Verified Validation for Affine Scheduling in Polyhedral Compilation

Xuyang Li, Hongjin Liang, and Xinyu Feng(✉)

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, Jiangsu, China
xuyang.li@smail.nju.edu.cn, {hongjin,xyfeng}@nju.edu.cn

Abstract. Structural nested loops can be abstracted into polyhedral models, based on which one can perform aggressive loop optimizations; however, the optimizations are often heuristic and complex, and therefore error-prone. Meanwhile, verified compilers, though rigorously correct, still miss powerful optimizing transformations and therefore produce less efficient code than industrial ones. To bridge this gap, this work provides a general verified validation framework based on Bernstein's conditions for affine scheduling, the core component of polyhedral optimization techniques. It is parameterized over the concrete definitions and proofs of the instruction language to be reusable. As shown in our evaluation, the framework is flexible enough to support both existing verified compilers like CompCert and existing polyhedral compilers like Pluto. The result is fully mechanized in the Coq proof assistant.

1 Introduction

Nested loops are heavily used in scientific computing, operating over multidimensional arrays. People have tried to analyze and transform such code fragments, seeking better memory locality or parallelization. Such efforts lead to compilation and optimization based on *polyhedral models* [14]. Common industrial compilers have integrated polyhedral optimizers to empower generated code, like LLVM [16] and GCC [25]. Some domain-specific areas also resort to polyhedral optimizations, like image processing [27], and tensor compilation in AI systems [6,34].

Though powerful, polyhedral optimizations are error-prone just like other loop-specific optimizations [21], which is partly due to its complex and heuristic algorithms (e.g., the Pluto algorithm [4]) based on integer programming. We aim to equip the powerful polyhedral techniques with formal guarantees to achieve verified and optimizing compilation.

This work reports the *verified validation* for the core component of the polyhedral compilation. Verified validation is a kind of compiler verification technique. The validation algorithm takes the source and target programs as its inputs and checks that the target always behaves the same as the source. The validation algorithm itself should be verified for a more rigorous correctness guarantee. Several passes of CompCert are verified this way, including the parser [17],

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
W.-N. Chin and Z. Xu (Eds.): TASE 2024, LNCS 14777, pp. 287–305, 2024.
https://doi.org/10.1007/978-3-031-64626-3_17

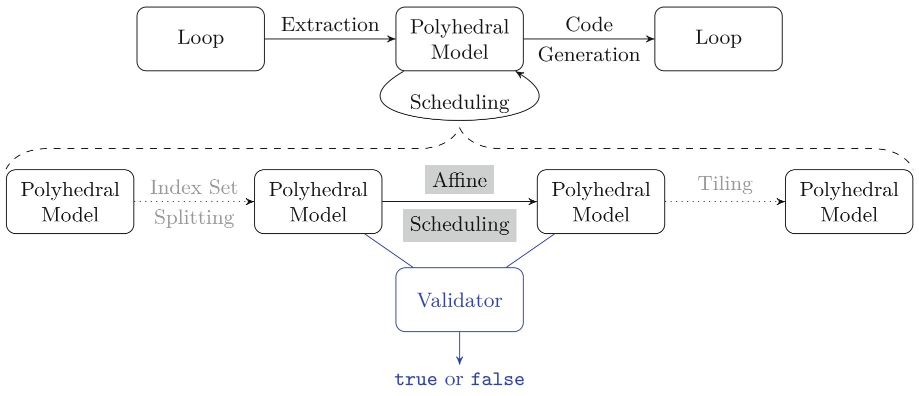

Fig. 1. The pipeline of polyhedral compilation. The blue part is our focus. (Color figure online)

register allocation [28], loop-invariant code motion [22], superblock scheduling [29], lazy code motion and strength reduction [15].

We focus on the validation for the *affine scheduling* pass in polyhedral compilation. As Fig. 1 shows, polyhedral compilation comprises three passes, namely *extraction*, *scheduling* and *code generation*. Extraction and code generation are responsible for the conversions between the structural nested loops and the mathematical polyhedral model. The scheduling pass does semantic-preserving transformation of polyhedral models. It can be further decomposed into several steps. Affine scheduling [11,12] is the main step. It reorders iterations of the loop for better locality and parallelization. Correctness of the reordering is known as *Bernstein's conditions* [13], which requires the preservation of write-after-write (WAW), write-after-read (WAR) and read-after-write (RAW) dependencies.

This work develops a general verified validation framework for affine scheduling in polyhedral compilation. It makes the following new contributions:

- We implement and verify a general validation framework in the Coq proof assistant, partly reusing previous mechanizations in Verified Polyhedron Library (VPL) [5] and the verified polyhedral code generator PolyGen [8]. Its algorithm checks Bernstein's conditions and is apt for affine scheduling. Further, as nested loops prevail in different programming languages, the framework is parameterized over instruction-level languages for reusability.
- We instantiate the framework with a variant of CompCert's instruction language, called CInstr, to show the practicality of the framework and the possibility towards an extension of CompCert with polyhedral optimization to complement CompCert's current loop optimizations.
- We evaluate the validation algorithm's *efficiency* and *completeness* with the Pluto compiler and its test suite. Shown in the result, the implementation is able to prove the equivalence between the polyhedral models before and after Pluto's affine scheduling of all the 62 test cases with reasonable overhead.

```
for (j1 = 1; j1 <= M; j1++) {
    for (j2 = j1; j2 <= M; j2++) {
        for (i = 1; i <= N; i++) {
            I0: symmat[j1][j2] += data[i][j1] * data[i][j2];
        }
        I1: symmat[j2][j1] = symmat[j1][j2];
}}
```

Fig. 2. Sample code π_{cov} for covariance matrix calculation.

In the following parts of the paper, we give an overview of polyhedral compilation in Sect. 2, and present the basic technical settings for our polyhedral model in Sect. 3. In Sect. 4, we outline the validation algorithm for affine transformations together with its proofs. In Sect. 5, we show how our framework can be applied to existing verified compiler CompCert and the polyhedral compiler Pluto. Finally, we discuss related work in Sect. 6, and conclude in Sect. 7. The complete artifact in Coq is available at https://github.com/verif-scop/PolCert.

2 Background

In this section, we give an overview of polyhedral models and show the key ideas to validate model transformations *w.r.t.* Bernstein's conditions.

2.1 Overview of Polyhedral Models

The polyhedral model is an expressive representation of a subclass of *nested loops* described by the following grammar:

$$s ::= \texttt{for (}i\texttt{ = }\varepsilon_1\texttt{; }i\texttt{ <= }\varepsilon_2\texttt{; }i\texttt{++) \{ }s\texttt{ \}} \mid s\texttt{;}s \mid \texttt{if (}\varepsilon\texttt{ <= 0) \{ }s\texttt{ \}} \mid \texttt{I}$$

where the variable i is called *loop iterators*, and ε stands for *affine* expressions in the form of $a_1 * x_1 + \cdots + a_n * x_n + c$. Here we use $a_1, \ldots, a_n$ and c to represent integer literals, and $x_1, \ldots, x_n$ for surrounding loop iterators (like i) or parameters (normally symbolic constants representing the problem size). `I` stands for primitive instructions such as assignments, which may contain array access expressions `arr`$[\varepsilon_1] \ldots [\varepsilon_n]$ and never modify loop iterators.

Such code fragments can be effectively analyzed and transformed into polyhedral models, with the assumption that all variables are non-aliased. Figure 2 shows sample fragment π_{cov} of covariance matrix calculation from PolyBench [26]. It has instructions $\texttt{I}_0$ and $\texttt{I}_1$, arrays `symmat` and `data`, loop iterators `j1`, `j2` and `i`, and parameters `M` and `N`.

Since instructions can be executed multiple times in the iterations, we use *iteration vector* p to identify individual execution *instances* of each instruction, like $[\texttt{M}, \texttt{N}; \texttt{j1}, \texttt{j2}, \texttt{i}]$ for $\texttt{I}_0$ and $[\texttt{M}, \texttt{N}; \texttt{j1}, \texttt{j2}]$ for $\texttt{I}_1$. Concrete values of `j1`, `j2` and

$\mathtt{i}$ specify the specific iteration where the instruction is executed. We use $\mathtt{I}(p)$ to locate the instance of the instruction $\mathtt{I}$ with the iteration vector p.

In the polyhedral model, we use a *domain* $\mathcal{D}$ to represent the set of iteration vectors for each instruction. Back to our example, domains $\mathcal{D}_0, \mathcal{D}_1$ of $\mathtt{I}_0$ and $\mathtt{I}_1$ respectively are shown below. With the affine bound constraints of each iterator, we can see each $\mathcal{D}_i$ is essentially a polyhedron.

$$\begin{aligned} \mathcal{D}_0 &= \{[\mathtt{M}, \mathtt{N}; \mathtt{j1}, \mathtt{j2}, \mathtt{i}] \mid 1 \le \mathtt{j1} \le \mathtt{M} \wedge \mathtt{j1} \le \mathtt{j2} \le \mathtt{M} \wedge 1 \le \mathtt{i} \le \mathtt{N}\} \\ \mathcal{D}_1 &= \{[\mathtt{M}, \mathtt{N}; \mathtt{j1}, \mathtt{j2}] \mid 1 \le \mathtt{j1} \le \mathtt{M} \wedge \mathtt{j1} \le \mathtt{j2} \le \mathtt{M}\} \end{aligned}$$

The execution order of all instances is specified with an affine function θ, namely *schedule*, mapping iteration vectors to *timestamps*. A timestamp is also a vector that can be lexicographically ordered, with each dimension broadly corresponding to a layer of nested loop. For identity transformations that do not reorder instruction instances, the schedules are simply identity functions, as shown by the schedules θ_0 and θ_1 below for $\mathtt{I}_0$ and $\mathtt{I}_1$ respectively.

$$\begin{aligned} \theta_0([\mathtt{M}, \mathtt{N}; \mathtt{j1}, \mathtt{j2}, \mathtt{i}]) &= (\mathtt{j1}, \mathtt{j2}, \mathtt{i}) \\ \theta_1([\mathtt{M}, \mathtt{N}; \mathtt{j1}, \mathtt{j2}]) &= (\mathtt{j1}, \mathtt{j2}) \end{aligned}$$

Polyhedral models also need information about instructions' memory access patterns to reason about their execution. We use a pair of an identifier and a vector to represent the location of a memory access, where the identifier is the name of a multidimensional array, and the vector represents the array indices.

$$(\mathit{MemCell})\ c \ \in\ \mathsf{Id} \times \mathit{List}(\mathbb{Z})$$

For each instruction, we use two *access functions* to describe the write-set and read-set of each instance. Each access function maps an iteration vector p to the corresponding set of memory cells. For our sample, read and write access functions $\mathcal{W}_0, \mathcal{R}_0, \mathcal{W}_1$ and $\mathcal{R}_1$ for $\mathtt{I}_0$ and $\mathtt{I}_1$ are given below.

$$\begin{aligned} \mathcal{W}_0([\mathtt{M}, \mathtt{N}; \mathtt{j1}, \mathtt{j2}, \mathtt{i}]) &= \{(\mathtt{symmat}, [\mathtt{j1}, \mathtt{j2}])\} \\ \mathcal{R}_0([\mathtt{M}, \mathtt{N}; \mathtt{j1}, \mathtt{j2}, \mathtt{i}]) &= \{(\mathtt{symmat}, [\mathtt{j1}, \mathtt{j2}]), (\mathtt{data}, [\mathtt{i}, \mathtt{j1}]), (\mathtt{data}, [\mathtt{i}, \mathtt{j2}])\} \\ \mathcal{W}_1([\mathtt{M}, \mathtt{N}; \mathtt{j1}, \mathtt{j2}]) &= \{(\mathtt{symmat}, [\mathtt{j2}, \mathtt{j1}])\} \\ \mathcal{R}_1([\mathtt{M}, \mathtt{N}; \mathtt{j1}, \mathtt{j2}]) &= \{(\mathtt{symmat}, [\mathtt{j1}, \mathtt{j2}])\} \end{aligned}$$

Then the whole polyhedral model $\mathcal{P}_{cov}$ of π_{cov} is defined as a set of tuples, each tuple consisting of an instruction, the corresponding domain, schedule, and access functions.

$$\mathcal{P}_{cov} = \{(\mathtt{I}_0, \mathcal{D}_0, \theta_0, \mathcal{W}_0, \mathcal{R}_0), (\mathtt{I}_1, \mathcal{D}_1, \theta_1, \mathcal{W}_1, \mathcal{R}_1)\}$$

Normally, as in Fig. 1, after we calculate the polyhedral model $\mathcal{P}_{cov}$ of π_{cov}, a *scheduling* algorithm is applied to transform $\mathcal{P}_{cov}$ to a new polyhedral model $\mathcal{P}'_{cov}$. Here we focus on the key *affine scheduling* sub-pass, which tries to find new schedules θ'_0, θ'_1 for $\mathtt{I}_0, \mathtt{I}_1$.

$$\begin{aligned} \mathcal{P}'_{cov} = \mathit{Schedule}(\mathcal{P}_{cov}) &= \{(\mathtt{I}_0, \mathcal{D}_0, \theta'_0, \mathcal{W}_0, \mathcal{R}_0), (\mathtt{I}_1, \mathcal{D}_1, \theta'_1, \mathcal{W}_1, \mathcal{R}_1)\} \\ \textbf{where } \theta'_0([\mathtt{M}, \mathtt{N}; \mathtt{j1}, \mathtt{j2}, \mathtt{i}]) &= (\mathtt{0}, \mathtt{i}, \mathtt{j1}, \mathtt{j2}), \theta'_1([\mathtt{M}, \mathtt{N}; \mathtt{j1}, \mathtt{j2}]) = (\mathtt{1}, \mathtt{j1}, \mathtt{j2}) \end{aligned}$$

```
for (i = 1; i <= N; i++) {
    for (j1 = 1; j1 <= M; j1++) {
        for (j2 = j1; j2 <= M; j2++) {
            I0: symmat[j1][j2] += data[i][j1] * data[i][j2];
}}}
for (j1 = 1; j1 <= M; j1++) {
    for (j2 = j1; j2 <= M; j2++) {
        I1: symmat[j2][j1] = symmat[j1][j2];
}}
```

Fig. 3. Optimized code π'_{cov} for covariance matrix calculation.

This specific transformation applies a combination of loop fission (see the extra dimension with values `0,1`) and loop interchange (see the fronted iterator `i`) to the old schedule. It will be intuitive after the final code generation, recovering the structural loop structure, say π'_{cov} in Fig. 3.

The above compilation pipeline from π_{cov} to π'_{cov} is producible by Pluto compiler [4]. It achieves about 4 times speed-up on our machine with `M=N=1500`.

2.2 Validation via Bernstein's Conditions

A compilation validation algorithm takes in source and target programs, and reports `true` if it can establish refinement relation, `false` otherwise. To design a validation algorithm for a specific compilation pass, a decidable correctness criterion of that pass is needed. Intuitively, affine scheduling is a *reordering* transformation of instances. Such transformation's correctness is well-studied as *Bernstein's conditions* [3,13].

Bernstein's conditions describe when it is correct to reorder two operations. Let $\mathsf{W}(u)$ and $\mathsf{R}(u)$ be the write-set and read-set of an operation u respectively. Bernstein's conditions require $\mathsf{W}(u)\cap\mathsf{W}(v) = \mathsf{W}(u)\cap\mathsf{R}(v) = \mathsf{R}(u)\cap\mathsf{W}(v) = \emptyset$. That is, Bernstein's conditions require the preservation of write-after-write (WAW), write-after-read (WAR) and read-after-write (RAW) dependencies.

In summary, to validate the polyhedral affine scheduling pass, we need to implement a validation algorithm based on Bernstein's conditions. Further, to achieve formal correctness guarantees, the algorithm should be itself formally verified by showing its sufficiency to guarantee an affine scheduling's correctness, that is the refinement relation between the target and the source.

3 Language Setting

In this section, we present the formalization of the polyhedral model, and the instruction language interface for our parameterized framework.

3.1 Polyhedral Models

We give the formal definition of polyhedral models *PolyProg* in Fig. 4. We use $\mathcal{P} = (\mathcal{I}, \mathcal{V}, \Gamma)$ to denote a polyhedral program, which contains a polyhedral instruction list $\mathcal{I}$, a parameter list $\mathcal{V}$ and a typing context Γ (which is explained in Sect. 3.2). For each list, say $\mathcal{I}$, we use $|\mathcal{I}|$ to represent its length, and $\mathcal{I}[n]$ for its n-th element (from zero).

PolyProg is parameterized over the instruction language (see Sect. 3.2), borrowing its definitions such as identifiers (of type Id), instructions (denoted by $\mathtt{I}$), states (denoted by σ), semantics of instruction instances (in the form of $p \vdash \mathtt{I}, \sigma \xrightarrow{\Delta_r, \Delta_w} \sigma'$) . We refer readers to Sect. 3.2 for their details.

$$\begin{array}{rcll rcll}
\textit{(Vec)} & p, t, v, x, \mathcal{E} & \in & \textit{List}(\mathbb{Z}) & \textit{(MemCells)} & \Delta_w, \Delta_r & \in & \textit{Set(MemCell)} \\
\textit{(Poly)} & \mathcal{D} & \in & \textit{Set(Vec)} & \textit{(AcsFunc)} & \mathcal{R}, \mathcal{W} & \in & \textit{Vec} \rightarrow \textit{MemCells} \\
\textit{(AffFunc)} & \theta & \in & \textit{Vec} \rightarrow \textit{Vec} & \textit{(PolyInstr)} & I & ::= & (\mathtt{I}, \mathcal{D}, \theta, \mathcal{R}, \mathcal{W}) \\
\textit{(Params)} & \mathcal{V} & \in & \textit{List}(\mathsf{Id}) & \textit{(PolyInstrs)} & \mathcal{I} & \in & \textit{List(PolyInstr)} \\
\textit{(MemCell)} & c & \in & \mathsf{Id} \times \textit{Vec} & \textit{(PolyProg)} & \mathcal{P} & ::= & (\mathcal{I}, \mathcal{V}, \Gamma)
\end{array}$$

Fig. 4. Syntax of *PolyProg*.

$$\begin{array}{rcl}
\textit{(Index)} \; n & \in & \mathbb{N} \\
\textit{(Instance)} \; \iota & ::= & (\mathtt{I}, n, p, t) \\
\textit{(Instances)} \; \mathcal{L} & ::= & \textit{List(Instance)}
\end{array}$$

Fig. 5. Auxiliary definitions of *PolyProg*'s semantics.

A polyhedral instruction I is a tuple $(\mathtt{I}, \mathcal{D}, \theta, \mathcal{R}, \mathcal{W})$. The instruction $\mathtt{I}$ is the base instruction. The polyhedron $\mathcal{D}$ is the domain of the instruction; we call every vector $p \in \mathcal{D}$ the *iteration vector*. θ is the affine schedule which maps each p in $\mathcal{D}$ to a timestamp t (also a vector, as explained in Sect. 2.1). $\mathcal{R}, \mathcal{W}$ denotes the instruction's read access function and write access function, respectively. Note that we omit the linear algebra representations and implementations of the notions for presentation simplicity, and use their more intuitive mathematical notions instead (e.g., sets and functions).

The polyhedral program's semantics judgment is of the form $\vdash \mathcal{P}, \sigma \longrightarrow \sigma'$ in Fig. 6, saying that, starting from an initial state σ, the polyhedral program $\mathcal{P}$ executes and terminates at the final state σ'. The semantics first flattens the polyhedral instruction list $\mathcal{I}$ into instances $\mathcal{L}$ (defined in Fig. 5) and sorts it (rule Init), then iteratively executes the instance with the least timestamp (rule Progress), until there are no remaining instances (rule Done).

Below we give explanations of the semantic components occurring in Fig. 6.

- An instance ι is a four-tuple $(\mathtt{I}, n, p, t)$ (see Fig. 5), recording its instruction $\mathtt{I}$, its n in list $\mathcal{I}$, its iteration vector p and its timestamp t. We order instances with their timestamps lexicographically, with notations like $\prec, \preccurlyeq, \succcurlyeq$.
- Propositions Compat, $\mathsf{Retrieve}$ and $\mathsf{NonAlias}$ together describe valid initial states and are defined within the instruction language (see Sect. 3.2).

$$\frac{\begin{array}{c}\mathcal{P} = (\mathcal{I}, \mathcal{V}, \Gamma) \\ \mathsf{Compat}(\Gamma, \sigma) \quad \mathsf{Retrieve}(\mathcal{V}, \mathcal{E}, \sigma) \quad \mathsf{NonAlias}(\sigma) \\ \mathsf{flatten}(\mathcal{I}, \mathcal{E}, \mathcal{L}) \quad \mathsf{Permut}(\mathcal{L}, \mathcal{L}') \quad \mathsf{Sorted}(\mathcal{L}', \preccurlyeq) \\ \vdash \mathcal{L}', \sigma \longrightarrow \sigma'\end{array}}{\vdash \mathcal{P}, \sigma \longrightarrow \sigma'} \text{(INIT)}$$

$$\frac{\begin{array}{c}\mathcal{L} = \iota :: \mathcal{L}' \quad \iota = (\mathtt{I}, n, p, t) \\ p \vdash \mathtt{I}, \sigma \xrightarrow{\Delta_r, \Delta_w} \sigma' \quad \vdash \mathcal{L}', \sigma' \longrightarrow \sigma''\end{array}}{\vdash \mathcal{L}, \sigma \longrightarrow \sigma''} \text{(PROGRESS)} \qquad \frac{}{\vdash \mathsf{nil}, \sigma \longrightarrow \sigma} \text{(DONE)}$$

Fig. 6. Semantics of *PolyProg*.

- The proposition $\mathsf{flatten}(\mathcal{I}, \mathcal{E}, \mathcal{L})$ defines how a list of polyhedral instructions $\mathcal{I}$ is expanded into a set of instances $\mathcal{L}$, with the parameters $\mathcal{E}$ (e.g., $[\mathtt{M}, \mathtt{N}]$ in the example in Sect. 2.1). It holds if and only if:
  1. Elements in $\mathcal{L}$ have unique (n, p) pairs.
  2. For any instance $\iota = (\mathtt{I}, n, p, t) \in \mathcal{L}$, we have $\mathcal{I}[n] = (\mathtt{I}, \mathcal{D}, \theta, \mathcal{R}, \mathcal{W})$, and
     (a) $p \in \mathcal{D}$ and p's prefix of length $|\mathcal{E}|$ is equal to $\mathcal{E}$,
     (b) $\theta(p) = t$.
- $\mathsf{Permut}(\mathcal{L}, \mathcal{L}')$ says $\mathcal{L}$ and $\mathcal{L}'$ are permutations of each other (they contain the same set of instances), and $\mathsf{Sorted}(\mathcal{L}, \preccurlyeq)$ says $\mathcal{L}$ is sorted by relation $\preccurlyeq$.

We define the refinement relation $\mathcal{P}_t \sqsubseteq \mathcal{P}_s$ between two polyhedral programs $\mathcal{P}_s$ and $\mathcal{P}_t$ in Definition 1, to which the validation's correctness refers. It says, starting from any valid state σ, if the target program $\mathcal{P}_t$ terminates at state σ', the source $\mathcal{P}_s$ also terminates at σ'.

Definition 1 (Refinement Relation between Polyhedral Programs).

$$\mathcal{P}_t \sqsubseteq \mathcal{P}_s \triangleq \forall \sigma, \sigma'. \ \vdash \mathcal{P}_t, \sigma \longrightarrow \sigma' \implies \vdash \mathcal{P}_s, \sigma \longrightarrow \sigma'.$$

3.2 Instruction Language Interface

The polyhedral compilation does high-level structural transformations, not relying on the concrete definition of the underlying instruction language. However, the underlying language must expose its memory access patterns and guarantee some properties. We modularize these essential definitions as the module type $\mathbb{I}$ (see Fig. 7). Our polyhedral model is parameterized over instances of the type $\mathbb{I}$.

An instruction language must define its identifier Id, its variable type T (normally multi-dimensional arrays), its base instruction's syntax I, and its state S. We use notations $\mathtt{I}$, τ and σ for concrete instruction, type, and state.

We use Γ to denote a typing context, which is a list of variable-type pairs that contains all live variables and their corresponding types for the code fragment.

$$\textit{(TypCtxt)}\ \Gamma \ \in \ \textit{List}(\mathsf{Id} \times \mathsf{T})$$

The semantics is defined in the form of $p \vdash \mathtt{I}, \sigma \xrightarrow{\Delta_r, \Delta_w} \sigma'$. It says that, the instance of the base instruction $\mathtt{I}$ under the iteration vector p executes from the initial state σ and terminates at the final state σ', reading and writing memory cells Δ_r and Δ_w (see Fig. 4 for the definitions). The iteration vector p contains the exact values for parameters $\mathcal{V}$ and all the iterators. That is, $p[i-1]$ is the value of the i-th parameter (if $i \le |\mathcal{V}|$), and $p[|\mathcal{V}|+j-1]$ is the value of the j-depth iterator.

$$
\textsf{Module Type } \mathbb{I} \triangleq
$$
$$
\left\{\begin{array}{l}
\textsf{Id}, \textsf{T}, \textsf{I}, \textsf{S} : \mathit{Type} \\
\vdash\ : \textsf{I} \to \mathit{List}(\mathbb{Z}) \to \mathit{MemCells} \to \mathit{MemCells} \to \textsf{S} \to \textsf{S} \to \mathit{Prop} \\
\textsf{Compat} : \mathit{TypCtxt} \to \textsf{S} \to \mathit{Prop} \\
\textsf{Retrieve} : \mathit{List}(\textsf{Id}) \to \mathit{List}(\mathbb{Z}) \to \textsf{S} \to \mathit{Prop} \\
\textsf{NonAlias} : \textsf{S} \to \mathit{Prop} \\
\textsf{NonAliasPsrv} : \\
\quad \forall \mathtt{I}, \sigma, \sigma'.\ \textsf{NonAlias}(\sigma) \wedge p \vdash \mathtt{I}, \sigma \xrightarrow{\_,\_} \sigma' \implies \textsf{NonAlias}(\sigma'). \\
\textsf{Check} : \textsf{I} \to \mathit{AcsFuncs} \to \mathit{AcsFuncs} \to \mathit{Bool} \\
\textsf{Correct}(\textsf{Check}) : \\
\quad \forall \mathtt{I}, \mathcal{W}, \mathcal{R}.\ \textsf{Check}(\mathtt{I}, \mathcal{W}, \mathcal{R}) = \textsf{true} \\
\qquad\quad \implies (\forall \sigma, \sigma', p, \Delta_r, \Delta_w.\ p \vdash \mathtt{I}, \sigma \xrightarrow{\Delta_r, \Delta_w} \sigma' \implies \Delta_r \subseteq \mathcal{R}(p) \wedge \Delta_w \subseteq \mathcal{W}(p)). \\
\textsf{BCPermut} : \\
\quad \forall \mathtt{I}_1, \mathtt{I}_2, p_1, p_2, \sigma, \sigma', \sigma'', \Delta_r, \Delta_w, \Delta'_r, \Delta'_w. \\
\quad (p_1 \vdash \mathtt{I}_1, \sigma \xrightarrow{\Delta_r, \Delta_w} \sigma' \wedge p_2 \vdash \mathtt{I}_2, \sigma' \xrightarrow{\Delta'_r, \Delta'_w} \sigma'' \\
\qquad\qquad \wedge \Delta_r \cap \Delta'_w = \emptyset \wedge \Delta_w \cap \Delta'_r = \emptyset \wedge \Delta_w \cap \Delta'_w = \emptyset) \wedge \textsf{NonAlias}(\sigma) \\
\qquad\quad \implies \exists \sigma^*.\ p_2 \vdash \mathtt{I}_2, \sigma \xrightarrow{\Delta'_r, \Delta'_w} \sigma^* \wedge p_1 \vdash \mathtt{I}_1, \sigma^* \xrightarrow{\Delta_r, \Delta_w} \sigma''.
\end{array}\right.
$$

Fig. 7. Definition of instruction language module $\mathbb{I}$.

The proposition Compat describes the validity of the initial states *w.r.t.* the program's typing context, e.g. correct allocations. The proposition Retrieve collects the parameters $\mathcal{V}$'s values $\mathcal{E}$ from the initial state σ, which will be the constant prefix of every iteration vector. The proposition NonAlias describes non-aliasing states; the preservation of the state's non-aliasing property *w.r.t.* instruction's semantics is captured by lemma NonAliasPsrv.

Polyhedral models summarize access patterns with read and write access functions. A procedure Check should be defined to validate the consistency between every instruction's semantics and its access functions. It should satisfy the correctness property Correct(Check), which says: if Check returns `true` for instruction $\mathtt{I}$ and its read and write access functions $\mathcal{R}$ and $\mathcal{W}$, then the read and write memory footprints Δ_r, Δ_w of $\mathtt{I}$'s any execution under iteration vector p are over-approximated by $\mathcal{R}(p)$ and $\mathcal{W}(p)$, respectively.

Finally, the instruction language should respect Bernstein's conditions, specified with lemma BCPermut. It says that, from a non-aliasing state, if two instructions $\mathtt{I}_1$ and $\mathtt{I}_2$ are executed in sequence under some iteration vectors and their footprints satisfy Bernstein's conditions, then the permutation of their executions does not affect the final state.

4 Validation Algorithm and Its Proofs

In this section, we outline our validation algorithm and its correctness proofs.

4.1 Validation Algorithm

Preprocessing. We first simplify the inputs with the function Compose, as defined below. The validator assumes the two input polyhedral programs only differ in each instruction's schedule functions and tries to compose them into an extended program $\hat{\mathcal{P}}$. If the two inputs are not composable, the validator returns `false` directly. Note that the function seq transforms a list of optional values into an optional list, returning `Some` if all elements are not `None`, and `None` otherwise.

$$\begin{aligned} \textit{(ExtPolyInstr)}\ \hat{I} &::= (\mathtt{I}, \mathcal{D}, \theta_s, \theta_t, \mathcal{R}, \mathcal{W}) \\ \textit{(ExtPolyInstrs)}\ \hat{\mathcal{I}} &::= \textit{List(ExtPolyInstr)} \\ \textit{(ExtPolyProg)}\ \hat{\mathcal{P}} &::= (\hat{\mathcal{I}}, \mathcal{V}, \Gamma) \end{aligned}$$

$$\begin{aligned} &\mathsf{Compose}((\mathcal{I}_s, \mathcal{V}_s, \Gamma_s), (\mathcal{I}_t, \mathcal{V}_t, \Gamma_t))::= \\ &\begin{cases} \mathtt{Some}\ (\hat{\mathcal{I}}, \mathcal{V}_s, \Gamma_s) & \text{if } \mathsf{seq}\ (\mathsf{map}\ \mathsf{Compose}'\ (\mathsf{zip}\ \mathcal{I}_s\ \mathcal{I}_t)) = \mathtt{Some}\ \hat{\mathcal{I}} \wedge \mathcal{V}_s = \mathcal{V}_t \wedge \Gamma_s = \Gamma_t \\ \mathtt{None} & \text{otherwise} \end{cases} \\ &\mathsf{Compose}'((\mathtt{I}_s, \mathcal{D}_s, \theta_s, \mathcal{R}_s, \mathcal{W}_s), (\mathtt{I}_t, \mathcal{D}_t, \theta_t, \mathcal{R}_t, \mathcal{W}_t))::= \\ &\begin{cases} \mathtt{Some}\ (\mathtt{I}_s, \mathcal{D}_s, \theta_s, \theta_t, \mathcal{R}_s, \mathcal{W}_s) & \text{if } \mathtt{I}_s = \mathtt{I}_t \wedge \mathcal{D}_s = \mathcal{D}_t \wedge \mathcal{R}_s = \mathcal{R}_t \wedge \mathcal{W}_s = \mathcal{W}_t \\ \mathtt{None} & \text{otherwise} \end{cases} \end{aligned}$$

The composability suggests instances of the two input polyhedral programs are of one-to-one correspondence. We also compose two corresponding instances into one extended instance $\hat{\imath}$, carrying both the old and new timestamps. It can be ordered either by its old timestamp using $\prec_s, \preccurlyeq_s, \succcurlyeq_s$ or its new timestamp using $\prec_t, \preccurlyeq_t, \succcurlyeq_t$. Similarly, lists of extended instances exist. We introduce operators $\langle\cdot\rangle_s$ and $\langle\cdot\rangle_t$ to split the extended definitions to the originals.

$$\begin{aligned} \textit{(ExtInstance)}\ \hat{\imath} &::= (\mathtt{I}, n, p, t_s, t_t) \\ \textit{(ExtInstances)}\ \hat{\mathcal{L}} &::= \textit{List}(\textit{ExtInstance}) \end{aligned}$$

Further, the proposition flatten is lifted, so that $\mathsf{flatten}(\hat{\mathcal{I}}, \mathcal{E}, \hat{\mathcal{L}})$ expands an extended polyhedral instruction list into an extended instance list. The extended instance list is just a compact representation for the non-extended, and we have $\langle\hat{\mathcal{L}}\rangle_s = \mathcal{L}_s$ and $\langle\hat{\mathcal{L}}\rangle_t = \mathcal{L}_t$ if $\mathsf{flatten}(\langle\hat{\mathcal{I}}\rangle_s, \mathcal{E}, \mathcal{L}_s)$ and $\mathsf{flatten}(\langle\hat{\mathcal{I}}\rangle_t, \mathcal{E}, \mathcal{L}_t)$.

Instruction-Level Validation. The function V_{instr} checks if Bernstein's conditions (*w.r.t.* access functions) are satisfied by every two permuted instances of instructions $\hat{I}$ and $\hat{I}'$, i.e., no violation of WAW, WAR or RAW dependencies. It invokes dependency preservation checkers *WAWPrsv*, *RAWPrsv*, *WARPrsv* on input instructions. Given that each program comprises multiple instructions, and potential dependencies could exist between any two of them, V_{instr} must be applied to all instruction pairs.

$$V_{instr}(\hat{I}, \hat{I}', m) = \mathit{WAWPrsv}(\hat{I}, \hat{I}', m) \wedge \mathit{RAWPrsv}(\hat{I}, \hat{I}', m) \wedge \mathit{WARPrsv}(\hat{I}, \hat{I}', m)$$

Taking *WAWPrsv* as an example, it constructs emptiness tests to find permuted instances accessing overlapped memory cells for writes. Its additional argument m will be the length $|\mathcal{V}|$ of parameters, used to enforce the same value assumption of parameters in every iteration vector. The notation $p[:m]$ slices the iteration vector p to its first m elements, i.e., the values of parameters. *RAWPrsv* and *WARPrsv* are defined in similar ways.

$$\begin{aligned}
&\mathit{WAWPrsv}(\hat{I}, \hat{I}', m) = \\
&\quad \textbf{let } \hat{I} \ = (\mathtt{I}, \mathcal{D}, \theta_s, \theta_t, \mathcal{R}, \mathcal{W}) \textbf{ in} \\
&\quad \textbf{let } \hat{I}' = (\mathtt{I}', \mathcal{D}', \theta_s', \theta_t', \mathcal{R}', \mathcal{W}') \textbf{ in} \\
&\qquad \begin{cases} \mathtt{true} & \text{if } \{(p, p') \mid p \in \mathcal{D} \wedge p' \in \mathcal{D}' \wedge p[:m] = p'[:m] \\ & \qquad \wedge\ \theta_s(p) \prec \theta_s(p') \wedge \theta_t(p) \succcurlyeq \theta_t(p') \wedge \mathcal{W}(p) \cap \mathcal{W}'(p') \neq \emptyset\} = \emptyset \\ \mathtt{false} & \text{otherwise} \end{cases}
\end{aligned}$$

Validation of Access Functions. However, the function V_{instr} directly uses access functions of each instruction for granted, not checking its consistency with the instruction's semantics. To achieve formal correctness guarantees, the user should provide a checking procedure Check and prove its correctness $\mathsf{Correct}(\mathsf{Check})$ for her language (see Sect. 3.2). Procedure Check is invoked on every instruction.

Only when the procedure Check reports `true` for every instruction, the results of V_{instr} are valid *w.r.t.* the language's semantics, and the premise of lemma $\mathsf{BCPermut}$, which describes Bernstein's condition with the semantics, is fulfillable.

Overall Validation Algorithm. So far, we have all the ingredients to define the overall validation algorithm *Validate*. It first composes the two input polyhedral programs into one and invokes V_{instr} for every two extended instructions. Finally, it invokes user-defined access function checker Check for every instruction. If all checks pass, the algorithm returns `true`; otherwise, it cannot establish the refinement relation and returns `false`.

Definition 2 (Definition of the Validation Algorithm).

$$\mathit{Validate}(\mathcal{P}_s, \mathcal{P}_t) = \begin{cases} \forall \hat{I}, \hat{I}' \in \hat{\mathcal{I}}.\ V_{instr}(\hat{I}, \hat{I}', |\mathcal{V}|) = \mathtt{true} \\ \quad \wedge \forall \hat{I} \in \hat{\mathcal{I}}.\ \mathsf{Check}(\hat{I}.\mathtt{I}, \hat{I}.\mathcal{W}, \hat{I}.\mathcal{R}) = \mathtt{true} \\ \qquad\qquad\qquad \text{if } \mathsf{Compose}(\mathcal{P}_s, \mathcal{P}_t) = \mathtt{Some}\ (\hat{\mathcal{I}}, \mathcal{V}, \Gamma) \\ \mathtt{false} \qquad\quad \text{if } \mathsf{Compose}(\mathcal{P}_s, \mathcal{P}_t) = \mathtt{None} \end{cases}$$

4.2 Proof Goal and Its Sketch

The correctness of the validator *Validate*, which is our proof goal, is given in Theorem 1. It says that whenever it reports `true` on input programs $\mathcal{P}_s$ and $\mathcal{P}_t$, it guarantees the establishment of their refinement relation $\mathcal{P}_t \sqsubseteq \mathcal{P}_s$.

Theorem 1 (Correctness of the Validator).

$$\forall \mathcal{P}_s, \mathcal{P}_t.\ \mathit{Validate}(\mathcal{P}_s, \mathcal{P}_t) = \texttt{true} \implies \mathcal{P}_t \sqsubseteq \mathcal{P}_s.$$

Now we outline the proof of Theorem 1.

Proof. Unfold the goal. We have an extended instance list $\hat{\mathcal{L}}$ for the composed program $\hat{\mathcal{P}}$. Now for every possible permutation $\hat{\mathcal{L}}_t$ of $\hat{\mathcal{L}}$ where $\mathsf{Sorted}(\hat{\mathcal{L}}_t, \preccurlyeq_t)$, which determines a possible iteration $\langle \hat{\mathcal{L}}_t \rangle_t$ for the target program, we need to find a semantic equivalent instance list $\hat{\mathcal{L}}_s$ where $\mathsf{Sorted}(\hat{\mathcal{L}}_s, \preccurlyeq_s)$ that determines the iteration $\langle \hat{\mathcal{L}}_s \rangle_s$ for the source program.

$$\frac{\begin{array}{c}\hat{\mathcal{L}}_1 = \hat{\iota}_1 :: \hat{\iota}_2 :: \hat{\mathcal{L}}' \\ \hat{\mathcal{L}}_2 = \hat{\iota}_2 :: \hat{\iota}_1 :: \hat{\mathcal{L}}' \\ \hat{\iota}_1 \sim \hat{\iota}_2\end{array}}{\mathsf{StablePermut}'(\hat{\mathcal{L}}_1, \hat{\mathcal{L}}_2, \sim)}\ (\textsc{SPmtSwap}) \qquad \frac{\begin{array}{c}\hat{\mathcal{L}}_1 = \hat{\iota} :: \hat{\mathcal{L}}_1' \\ \hat{\mathcal{L}}_2 = \hat{\iota} :: \hat{\mathcal{L}}_2' \\ \mathsf{StablePermut}'(\hat{\mathcal{L}}_1', \hat{\mathcal{L}}_2', \sim)\end{array}}{\mathsf{StablePermut}'(\hat{\mathcal{L}}_1, \hat{\mathcal{L}}_2, \sim)}\ (\textsc{SPmtSkip})$$

$$\frac{\hat{\mathcal{L}}_1 = \hat{\mathcal{L}}_2 = \mathsf{nil}}{\mathsf{StablePermut}(\hat{\mathcal{L}}_1, \hat{\mathcal{L}}_2, \sim, 0)}\ (\textsc{SPmtNil}) \qquad \frac{\begin{array}{c}\mathsf{StablePermut}'(\hat{\mathcal{L}}_1, \hat{\mathcal{L}}_2, \sim) \\ \mathsf{StablePermut}(\hat{\mathcal{L}}_2, \hat{\mathcal{L}}_3, \sim, n)\end{array}}{\mathsf{StablePermut}(\hat{\mathcal{L}}_1, \hat{\mathcal{L}}_3, \sim, n+1)}\ (\textsc{SPmt})$$

Fig. 8. Definition of stable permutation.

First we define the relation $\sim$ to $\lambda \hat{\iota}, \hat{\iota}'.\ \hat{\iota} \succ_s \hat{\iota}' \wedge \hat{\iota} \preccurlyeq_t \hat{\iota}'$, which establishes if two extended instances are permuted.

Now we sort $\hat{\mathcal{L}}_t$ and let $\hat{\mathcal{L}}_s = \mathsf{sort}(\hat{\mathcal{L}}_t, \preccurlyeq_s)$. The sorting algorithm sort establishes $\mathsf{Sorted}(\hat{\mathcal{L}}_s, \preccurlyeq_s) \wedge \mathsf{Permut}(\hat{\mathcal{L}}_t, \hat{\mathcal{L}}_s)$ and the additional property:

$$\exists n.\ \mathsf{StablePermut}(\hat{\mathcal{L}}_t, \hat{\mathcal{L}}_s, \sim, n)$$

where $\mathsf{StablePermut}$ is defined in Fig. 8. $\mathsf{StablePermut}(\hat{\mathcal{L}}_t, \hat{\mathcal{L}}_s, \sim, n)$ says there exists a permuting sequence of length n that transforms $\hat{\mathcal{L}}_t$ to $\hat{\mathcal{L}}_s$, and each permutation always swaps adjacent instances satisfying relation $\sim$ (i.e., only do *necessary* swaps); this fact is obvious as $\hat{\mathcal{L}}_t$ is sorted by $\preccurlyeq_t$ and if sort is designed efficiently (so that it never increases inversion number during its operation *w.r.t.* $\preccurlyeq_s$, i.e., only swaps two instances satisfying $\succ_s$).

Now we move on to prove two list's equivalence, by induction on $\mathsf{StablePermut}$. The rule SPmtSwap is the only rule that enables the permutation, swapping two adjacent instances; other rules are correct trivially or by induction hypothesis.

The `true` result of the validation guarantees that any two necessarily swapped instances (i.e., satisfy relation $\sim$) have no WAW, WAR, or RAW dependencies *w.r.t.* access functions. From $\mathsf{Correct}(\mathsf{Check})$, we know the access functions over-approximate every instruction's real memory access pattern. Then whenever we swap two adjacent instances $\hat{\imath}_1, \hat{\imath}_2$ with the rule SPmtSwap, we know that they have no real WAW, RAW, or WAR dependencies.

Recall that all variables in the initial state are non-aliased. This property is preserved during execution thanks to the language's property $\mathsf{NonAliasPsrv}$.

By the provided property $\mathsf{BCPermut}$ of the instruction language (see Sect. 3.2), saying that two instances are permutable if they have no real WAW, WAR, or RAW dependencies and all variables are non-aliased, we finish the proof. □

5 Evaluation

In this section, we show how the framework can be applied to existing verified compilers and polyhedral compilers, and discuss our engineering efforts.

5.1 Support Existing Verified Compiler

To demonstrate the practicality of our approach, we instantiate the validator with a variant of CompCert's instruction language (called CInstr), making the proof complete. Several manual-written tests, like $\mathcal{P}_{cov}$ and $\mathcal{P}'_{cov}$ in Sect. 2, are successfully validated.

Now we explain the main components of CInstr's formalization.

- The syntax `I` of CInstr is a subset of CompCert's instruction language[1], including only assignments with multi-dimensional array accesses. We reuse CompCert's unary and binary operators while defining expressions. Id is defined as positive integers as in CompCert. Type T is defined as k-dimensional array types of the base type (CompCert's signed 32-bit integer).
- The state S is directly inherited from CompCert's formalization, which contains an environment that records definitions of global and local variables, and the CompCert memory model [18,19]. In such formalization, each variable is associated with a block identifier in the memory model.
- The semantics of CInstr reuses most CompCert's semantics, like its evaluation of normal expressions. The exception is that CInstr evaluates array subscripts in $\mathbb{Z}$, rather than fixed-width integer arithmetic. The CompCert memory model is one-dimensional, so multi-dimensional array accesses should be correctly mapped. Without loss of generality, loop iterators are considered ghost variables in $\mathbb{Z}$ that do not write to the state during iteration; their values are converted to fixed-width integers when necessary.
- The proposition $\mathsf{Compat}(\Gamma, \sigma)$ is defined as the appropriate allocation of typing context Γ in state σ, similar to how CompCert initializes variables. The proposition $\mathsf{Retrieve}(\mathcal{V}, \mathcal{E}, \sigma)$ asserts how each parameter's value is retrieved

[1] https://compcert.org/doc/html/compcert.cfrontend.Csyntax.html.

from the state; it involves casting fix-width integers to $\mathbb{Z}$. The proposition $\mathsf{NonAlias}(\sigma)$ signifies that each variable's reference (i.e., the assigned block identifier) is not equivalent to any other.
- The procedure $\mathsf{Check}(\mathtt{I}, \mathcal{W}, \mathcal{R})$ is defined to symbolically evaluate every array-access expression of instruction `I` to get its access functions and check if it is contained by input access functions $\mathcal{W}$ and $\mathcal{R}$. For example, the instruction `I: res += arr[i][(j+i)+(j-i)]` has write and read access functions $\mathcal{W}_{\checkmark}([\mathtt{i},\mathtt{j}]) = \{(\mathtt{res}, [])\}$ and $\mathcal{R}_{\checkmark}([\mathtt{i},\mathtt{j}]) = \{(\mathtt{res}, []), (\mathtt{arr}, [\mathtt{i}, 2{*}\mathtt{j}])\}$; if the inputs are $\mathcal{W} = \mathcal{W}_{\checkmark}$ and $\mathcal{R}([\mathtt{i},\mathtt{j}]) = \{(\mathtt{res}, []), (\mathtt{arr}, [\mathtt{i}, 2{*}\mathtt{j}]), (\mathtt{redundant}, [])\}$, the procedure returns `true`.
- Finally, with tedious yet straightforward proofs, we establish properties $\mathsf{NonAliasPsrv}$, $\mathsf{Correct}(\mathsf{Check})$, and $\mathsf{BCPermut}$ for the above definitions.

5.2 Support Existing Polyhedral Compiler

Polyhedral compilers like Pluto use similar polyhedral representations as *PolyProg*, like the OpenScop format [2]. After developing the converter for OpenScop and *PolyProg*, we can validate the Pluto's affine scheduling pass.

However, we have to give up some formal guarantees because Pluto underspecifies its inputs. To be concrete, Pluto's validator is instantiated with empty instruction language, and the procedure Check always returns `true`. This is because Pluto has no formal specification of its inputs, which only contain textual code fragments without sufficient contexts like typing to fully describe a nested loop. Consequently, as instructions have no formal semantics, we cannot define a verified checker for their semantics and access functions.

Nevertheless, it still makes sense to adopt our framework to Pluto. Like other polyhedral compilers, Pluto implicitly presumes its input programs agree on some properties like non-aliasing states and in-bound array accesses [10], which we believe are well-summarized by our instruction interface $\mathbb{I}$ in Sect. 3.2. Though we have to suspend the definition of the instruction language and give up checking the input access functions, we can still utilize the verified validation algorithm.

We collect all relevant tests and examples from Pluto's repository[2] as our test suite, counting to 62. We configure Pluto so that it only does affine scheduling, but still does its best:

```
pluto --smartfuse --rar \
      --notile --noparallel --noprevector --nounrolljam [.c]
```

Let's say Pluto compiles $\mathcal{P}_s$ to $\mathcal{P}_t$. Table 1 gives Pluto affine scheduler's running time (P-Time) and validator's running time (V-Time, for both *Validate*$(\mathcal{P}_s, \mathcal{P}_t)$ and *Validate*$(\mathcal{P}_t, \mathcal{P}_s)$) for each test[3]. The validator successfully validates every test's compilation, guaranteeing $\mathcal{P}_s \sqsubseteq \mathcal{P}_t \wedge \mathcal{P}_t \sqsubseteq \mathcal{P}_s$. Measurements are performed on an Intel i7-11700 processor with 64 GB memory, running in Windows Subsystem for Linux (WSL2) of Ubuntu 18.04 on Windows

[2] https://github.com/bondhugula/pluto.

[3] Please note that P-Time and V-Time for each test are not comparable, as they pertain to implementations of different algorithms in different programming languages.

10.0.19045. The result shows the validation algorithm runs in a reasonable time. As examples, the test case `tce` (Tensor Contraction Engine for four indexes) with the largest runtime (around 4.5 s) has 4 instructions operating over four-dimensional arrays, nested within loops at a depth of 5; for the corner test `noloop`, which contains no loops but one trivial assignment, the validation spends zero time.

Discussion on Validation's Completeness. We already mechanize formal proofs for the validator's *soundness*, which guarantees that the validator never

Table 1. Evaluation results on Pluto's test suite.

Test	P-Time (ms)	V-Time (ms,ms)	Test	P-Time (ms)	V-time (ms,ms)
covcol	3.5	434.6, 320.7	dsyr2k	2.6	106.0, 83.4
fdtd-2d	46.4	1615.5, 1296.3	gemver	7.0	247.9, 240.4
lu	6.1	410.6, 331.2	mvt	2.2	70.2, 56.3
ssymm	40.7	726.0, 551.2	tce	568.6	4442.0, 4422.5
adi	77.5	2531.7, 2377.8	corcol	5.5	442.5, 362.1
dct	21.8	879.4, 739.4	dsyrk	1.8	96.8, 78.9
floyd	12.1	502.6, 421.7	jacobi-1d-	3.8	184.0, 167.8
matmul-init	2.9	257.8, 192.4	pca	202.5	2923.6, 2679.5
strmm	1.9	141.4, 110.8	tmm	1.6	109.7, 89.6
advect3d	1023.1	579.1, 498.1	corcol3	13.6	851.3, 733.4
doitgen	10.4	1069.2, 837.4	fdtd-1d	6.0	268.7, 229.9
jacobi-2d-...	17.7	619.5, 543.5	matmul	3.2	157.1, 125.5
seidel	24.5	818.1, 725.5	strsm	6.4	209.3, 161.2
trisolv	5.1	338.9, 248.8	1dloop-invar	0.3	6.7, 6.0
costfunc	0.8	47.4, 35.0	fusion1	0.9	15.3, 13.9
fusion2	4.7	36.8, 31.6	fusion3	3.3	33.6, 33.8
fusion4	4.3	26.2, 23.4	fusion5	1.3	41.3, 38.9
fusion6	0.5	24.2, 25.4	fusion7	0.4	14.3, 13.8
fusion8	0.6	4.6, 3.8	fusion9	4.3	111.5, 110.2
fusion10	4.6	32.4, 25.7	intratileopt1	0.4	11.0, 7.8
intratileopt2	0.3	28.7, 21.3	intratileopt3	0.7	40.8, 35.9
intratileopt4	0.8	40.6, 33.3	matmul-seq	7.9	329.8, 275.7
matmul-seq3	19.8	531.4, 502.4	multi-loop-	1.2	46.3, 45.6
multi-stmt-...	14.7	57.4, 53.6	mxv	1.2	73.9, 61.8
mxv-seq	2.6	102.4, 96.2	mxv-seq3	6.3	200.4, 194.5
negparam	1.5	128.4, 118.4	nodep	0.4	26.5, 18.1
noloop	0.0	0.2, 0.0	polynomial	1.4	100.8, 90.0
seq	1.1	54.2, 50.5	shift	1.9	101.6, 88.2
spatial	0.3	29.9, 21.0	tricky1	5.7	105.6, 93.7
tricky2	0.5	19.3, 16.2	tricky3	3.7	105.1, 107.0
tricky4	0.2	6.1, 4.8	wavefront	0.7	39.0, 32.3

gives false positive results. Now we discuss its *completeness*: does the validator try its best to prove the correctness, validating valid transformations as much as possible? This matters because a validator always returning `false` is useless.

We argue that the validator is complete enough in practice, as shown in the evaluation. The completeness of the validation algorithm relies on the completeness of VPL's emptiness check $\mathsf{isEmpty}()$, which only gives one-way implication $\forall \mathcal{P}.\ \mathsf{isEmpty}(\mathcal{P}) \implies \mathcal{P} = \emptyset$. If we assume VPL's implementation is complete enough, then the validation algorithm should be equally complete. That's because the well-studied Bernstein's conditions basing the validation, should be the very correctness criterion of all reordering transformations like affine scheduling. Polyhedral scheduling algorithms were designed with this in mind, such as the *legality constraints* in Pluto's core algorithm that allow permutations of instances only when Bernstein's conditions are established [4].

5.3 Engineering Efforts

This work is fully mechanized in Coq, based on the Verified Polyhedron Library (VPL) [5] and the basic libraries and formalizations of PolyGen [8].

We've written around 17000 lines of Coq and 1000 lines of OCaml not counting blanks, comments and original proofs. In Coq, our proof efforts mainly involve additions to existing linear algebra and polyhedron libraries, formalization of *PolyProg*'s properties based on the instruction interface $\mathbb{I}$, implementation and verification of the validator and CInstr. In OCaml code, we provide basic functionalities like parser and printer for OpenScop format, compiler driver, building and testing scripts, etc.

6 Related Work

In this section, we discuss verification of polyhedral compilation and tensor compilation, as well as verified validation for other compiler optimizations.

6.1 Verification of Polyhedral Compilation

Courant and Leroy [8] provide basic formalizations of the polyhedral model and the first verified polyhedral code generator based on VPL [5]. Pilkiewicz [24] provides a prototypical verified validator (called s2sloop) of polyhedral scheduling ten years ago, sharing the idea to base the validation algorithm on Bernstein's conditions. In our hindsight, its abeyance is attributed to its excessive use of dependent type (making its proof engineering harder) and the lack of supported libraries, such as VPL. As emphasis, we do not merely reimplement his work. We go a step further, giving stronger proof and a well-designed framework with extensive evaluation. To be more specific with the first point, s2sloop's final theorem demands determinism (say, $\mathsf{Det}()$) of its first input program as an additional assumption, which makes it weaker than ours (see Sect. 4.2):

$$\mathsf{Correct}(\mathit{Validate}) \triangleq \forall \mathcal{P}_s, \mathcal{P}_t.\ \mathit{Validate}(\mathcal{P}_s, \mathcal{P}_t) = \mathtt{true} \land \boxed{\mathsf{Det}(\mathcal{P}_s)} \Rightarrow \mathcal{P}_t \sqsubseteq \mathcal{P}_s$$

This matters as a polyhedral compiler may make use of non-determinism to express parallelism [33], and the validator should be able to handle two parallelized inputs.

Similarly, Namjoshi and Singhania [23] provides a validator based on dependency violation (i.e., Bernstein's condition) for their semi-automatic polyhedral scheduling framework. However, the validator is itself unverified and only supports sequential programs. Our work can equip it with stronger correctness guarantees.

Polyhedral models assume arbitrary precision arithmetic for expressions in specific positions, which is not compatible with real-world fixed-width integer arithmetic that possibly overflows. Cuervo Parrino et al. [9] attempt to bridge such semantics gap using an optimistic approach [10].

6.2 Verification of Tensor Compilation

Tensor compilations allow users to model complex matrix computations with high-level mathematical representations and help to compile them to efficient low-level codes (involving nested loop), and validation in such domain also attracts attention. Clément and Cohen [7] design an end-to-end validation method between affine Halide algorithm [27] and its low-level imperative generated code. Liu et al. [20] design a Halide-like system within Coq, enabling developers to design and verify optimized schedules at the same time with the provided tactics, semi-automatically. Bang et al. [1] supports validating tensor-related dialects of `mlir` with appropriate SMT encoding; it does not support aggressive optimizations like the polyhedral-based yet.

6.3 Verified Validation for Compiler Optimization

Only a few validation algorithms for compiler optimization are formally verified. Tristan and Leroy [30–32] provides verified validators for instruction scheduling, software pipelining, and lazy code motion within CompCert; Gourdin et al. [22],Monniaux and Six [29],Six et al. [15] use block-level simulation to support verified validation of loop unrolling, loop invariant code motion, superblock scheduling, strength reduction and others within CompCert. All of these works focus on low-level RTL code, which is conceptually a control flow graph. These validation algorithms must synchronize two graphs correctly. To validate reordered instructions, they compare the summaries of subgraphs (such as basic blocks) obtained through symbolic execution, rather than directly checking Bernstein's conditions, which may require heavy dependence analysis.

7 Conclusions and Future Work

In this paper, we provide a well-designed verified validation framework for affine scheduling in polyhedral compilation, fully mechanized in Coq proof assistant. It is highly extensible, as we demonstrate by instantiating it with CInstr, a variant of CompCert's instruction language. Its core algorithm checks Bernstein's conditions for two input polyhedral models, which is complete and efficient enough to validate all 62 Pluto's tests as shown in our evaluation. We believe this work makes solid progress towards verified optimizing compilation.

As future work, it would be interesting to support advanced polyhedral techniques like tiling. Moreover, to seamlessly integrate polyhedral compilation into CompCert, we still need efficient extraction and its verification, which requires bridging the semantics gaps between C and polyhedral model [9,10].

Acknowledgments. We thank anonymous referees for their suggestions and comments on earlier versions of this paper. This work is supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62232015.

References

1. Bang, S., Nam, S., Chun, I., Jhoo, H.Y., Lee, J.: Smt-based translation validation for machine learning compiler. In: Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, 7–10 August 2022, Proceedings, Part II, p. 386-407, Springer, Berlin (2022). https://doi.org/10.1007/978-3-031-13188-2_19
2. Bastoul, C.: Openscop: A specification and a library for data exchange in polyhedral compilation tools. Paris-Sud University, France (September, Technical Report (2011)
3. Bernstein, A.J.: Analysis of programs for parallel processing. IEEE Trans. Electron. Comput. **15**, 757–763 (1966). https://doi.org/10.1109/PGEC.1966.264565
4. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN Not. **43**(6), 101–113 (2008). https://doi.org/10.1145/1379022.1375595
5. Boulmé, S., Maréchaly, A., Monniaux, D., Périn, M., Yu, H.: The verified polyhedron library: an overview. In: 2018 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 9–17, September 2018. https://doi.org/10.1109/SYNASC.2018.00014
6. Chen, T., et al.: Tvm: an automated end-to-end optimizing compiler for deep learning. In: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI 2018, pp. 579–594, USENIX Association, USA (2018)
7. Clément, B., Cohen, A.: End-to-end translation validation for the halide language. Proc. ACM Program. Lang. **6**(OOPSLA1) (2022). https://doi.org/10.1145/3527328
8. Courant, N., Leroy, X.: Verified code generation for the polyhedral model. Proc. ACM Program. Lang. **5**(POPL) (2021). https://doi.org/10.1145/3434321
9. Cuervo Parrino, B., Narboux, J., Violard, E., Magaud, N.: Dealing with arithmetic overflows in the polyhedral model. In: Bondhugula, U., Loechner, V. (eds.) IMPACT 2012–2nd International Workshop on Polyhedral Compilation Techniques. Louis-Noel Pouchet, Paris, France, January 2012
10. Doerfert, J., Grosser, T., Hack, S.: Optimistic loop optimization. In: Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, pp. 292-304, IEEE Press (2017). https://doi.org/10.1109/CGO.2017.7863748
11. Feautrier, P.: Some efficient solutions to the affine scheduling problem: Part i. one-dimensional time. Int. J. Parall. Program. **21**(5), 313–348 (1992a). https://doi.org/10.1007/BF01407835

12. Feautrier, P.: Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. Int. J. Parall. Program. **21**, 389–420 (1992b). https://doi.org/10.1007/BF01379404
13. Feautrier, P.: Bernstein's Conditions, pp. 130–134. Springer US, Boston, MA (2011).https://doi.org/10.1007/978-0-387-09766-4_521
14. Feautrier, P., Lengauer, C.: Polyhedron model. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1581–1592, Springer, Boston (2011). https://doi.org/10.1007/978-0-387-09766-4_502
15. Gourdin, L., Bonneau, B., Boulmé, S., Monniaux, D., Bérard, A.: Formally verifying optimizations with block simulations. Proc. ACM Program. Lang. **7**(OOPSLA2) (2023). https://doi.org/10.1145/3622799
16. Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Größlinger, A., Pouchet, L.N.: Polly - polyhedral optimization in LLVM. In: Alias, C., Bastoul, C. (eds.) 1st International Workshop on Polyhedral Compilation Techniques (IMPACT). Chamonix, France (2011)
17. Jourdan, J.-H., Pottier, F., Leroy, X.: Validating $LR(1)$ parsers. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 397–416. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_20
18. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert memory model, version 2. Research report RR-7987, INRIA (Jun 2012)
19. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. J. Autom. Reasoning **41**(1), 1–31 (2008). https://doi.org/10.1007/s10817-008-9099-0
20. Liu, A., Bernstein, G.L., Chlipala, A., Ragan-Kelley, J.: Verified tensor-program optimization via high-level scheduling rewrites. Proc. ACM Program. Lang. **6**(POPL) (2022). https://doi.org/10.1145/3498717
21. Livinskii, V., Babokin, D., Regehr, J.: Fuzzing loop optimizations in compilers for c++ and data-parallel languages. Proc. ACM Program. Lang. **7**(PLDI) (2023). https://doi.org/10.1145/3591295
22. Monniaux, D., Six, C.: Formally verified loop-invariant code motion and assorted optimizations. ACM Trans. Embed. Comput. Syst. **22**(1) (2022). https://doi.org/10.1145/3529507
23. Namjoshi, K.S., Singhania, N.: Loopy: programmable and formally verified loop transformations. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 383–402. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_19
24. Pilkiewicz, A.: s2sloop: a validator for polyhedral transformations. https://github.com/pilki/s2sLoop (2010-2013)
25. Pop, S., Cohen, A., Bastoul, C., Girbal, S., Silber, G.A., Vasilache, N.: Graphite: polyhedral analyses and optimizations for QCC. In: Proceedings of the 2006 GCC Developers Summit, p. 2006 (2006)
26. Pouchet, L.N., Bondhugula, U., et al.: The polybench benchmarks. https://www.cs.colostate.edu/~pouchet/software/polybench/ (2010-2015)
27. Ragan-Kelley, J., et al.: Halide: decoupling algorithms from schedules for high-performance image processing. Commun. ACM **61**(1), 106–115 (2017). https://doi.org/10.1145/3150211
28. Rideau, S., Leroy, X.: Validating register allocation and spilling. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 224–243. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11970-5_13
29. Six, C., Gourdin, L., Boulmé, S., Monniaux, D., Fasse, J., Nardino, N.: Formally verified superblock scheduling. In: Proceedings of the 11th ACM SIGPLAN Inter-

national Conference on Certified Programs and Proofs, CPP 2022, pp. 40–54, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3497775.3503679

30. Tristan, J.B., Leroy, X.: Formal verification of translation validators: a case study on instruction scheduling optimizations. SIGPLAN Not. **43**(1), 17–27 (2008). https://doi.org/10.1145/1328897.1328444
31. Tristan, J.B., Leroy, X.: Verified validation of lazy code motion. SIGPLAN Not. **44**(6), 316–326 (2009). https://doi.org/10.1145/1543135.1542512
32. Tristan, J.B., Leroy, X.: A simple, verified validator for software pipelining. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, pp. 83–92, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1706299.1706311
33. Verdoolaege, S., Guelton, S., Grosser, T., Cohen, A.: Schedule trees. In: Rajopadhye, S., Verdoolaege, S. (eds.) Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques (IMPACT), Vienna, Austria, January 2014
34. Zhao, J., et al.: Akg: automatic kernel generation for neural processing units using polyhedral transformations. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021, pp. 1233–1248, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454106