# Recover Polyhedral Transformations From Polyhedral Scheduler

Nelson Lossing
Huawei Technologies France
France

Walid Astaoui
Huawei Technologies France
France

Gianpietro Consolaro
Huawei Technologies France
France

Harenome Razanajato
Huawei Technologies France
France

Zhen Zhang
Huawei Technologies France
France

Denis Barthou
Huawei Technologies France
France

## Abstract

While Polyhedral optimization offers good performance results, it is often at the price of understandability. The Chlore project — using Clay transformations — is a first attempt at describing Polyhedral optimizations as sequences of high-level transformations. However, Chlore's output may often be overly complex or contain redundant transformation passes.

In this paper, we extend on Chlore's ideas. We present techniques to reduce the number of generated transformation primitives and to optimize the recovery algorithm, showing some important speedups compared to the original Chlore algorithm.

***Keywords:*** Loop Transformations; Polyhedral Scheduling; Polyhedral Transformations

## 1 Introduction

The polyhedral model is used in optimizing compilers to find and apply loop transformations automatically in order to maximize the input kernel performance. While much of the research focuses on developing better heuristics used by the polyhedral scheduler, a significant amount of effort has also been directed towards the explainability of the final transformation. A deeper understanding of this process can reveal the drawbacks of current heuristics and help identify areas for improvement, particularly when dealing with the heterogeneous characteristics of different architectures. Moreover, frameworks such as Halide [10] or TVM [5] provide users with high-level interfaces for manual scheduling and understanding good automatic optimization may help leveraging such frameworks, especially if the developper is not a polyhedral model expert.

Chlore [2, 12] attempted to translate the full polyhedral transformation, which is typically described as an algebraic transformation, into a series of transformation primitives. These transformation primitives include loop reordering, loop fusion, shifting, and more. This type of translation, or recovery, allows for a better understanding of the transformation applied by the polyhedral scheduler. However, Chlore's output may sometimes remain overly complex or contain redundant transformation passes.

In this paper, we propose extensions to address these issues. Experiments show that our approach significantly reduces the required number of transformations to retrieve an equivalent polyhedral optimization. Moreover, our recovery is much faster.

Section 2 will present related work, mostly Chlore, and its drawbacks. We redefine a smaller set of transformation primitives presented in Section 3. Section 4 explains our recovery algorithm. Experimental results are presented in Section 5, and some future work is proposed in Section 6.

## 2 Related Work

Chlore [2, 12] uses Clay as the set of primitives to recover. Clay provides a series of scheduling primitives covering the full space of polyhedral scheduling transformation. It can be used to apply a list of transformations directly.

Chlore is a tool that translates polyhedral scheduling functions into a list of scheduling primitives. This list of primitives represents an equivalent transformation and is chosen from the set defined by Clay. The main advantage is that lists of scheduling primitives are oftentimes far easier to understand and customize for humans than mathematical formulas. Chlore expects as input two polyhedral schedules, in Openscop format [3]: a source and a target schedule. The recovery algorithm will try to generate a list of scheduling primitives that transforms the former into the latter.

The algorithm assumes that the source and target scheduling is represented using the 2d+1 format [11]. In this representation, even dimensions, named *$\alpha$-dimensions*, represent *loop iterator* transformations while odd dimensions, named *$\beta$-dimensions*, represent statement orderings. If this representation is not respected, an initial preprocessing may be necessary.

The recovery algorithm implemented in Chlore can be decomposed into two components: *$\alpha$-recovery* and *$\beta$-recovery*.

*IMPACT 2024, January 17, 2024, Munich, Germany*
2024.

| Name | Type | Description |
|---|---|---|
| *Reorder*($\vec{\beta}, \vec{p}$) | $\beta$-primitive | Reorder inner-loops or statements directly beneath the given outer-loop. |
| *Split*($\vec{\beta}$) | $\beta$-primitive | Split outer-loop just before given inner-loop or statement. |
| *Fuse*($\vec{\beta}$) | $\beta$-primitive | Fuse given loop with the next one on the same depth. |
| *Embed*($\vec{L}$)<br>*Unembed*($\vec{L}$) | $\alpha$-primitive | Embed given statements beneath an innermost one-iteration extra loop. Unembed removes the added innermost extra loop. |
| *Reverse*($\vec{L}, d$) | $\alpha$-primitive | Reverse given output dimension for given statements. |
| *Grain*($\vec{L}, d, c$)<br>*Densify*($\vec{L}, d, c$) | $\alpha$-primitive | Add some pad between consecutive iterations in given output dimension for given statements. Densify removes some/all of the pad. |
| *Shift*($\vec{L}, d, c, \vec{C}$) | $\alpha$-primitive | Shift given output dimension by some (parametric) coefficient(s) for given statements. |
| *Interchange*($\vec{L}, d_1, d_2$) | $\alpha$-primitive | Interchange two given output dimensions for given statements. |
| *Skew*($\vec{L}, d_1, d_2, c$) | $\alpha$-primitive | Skew first output dimension by a coefficient of the second output dimension. |
| *Reshape*($\vec{L}, d, d_{input}, c$) | $\alpha$-primitive | Skew given output dimension by a coefficient of the given input dimension. |

**Table 1.** List of supported primitives by the recovery tool

## 3 Transformation primitives

Table 1 shows the list of atomic transformations supported by our tool along with expected parameters and short summaries using the following notations:

- $\vec{L}$ : A list of schedule IDs, an injection in $[0 \ldots N-1]$ where $N$ is the number of statements.
- $\vec{\beta}$ : A beta-vector targeting an entity (loop or statement).
- $\vec{p}$ : A permutation vector of size equal to the number of entities directly beneath the loop that is the target of $\vec{\beta}$.
- $\vec{C}$ : A list of parametric shift coefficients of size equal to the number of parametric bounds relevant to the schedules, which are the target of $\vec{L}$.
- $d, d_1, d_2$ : An output dimension
- $d_{input}$ : An input dimension
- $c$ : A scalar coefficient

All primitives are invertible. The $\alpha$-primitives are versatile and not tied to a specific scheduling representation. Individual statements or groups of statements can be targeted, no matter their inner loop placement. In contrast, the $\beta$-primitives, *Reorder*, *Fuse*, and *Split*, are inherently loop-centric. In this context, the 2d+1 format is the most practical for loop transformations. Hence, it is the format of choice for the recovery algorithm.

We can note that our list of transformation primitives does not include *Stripmine* or *Tile* transformations. Our focus in this paper is to explain polyhedral scheduler transformations. We omit these transformations as current polyhedral scheduling algorithms would not propose such transformations.

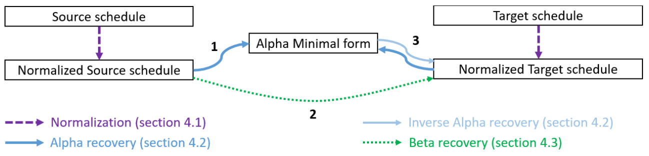
Source schedule
Normalized Source schedule
1
Alpha Minimal form
3
Target schedule
Normalized Target schedule
2
Normalization (section 4.1)
Alpha recovery (section 4.2)
Inverse Alpha recovery (section 4.2)
Beta recovery (section 4.3)

**Figure 1.** Recovery Algorithm

## 4 Recovery Algorithm

The recovery algorithm requires a *source* and a *target* schedule. Both schedules are assumed to be defined over the same space (i.e. number of statements, identical domains, etc.). Figure 1 illustrates the main steps of the recovery algorithm.

Before the recovery algorithm, both input schedules are normalized. This *normalization* preprocessing is described in Section 4.1. The first step of the recovery algorithm is $\alpha$*-recovery* and is explained in Section 4.2. The output of the $\alpha$*-recovery* is then inversed for the *target* schedule. The second step, $\beta$*-recovery* is detailed in Section 4.3.

The output list of primitives is constructed as the concatenation of three lists: the $\alpha$*-recovery* list for the source schedule, the $\beta$*-recovery* list, and the inverted $\alpha$*-recovery* list for the target schedule.

### 4.1 Normalization step

The *normalization* step guarantees a normalized 2d+1 format [11] for the polyhedral representation. Indeed, polyhedral scheduling algorithms may not always output schedules in 2d+1 format.

This step is composed of three sub-steps:

$\beta$***-collapsing*** fuse consecutive $\beta$-dimensions into a single $\beta$-dimension

***2d+1 format*** insert full-zero $\beta$-dimensions between consecutive $\alpha$-dimensions

$\beta$***-normalization*** normalize the $\beta$-dimensions into minimal $\beta$ vector for each statement.

Figure 2 shows an example of the *normalization*. This step does not introduce, modify or drop primitives to recover since it simply modifies the representation while preserving the scheduling semantics.

### 4.2 $\alpha$-recovery step

The $\alpha$*-recovery* strategy is to transform the *source* and *target* schedules into a common representation on the $\alpha$-dimensions. This common representation is called *minimal form* [12, Section 5.3.4]. It exhibits the following properties: (1) it contains exactly one polyhedral relation; (2) the scheduling relation union is defined exclusively by identity equalities between the input and the output dimensions in their respective orders; (3) $\beta$-dimensions are removed.

To achieve this *minimal form*, the following $\alpha$*-primitives* are considered: *Unembed*, *Densify*, *Reverse*, *Shift*, *Skew*, *Interchange* and *Reshape*. The $\alpha$*-recovery* algorithm will iterate

$$
\begin{array}{l}
\phi_{S0}: \begin{bmatrix} 5 & i & j & 1 & 0 & 0 & k \end{bmatrix} \\
\phi_{S1}: \begin{bmatrix} 5 & i & j & 1 & 1 & 0 & k \end{bmatrix} \\
\phi_{S2}: \begin{bmatrix} 5 & -i & j & 1 & 1 & 0 & k \end{bmatrix} \\
\phi_{S3}: \begin{bmatrix} 5 & i & j & 0 & 0 & 0 & k \end{bmatrix} \\
\phi_{S4}: \begin{bmatrix} 5 & i & 0 & 0 & 0 & 0 & k \end{bmatrix}
\end{array}
\quad
\begin{array}{l}
\begin{bmatrix} 5 & i & j & 1 & k \end{bmatrix} \\
\begin{bmatrix} 5 & i & j & 2 & k \end{bmatrix} \\
\begin{bmatrix} 5 & -i & j & 2 & k \end{bmatrix} \\
\begin{bmatrix} 5 & i & j & 0 & k \end{bmatrix} \\
\begin{bmatrix} 5 & i & 0 & 0 & k \end{bmatrix}
\end{array}
\quad
\begin{array}{l}
\begin{bmatrix} 5 & i & 0 & j & 1 & k & 0 \end{bmatrix} \\
\begin{bmatrix} 5 & i & 0 & j & 2 & k & 0 \end{bmatrix} \\
\begin{bmatrix} 5 & -i & 0 & j & 2 & k & 0 \end{bmatrix} \\
\begin{bmatrix} 5 & i & 0 & j & 0 & k & 0 \end{bmatrix} \\
\begin{bmatrix} 5 & i & 0 & 0 & 0 & k & 0 \end{bmatrix}
\end{array}
\quad
\begin{array}{l}
\begin{bmatrix} 0 & i & 0 & j & 1 & k & 0 \end{bmatrix} \\
\begin{bmatrix} 0 & i & 0 & j & 2 & k & 0 \end{bmatrix} \\
\begin{bmatrix} 0 & -i & 0 & j & 2 & k & 1 \end{bmatrix} \\
\begin{bmatrix} 0 & i & 0 & j & 0 & k & 0 \end{bmatrix} \\
\begin{bmatrix} 0 & i & 0 & 0 & 0 & k & 1 \end{bmatrix}
\end{array}
$$

**(a)** Initial **(b)** $\beta$-collapsing **(c)** $2d + 1$ format **(d)** $\beta$-normalization

**Figure 2.** *Normalization* step example on a schedule for 5 statements $S0 \dots S4$

over these primitives and over statements. Primitives are applied to the schedule if their application helps progressing towards the *minimal form* until it is reached.

The *α-recovery* algorithm may either first iterate over statements or over primitives. Option *alpha-priority* controls this behaviour:

**schedule** prioritize statement-by-statement recovery. The process iterates focuses on a single statement and modifies only on scheduling function.

**primitive** prioritize the recovery of a given primitive over all statements before another primitive is considered. It is applied if it useful for at least one statement.

The *schedule* option is more intuitive and groups transformations by statements. This option is equivalent to Chlore's behaviour. The *primitive* option has the advantage of grouping similar transformations together.

### 4.3 $\beta$-recovery step

The *β-recovery* focus on the $\beta$ dimensions of the source schedule to transform it to the target schedule. The *β-recovery* repeatedly considers the following *β-primitives*: *Reorder*, *Split* and *Fuse*.

Multiple strategies for *β-recovery* may be selected using option *beta-strategy*. The first two strategies, *maxfuse* and *maxsplit* attempt to find a common form, as in *α-recovery*, considering a maximum fused form or a maximum split form. Strategy *isolate* attempts to directly find a path from the source schedule to the target schedule since *β-dimensions* intuitively express an explicit order. None of the presented strategies is optimal for all the source and target schedules tried in our results section.

The $\beta$-recovery can be configured to attempt correcting a block of statements at once instead of one by one. Option *beta-targeting* can be set to *statement*, *segment* or *group*. For all variations, the first element of the said target, the *anchor* remains unchanged but the sizes of the blocks may vary.

The recovery iterates from outermost to innermost dimensions. *Anchors* are chosen by finding mismatches in $\beta$ values at the current depth between the source and target schedules. If multiple *anchors* candidates are found, the *anchor* is selected as follows: (1) it has not been correctly recovered in the source schedule and (2) its $\beta$-vector in the target schedule is the lexicographic minimum (among the *anchors* that satisfy the first condition). Thanks to this choice, correctly placed statements are never impacted by subsequently found $\beta$-primitives.

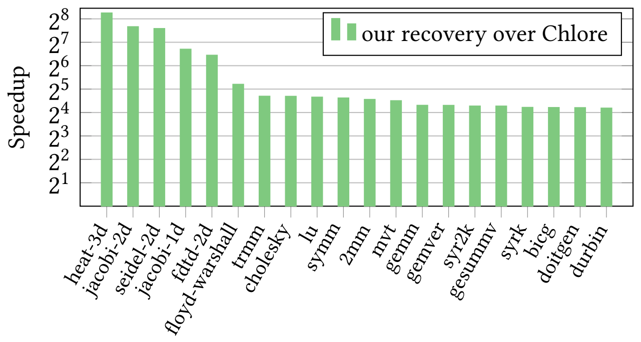

**Figure 3.** Speedup to recover transformations in comparison with Chlore

Option *block-targeting* influences how blocks expand. For *segment* targeting, they will expand as long as the next entity (loop or statement at the same depth as the *anchor*) is a statement and is also mismatched. For *group* targeting, blocks will expand as long as the next entity is also mismatched. The bigger the blocks corrected in one iteration, the better, which is why the default *group* option usually provides the simplest and fastest results.

## 5 Results

We implement our approach into PolyTOPS [6]. We experiment on the Polybench [9] and compare Polytops result with Chlore. Results show that, with our sub-set of transformation primitive defined in Section 3 and the new recovery algorithm defined in Section 4, we can (1) greatly reduce the recovery time and (2) reduce the number of transformation primitives to apply.

Figure 3 shows the speed up comparing our recovery algorithm to Chlore. Our algorithm is at least x18 and up to x300 faster than Chlore recovery time. These experiments were done on an AMD EPYC 7452, 32 cores (2 threads per core), 2 sockets, 256 MiB of L3 cache. The compiler is gcc-11.3.

Table 2 shows the number of transformations found for some Polybench cases by, respectively, Chlore and PolyTOPS. Test cases *atax*, *deriche*, *nussinov*, *adi* are not present in the

| Name | Chlore | PolyTOPS |
|---|---|---|
| correlation | NA | 45 |
| covariance | NA | 21 |
| 2mm | 16 | 7 |
| 3mm | 22 | 13 |
| bicg | 14 | 10 |
| cholesky | 13 | 9 |
| doitgen | 1 | 5 |
| durbin | 40 | 28 |
| gemm | 3 | 3 |
| gemver | 7 | 6 |
| gesummv | 7 | 5 |
| gramschmidt* | 29 | 13 |
| lu | 2 | 4 |
| mvt | 2 | 2 |
| symm | 3 | 5 |
| syr2k | 3 | 3 |
| syrk | 3 | 3 |
| trisolv | 9 | 5 |
| trmm | 3 | 2 |
| floyd-warshall | 0 | 0 |
| fdtd-2d | 41 | 15 |
| heat-3d | 38 | 12 |
| jacobi-1d | 16 | 4 |
| jacobi-2d | 26 | 8 |
| seidel-2d | 12 | 3 |

**Table 2.** Number of primitives recovered with Chlore and our approach in PolyTOPS

array because no new schedule is found. For *correlation* and *covariance*, Chlore recovery falls into an infinite loop. For *gramschmidt**, Chlore and PolyTOPS do not use exactly the same target schedule, but an equivalent one. Over the 25 cases, 11 cases have more than 10 transformations for Chlore, when only 6 cases are over 10 transformations for PolyTOPS. Moreover, except *doitgen* and *lu*, our solution always find the same number of transformation or much more less than Chlore solution. We can highlight all the stencil cases, *fdtd-2d*, *heat-3d*, *jacobi-1d*, *jacobi-2d* and *seidel-2d*, where PolyTOPS find at least 3 times less transformations to apply than Chlore.

## 6 Future Work

We focused on understanding the output of a polyhedral scheduler and hence omitted stripmining and tiling. Future work may extend the scope to arbitrary polyhedral schedules.

Future work may build upon this approach to make it possible to use polyhedral optimization in a wider variety of scenarios. In particular, translating automatic polyhedral schedules into sequences of transformations may ease the use of frameworks such as TVM [5] or Halide [10] where experts need to provide manual schedules. This could also increase inter-operability with existing work such as Clay [2], CHiLL [4], URUK [7], Tiramisu [1], or even MLIR [8].

## 7 Conclusion

In this paper, we present new techniques to recover transformation primitives from the output of a polyhedral scheduler. We show that an efficient recovery algorithm can be designed with a limited number of transformation primitives. There is no unique way to recover the transformation primitives that correspond to the output of a polyhedral scheduler and corresponding decisions can greatly influence the length of recovered sequences. Indeed, our experiments on PolyBench show that a good set of options can result in very short transformation primitive sequences for simple cases. Recovering transformation primitives can help non-experts understand and possibly tailor polyhedral scheduling. It may be useful in future work to leverage existing manual-scheduling tools such as TVM or Halide.

## References

[1] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

[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*. 128–138. https://doi.org/10.1145/2854038.2854048

[3] Cédric Bastoul. 2011. *Openscop: A specification and a library for data exchange in polyhedral compilation tools.* Technical Report. Université Paris-Sud. 22 pages.

[4] Chun Chen, Jacqueline Chame, and Mary W. Hall. 2007. CHiLL : A Framework for Composing High-Level Loop Transformations.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[6] Gianpietro Consolaro, Zhen Zhang, Harenome Razanajato, Nelson Lossing, Nassim Tchoulak, Adilla Susungi, Artur Cesar Araujo Alves, Renwei Zhang, Denis Barthou, Corinne Ancourt, and Cedric Bastoul. 2024. PolyTOPS: Reconfigurable and Flexible Polyhedral Scheduler. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[7] 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 (2006), 261–317.

[8] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[9] Louis-Noël Pouchet et al. 2010. Polybench: The polyhedral benchmark suite. https://sourceforge.net/projects/polybench

[10] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[11] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule Trees. In *4th International Workshop on Polyhedral Compilation Techniques (IMPACT 14)*. Vienna, Austria.

[12] Oleksandr Zinenko. 2016. *Interactive Program Restructuring*. Ph. D. Dissertation. Université Paris Saclay (COmUE). https://theses.hal.science/tel-01414770