported *Preference*, depicted on Fig. 12b, shows the same tendency. The preference for *Code* drops from 56% in *Easy Execution* tasks to 6% in *Medium* and *Hard Execution* tasks. Since we asked which representation they found "most useful", the difference between reported *Preference* and *Visual Preference* suggests that participants tend to look at both representations even though they do not find one of them useful. Nevertheless, we observed a positive correlation between reported *Preference* and *Visual Preference*, $r = 0.41$ ($95\%CI = [0.20, 0.57]$), suggesting that participants tend to use more the representation they find useful.

Overall, we observed a correlation between *Representation Uncertainty* rate and *Completion Time*, $r = 0.41$ ($95\%CI = [0.19, 0.58]$) as well as a negative correlation between *Representation Uncertainty* rate and *Correctness*, $r = -0.27$ ($95\%CI = [-0.47, -0.04]$): the more participants' attention was distributed between representations, the less correct answers they gave. Although the correlation

does not imply causality, the connection between the simultaneous use of different representations and the total trial duration suggests that one *matching* representation should be preferred to two.

6 RELATED WORK

Interactive Program Parallelization. Program editors supporting interactive program parallelization date back to wide adoption of parallelism for scientific programming. We review those specifically targeting loop-level optimizations. The *ParaScope* editor [33] provided dependence analysis and interactive loop transformation for High-Performance Fortran (HPF). It reported the dependence analysis results and allowed the user to perform various loop transformations, including parallelization. The *D Editor* interacted with an distributed HPF compiler to report optimization choices regarding data distribution and parallelization [29]. *SUIF Explorer* took a different approach, collecting dynamic execution and dependence data to suggest loops (or parts thereof thanks to program slicing [51]) for parallelization [35]. Similarly, *DECO* records traces of the memory accesses along with cache hit information and uses pattern recognition algorithms to suggest memory optimizations [47]. *NaraView* provides a navigable 3D visualization of loop-level access patterns [44]. Contrary to these tools, *Clint* uses the polyhedral model with its instance-wise dependence analysis and static guarantees of loop transformation legality. It also allows for transforming the program using its visualization. *Chlore*-based transformation replay is not tied to particular compiler transformations.

Semi-Automatic Polyhedral Transformations. User-assisting tools based on the polyhedral model emerged as a means to express "classical" loop transformations [52] in the model, the Unified Transformation Framework (*UTF*) stemming from the first approach [31]. *URUK* was proposed to improve loop transformation composability and enable automated traversal of a transformation search space [22], delaying the legality analysis until code generation. *Loop Transformation Recipes* combine loop transformations, mapping to accelerators and code generation directives from *CHiLL* [12] with the *POET* [54] language for auto-tuning specification. *AlphaZ* focuses on equational programming and enables complex memory mapping and management [56]. *Clay* is arguably the first complete set of directives for polyhedral program transformations [2]. *Clint* uses visualization and direct manipulation to address the challenges of directive-based approaches, such as identifying a promising transformation, targeting it at a program entity or evaluating its effects.

Visualizations for the Polyhedral Model. The literature on the polyhedral model heavily relies on scatterplot-like visualizations of iteration domains. Polyhedral libraries include components for visualization, including *VisualPolylib* [36] for *Polylib* and `islplot` [24] for `isl` [49]. *LooPo* was arguably the first tool to visualize the polyhedral dependence analysis information during program transformation [23]. *Tulipse* integrates polyhedral visualization into Eclipse IDE [53]. *Clint* goes beyond static visualization by enabling direct manipulation to transform the program.

3D *iteration space visualizer* lets the user interactively request loop parallelization through a visual representation [55]. Polyhedral Playground [27] augments a web-based polyhedral calculator with domain and dependence visualizations. *PUMA-V* provides a set of visualizations that expose internal operation of the *R-Stream* compiler [39, 40]. It allows the user to control the optimization-related compiler options from the visualization. *Clint* builds on *Clay* as intermediate abstraction and does not require the user to control or even understand the operation of a compiler.

7 CONCLUSION

Clint addresses the issues of directive-based approaches in the polyhedral model: target identification is made direct without exposing polyhedral-specific concepts; transformation legality and

effects are visible immediately during manipulation; reading polyhedrally-transformed code is no longer necessary. It makes loop optimization accessible, interactive and independent of a particular algorithm. Our approach enables human-machine partnership where an automatic framework performs heuristic-driven transformation and provides feedback on demand while a user brings in domain knowledge to tweak the transformation without modifying the heuristics. Such domain knowledge may be unavailable to framework designers and differ between use cases.

Experiments suggest that visualizations lower the expertise necessary to perform aggressive program restructuring and decrease the time necessary for program analysis. Semi-automatic transformation decreases the time of program transformation. In our studies, visual semi-automatic approach to program transformation doubled the success rate and decreased the required time by a factor of 5 for some program structures. We also contribute to the discussion on visualization acceptance, suggesting its perceived utility increases with the relative complexity of the task.

Limitations. As *Clint* was designed using a set of polyhedral test cases with small number of statements nested in shallow loops, it may be subject to cluttering for larger program parts. Long blocks of interdependent statements may result in a profusion of dependence arrows. Visual replay may become distracting when multiple projections are rendered for deep loops. However, program parts amenable to the polyhedral model are typically small yet require aggressive transformation.

Future Work. Drawing from the eye-tracking study conclusions and existing limitations, the visual approach seems promising yet restricted for difficult cases. We plan to address those by interleaving visual representations and code fragments and by proposing a zoomable interface with different levels of detail. At the same time, the visualization may be beneficial for learning, which can be supported with a smooth transition between code and visual representation.

Visual cluttering can be addressed by only displaying salient parts. They can be identified directly by the users, or inferred from their behavior. On the other hand, a polyhedral compiler may provide additional feedback on, e.g., dependences that prevent parallel execution. Finally, *Clint* visualizations may be used conjointly with performance models and runtime evaluators, and integrated into a larger development environment in order to account for program parallelization all along the development process.

REFERENCES

[1] Corinne Ancourt and François Irigoin. 1991. Scanning Polyhedra with DO Loops. *ACM Sigplan Notices* 26, 7 (1991), 39–50.

[2] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016. Opening Polyhedral Compiler's Black Box. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO 2016)*. ACM, New York, NY, USA, 128–138.

[3] Cedric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. IEEE Computer Society, Washington, DC, USA, 7–16.

[4] Cédric Bastoul. 2016. Mapping Deviation: A Technique to Adapt or to Guard Loop Transformation Intuitions for Legality. In *Proc. of the 25th International Conference on Compiler Construction*. ACM, New York, NY, USA, 229–239.

[5] Michel Beaudouin-Lafon. 2004. Designing Interaction, Not Interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '04)*. ACM, New York, NY, USA, 15–22.

[6] Michel Beaudouin-Lafon and Wendy E. Mackay. 2000. Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '00)*. ACM, New York, NY, USA, 102–109.

[7] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The Polyhedral Model Is More Widely Applicable Than You Think. In *Compiler Construction*, Rajiv Gupta (Ed.). Number 6011 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 283–303.

[8] Uday Bondhugula, Aravind Acharya, and Albert Cohen. 2016. The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests. *ACM Transactions on Programming Languages and*

Systems 38, 3 (April 2016), 12:1–12:32.

[9] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Compiler Construction*. Springer, Budapest, Hungary, 132–146.

[10] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. 2010. A Model for Fusion and Code Motion in an Automatic Parallelizing Compiler. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. ACM, New York, NY, USA, 343–352.

[11] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *ACM SIGPLAN Notices* 43, 6 (2008), 101–113.

[12] Chun Chen. 2008. *CHiLL: A Framework for Composing High-Level Loop Transformation*. Technical Report 08-897. University of Southern California.

[13] Chun Chen. 2012. Polyhedra Scanning Revisited. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 499–508.

[14] Andy Cockburn and Bruce McKenzie. 2001. 3D or Not 3D?: Evaluating the Effect of the Third Dimension in a Document Management System. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '01)*. ACM, New York, NY, USA, 434–441.

[15] Geoff Cumming. 2013. The New Statistics Why and How. *Psychological Science* 25 (Nov. 2013), 7–29. Issue 1.

[16] Geoff Cumming and Sue Finch. 2005. Inference by Eye: Confidence Intervals and How to Read Pictures of Data. *American Psychologist* 60, 2 (2005), 170–180.

[17] Pierre Dragicevic. 2016. Fair Statistical Communication in HCI. In *Modern Statistical Methods for HCI*, Judy Robertson and Maurits Kaptein (Eds.). Springer International Publishing, 291–330.

[18] N. Elmqvist, P. Dragicevic, and J. D. Fekete. 2008. Rolling the Dice: Multidimensional Visual Exploration Using Scatterplot Matrix Navigation. *IEEE Transactions on Visualization* 14, 6 (Nov. 2008), 1539–1148.

[19] Paul Feautrier. 1991. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming* 20, 1 (1991), 23–53.

[20] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time. *International Journal of Parallel Programming* 21, 6 (1992), 389–420.

[21] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer US, 1581–1592.

[22] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *International Journal of Parallel Programming* 34, 3 (July 2006), 261–317.

[23] Martin Griebl and Christian Lengauer. 1997. The Loop Parallelizer LooPo-Announcement. In *Proc. of the 9th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC '96)*. Springer-Verlag, London, UK, UK, 603–604.

[24] Tobias Grosser. 2016. islplot: Library to Plot Sets and Maps. (2016). http://tobig.github.io/islplot

[25] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters* 22, 04 (Dec. 2012), 1250010.

[26] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Transactions on Programming Languages and Systems* 37, 4 (July 2015), 12:1–12:50.

[27] Tobias Grosser and Oleksandr Zinenko. 2017. PollyLabs Polyhedral Playground. (2017). http://playground.pollylabs.org/

[28] Jeffrey Heer, Michael Bostock, and Vadim Ogievetsky. 2010. A Tour Through the Visualization Zoo. *ACM Queue* 53, 6 (June 2010), 59–67.

[29] Seema Hiranandani, Ken Kennedy, Chau Wen Tseng, and Scott Warren. 1994. The D Editor: A New Interactive Parallel Programming Tool. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing (Supercomputing '94)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 733–742.

[30] F. Irigoin and R. Triolet. 1988. Supernode Partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. ACM, New York, NY, USA, 319–329.

[31] W. Kelly and W. Pugh. 1995. A Unifying Framework for Iteration Reordering Transformations. In *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*, Vol. 1. 153–162 vol.1.

[32] Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[33] K. Kennedy, K. S. McKinley, and C. W. Tseng. 1991. Interactive Parallel Programming Using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems* 2, 3 (July 1991), 329–341.

[34] Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: Shareable Dynamic Media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. ACM, New York, NY, USA, 280–290.

[35] Shih-Wei Liao, Amer Diwan, Robert P. Bosch, Jr., Anwar Ghuloum, and Monica S. Lam. 1999. SUIF Explorer: An Interactive and Interprocedural Parallelizer. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '99)*. ACM, New York, NY, USA, 37–48.

[36] Vincent Loechner. 1999. PolyLib: A library for manipulating parameterized polyhedra. (1999). http://icps.u-strasbg.fr/polylib/#visualpolylib

[37] Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. 2011. R-Stream Compiler. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer US, 1756–1765.

[38] Donald A. Norman. 2010. *Living with Complexity*. MIT Press.

[39] E. Papenhausen, K. Mueller, H. Langston, B. Meister, and R. Lethin. 2016. PUMA-V: An Interactive Visual Tool for Code Optimization and Parallelization Based on the Polyhedral Model. In *New York Scientific Data Summit*. 1–4.

[40] E. Papenhausen, B. Wang, M. H. Langston, M. Baskaran, T. Henretty, T. Izubuchi, A. Johnson, C. Jung, M. Lin, B. Meister, K. Mueller, and R. Lethin. 2015. Polyhedral User Mapping and Assistant Visualizer Tool for the R-Stream Auto-Parallelizing Compiler. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. 180–184.

[41] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. 2006. GRAPHITE: Polyhedral Analyses and Optimizations for GCC. In *Proc. of the GCC Developers Summit*. 179–197.

[42] Louis-Noël Pouchet. 2016. PolyBench/C 4.2. Polyhedral Benchmark Suite. (2016). http://web.cse.ohio-state.edu/~pouchet.2/software/polybench

[43] William Pugh. 1991. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Proc. of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 4–13.

[44] Mariko Sasakura, Kazuki Joe, Yoshitoshi Kunieda, and Keijiro Araki. 1999. NaraView: An Interactive 3D Visualization System for Parallelization of Programs. 27, 2 (April 1999), 111–129.

[45] J. Shirako, L. N. Pouchet, and V. Sarkar. 2014. Oil and Water Can Mix: An Integration of Polyhedral and AST-Based Transformations. In *Intl. Conference for High Performance Computing, Networking, Storage and Analysis*. 287–298.

[46] Ben Shneiderman. 1981. Direct Manipulation: A Step Beyond Programming Languages. In *Proceedings of the Joint Conference on Easier and More Productive Use of Computer Systems. (Part - II): Human Interface and the User Interface - Volume 1981 (CHI '81)*. ACM, New York, NY, USA, 143–.

[47] Jie Tao, Thomas Dressler, and Wolfgang Karl. 2007. An Interactive Graphical Environment for Code Optimization. In *Computational Science–ICCS 2007*. Springer, 831–838.

[48] Nicolas Vasilache, Cedric Bastoul, Albert Cohen, and Sylvain Girbal. 2006. Violated Dependence Analysis. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS '06)*. ACM, New York, NY, USA, 335–344.

[49] Sven Verdoolaege. 2010. Isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software – ICMS 2010*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Number 6327 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 299–302.

[50] Colin Ware. 2012. *Information Visualization: Perception for Design* (3 ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[51] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 439–449.

[52] Michael Joseph Wolfe. 1995. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[53] Yi Wen Wong, Tomasz Dubrownik, Wai Teng Tang, Wen Jun Tan, Rubing Duan, Rick Siow Mong Goh, Shyh-hao Kuo, Stephen John Turner, and Weng-Fai Wong. 2012. Tulipse: A Visualization Framework for User-Guided Parallelization. In *Euro-Par 2012 Parallel Processing*. Springer, 4–15.

[54] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. 2007. POET: Parameterized Optimizations for Empirical Tuning. In *2007 IEEE International Parallel and Distributed Processing Symposium*. 1–8.

[55] Yijun Yu and Erik H. D'Hollander. 2001. Loop Parallelization Using the 3D Iteration Space Visualizer. *Journal of Visual Languages and Computing* 12, 2 (April 2001), 163–181.

[56] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2012. Alphaz: A System for Design Space Exploration in the Polyhedral Model. In *Languages and Compilers for Parallel Computing*. Springer, 17–31.

[57] Oleksandr Zinenko, Cédric Bastoul, and Stéphane Huot. 2015. Manipulating Visualization, Not Codes. In *IMPACT 2015, Fifth International Workshop on Polyhedral Compilation Techniques, In Conjunction with HiPEAC 2015*. 8.

[58] Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2014. Clint: A Direct Manipulation Tool for Parallelizing Compute-Intensive Program Parts. In *Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 109–112.

Received May 2017; revised December 2017; accepted December 2017