Manipulating Visualization, Not Codes

Oleksandr Zinenko
Inria and Univ. Paris-Sud
Orsay, France
oleksandr.zinenko@inria.fr

Cédric Bastoul
Univ. of Strasbourg and Inria
Strasbourg, France
cedric.bastoul@unistra.fr

Stéphane Huot
Inria
Lille, France
stephane.huot@inria.fr

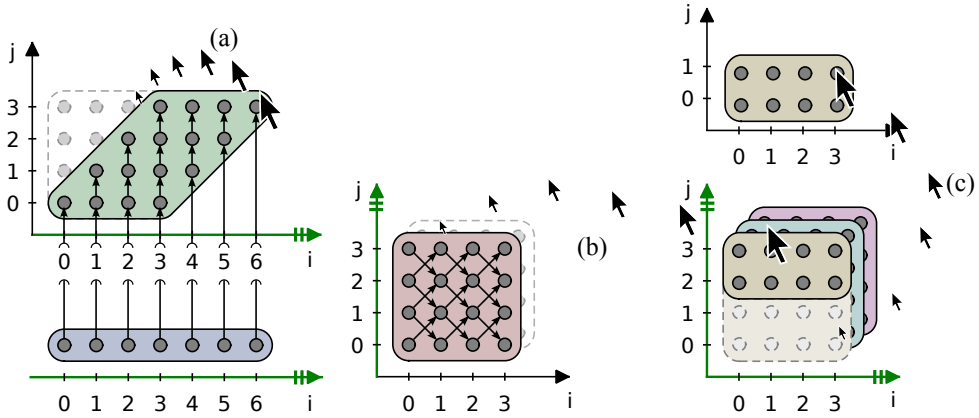


Figure 1: Directly Manipulating Visualized Polyhedra: (a) Skewing; (b) Reordering and Fusion; (c) Index-Set Splitting.

ABSTRACT

Manual program parallelization and optimization may be necessary to reach a decent portion of the target architecture's peak performance when automatic tools fail at choosing the best strategy. While a broad range of languages and libraries provide convenient ways to express parallelism, the difficult, time consuming and error-prone parallelism identification and extraction task is mostly left under the programmer's responsibility. To address this issue, we introduce a visualization-based approach to ease parallelism extraction and expression that leverages polyhedral compilation technologies. Our interactive tool, *Clint*, maps direct manipulation of the visual representation to polyhedral program transformations with real-time semantics preservation feedback. We conducted two user studies showing that Clint's visualization can be accurately understood by both experts and non-expert programmers, and that the parallelism can be extracted better from Clint's representation than from the source code in many cases.

1. INTRODUCTION

The massive adoption of modern and heterogeneous parallel architectures requires adequate solutions to support the creation and debugging of programs that efficiently exploit the full power of available parallel resources. Tremendous effort was made towards simplification of parallel programming by creating dedicated programming models, libraries or languages that express parallelism with high-level constructs. However, identifying parallelism remains a challenging task, especially when a deep data dependence analysis is required before parallelizing a sequential program.

The polyhedral model [8] has proven to be useful for loop-level parallelization or vectorization. Its unique instance-wise dependence analysis and transformation expressiveness makes it possible to apply aggresive program restructuring while preserving the original semantics. However, automatic polyhedral compiler techniques for loop-level parallelism extraction [5, 18] operate as heuristics-driven black-boxes that provide limited help and feedback to programmers when the computed transformation does not suit their needs. Semi-automatic tools based on directive scripts [6, 13, 9] offer more flexibility and control for program transformation, but they also require significant expertise from the end user. In this paper, we report on an interdisciplinary research project (Human-Computer Interaction and Optimizing Compilation) that aims to design and evaluate a new way to interact with a polyhedral framework through direct manipulation of visualizations.

Due to the geometric nature of its algebraic structures, the polyhedral model has a direct visual representation that is extensively used to describe program transformations in the literature. We have thus designed an interactive version of this visualization that allows to perform loop transformations, such as shifting, fusion or index-set splitting, through direct manipulation. This visualization is integrated into the interactive tool *Clint*[1] (Fig. 1) and features projections of the multidimensional loop nests containing individual iterations and dependences between them. *Clint* maps graphical manipulation of these objects to the corresponding polyhedral transformations and provides precise and direct feedback on semantics preservation during the manipulation.

IMPACT 2015
Fifth International Workshop on Polyhedral Compilation Techniques
Jan 19, 2015, Amsterdam, The Netherlands
In conjunction with HiPEAC 2015.

http://impact.gforge.inria.fr/impact2015

[1]Note: *Clint* will be demonstrated at the workshop. All visualizations in this paper are generated with *Clint*. A version of the tool with limited polyhedral backend was presented at the VL/HCC symposium in 2014 [26].

Visualization is updated automatically whenever the source code is modified, while the transformed code may be generated on demand. Overall, our goal with *Clint* is not designing just an "interface" but consistent "interaction" between the programmer and the program restructuring engine [3].

In the following section, we present our visualization and the related interaction techniques. Section 3 details the support provided by the polyhedral framework to enable interactive manipulation. We report on the results of empirical evaluation of *Clint* in Section 4. Related work is discussed in Section 5, and conclusions are drawn in Section 6.

2. INTERACTIVE VISUAL FRONTEND

2.1 Design Rationale

Clint leverages the geometric nature of the polyhedral model by presenting statement instances and dependences in a scatterplot-like visualization. This approach is similar to the one commonly used in the polyhedral compilation community to illustrate iteration domains. However, it goes beyond these common static visualizations by allowing the direct manipulation [11] of the graphical objects in order to restructure the program. Each action performed by the user is mapped to a sequence of program transformations that, if applied, would change the original program structure so that its new visualization would corresponds to the one obtained after the direct manipulation. Furthermore, the set of possible interactive manipulations is based on the geometry-related vocabulary of classical loop transformations, such as skewing or shifting, providing the user with an intuition on the effect of the transformation.

The design of *Clint* is motivated by the needs for (1) a single and consistent interface for polyhedral program transformation and dependency analysis; (2) easier exploration of alternative loop transformations; and (3) reduced manual code and directive script editing. It relies on the polyhedral framework, but is not bound to any particular directive set or programming language as long as they may be expressed in the polyhedral model. It seamlessly combines loop transformations to allow for reasoning about execution order and dependences rather than loop structure and branch conditions. Finally, the interactive visual approach reduces parallelism extraction to visual pattern recognition [21] and code transformation to geometrical manipulations, giving non-expert programmers a way to manage the complexity of the underlying model [17].

2.2 Structure of the Visualization

One Statement Occurrence – The main structure of our visualization is a *polygon* that contains *points* on the integer lattice. Each point corresponds to an execution of a particular statement in the iteration of a loop nest, which is a statement instance in the polyhedral model. These points are linked by arrows to denote dependences between iterations. In Fig. 2(left), this polygon is displayed in the *coordinate system* where axes correspond to loop iteration variables. The polygon shape delimits loop iteration bounds.

Multiple Statement Occurrences – A transformation may result in a case where executions of a statement are distributed to multiple different loops. We then assume that this statement has multiple *occurrences*. *Clint* uses color coding scheme to match occurrences of the same statement both in the visualization and in the source code (see Fig. 3).

Multiple Coordinate Systems – Each coordinate system is at most two-dimensional and represents two nested loops. Statement occurrences that are enclosed in both loops are displayed in the same coordinate system but with optional slight displacement to discern them (see Fig. 3). Statement occurrences enclosed only in the outer loop share one axis of the coordinate system, forming a *pile* (see Fig. 1(left)). Finally, statement occurrences not sharing loops are displayed as a sequence of piles (see Fig. 1(right)). This structure represents the lexical ordering of the statement and loops in the source code and conveys their overall execution order.

Multiple Projections – The overall visualization is a set of two-dimensional projections, where loops that are not matched to the axes are ignored, and the program blocks containing statements are arranged according to their lexicographic order. For a single statement occurrence, they may be ordered in a scatterplot matrix as in Fig. 4. The points are displayed with different intensity of shade depending on how many multidimensional instances were projected on this point. We motivate this choice of two-dimensional projections by easier direct manipulation with a standard 2D input device (e.g. mouse) [3] as well as maintaining the consistency of the visualization for any dimensionality.

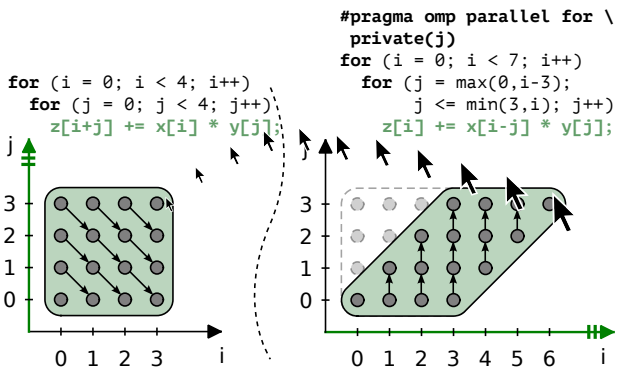


Figure 2: Performing a skew transformation to parallelize polynomial multiplication loop by deforming the polygon. The code is automatically transformed from its original form (left) to the skewed one (right).

2.3 Direct Manipulation to Restructure Loops

This visualization affords direct manipulation of its components. Depending on the strategy of restructuring to enable parallelism, points or groups of points can be dragged outside of their container polygon thus creating a new one (see Fig. 1(c)) in order to isolate irregular dependences or iteration groups that require strict execution order. Polygons can also be dragged within (Fig. 3) or between coordinate systems (Fig. 1(b,c)) to adjust the execution order between statements in the loop nest or move them to another loop nest. They can be reshaped so that the loop iterations are executed in a different order: for example, skewing prevents uniform dependences from spanning between iterations of the outer loop (see Fig. 2).

Coordinate systems within a pile or entire piles can be reordered by a dragging operation, as if they represented a list. This enables generalized reordering of statements and loops in the program and allows to analyze the overall data flow in order to find coarser-grain parallelism.

These manipulations can also be composed: for example, a group of points may be detached, dragged to another co-

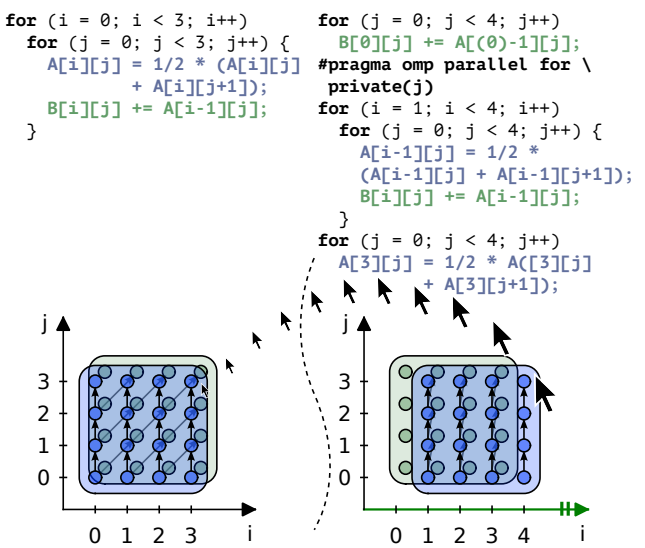


Figure 3: Manipulation for *shift* Transformation: the darker polygon is dragged right so that dependence arrows become vertical without spanning between different iterations on `i`. On the right, the visualization is decoupled from the code structure, and both statements can still be manipulated as if they were not split between two loops.

ordinate system and placed in a particular position during a single manipulation. After each action is performed, "legality" and "parallelism" feedback is provided: dependence arrows turn red and become thicker if the corresponding dependence is violated, and the axis becomes thicker and green if the corresponding loop may be executed in parallel (Fig. 3, right). This allows the user to resolve dependences by following visual intuition, e.g. by aligning all dependency arrows in parallel to each other, and thus to reveal parallelism without editing the source code or compiler directives.

In the case of a visualization with multiple projections, the selection of statement instance points has to be performed one by one in each projection, or by using a rubber band rectangle technique. The overall multidimensional selection is thus the intersection of constraints imposed by each separate two-dimensional selection. If there are no selected points in a projection, it is discarded, as it would result in an overall empty selection. *Clint* also handles parametric control flow conditions by providing identical visualizations and manipulation techniques for parameter values. It infers the parameters used in the selection and transformations and prefers parametric transformations in case of ambiguity.

In *Clint* we keep the visualization consistent with the original program structure unless the user explicitly applies the transformations to the code. This allows the manipulation of multiple disparate statement occurrences as a whole, for example in case of the partial loop fusion shown in Fig. 3.

Direct manipulation interfaces feature three promising capabilities for semi-automatic code restructuring. First, *selection of transformation target* – a particular iteration, group of iterations, statement or loop – is done directly on the graphical object, while in the code or polyhedral representation it may require additional identification methods relying on, e.g., lexicographic ordering of statements and inequations for iteration grouping. Second, *transformation composition* is as easy as sequencing graphical actions on the persistent visualization components with immediate feedback, even in cases where the underlying program structure evolves quickly. Finally, *transformation refinement* is possible thanks to editable transformation history view.

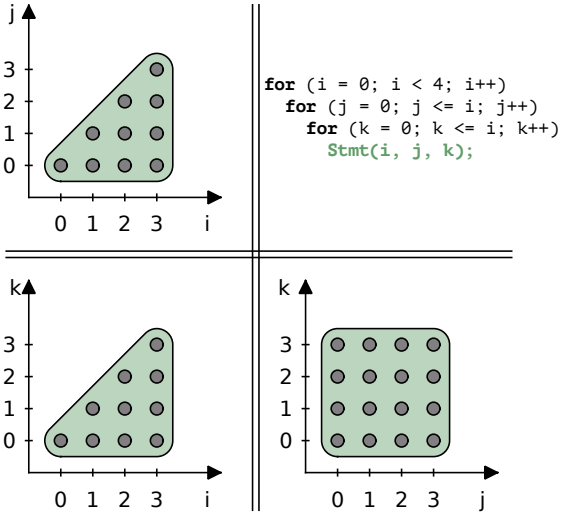


Figure 4: Multidimensional iteration domains are shown as two-dimensional projections in a scatter-plot matrix.

2.4 Clint Interface

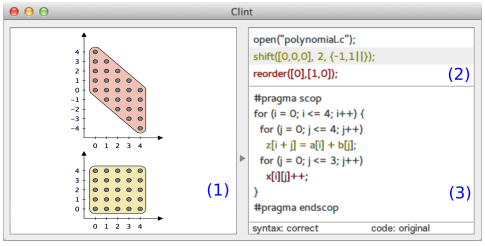


Figure 5: Clint interface includes: (1) interactive visualization, (2) editable history view, and (3) source code editor.

Clint combines three editable and synchronized representations: (1) the interactive visualization described above; (2) a navigable and editable transformation history view; and (3) the source code editor as shown in Fig. 5. A consistent color scheme is used between the views to match code statements to the visualization and to the history entries that affected these statements. User's manipulations are immediately appended to the history view using a special syntax to express high-level loop transformations [22]. The user can then navigate through the history by selecting an entry, which will update the visualization to the corresponding previous state. Entries may be edited as long as the syntax is respected (the system provides real-time feedback on syntax correctness and transformation legality). As the target code tends to become complex and unreadable after several manipulations, the user has the choice to update it or not in order to reflect the state of the visualization. Finally, when the code is edited, the visualization is updated, thus making *Clint* a dynamic visualizer for polyhedral code.

3. POLYHEDRAL BACKEND

The support for visualizing transformed instance sets and dependencies, for checking the legality of the complete transformation sequence, for marking axes as parallel and for ultimately generating the code that implements the transfor-

mation sequence is provided by a specific polyhedral framework. The overall architecture of Clint's framework is depicted in Fig. 6. Clint relies on well-known polyhedral compiler building blocks to achieve specific parts of the work. *Clan* [2] raises a C program to its polyhedral representation counterpart, *Candl* [2] achieves the data dependence analysis and the parallelism detection, and *CLooG* [1] generates a C+OpenMP code that implements a given transformation. A key aspect of these tools with respect to Clint's purpose is that they support the "*unions of relations*" polyhedral representation as recalled in Section 3.1. Clint also relies on two dedicated building blocks: *Clay* [2], which provides a high-level transformation formalism based on the unions of relations representation (Section 3.2); and a specific support to build the visualization and to translate user's actions to *Clay*'s formalism (Section 3.3).

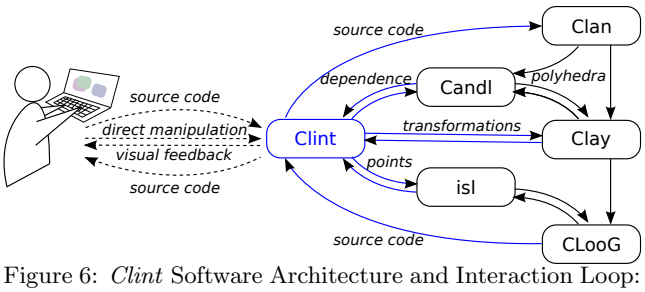


Figure 6: *Clint* Software Architecture and Interaction Loop: The user interacts only with *Clint* by entering the code and manipulating the visual representation. The system returns immediate feedback in the same interface and generates the transformed code on-demand.

3.1 Union of Relations Representation

Our work is based on a state-of-the art *union of relations* abstraction [12]. A union of relations is a piece-wise mapping from *input* dimensions to *output* dimensions according to affine constraints. We use this abstraction to represent all relevant components of an input program [2], and we consider only programs that can be abstracted in such a way. The relevant components include, for each statement, the statement's *iteration domain*, its *scheduling* and the list of its *memory accesses*.

The iteration domain of a statement captures the control structures surrounding it and abstracts all dynamic executions, or *instances*, of that statement. It is represented with a degenerate relation without input dimensions and where output dimensions correspond to the statement's iteration space. Disjunctions in control conditions result in the iteration domain being represented as a union of relations.

The scheduling of a statement expresses the ordering of its instances with respect to each other and with respect to instances of other statements. It is represented with a relation where input dimensions correspond to the original iteration space and where output dimensions correspond to the target multidimensional execution time. Each component of a union of scheduling relations may include constraints to limit the applicability of that scheduling component to specific parts of the original iteration space. A given iteration may have multiple mappings from multiple scheduling components (as a result, it will be duplicated in the final code).

A memory access relation abstracts accesses to a given variable or to indexed elements of a given array in a given statement. It is represented as a relation where input dimensions correspond to the original iteration space and output dimensions are the array dimensions. Approximations of memory accesses, e.g., when array indices are not affine expressions, may be represented as a union of relations.

3.2 High-Level Transformations

Supporting the direct manipulation of the polyhedral representation of a program requires a transformation mechanism with specific properties. First, it must be possible to precisely and independently select and transform any subset of a given iteration domain, or a complete iteration domain, or a group of iteration domains (*selection challenge*). Next, it must be possible to check at any moment the legality of the current state of the transformation and to allow illegal intermediate states while the user is designing the optimization (*composition challenge*). Finally, the user should be able to replay and to refine its optimization (*refinement challenge*).

Our solution to meet these requirements is a new high-level transformation formalism. Each user action is translated to a high-level directive which in turn modifies the scheduling relations. This new formalism, named after its implementation *Clay*, generalizes previous approaches that build high-level transformation directives on top of a polyhedral engine such as *UTF* [13], *URUK* [9] or *CHiLL* [6] by being based on the more general union of relations abstraction discussed in Section 3.1 and by strongly taking advantage of it. *Clay* and its dozen of directives corresponding to extended versions of classical loop transformations (reordering, shifting, interchange, fusion, splitting, index set splitting, strip-mining, grain, reversal, skewing, tiling etc.), are detailed by Bastoul in [2].

Clay addresses the selection challenge thanks to a specific structure of the scheduling relation, an extension to unions of relations of the so-called "2d+1" scheduling relation structure where odd dimensions are constant and represent the lexical order of statements at a given loop depth. The vector of such constants, named β-vector for consistency with URUK's formalism, is unique for each union of relations component and is guaranteed to remain unique by the formalism. Combined with the ability to apply index-set splitting as an additional scheduling relation constraint if a subset of a given iteration domain is involved, any selection corresponds to a set of β-vectors or β-vector prefixes and the desired transformation can be applied only to each scheduling component that has one of them.

The composition challenge is met because the formalism only modifies the scheduling relations, even for transformations that would require iteration domain alterations or duplications in previous approaches, such as *tiling* or *index-set splitting*. As a result, it is not necessary to apply intermediate dependence graph updates, as in URUK, or to enforce each step is legal, as in CHiLL, to ensure the legality of the complete transformation sequence. Because all iteration domains are immutable in *Clay*'s formalism, only the final scheduling has to be checked for dependence violations.

Finally, *Clay* meets the refinement challenge because the sequence of user actions translates to a list of directives: it is thus possible to undo, save, replay or refine. The user may keep the original code along with the transformation script made with *Clint* to achieve a clear decoupling between the program and its optimization.

3.3 Visualization Support

Scheduled Iteration Domains and Dependences – To convey information about the original or transformed execution order of statement instances, we use *scheduled iteration domain* visualizations. They are built separately for each scheduling relation component of each statement. First, we set parameters to (user-)defined constant values in order to generate a finite visualization. Next, we remove special β-vector dimensions (see Section 3.2). Then we apply the Generalized Change of Basis to the iteration domain with respect to the scheduling relation component to get a scheduled iteration domain part [14, 1]. Finally, we rely on *isl* [20] to enumerate points in this domain to be displayed. In the same way, we use *Candl* [2] to compute instance-wise data dependence sets restricted to displayed statement instances only and to expose them. We also perform a violated dependence analysis generalized to union of relations in order to highlight violated dependences [19, 2].

Coordinate Systems and Piles – Special β-vector dimensions are not considered for scheduled iteration domain visualizations but are used to organize them into polygon stacks and coordinate systems piles instead. Two scheduled iteration domain visualizations share their first n coordinates if the first n components of their β-vectors are the same. For a projection on the iteration space dimensions n and m where $n < m$, the visualizations are stacked in one coordinate system if they share m β-vector components, while coordinate systems are arranged in piles for visualizations sharing only n β-vector components.

Instance-Wise Selection – To operate on an arbitrary set of selected points, a polyhedron containing these points must be defined first. As an initial approximation, we compute a convex hull for these points and construct a set of all integer points within it. Then, we compute a set of difference between convex hull points and selected points. If it is empty, the convex hull is used, otherwise we check if the remaining points fit a multidimensional linear function. In this case, the function is used as a constraint with an existentially quantified dimension instead of the constant to complete the convex hull, otherwise we discard the transformation until the selection is updated. Although discarding several irregular selection cases, this algorithm offers reasonable trade-off between performance and typical case coverage.

4. EVALUATION OF CLINT

We conducted two controlled experiments to evaluate *Clint*'s design and assess its benefits over manual parallelization methods. In the first experiment, we focused on the visual representation and in the second we compared its direct manipulation approach with manual code transformation.

4.1 Exp. 1: Suitability of the Visualization

In this experiment, we assess the suitability of our visual representation of program statements in the polyhedral model. Although similar visualizations have been already used for descriptive or pedagogical purposes, there is no empirical evidence of their appropriateness for conveying program structures. In order to inform the design of *Clint*, we are testing whether programmers with different expertise in parallel programming and optimizing transformations are able to deduce the corresponding code from a visualization and vice versa, at several levels of difficulty.

Participants – We recruited 16 participants – 12 male, 4 female, aged 18-53 – from our organizations. All of them have experience in imperative programming with C-like languages and previous knowledge of the polyhedral model. Six participants already used iteration domain visualizations in their work and were thus considered as experts.

Procedure – The experiment is a $[3 \times 2]$ between-subject design with two factors:

- Task: (i) writing a code snippet which corresponds to the given visualization using a C-like programming language, which had loops and branches with affine conditions (VC); (ii) drawing an iteration domain visualization given the corresponding code (CV).
- Difficulty: problems may be (i) two-dimensional with constant bounds (*Simple*); (ii) multi-dimensional with constant bounds (*Medium*); (iii) two-dimensional with branches and mutually dependent bounds (*Hard*).

In order to avoid learning effect and to ensure consistent difficulty over tasks, participants were divided in two groups with the same number of experts. Group 1 was asked to perform the visualization to code task (VC), and group 2 the code to visualization task (CV). The order of task difficulty was counterbalanced across participants. Both tasks were performed on paper, with squared graph paper for the CV condition. Participants were presented with the visualization and did two practice tasks at the beginning of the session. They were instructed to perform the tasks 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 min.

Data Collection – For each trial, we measured Completion Time, Error and Abandon rates. The errors were split in two categories: *type I*, the shape of the resulting polyhedron was drawn correctly, but linear sizes or position were wrong; *type II*, the shape of the polyhedron was incorrect. Codes describing the same iteration domain were considered equivalent (e.g. `i <= 4` and `i < 5`). We also videotaped participants activity and collected the materials they produced. After they completed the study, participants were asked about their strategies to accomplish the task as well as any suggestion on the visualization.

4.1.1 Results

We did not observe any significant learning effect and we discarded the trials in which the participants produced syntactically incorrect or not static control code.

Completion Time – We found a statistically significant effect of Difficulty with all difficulty levels being different (*Easy* = 114.3s, *Medium* = 235.8s and *Hard* = 437.9s). We also found a significant Expert×Difficulty interaction, which is explained by better performance of experts for the *Hard* tasks (319s vs 556s, see Fig. 7).

Errors – Participants performed the tasks with very low error rates (VC = 8.3%, CV = 4.1%). We observed only two withdrawals during a trial, both from non-experts, and after a significant amount of time. For the type of errors, some non-experts were not able to propose code for some hard tasks, while experts mostly made type I errors for some medium tasks (Fig. 8). However, it is hard to conclude on the causes of errors with such low error rates.

Qualitative Data – Participants' feedback also allowed us to improve the visualization since some of them noticed that overlapping statements and visual axis sharing could be interpreted ambiguously. Half of them also stated that the

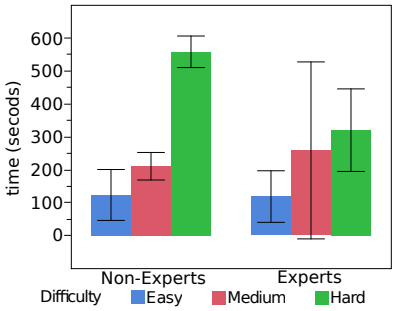


Figure 7: COMPLETION TIME increases with task difficulty but is lower for *experts*.(Error bars show 95% confidence intervals)

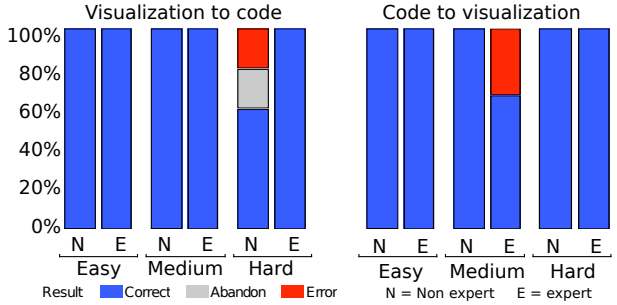


Figure 8: Percentage of errors and withdrawal: experts were slightly more successful than non-experts, but failed at simpler tasks. Only non-experts abandoned tasks. The overall error rate is less than 10% for each task.

visualization significantly helps to understand the program structure, 31% that it rather helps and 19% that it does not change their level of understanding, but does not harm.

Overall, these results suggest that both experts and non-experts programmers were able to reliably map our visualization to the corresponding code, most of them stating that it has potential in assisting them in understanding programs. While the task they performed in this study does not belong to the program parallelization process, it shows that the scheduled iteration domain visualization is an efficient representation of static control parts of the program.

4.2 Exp. 2: Benefits of Direct Manipulation

In this second experiment, we compare *Clint* with common manual code transformation. Beyond the preliminary assessment of its efficiency, we are also interested in its acceptability by expert programmers who are more used to text-based interfaces. Participants who already took part in the first experiment were asked to perform some parallelization tasks at several levels of difficulty and in three conditions: source code (the baseline), *Clint* without source code, and *Clint*, the latter assessing participants' preference between direct manipulation and source code editing. Our hypotheses are that *Clint* can improve programmers accuracy and efficiency when parallelizing code, but also that the direct manipulation approach is likely to change their strategy when they address a parallelization problem.

Participants – Eight participants took part in this experiment (5 male, 3 female, aged 23-47). All of them had participated in the first experiment, and were thus familiar with the polyhedral model and our visualization.

Apparatus – The experiment was conducted with a specific version of our *Clint* prototype, implemented in C++, on a 15" MacBook Pro. Participants were interacting with a keyboard and a mouse. The language used was a subset of an imperative language with C-like syntax.

Procedure – The task consists in transforming a loop-based program so that the maximum number of loops becomes parallelizable, i. e. without any dependences that prevent parallel execution. Participants were asked to transform the program, but not to write the actual parallel code in order to avoid bias from individual expertise in using a particular parallel language. The experiment is a $[3 \times 3]$ within-subject design with two factors:

- TECHNIQUE: (i) code editing (*Code*); (ii) direct manipulation without code (*Viz*); (iii) full interface, with direct manipulation and source code editing (*Clint*).
- DIFFICULTY: (i) two-dimensional case with at most two transformations (*Simple*); (ii) two- or three-dimensional case with rectangular boundaries and at most three transformations (*Medium*); (iii) two- or three-dimensional case with non-trivial boundaries and at least two transformations (*Hard*).

Trials were grouped in three blocks by TECHNIQUE. The *Code* and *Viz* blocks were presented first in counterbalanced order across participants. *Clint* was always presented last, in order to assess participants' preference in using code editing or direct manipulation. In each block, participants were presented with one task of each difficulty level in random order (tasks were different from one block to another). The tasks were drawn from real-world program examples and simplified (see Appendix B). Trials were not limited in time and participants were asked to explicitly end the trial when they thought to be done, whether they succeed or not. 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 and were allowed to perform two "recall" practice trials before each TECHNIQUE block. Each session lasted about 60 minutes and participants answered a short questionnaire at the end.

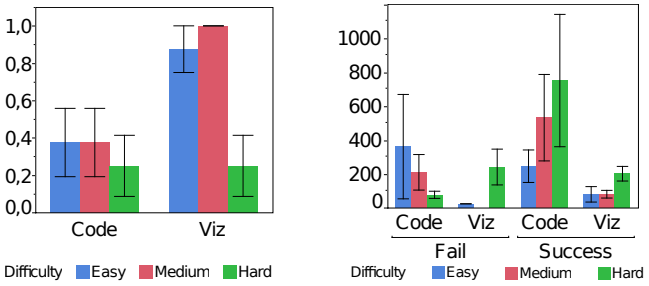
Data Collection – For each trial, we measured: (i) the overall trial COMPLETION TIME; and (ii) TRANSFORM TIME, the amount of time from the start to the first change in the program structure (code edited or visualization manipulated). We recorded both final and intermediate transformations to the program.

4.2.1 Results

We did not observe any significant ordering effect of TECHNIQUE or DIFFICULTY on COMPLETION TIME and SUCCESS RATE. Because this experiment was conducted with a small sample, we opted not to conduct any statistical analysis.

Accuracy and Efficiency – Fig. 9a shows the SUCCESS RATE, defined as the percentage of trials where all possible loops became parallelizable, for each TECHNIQUE and in each DIFFICULTY condition. Despite large variability, it suggests that participants were in general more successful to find the expected transformations with direct manipulation than with code editing for *Easy* and *Medium* tasks (about 90% success rate vs 40%). For the *Hard* condition however, SUCCESS RATES are very similar (around 25%). Fig. 9b also suggests that participants often performed faster and that COMPLETION TIME is likely to be more consistent over participants with the direct manipulation interface (smaller standard error).

Strategy and exploration – In terms of strategy, the ratio of tasks were participants at least tried to perform a transformation is of 76% with *Code*, against 94% for *Viz*.



(a) Success Rate (b) Completion Time (s)

Figure 9: (a) Success Rate is higher with *Clint* for *Easy* and *Medium* tasks, but similar to *Code* for *Hard* tasks. (b) Average Completion Time is often lower with the *Viz* technique, especially when tasks were successfully performed. (error bars show 1 standard error from the mean)

Additionally, we observed that the time it took to participants to start modifying the program is of 135s on average with *Code* against 13s with direct manipulation. We also computed the ratio Transform Time/Completion Time as a measure of "engagement" of the participants (a lower value meaning that the participant started to transform the program faster). As shown in Fig. 10, this ratio increases with difficulty for *Code*, but drastically decreases for *Viz*. It suggests that participants were more likely to adopt an exploratory strategy for hard transformation problems with the interactive visualization than with code editing.

Code editing or direct manipulation? – For the *Clint* condition, we observed that all the participants used the interactive visualization and that only three of them edited the code during the first 30s of two trials on average before switching to the visual interface (12% of all the trials). In the post-experiment interview, these participants explained that they were trying out the code-visualization mapping or changing the code for the sake of analysis. We found Success Rate and Completion Time to be very similar to those with only the visualization. Qualitatively, we observed that several participants were examining the original and transformed source code, but not editing or selecting it. These results suggest that most users would prefer using the visual interface to perform transformations, but still need the source code view to have a link with conventional program editing approach.

4.3 Discussion

Although conducted with a small number of participants and on targeted tasks, this preliminary study gives interesting insights into the appropriateness of *Clint*'s direct manipulation approach for program parallelization. First, in terms of performance and accuracy, it suggests that the interactive approach could help programmers to accurately extract parallelism and apply transformations faster than with standard tools. However, we only compared *Clint* with manual code editing as a baseline, and we did not consider automatic/semi-automatic approaches (e.g. *Pluto* [5] or *Clay* [2]) that could also assist users in managing the complexity of parallelization tasks. We can expect *Clint* to be a good complementary approach anyway, since it builds upon these tools in order to give more control to the programmer.

We also observed that *Clint* effectively changed programmers strategy. It allows them to explore and manipulate

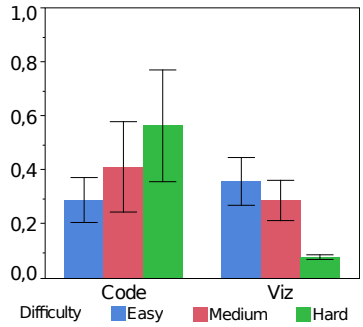


Figure 10: Ratio of first change time to completion time. The change in trend between different techniques can be caused by an improved problem understanding and favorisation of exploring different transformations. (error bars show 1 standard error from the mean)

programs structure from the very beginning of the task, thanks to its visual affordances and transformation undoability. Clint may also favor attention switch from syntactical constructs like loops to dependences in data flow. We believe that this "active" exploration approach could help programmers to better learn some typical solutions to given situations, to recognize those situations thanks to visual patterns, and to reuse the gathered knowledge in new situations [21]. This would however require deeper investigations and long-term field studies on the usage of *Clint*.

Our preliminary tests and studies of *Clint* revealed good acceptance by expert programmers, who are known to be reluctant to use visual programming tools. We believe that the way *Clint* allows direct manipulation of the concepts that programmers use for parallelization favors its acceptance: instead of hiding its underlying complex model, *Clint* "reveals" it and helps to manage its complexity [17].

5. RELATED WORK

Visual Representations for Polyhedral Model – Scatterplot-like projections of loop iteration domains are extensively used in the literature on the polyhedral model. Popular polyhedral libraries provide interface to generate visualizations, such as *VisualPolylib* component for PolyLib [16] and islplot[2] for isl [20]. *LooPo* [10] visualizes dependences in iteration domains before and after automatic parallelization while *3D iteration space visualizer* [24] allows to mark desired dimension parallel to start automatic transformation search. Tulipse [23] integrates visual dependency analysis into the Eclipse IDE. These tools allow to interact only with the visualization while *Clint* translates manipulations back to the polyhedral representation and ultimately transforms the code to match the visualization.

Semi-Automatic Polyhedral Program Transformations – A handful of polyhedral frameworks provide a semi-automatic approach to program restructuring based on directive scripts implementing classical loop transformations, *UTF* being arguably the first of them [13]. *URUK* enables the composition of a complex sequence of transformations decoupled from any syntactic form of the program [9]. *CHiLL* enforces legality of each transformation in a sequence by intermediate dependence checks [6]. *AlphaZ* allows to redistribute data in the memory [25]. We propose an alternative formalism, *Clay*, that builds on unions of relations to

[2]http://tobig.github.io/islplot/

provide high composition capabilities. We rely on it to convert visual actions to mapping relations without having user to input the textual form of the transformation sequence.

6. CONCLUSION

Clint brings intuition in loop parallelization by visualizing iterations with real-time feedback on data dependences, and enables program restructuring through graphical actions. It addresses challenges of semi-automatic approaches to loop transformations such as transformation composition and refinement or target selection. The results of our preliminary studies provided empirical evidence that the visualization approach is efficient and reliable, and confirmed the benefits of direct manipulation for the efficient exploration of possibilities for program parallelization. We believe our approach to be a promising first step towards better parallelization tools that leverage the power of analytical models by giving more control and expressiveness to programmers.

Our studies also revealed several possible improvements to *Clint* as well as new research directions: (1) enrich the editor with smooth transition between the original and transformed code and the visualization using advanced animation techniques [7]; (2) use three-dimensional transforms to reveal hidden or overlapping points and dependences; (3) provide dynamic visual feedback on the transformation legality and interactive guidance through manipulation restrictions/enhancements (e.g. pseudo-haptic feedback [15] or semantic pointing [4]); (4) investigate scaling of this approach to represent data flow in programs and expose coarser-grain parallelism; and (5) investigate the use of interactive visualization for learning parallelization and the polyhedral model.

7. REFERENCES

[1] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proc. of PACT '04*, pages 7–16. IEEE, 2004.

[2] C. Bastoul. *Contributions to High-Level Program Optimization*. Habilitation Thesis. Paris-Sud University, France, 2012.

[3] M. Beaudouin-Lafon. Designing interaction, not interfaces. In *Proc. of AVI '04*, pages 15–22. ACM, 2004.

[4] R. Blanch, Y. Guiard, and M. Beaudouin-Lafon. Semantic pointing: Improving target acquisition with control-display ratio adaptation. In *CHI '04*, pages 519–526. ACM, 2004.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. of PLDI '08*, volume 43, pages 101–113. ACM, 2008.

[6] C. Chen, J. Chame, and M. Hall. Chill: A framework for composing high-level loop transformations. *U. of Southern California, Tech. Rep*, pages 08–897, 2008.

[7] P. Dragicevic, S. Huot, and F. Chevalier. Gliimpse: Animating from markup code to rendered documents and vice versa. In *Proc. of UIST 11*, pages 257–262. ACM, 2011.

[8] P. Feautrier and C. Lengauer. Polyhedron model. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer US, 2011.

[9] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[10] M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004. habilitation thesis.

[11] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. *Human–Computer Interaction*, 1(4):311–338, 1985.

[12] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The omega calculator and library, version 1.1. 0. Technical report, College Park, MD, 1996.

[13] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report UMIACS-TR-92-126.1, University of Maryland Institute for Advanced Computer Studies, 1993.

[14] H. Le Verge. Recurrences on lattice polyhedra and their applications, April 1995. Unpublished work based on a manuscript written by H. Le Verge just before his untimely death in 1994.

[15] A. Lécuyer, J.-M. Burkhardt, and L. Etienne. Feeling bumps and holes without a haptic interface: The perception of pseudo-haptic textures. In *CHI '04*, pages 239–246. ACM, 2004.

[16] V. Loechner. Polylib: A library for manipulating parameterized polyhedra. 1999.

[17] D. Norman. *Living with complexity*. MIT Press, 2011.

[18] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *PLDI '08*, pages 90–100. ACM, 2008.

[19] N. Vasilache, C. Bastoul, S. Girbal, and A. Cohen. Violated dependence analysis. In *Proc. of ICS'06*, pages 335–344, Cairns, Australia, June 2006.

[20] S. Verdoolaege. isl: An integer set library for the polyhedral model. *Mathematical Software–ICMS 2010*, pages 299–302, 2010.

[21] C. Ware. *Information visualization: perception for design*. Elsevier, 2012.

[22] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[23] Y. W. Wong, T. Dubrownik, W. T. Tang, W. J. Tan, R. Duan, R. S. M. Goh, S.-h. Kuo, S. J. Turner, and W.-F. Wong. Tulipse: a visualization framework for user-guided parallelization. In *Euro-Par 2012 Parallel Processing*, pages 4–15. Springer, 2012.

[24] Y. Yu and E. D'Hollander. Loop parallelization using the 3d iteration space visualizer. *Journal of Visual Languages & Computing*, 12(2):163–181, 2001.

[25] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye. Alphaz: A system for design space exploration in the polyhedral model. In *Languages and Compilers for Parallel Computing*, pages 17–31. Springer, 2013.

[26] O. Zinenko, S. Huot, and C. Bastoul. Clint: A direct manipulation tool for parallelizing compute-intensive program parts. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, pages 109–112. IEEE, 2014.

APPENDIX

A. USE CASE

In this Appendix, we give more details about *Clint*'s interactive approach through a step-by-step scenario based on a larger example with several multidimensional loop nests. This example taken from our preliminary study illustrates both the benefits and the limitations of the approach. The code of this example (see Fig. 11(right)) was adapted from an ad-hoc physical process simulation program developed in one of our institutions. This adaptation involved reformatting and variable renaming for the sake of brevity. We also modified the branch condition to fit the restriction of the Clan polyhedral raising tool, and added `scop/endscop` compiler pragmas around the loop nests to target this specific part of the code (automatic SCoP extraction is possible *via* Clan, but requires to choose a specific SCoP to interact with). The code preforms an operation similar to a triangular decomposition of a matrix with a subsequent search for the index of a row with particular properties.

The baseline for the parallelization task uses Pluto 0.11.1 automatic optimizer called with parallelization objective as `polycc -parallelize`. The generated code was compiled using GCC 4.9 with level 3 optimization enabled, and with OpenMP 4.0 on a x86_64 Linux machine with an Intel i5 T2405S quad-core processor. For the constant parameter value $n = 1000$, the execution of the original code took 86 msec on average out of 5 consecutive executions, while the Pluto-generated version took 80 msec on average. We used PolyBench infrastructure for benchmarking this code.

A typical parallelization of this code with our interactive approach may be done as follows:

1 - Program Fragment Loading: The interactive session starts when the user loads a code file with *Clint*. Since it contains a manually marked static control part, it uses Clan to extract polyhedra from the selected loop nests and displays the corresponding visualization (see Fig. 11(left)). Despite many dependency arrows, the user can visualize the structure of the loop nests projected on the two outer dimensions as well as the distribution of the statements in the loops. Since *Clint* builds upon the mathematical representations of lattice polyhedra, the visualization is read from the bottom to the top (following the conventional direction of the vertical axis). However, this can also be configured to match the top-to-bottom order of statements in the code. As described in the paper, statements sharing a loop in the code share an axis in the visualization, e.g. all the statements of the outer loop on `j` are displayed in the leftmost vertical pile. Statements sharing two loops are displayed in the same two-dimensional coordinate system and may overlap, e.g. the top left coordinate system in the Fig. 11. Depending on the configuration of the visualization, the polygons representing statements may perfectly overlap (as in Fig. 11) or be slightly displaced with respect to each other as described in the section 2.2 of the paper. Finally, a crossing arrow sign is displayed between coordinate systems to notify that multiple dependency lines are connecting the corresponding loops, preventing visual cluttering. Conversely, the dependency lines within a coordinate system are always fully drawn, which can drastically reduce the readability of the polygon visualization for certain statements.

2 - Direct Manipulation Restructuring (shift): After the visualization is displayed, the user can check if all of the nested statements in two loops on `j` and `i` are involved in the dependences. To that end, she uses the *displaced visualization mode* (see section 2.2 in the paper) in order to select each particular polygon representing statement, and to drag them on the right side so that they do not overlap anymore (see Fig. 12). This direct manipulation results in two *shift* transformations being applied to the code, essentially breaking the initial internal loop nest into three different loops. As this operation significantly restructured the code, the user set up the code view to display the original version along the visualization, while the transformation sequence is preserved.

3 - Code Editing: The user observes that all statements instances in this loop nest are connected by dependences to multiple other instances of the same statement, and that they are also involved into into dependences between statements. She notices the homogeneity of the dependence arrows across different statement instances and supposes that each statement instance depends on all other instances of the same statement, which corresponds to a scalar dependence To check this hypothesis, she manually modifies the source code by introducing an array subscript to the only scalar used in the last statement (Fig. 13(right)). *Clint* automatically reapplies the transformation sequence to the new code and, as the transformation is applicable without errors, it regenerates the visualization to match the generated code. The visualization is then automatically updated to match the new source code.

4 - Direct Manipulation Restructuring (shift/undo): The user then notices that scalar expansion indeed removed most of the dependences between the instances of this statement, leaving only those of the loop on `j` for a fixed `i` iteration (see Fig. 13(left) with horizontal lines). These dependences reflect the reuse of `L[i][j]` and `L[j][j]` in the last statement. The user drags the polygons back to their original positions thus performing another group of *shift* transformations, effectively undoing the previous ones. Although this transformation is not guaranteed to achieve parallelization, it allows a better understanding of the code, with a better visual representation of the non-scalar dependences that might prevent parallelization.

This step illustrates the benefits of the real-time visual feedback and the implicit undoability of the transformations for a parallelization task. It also reveals some of the limitations of the approach and possible improvements: (1) if the scalar dependences were automatically detected by the polyhedral back-end, it would make it possible to display them without making the visualizations more complex; additional instruments for data layout manipulation would allow to perform scalar expansion or privatization for such loops when necessary; (2) although the second group of shift transformations is reciprocal to the first one, both are added to the transformation history and reapplied each time needed, thus introducing syntactic changes from the code generator (i.e. `[i+n-n]` indexing expressions).

5 - Code Editing: The user then decides to expand scalars in all statements by following the same principle. When she edits the code to expand the scalars `s` and `m`, since they are only used within one outer loop iteration, she "clears" the visualization of the scalar dependences which makes it easier to address the actual parallelization task (see Fig. 14(left)). In this example, the expansion can be replaced by scalar privatization to reduce the memory foot-

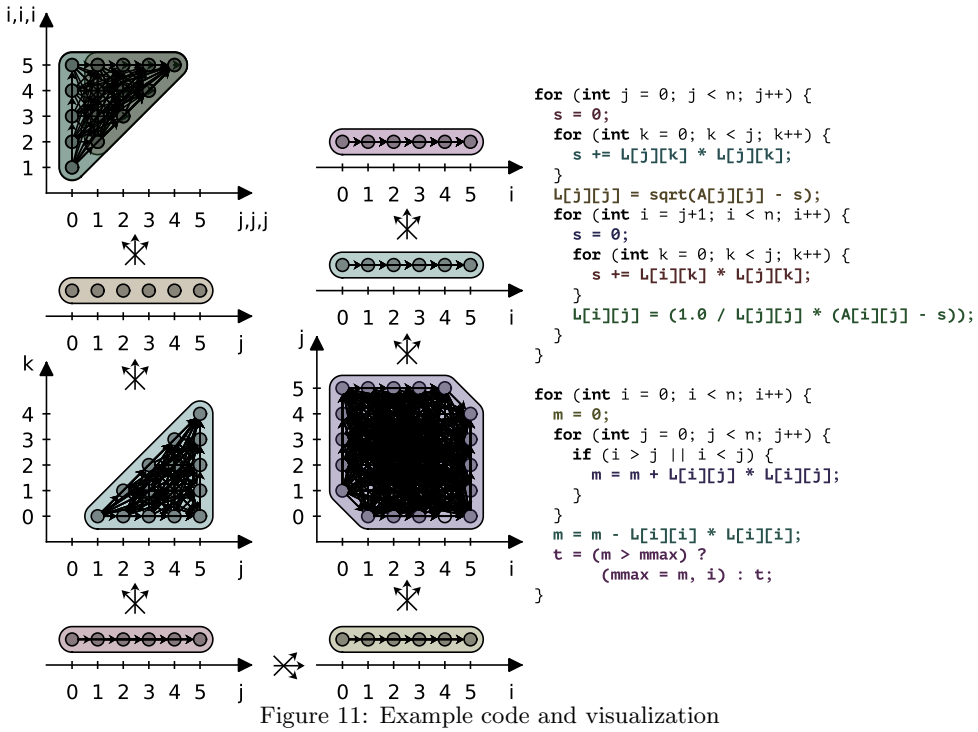


Figure 11: Example code and visualization

print. This modification by itself renders the inner loop on `i` parallelizable, and the corresponding code is generated as highlighted by the arrow in Fig. 14. Furthermore the dependences in the second loop nest do not split anymore between different iterations of the loop, which suggest the loop may be parallelizable. *Clint* emphasizes this change by actually displaying the parallel dependence arrows between coordinate systems in the vertical pile. However the last statement reuses the scalar `t` which can not be expanded, preventing the parallelization.

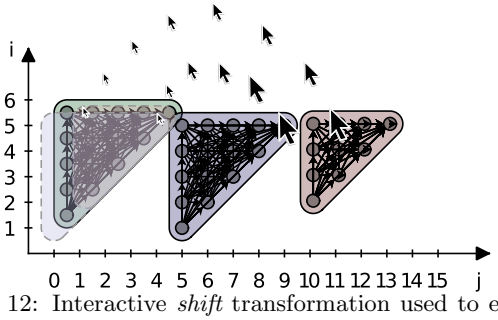


Figure 12: Interactive *shift* transformation used to explore dependences between statement in the loops, the code is freezed in the original version during transformation

6 - Direct Manipulation Restructuring (splitting): The user then decides to grab the polygon corresponding to the last statement and put it to the right, creating a new pile of coordinate systems (Fig. 15(left)). This operation corresponds to the *loop splitting* transformation. Thanks to the scalar expansion on the previous steps, the statement that was put in a separate loop uses have all input data generated by the previous loop nest and stored in memory. The transformation is this legal since no dependences are violated (as depicted by black crossing arrows between piles) and the semantics is preserved. The outer loop of the second loop nest became parallel and the corresponding code is generated (Fig. 15(right)).

7 - Code Generation: The resulting code now looks sufficiently parallelized and the user decides to benchmark its execution. She saves the generated code to the file and uses his standard procedure to compile and execute the program.

The newly generated code, with OpenMP pragmas, now takes only 17 msec to execute in the same setup as before and outperforms the result obtained by the automatic parallelizer. We did not specifically tried to optimize cache use with this example, but manual data manipulations may have resulted in an additional speedup.

Overall, this example demonstrates the possibilities of the interactive approach to polyhedral transformations implemented in *Clint*, with both its benefits and limitations. In particular, some work is still required to manage programs of increasing complexity in an interactive way, but we advocate for an approach combining semi-automatic and controllable polyhedral tools with advanced interactive visualizations as a way to achieve better program performance. Even in cases of a significant visual load, important information may be extracted from the visualization such as frequent dependence patterns. A successful visual approach may not only help explaining and using the polyhedral model, but to improve the polyhedral tools and frameworks themselves.

B. EXPERIMENTAL DATA

B.1 Suitability of the Visualization

Scatterplot-like visualizations are widely adopted in the literature on polyhedral model. Most of them use color and shape-coded dots to represent statement instances and arrows to represent dependences between them with multiple variations depending on the particular tool or task. In *Clint*

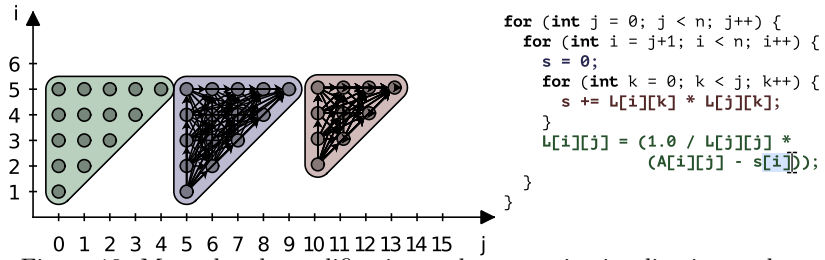


Figure 13: Manual code modification and automatic visualization update

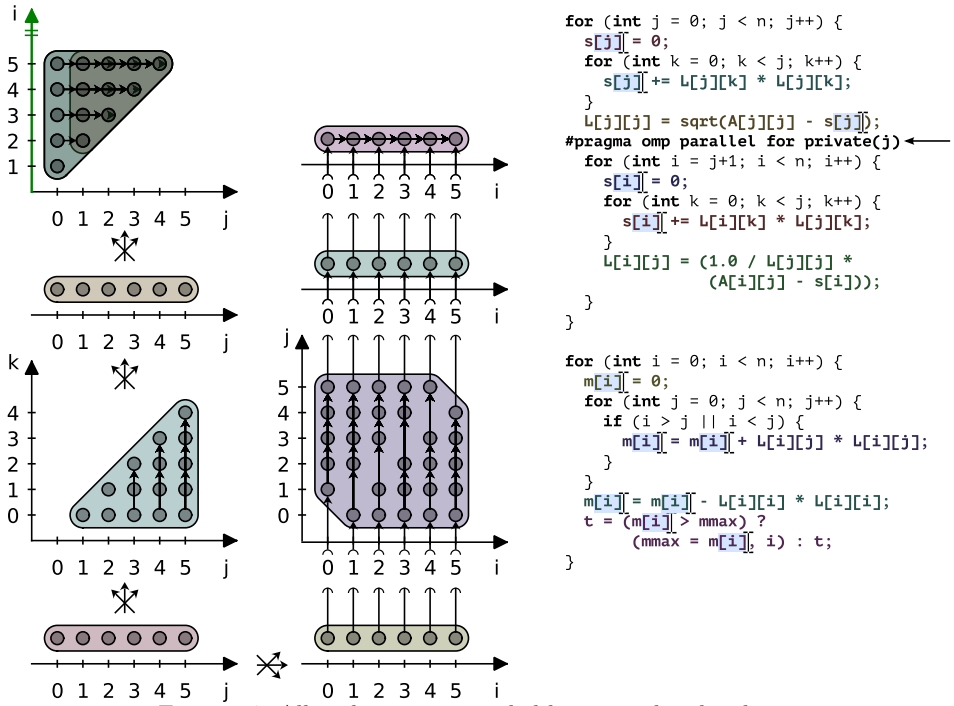


Figure 14: All scalars are expanded by manual code editing

we decided to use color-coded polygons in order to visually represent statements while keeping conventional use of dots and arrows. This choice allows to manipulate statements directly without having to select a set of statement instances subject to transformation.

Code Fragment Sources: Since no empirical evidence existed to support this kind of visualizations, we conducted a controlled experiment to evaluate it. In order to evaluate the expressiveness and the suitability of the visualization approach in the first experiment, we used a set of automatically generated possibly multidimensional loop nests. The code generation was done by a Python script in two steps: first, loop structure was generated to satisfy the difficulty condition; then a statement or group of statements without dependences was introduced into each loop nest. We decided not to introduce the dependences in the visualization in order to focus on the notion of statements and statement instances as well as on the relation between nested loops and axes. The difficulty was determined as a function of the number of nested loops and the number of non-constant loop boundaries. Before running the measured experiment, we observed that non-constant loop bounds are more difficult to handle in both tasks. We also discarded 5-dimensional and more loop nests: with constant boundaries, they can be treated similarly to the three-dimensional, while with non-constant mutually dependent variables physical execution of the task would take unreasonable amount of time for the experiment. A set of 24 code fragments, 8 of each difficulty, was chosen for the experiment. They were manually verified for correctness by the experimenters and compiled with GCC 4.8 compiler for syntax check. We chose not to use PolyBench or any other examples from the literature on the polyhedral model since several participants of our studies were familiar with the relevant literature on the polyhedral model in general and with the PolyBench codes in particular thus biasing the results.

A visual representation for each code fragment used was generated by an early prototype of *Clint*. It was verified to match the code fragment manually by the experimenter.

In the Fig. 16, we provide examples of the code fragments and respective visualizations used for this experiment.

Particularities of the Prototype: After the experiment, we asked participants to comment on their choices in the visual representations designed on paper and incorporated some of them in the final version of the visualization. The visualization used at this prototype differs from a more recent versions in an inverted vertical axis that matched the order of statements in the code and the absence of dis-

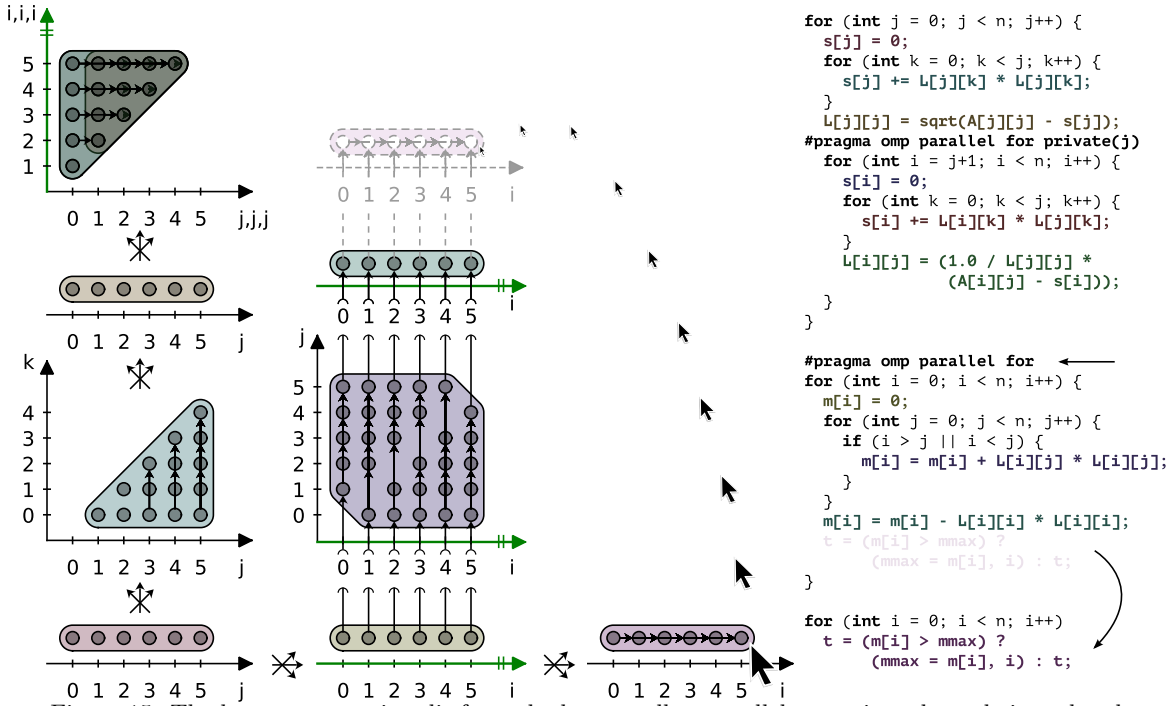


Figure 15: The last statement is split from the loop to allow parallel execution, the code is updated

placement between dots that represent instances of different statements. Although both changes are related to interaction: axis direction may be switched and displacement may be turned on and off as the program run, their configurability was found quite important by multiple participants. This experiment also gave as feedback on the limitations of the visualization techniques related to the large numbers of statements or deeply nested loops.

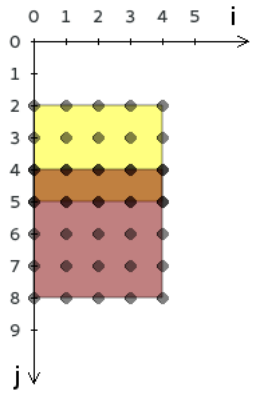
B.2 Benefits of Direct Manipulation

Sources of Code Fragments: For the evaluation of the interactive part that allowed us to gather insights about the benefits of the direct manipulation, we created a set of 20 programs of varying levels of difficulty following the same reasoning as before. The program fragments were created as combinations of loop nest structure and code statements with dependences. In order to find such code statements, we ran Clan polyhedral extractor on several computational programs and libraries, including Overture [3] partial differential equation solver framework and matrix manipulation library, PETSc [4] scientific computation library and multiple smaller ad-hoc physics numerical simulation programs. We ran Candl to analyze dependences in the extracted static control parts of the programs and removed those fragments that did not have or had a very large number of dependences. From the remaining set of program parts, we arbitrarily chose several static control parts and analyzed them manually. For each such part, we generated a visualization with Clint and tried to create a sequence of transformation that would enable parallelization. We further focused on the program parts coming from the specific computation-related parts of these code bases relying on the fact that parallel versions of the same computational operations may have been developed. For a selected subset of static control parts, we removed all statements that were not related to the computation and dependences, we also inlined short functions. These sequences of statements were then put into different loop structures with more or less trivial loop bounds and branching conditions. The resulting codes were thus based in the real-world programs, but were not the carbon copies of existing code.

Avoiding Task Recognition: Using this approach, we wanted to reduce the complexity of the code and focus our study on the interaction with loop-based program parts amenable to the polyhedral model. We also intended to diminish the possibility of a code fragment being recognized by the user and completely rewritten to a parallel version. Although frequent in the real setting, this situation should be studied separately from the evaluation of the interactive approach. In the post-experiment questionnaire, we asked participants if they had recognized the operation performed by the code fragment (for example LU matrix decomposition) and if so, did they use prior knowledge about parallel version of this operation. Several participants recognized the general type, for example "matrix manipulation" but were not able to tell which operation is was. Thus no participants relied on the previous knowledge of the parallel version of the operation, but some of them mentioned "general approaches" for loop parallelization. For example, one expert participant said he mechanically applied skewing transformation when he encountered a statement with array access indexed by a sum of iterators (like `Z[i + j] *= A[i] * B[j]`).

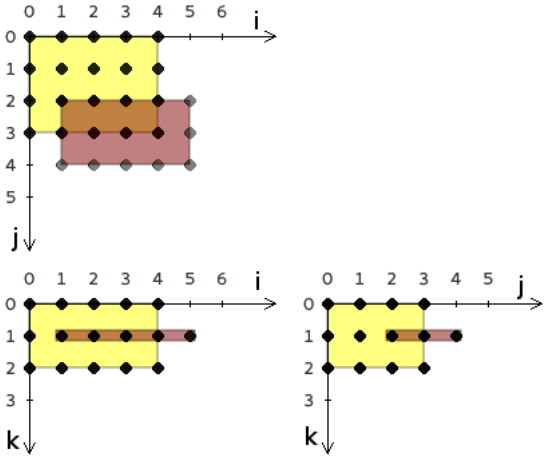
In the Fig. 17, we demonstrate examples of the code fragments and respective visualizations used as for the parallelization task.

[3] http://overtureframework.org/
[4] http://www.mcs.anl.gov/petsc/



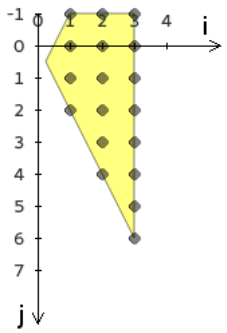

```
for (i = 2; i < 9; i++)
  for (j = 0; j < 5; j++) {
    if (i < 6) S1();
    if (i > 3) S2();
  }
```

(a) easy case: two statements partially overlap in a 2d shared loop nest with constant boundaries




```
for (i = 0; i < 6; i++)
  for (j = 0; j < 5; j++) {
    if (i < 5)
      if (j < 4)
        for (k = 0; k < 3; k++)
          S1(i, j, k);
    if (i > 0)
      if (j > 1)
        S2(i, j);
  }
```

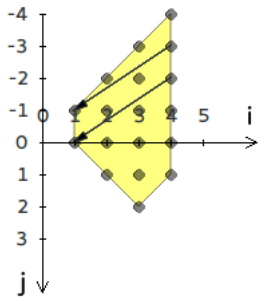
(b) medium case: two statements partially overlap in a 3d shared loop nest with constant boundaries




```
for (i = 0; i <= 3; i++)
  for (j = 1; j <= 2*i; j++)
    if (2*i + j >= 1)
      S(i, j);
```

(c) hard case: one statement in a 2d loop nest and with non-constant boundaries depending on the outer loop variables

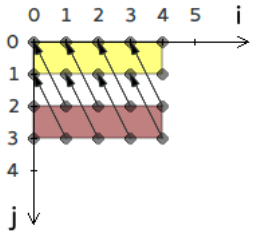
Figure 16: Examples of scatterplot-like visualizations (left) and code fragments (right) used in the Suitability of the Visualization experiment.




```
for (i = 0; i < N + 1; i++)
  for (j = -i; j < i; j++)
    if (i + j - N - 1 <= 0)
      z[2*i + 3*j] +=
        A[i][j] * B[i][i] * m[j];
```

Skew the statement right to make the dependence lines vertical in order to parallelize the outer loop. Such transformation is legal given that the `+=` operation is associative. The inner loop is directly parallelizable without transformation, but no participant made this observation with parallelization feedback turned off.

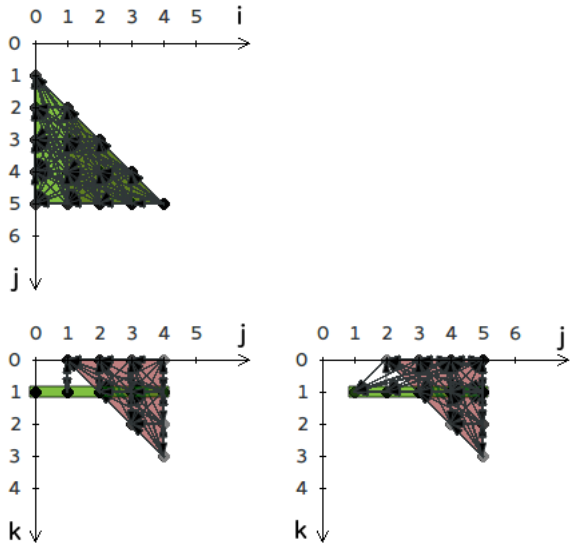
(a) easy case: one statement featuring parallelism in the inner loop that may be kept as is or transformed to expose parallelism in the outer loop




```
for (i = 0; i < 5; i++) {
  for (j = 0; j < 2; j++)
    A[i][j] = init(i, j);
  if (i > 0)
    for (j = 2; j < N; j++)
      A[i - 1][j - 2] += func(P[i][j]);
}
```

Shift the first statement right one time (equivalent to replacing `i` with `i - 1` including in loop boundaries) and down two times (equivalent to replacing `j` with `j - 2`), then fuse internal loops and parallelize.

(b) medium case: two statements in different loops that require three transformations to parallelize




```
for (i = 0; i < N ; i++)
  for (j = i + 1; j < N; j++) {
    s[i] = 0;
    for (k = 0; k < i; k++)
      s[i] += L[i][k] * L[j][k];
    L[j][k] = L[i][i] * A[j][i] - s[i];
  }
```

Expand `s` over two outer loops then split the third statement out of these loops. The first loop nest becomes parallelizable. This case offers a tradeoff between memory usage and performance in case of parallel execution since size of `s` will grow significantly. Another possible outcome of this trade off would be to privatize `s` in the second loop, which does not make the loop parallelizable. The innermost loop may be transformed to a parallel reduction, but existing polyhedral tools do not support such operations.

(c) hard case: three statements in a 3D loop nest with numerous dependences that require data layout adjustments

Figure 17: Examples of *Clint* prototype interactive visualizations used in the Benefits of the Direct Manipulation experiment along with the corresponding code fragments and solution suggestions.