FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Reusable Loop Transformation Pipeline for Fortran Using Metafor

Sergei Korepanov

Mestrado em Engenharia de Software

Supervisor: Prof. João Carlos Viegas Martins Bispo

Second Supervisor: Eng. Luis Miguel Mendes Pimentel Alves de Sousa

July 24, 2026

A Reusable Loop Transformation Pipeline for Fortran Using Metafor

Sergei Korepanov

Mestrado em Engenharia de Software

Approved in oral examination by the committee:

President: Prof. Bruno Miguel Carvalhido Lim

Examiner: Prof. João Saraiva

Member: Prof. João Carlos Viegas Martins Bispo

July 24, 2026

Resumo

A otimização de programas científicos é frequentemente um processo iterativo. Uma versão inicial pode expressar o algoritmo de forma clara, enquanto versões otimizadas tendem a acumular transformações manuais, como fusão, fissão, intercâmbio, tiling e desenrolamento de ciclos. Embora estas transformações possam melhorar o desempenho, também podem reduzir a manutenibilidade do código e tornar mais difícil compreender a intenção original do algoritmo.

Esta dissertação investiga uma abordagem de transformação source-to-source para programas Fortran, integrada na infraestrutura METAFOR. O trabalho segue uma metodologia baseada em artefacto: o principal resultado é uma extensão do METAFOR com suporte explícito e controlável para transformações de ciclos DO em Fortran. Foram implementadas cinco transformações clássicas: loop fusion, loop fission, loop interchange, loop tiling e loop unrolling.

A solução desenvolvida utiliza a infraestrutura do METAFOR para representar programas Fortran como árvores sintáticas abstratas, expor elementos do programa através de join points e aplicar transformações através de scripts. O código transformado é novamente gerado como código Fortran, permitindo que seja inspecionado, compilado, testado e comparado com a versão original.

A correção foi tratada como requisito central. Para cada transformação foram definidos controlos conservadores de legalidade, incluindo validações sobre a estrutura dos ciclos, acessos a arrays, dependências escalares, padrões de redução e limites triangulares ou dependentes de variáveis. Quando a ferramenta não consegue justificar que uma transformação é segura, a transformação é rejeitada e o código original é preservado.

A avaliação foi realizada com benchmarks PolyBench/Fortran. Os resultados mostram que as transformações implementadas preservam a saída dos programas nos casos testados e que os efeitos de desempenho variam consoante a transformação e o padrão de acesso à memória. A dissertação também avalia a composição de transformações, mostrando que aplicar fissão antes de tiling aumenta o número de benchmarks em que o tiling pode ser aplicado com sucesso.

O trabalho demonstra que o METAFOR pode ser estendido com uma pipeline reutilizável de transformações de ciclos para Fortran. A principal contribuição não é substituir compiladores otimizadores, mas fornecer uma forma explícita, inspecionável e ajustável de aplicar transformações source-to-source a programas Fortran existentes.

Palavras-chave: Fortran, compilação source-to-source, transformações de ciclos, METAFOR, LARA, computação de alto desempenho, engenharia de software.

Abstract

Scientific software optimization is often an iterative process. An initial version of a program may express the algorithm clearly, while optimized versions tend to accumulate several manual transformations, such as loop fusion, fission, interchange, tiling, or unrolling. These transformations may improve performance, but they can also reduce code maintainability and make the original algorithmic intent harder to understand.

This dissertation investigates a source-to-source transformation approach for Fortran programs, integrated into the Metafor infrastructure. The work follows an artifact-based methodology: the main result is an extension of Metafor with explicit and controllable support for Fortran DO loop transformations. Five classical loop transformations were implemented: loop fusion, loop fission, loop interchange, loop tiling, and loop unrolling.

The developed solution uses the existing Metafor infrastructure to represent Fortran programs as abstract syntax trees, expose program elements through join points, and apply transformations through scripts. The transformed program is regenerated as Fortran source code, allowing it to be inspected, compiled, tested, and compared with the original version.

Correctness was treated as a central requirement. Each transformation is guarded by conservative legality checks, including loop-shape validation, array access analysis, scalar dependency checks, reduction rejection, and handling of triangular or variable-dependent bounds. When we cannot guarantee that a transformation is safe, the transformation is rejected and the original code is preserved.

The evaluation was conducted using PolyBench/Fortran benchmarks. The results show that the implemented transformations preserve program output in the tested cases and that performance effects depend on the transformation and memory-access pattern. The dissertation also evaluates transformation composition, showing that applying loop fission before loop tiling increases the number of benchmarks where tiling can be applied successfully.

The work demonstrates that Metafor can be extended with a reusable loop transformation pipeline for Fortran. The main contribution is not to replace optimizing compilers, but to provide an explicit, inspectable, and adjustable source-to-source workflow for applying loop transformations to existing Fortran programs.

Keywords: Fortran, source-to-source compilation, loop transformations, Metafor, LARA, high-performance computing, software engineering.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. It includes 17 goals to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

The specific Sustainable Development Goals mentioned have the following names:

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

SDG 12 Ensure sustainable consumption and production patterns

SDG	Target	Contribution	Performance Indicators and Metrics
9	9.4	Metafor enables the modernization of long lived scientific code and enhances reproducibility and reusability of scientific artifacts in performance computing.	Number of benchmarks transformed; correctness match rate; transformation applicability; execution-time speedups and slowdowns.
	9.5	Development of a reusable source-to-source transformation artifact for Fortran programs inside Metafor.	Number of loop transformations; generated Fortran output; support for script-controlled transformation.
12	12.2	Indirect support for future resource-efficiency studies through reproducible transformation experiments.	Loop Transformation transparency reduces the cycle of experiments in achieving desired performance metrics.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Turning a clear scientific algorithm into high-performance code is rarely a single-step process. In scientific and high-performance computing, performance-critical computation is often concentrated in nested Fortran `DO` loops over arrays. Optimizing these loops typically involves repeated transformations such as fusion, fission, interchange, tiling, and unrolling to improve locality, vectorization, or parallel execution. While these transformations can improve performance, they also make the resulting code progressively harder to understand, validate, and maintain. After several optimization iterations, the connection between the optimized implementation and the original algorithm may become difficult to recognize.

Existing approaches address this problem in different ways, but they differ in the degree of control they provide and in their suitability for existing Fortran applications.

In practice, performance tuning often remains a manual and iterative activity. A developer identifies a critical kernel, modifies the code, compiles it, tests it, measures performance, and then repeats the process. This workflow is expensive. It limits how much of the optimization space can be explored. It also increases the risk of subtle correctness errors, because every transformation must preserve the original program behavior. Most importantly, it tends to leave behind only the final optimized source code, not a clear record of the transformation steps that produced it.

This dissertation investigates a different workflow: source-to-source loop transformation guided by explicit transformation scripts. Instead of permanently replacing the original code with a manually optimized version, the developer keeps the high-level Fortran source as the reference program. Transformations are then applied to that source before compilation, producing transformed Fortran code that can be inspected, compiled, tested, and measured. The optimization process becomes explicit: the transformation script records what was changed, where it was changed, and with which parameters.

The dissertation is artifact-based. Its main contribution is separating performance optimization from algorithm development and maintenance in Fortran programs. This is achieved by extending the METAFOR compiler with support for selected Fortran loop transformations. The transformations are exposed as script-controlled operations. A user can select a program region, apply a transformation, choose parameters such as tile size or unrolling factor, regenerate Fortran code,

and validate the result against the original program.

METAFOR is a source to source compiler for Fortran language that supports source code transformation via scripts written in TypeScript. Detailed overview of the architecture can be found in Chapter 5.1 The compiler supports five classical loop transformations: fusion, fission, interchange, tiling, and unrolling. These transformations are evaluated individually and as part of transformation pipelines.

A central argument of this dissertation is that individual transformations are not enough. Real optimization is often a pipeline. One transformation prepares the code for another, and the value of the first transformation may only become visible later in the sequence. For example, loop tiling requires a suitable loop nest. Some loops are not directly tileable because independent computations are mixed in the same loop body or because the structure does not match the expected transformation pattern. Loop fission can separate those computations and expose simpler loop nests that can then be tiled. For this reason, the evaluation includes a pipeline experiment comparing tiling alone with fission followed by tiling. The goal is to show that transformation pipelines can enable successful transformations that would not be possible when each transformation is applied in isolation.

This source-to-source approach does not replace optimizing compilers. Instead, it complements them. The transformed Fortran code is still compiled by a normal compiler, but the source-level restructuring is made visible and controllable before compilation. This gives the developer a practical way to combine human insight with automated code generation: the human decides which transformations to apply, while the framework performs the rewriting and preserves a reproducible transformation path.

1.1 Context

Fortran remains a major language in scientific and high-performance computing. Many numerical applications have been developed and validated over long periods, and their most expensive computations are often expressed as loops over arrays. These loops are central both to the meaning of the program and to its performance.

Correct application of loop transformations requires preserving program semantics. They can improve memory locality, reduce loop overhead, expose vectorization opportunities, or prepare the code for parallel execution. General-purpose compilers and directive-based models already support some of these optimizations, while systems such as CHiLL [4], POET [14], OptiTrust [3], and Xevolver [13] demonstrate different forms of source-level or script-controlled transformation. However, these existing approaches differ in language focus, transformation model, and integration context. The work in this dissertation is concerned specifically with extending METAFOR with explicit loop transformation support for Fortran.

Loop transformations are not simple textual rewrites. A legal transformation must preserve the order of dependent reads and writes. If this order is changed incorrectly, the transformed program may still compile and run but produce different results.

The METAFOR infrastructure provides the basis for the work developed in this dissertation. It supports source-to-source transformation of Fortran programs by parsing source code, representing it as an abstract syntax tree, exposing program elements through a join-point model, and regenerating Fortran source after modification.

1.2 Motivation

The motivation for this work is the maintainability cost of manual performance optimization. A high-level scientific program usually expresses what the computation does. An optimized version often expresses how the computation has been adapted to a particular machine or performance goal. After several optimization iterations, these two concerns become mixed in the same source code.

This creates several practical problems. First, optimization becomes expensive because each candidate version must be implemented manually. Second, performance exploration becomes limited because there is not enough time to try many transformation sequences. Third, correctness becomes harder to guarantee because every manual rewrite can introduce a subtle change in program behavior. Finally, maintainability suffers because future changes to the algorithm must be transferred into code that already contains many baked-in optimization decisions.

Existing systems partially address these issues. OpenMP gives programmers a portable way to express selected optimization and parallelization intentions. DSLs such as Halide [12] and Tiramisu [1] separate the algorithm from the schedule, which can make optimization more systematic in their target domains. Scriptable transformation tools such as OptiTrust [3], CHiLL [4], POET [14], and Xevolver [13] make transformation steps more explicit than manual rewriting. These approaches show the value of exposing optimization decisions, but they also motivate the need for a Fortran-oriented, METAFOR-integrated workflow where loop transformations can be applied, inspected, validated, and composed through scripts.

The motivation is stronger when transformations are composed. A single transformation may not be applicable to the original code, but an earlier restructuring step may make it applicable. In this dissertation, loop fission followed by loop tiling is used as a concrete example of this idea. Fission can separate independent statements and expose loop nests that satisfy the requirements for tiling, increasing the number of successful tiling transformations compared with tiling alone.

1.3 Problem Statement

This dissertation addresses the lack of a reusable, script-controlled source-to-source loop transformation workflow for Fortran programs within METAFOR. It designs, implements, and evaluates an artifact that identifies Fortran `DO` loops, applies selected loop transformations, rejects unsafe cases, regenerates compilable Fortran source, and supports transformation pipelines.

The artifact must also support composing transformations into reusable pipelines, allowing one transformation to prepare the code for another.

1.4 Objectives

This dissertation aims to achieve the following objectives:

Objective 1. Extended METAFOR with reusable source-to-source operations for classical Fortran loop transformations.

Objective 2. Implement legality checks to prevent unsafe transformations while preserving the output of benchmark programs for supported cases.

Objective 3. Provide a practical workflow in which users select program regions, transformation parameters, and transformation order through script-controlled transformations.

Objective 4. Include transformation composition to increase transformation applicability.

1.5 Hypothesis

The main hypothesis is that source-to-source loop transformation can reduce the maintainability cost of performance optimization by keeping the original Fortran program separate from generated optimized variants. If transformations are expressed as scripts, the optimization process becomes explicit and reproducible instead of being hidden inside manually rewritten code.

A second hypothesis is that transformation pipelines can expose opportunities that isolated transformations miss. In particular, applying loop fission before loop tiling is expected to increase the number of successful tiling transformations, because fission can simplify loop bodies and produce loop nests that better satisfy the structural requirements of tiling.

1.6 Contributions

This dissertation makes the following contributions.

First, it extends METAFOR with five script-controlled source-to-source loop transformations for Fortran: fusion, fission, interchange, tiling, and unrolling. Users can select program regions, apply transformations, set parameters, and inspect the generated Fortran source before compilation.

Second, it implements conservative legality checks that reject unsafe transformations while preserving program correctness.

Third, it validates transformed programs against the original benchmark outputs. This validation is central because a transformation framework must preserve program behavior before its performance results can be trusted.

Fourth, it evaluates the performance effects of the implemented transformations on Fortran benchmark programs.

Finally, it evaluates transformation pipelining through a fission-then-tiling experiment. This experiment shows that source-level transformation sequences can increase the applicability of later transformations compared with applying those transformations directly to the original program.

1.7 Dissertation Structure

The remainder of this dissertation is organized as follows.

Chapter 2 presents the background required for the work. It introduces Fortran in scientific computing, source-to-source compilation, abstract syntax trees, loop transformations, legality checking, and the need for user control in optimization workflows.

Chapter 3 reviews related work. It discusses optimizing compilers, compiler directives, annotation-based systems, domain-specific languages, rewrite-oriented tools, and scriptable source-to-source transformation frameworks. The chapter identifies the gap addressed by this dissertation.

Chapter 4 describes the methodology. It explains the artifact-based research approach, the selected transformations, the general transformation procedure, the legality-checking strategy, and the design choices behind each transformation.

Chapter 5 presents the implementation. It describes the METAFOR pipeline, the integration with the Fortran parsing and AST infrastructure, the scripting interface, the implemented transformations, and the correctness-driven development process.

Chapter 6 evaluates the artifact. It presents the experimental setup, benchmark selection, correctness validation, performance results, threats to validity, and the pipeline experiment comparing tiling alone with fission followed by tiling.

Chapter 7 concludes the dissertation by summarizing the findings, discussing limitations, and outlining future work.

Chapter 2

Background

2.1 Introduction

This chapter introduces the concepts needed to understand the dissertation: Fortran in scientific computing, source-to-source compilation, program representation, loop transformations, and legality checking. The focus is on loop transformations applied to Fortran programs, not only as performance optimizations, but also as explicit source-level operations that can be inspected, adjusted, and reused.

The chapter also establishes the main software engineering concern behind the work. Loop transformations are widely used in compilers, high-performance computing tools, domain-specific languages, and source-to-source systems, but their selection and application are often hidden inside a compiler or tied to a specific programming model. This makes it difficult for developers to control optimization decisions in large scientific codebases.

2.2 Fortran in Scientific and High-Performance Computing

Fortran remains important in scientific and high-performance computing because many numerical applications are built around arrays, loops, and long-running computational kernels. Although it is often associated with legacy software, this view is incomplete. Many actively maintained scientific systems still use Fortran, and modern Fortran supports structured programming, modules, array operations, and interoperability features.

This matters for source-to-source transformation. Scientific projects may combine older coding styles with newer Fortran features, so a transformation framework should not be limited to a narrow subset of the language. In this dissertation, the benchmarks are written in a Fortran 90 style, while the broader engineering goal is to support more modern Fortran constructs as the framework evolves.

Fortran programs are also strongly connected with performance-critical loop nests. Many kernels are expressed as `DO` loops over one-dimensional or multi-dimensional arrays. These loops

often dominate execution time and are natural candidates for transformation. Changing their structure can improve cache locality, reduce loop overhead, expose parallelism, or help the backend compiler generate better machine code.

A Fortran-specific aspect relevant to loop transformations is array layout. Fortran stores multi-dimensional arrays in column-major order, meaning that the leftmost subscript varies fastest in memory. As a result, loop order can strongly affect memory locality. This dissertation does not treat array layout as a separate optimization topic, but it is relevant when transformations such as loop interchange and tiling are applied.

2.3 Source-to-Source Compilation

A traditional optimizing compiler lowers source code into intermediate or machine-level representations and applies many optimizations after the original source structure has been transformed. This approach is powerful, but the developer usually cannot inspect the exact source-level sequence of transformations applied by the compiler.

Source-to-source compilation follows a different model. It takes a source program as input and produces another source program as output. The generated program remains written in a high-level language and can be compiled by a normal compiler. In this dissertation, source-to-source compilation means structured transformation of Fortran source code into transformed Fortran source code.

This model has three practical advantages. First, the transformed code can be inspected, compiled, tested, and compared with the original version. Second, transformation decisions can be controlled outside the compiler. Third, the original algorithmic source can remain separate from generated variants used for performance experiments. This separation is important because heavily optimized code is often harder to read and modify than the original version.

The term is related to transpilation and program rewriting, but the emphasis here is different. Transpilation often refers to translation between languages at a similar abstraction level, while program rewriting may refer to smaller pattern-based changes. The work in this dissertation uses structured source-to-source transformation within the same language: Fortran in, Fortran out.

2.4 Abstract Syntax Trees and Program Representation

Program transformation can be performed directly on text, but text-based rewriting is fragile for non-trivial changes. It is difficult to reason about nested structures, declarations, expressions, scopes, and dependencies using plain strings. For this reason, source-to-source transformation systems usually operate on a structured program representation such as an abstract syntax tree (AST).

An AST represents a program as a tree of nodes. A loop, assignment, variable reference, array access, function call, or expression can be represented as a node with properties and children. This makes it possible to locate constructs and rewrite them in a controlled way. For example, a loop

transformation can find a `DO` loop node, inspect its bounds and body, modify the relevant subtree, and regenerate source code from the modified AST.

AST-based rewriting is not automatically safe or complete. The tool still needs accurate parsing, a sufficiently rich representation of language constructs, and reliable code generation. However, it provides a stronger engineering basis than textual rewriting when transformations must preserve program structure and behavior.

2.5 Loops as Optimization Targets

Loops are central in scientific computing because they express repeated numerical work over arrays. Small changes in loop order, nesting, or body structure can affect locality, vectorization, instruction scheduling, and parallelization. A loop may also be transformed not for immediate speedup, but to prepare the program for another optimization.

The five transformations considered in this dissertation illustrate these different roles. Loop fission can separate independent computations and expose simpler loop bodies. Loop fusion can reduce loop overhead and improve locality between related operations. Loop tiling can improve cache reuse by processing iteration spaces in blocks. Loop unrolling can reduce loop-control overhead and expose more straight-line code. Loop interchange can improve memory access order or create a structure more suitable for later transformations.

It is important to distinguish legality from profitability. A transformation is legal if it preserves the observable behavior of the program. It is profitable if it improves a chosen metric, such as execution time. Profitability depends on memory layout, cache behavior, compiler flags, processor architecture, dataset size, and interactions with other compiler optimizations. A transformation framework therefore needs both safety checks and an evaluation workflow.

2.6 Basic Loop Transformations

This dissertation focuses on five classical loop transformations: loop fusion, loop fission, loop interchange, loop tiling, and loop unrolling. They are introduced here conceptually; implementation details are discussed later.

Loop fusion combines two adjacent loops into one loop. It is applicable when the loops have compatible iteration spaces and their bodies can be executed together without changing program behavior. Fusion may reduce loop overhead and improve locality, but it is unsafe if merging the loops changes the order in which dependent values are produced and consumed.

Loop fission, also called loop distribution, splits one loop into two or more loops. It can separate independent computations, simplify loop bodies, and expose opportunities for vectorization, tiling, or parallelization. It is unsafe when statements inside the original loop communicate through scalar temporaries or depend on the original per-iteration execution order.

Loop interchange swaps the order of nested loops. It is commonly used to improve memory access order or move a more profitable loop to the innermost position. In Fortran, this is especially

relevant because column-major array layout makes some loop orders more cache-friendly than others. Interchange requires careful checks when bounds or inner computations depend on outer loop variables.

Loop tiling, also known as blocking, divides a loop nest into smaller blocks. Its main purpose is to improve data reuse by keeping a working set in cache for longer. Tiling is common for nested loops over arrays, but it requires regular iteration spaces and careful handling of boundary tiles.

Loop unrolling duplicates the loop body several times and increases the loop step accordingly. It can reduce loop-control overhead and expose more operations to the compiler. However, it can also increase code size and register pressure, so it does not always improve performance.

2.7 Legality and Dependence Checking

The main difficulty in loop transformation is preserving program meaning. A transformation must respect data dependences between reads and writes. If transformed code reads a value before it is computed, or overwrites a value before it is used, it may still compile but produce incorrect results.

Full dependence analysis is complex. It may need to reason about array subscripts, loop bounds, aliases, scalar variables, reductions, procedure calls, and control flow. Practical source-to-source systems often use conservative checks: if the tool cannot prove that a transformation is safe, it rejects the transformation. This may reject some legal cases, but it reduces the risk of generating incorrect code.

The checks used in this work are designed for common loop structures in scientific benchmarks. They include syntactic loop-shape checks, array read/write comparisons, iterator-based subscript checks, scalar dependency checks, and conservative handling of reductions. These checks do not replace a complete compiler dependence analysis; they provide a practical safety layer for the implemented transformations.

2.8 Transparency and User Control

A central motivation for this dissertation is the need for transparent and adjustable loop transformation workflows. Compilers already perform many optimizations, but their decisions are often internal. Directives and pragmas expose some programmer intent, but the final transformation still depends on compiler support and implementation choices. Domain-specific languages give stronger control in selected domains, but often require existing applications to be rewritten.

Source-to-source transformation offers a different compromise. It exposes transformation decisions at the source level, allows scripts to be reused, and makes optimization experiments easier to reproduce. Instead of relying only on compiler heuristics, the developer can choose the target region, transformation, and parameters. This is especially useful in scientific software, where domain knowledge and manual tuning often remain important.

The next chapter reviews existing optimization and transformation approaches with this distinction in mind: how much control they give to the developer, whether they preserve source-level visibility, and how suitable they are for existing Fortran codebases.

Chapter 3

State of the Art

3.1 Introduction

This chapter reviews approaches related to loop transformation and source-to-source optimization. The aim is not to cover every compiler or transformation system, but to identify the design choices most relevant to this dissertation. The reviewed work is organized into five categories: optimizing compilers, directive and annotation-based approaches, domain-specific languages, rewrite-oriented systems, and scriptable source-to-source tools.

The central distinction is how each approach balances automation and user control. Some systems hide most optimization decisions inside compiler heuristics. Others expose decisions through directives, annotations, schedules, rewrite rules, or scripts. These approaches are not simply competing alternatives; each makes a different trade-off between portability, automation, transparency, and developer effort. Table 3.1 summarizes these categories.

3.2 Optimizing Compilers

Optimizing compilers are the default path for performance improvement in most Fortran programs. They apply transformations to source code, intermediate representations, or machine-level forms to generate efficient executables.

Their strength is automation, but this also defines their limitation for the present work. A compiler must preserve language semantics across a wide range of programs and avoid transformations that may be unsafe or likely to degrade performance. For example, floating-point transformations can be constrained by language semantics, and profitable transformations may require domain knowledge that the compiler cannot infer. Charguéraud et al. [3] discuss this limitation in the context of source-to-source optimization, noting that some useful transformations remain outside what general-purpose compilers can reliably apply automatically.

Visibility is another limitation. A developer may compile with an optimization flag such as `-O3`, but the exact transformation sequence is usually not visible at the source level. This does

not reduce the importance of compiler optimization; it only means that compilers are not always sufficient when the goal is to make transformation decisions explicit, repeatable, and adjustable.

3.3 Compiler Directives, Pragmas, and Annotation-Based Tools

Compiler directives and pragmas allow programmers to guide optimization by embedding annotations in the source code. OpenMP[1] and OpenACC[2] are widely used in HPC to express parallelism and accelerator-oriented execution while keeping the original language.

This approach is attractive for large scientific codebases because a Fortran or C program can be extended without being rewritten in a new DSL. The trade-off is that directives still depend on compiler support, and the final generated code may vary across compilers and platforms.

3.3.1 OpenMP 5.x Loop Transformations

OpenMP is best known for shared-memory parallel programming, but recent versions also include loop transformation constructs. OpenMP 5.1 and 5.2 describe constructs such as `tile` and `unroll` .[3][4] These directives make selected loop transformations more explicit than compiler flags and show that loop transformation is becoming part of mainstream programming models.

Their main limitation is flexibility. OpenMP constructs are useful when the desired transformation fits the standard and the compiler implements it, but they are less suitable for custom multi-step transformation pipelines or for inspecting each intermediate source-level result.

3.3.2 Orio

Orio is an annotation-based empirical tuning system developed by Albert Hartono, Boyana Norris, and P. Sadayappan [7]. Developers mark regions of source code and describe performance transformations through structured annotations. Orio can then generate and measure multiple variants, exploring parameters such as tile sizes or unrolling factors.

Orio supports transformations including loop unrolling, tiling, permutation, scalar replacement, register tiling, array copy optimization, and OpenMP-based multicore parallelization. Its strength is empirical search over optimized variants. For this dissertation, its main limitation is that the described workflow is primarily centered on C and annotation-guided tuning, while this work focuses on controlled source-to-source transformation of Fortran `DO` loops.

3.3.3 X Language

The X Language is an annotation-based system for generating high-performance code, especially for matrix-matrix multiplication routines [6]. It uses annotated C or C++ programs, parses them

[1] https://www.openmp.org/
[2] https://www.openacc.org/
[3] https://www.openmp.org/spec-html/5.1/openmp.html
[4] https://www.openmp.org/spec-html/5.2/openmp.html

into an AST, identifies loops and directives, and rewrites marked regions using parameterized transformations such as unrolling and strip-mining.

It is relevant because it shows an early approach to explicit transformation control through annotations and transformation rules. Its focus, however, is mainly C/C++ and a specific style of parameterized optimization, rather than modern Fortran source-to-source transformation.

3.3.4 Scout

Scout is a source-to-source transformation tool for SIMD vectorization of C code [8]. It uses `#pragma` directives to mark loops, builds an AST using Clang, applies transformations, and rewrites the result back to C. Its transformations include loop simplification, unrolling, partial vectorization, and generation of SIMD intrinsics.

Scout shows how source-to-source transformation can expose architecture-specific optimization while still producing ordinary source code. Its scope is narrower than this dissertation because it focuses on SIMD vectorization for C, not a broader set of classical loop transformations for Fortran.

3.4 Domain-Specific Languages

Domain-specific languages offer another route to high performance. Instead of transforming an existing program directly, they provide a programming model designed for a specific class of computations. In HPC, DSLs often separate the mathematical algorithm from the implementation schedule, enabling aggressive optimization by a specialized compiler or code generator.

The cost is adoption. Existing scientific applications may contain years of development and validation in languages such as Fortran. Rewriting them in a DSL can be expensive and risky, as noted by Park et al. [11]. This concern also applies to kernel-generation systems such as Loopy, where the programming model is powerful but requires adaptation from existing source code [10]. Table 3.4 compares the DSL-based systems reviewed below.

3.4.1 Halide

Halide is a DSL for image-processing pipelines [12]. Its main idea is to separate the algorithm, which describes what is computed, from the schedule, which describes how it is executed. This separation lets developers explore tiling, fusion, vectorization, parallelization, and storage decisions without rewriting the algorithm itself.

Halide is highly effective in its target domain, but it is not designed to transform existing Fortran programs directly. It is most useful when the computation can be expressed in Halide's functional model.

3.4.2 Tiramisu

Tiramisu is a polyhedral framework and C++ embedded DSL for generating high-performance code across multicore CPUs, GPUs, and distributed machines [1]. Like Halide, it separates algorithm from schedule, but it targets a broader range of computations, including image processing, deep learning, linear algebra, tensor operations, and stencils.

Tiramisu supports affine transformations such as tiling, splitting, shifting, fusion, parallelization, vectorization, and GPU mapping. Its schedule-based model gives fine-grained control over generated code, but existing Fortran computations must be expressed in Tiramisu's representation before those benefits can be used.

3.5 Code Transformations via Rewrite Tools

Rewrite-oriented systems transform programs through pattern matching, transformation rules, or optimization patterns. They can work with existing codebases when the language support and program representation are sufficient. Their main limitation is that they are often constrained by the patterns they can recognize and by the safety conditions they can check.

3.5.1 EPOD

EPOD, or Extendable Pattern-Oriented Optimization Directives, encapsulates algorithm-specific optimizations into reusable patterns [5]. It supports patterns such as stencils, dense matrix multiplication, dynamically allocated arrays, and compressed arrays. These patterns can trigger loop transformations, data-layout changes, and synchronization.

EPOD starts from C or Fortran source annotated with pragmas. It checks whether a region satisfies the conditions for a pattern, normalizes loop nests, analyzes data accesses, and applies an EPOD script that defines the optimization scheme. It is relevant because it supports Fortran and combines source-to-source translation with domain-specific knowledge. Its focus is more pattern-oriented than the approach in this dissertation, which emphasizes user-selected general loop transformations.

3.6 Scriptable Source-to-Source Transformation Tools

Scriptable source-to-source tools use external scripts to describe where and how transformations should be applied. A script can specify transformation order, parameters, and program locations. This approach is especially relevant here because it separates the original source from generated variants and makes optimization steps easier to inspect and reproduce.

3.6.1 OptiTrust

OptiTrust is an interactive framework for source-to-source transformations of C code [3]. It aims to make high-performance code development more systematic, reviewable, and maintainable than manual optimization.

OptiTrust uses OCaml transformation scripts over an internal AST and can show textual differences after each transformation step. It supports loop fission, fusion, unrolling, tiling, data-layout changes, and lower-level rewrites. Conceptually, it is one of the closest references for this dissertation because it emphasizes transparency and scripted transformation. The main difference is language focus: OptiTrust targets C and parts of C++, while this work targets Fortran.

3.6.2 Clava

Clava is a Clang-based source-to-source compilation framework for C/C++ that uses LARA as its scripting language [2]. It parses C/C++ code with Clang, builds a Clava AST, exposes program elements through the Clava Weaver, and applies LARA strategies for analysis, instrumentation, transformation, compilation, and execution.

Clava is relevant to this dissertation because it demonstrates the use of LARA for script-controlled source-to-source transformation. It supports reusable strategies, custom targetability, and design exploration, where the same program can be transformed in different ways or with different parameter values. It also provides built-in actions for program manipulation, including loop transformations such as interchange and tiling.

The main limitation for this dissertation is language focus. Clava targets C/C++ and OpenCL rather than Fortran. However, it is conceptually close to the work presented here because both approaches use LARA-based scripting, join-point selection, AST manipulation, and source-code regeneration.

3.6.3 Xevolver

Xevolver is an XML-based translation framework for HPC application migration and evolution [13]. It parses C or Fortran programs into an AST, exposes the AST as XML, and lets users manipulate it through XSLT-based recipes. After transformation, the modified representation is converted back to source code.

Xevolver is relevant because it supports Fortran and user-defined transformations, including examples such as loop interchange. Its XML/XSLT representation is flexible, but can become verbose for complex rewrites.

3.6.4 Loopy

Loopy is a Python-based tool for representing and transforming array-based computational kernels [10]. A Loopy kernel is a symbolic computation description, and users apply Python-based transformations to control loop structure, memory mapping, scheduling, and parallelization. Loopy can

generate C or OpenCL C code.

Loopy provides strong programmable control for high-performance kernels, especially on GPU and many-core targets. For existing Fortran projects, however, computations must first be represented in Loopy's model, which introduces an adaptation step.

3.6.5 CHiLL

CHiLL, or Composing High-Level Loop Transformations, applies high-level loop transformations through scripts [4]. It supports transformations such as permutation, tiling, unroll-and-jam, data copying, iteration-space splitting, fusion, and distribution.

CHiLL uses a polyhedral representation of iteration spaces and statements, allowing transformations to be composed without generating intermediate source code after each step. It is relevant because it focuses directly on loop transformation scripts and composition. Compared with CHiLL, this dissertation takes a more source-level engineering route based on Fortran AST rewriting.

3.6.6 ASSIST

ASSIST is a source-to-source transformation tool for HPC applications integrated into the MAQAO toolset [9]. It combines source manipulation with static and dynamic profiling information. It supports transformations such as interchange, unrolling, strip mining, tiling, specialization, constant propagation, local dead-code elimination, and block vectorization.

ASSIST can use directives or Lua scripts and relies on ROSE to parse and manipulate C, C++, and Fortran ASTs. It is relevant because it supports Fortran and script-based transformation. Its focus is broader performance engineering, while this dissertation concentrates on implementing and validating a selected set of Fortran loop transformations inside METAFOR.

3.6.7 POET

POET, or Parameterized Source-to-source Program Transformations, is a scripting language for programmable transformation [14]. It uses external syntax descriptions to parse and unparse several languages, including C, C++, Java, Fortran, and DSLs. Transformations are defined as `xform` routines and can include pattern matching, AST replacement, and parameterized code generation.

POET is relevant because it provides language-independent scriptable transformation and has been used for loop interchange, blocking, fusion/fission, parallelization, scalar replacement, and vectorization. Its generality is also its limitation for this dissertation: it is a broad transformation infrastructure rather than a focused Fortran loop transformation artifact.

3.7 Discussion of the Research Gap

The reviewed systems show that loop transformations are supported in many forms. Optimizing compilers automate transformations but usually hide decisions from the developer. Directives

expose some intent, but remain limited by standards and compiler support. DSLs provide strong scheduling control, but often require existing programs to be rewritten. Rewrite-oriented and scriptable source-to-source tools offer more explicit transformation, but differ in language support, representation, and suitability for composing transformations over existing Fortran code.

The gap addressed by this dissertation is therefore not the absence of loop transformations. The gap is the need for a Fortran-oriented source-to-source workflow in which loop transformations are transparent, adjustable, and controlled by the user. Such a workflow should allow the developer to select a program region, choose transformation parameters such as tile size or unrolling factor, inspect the generated source, and evaluate the result against the original program.

The approach investigated here tests whether METAFOR can support this workflow for selected Fortran loop transformations. It does not aim to replace optimizing compilers, DSLs, or directive standards. Instead, it explores an engineering path that combines source-level visibility with explicit user control over transformation selection and composition.

3.8 Comparison Tables

The following tables show cross comparison of different approaches for optimizations in Table 3.1 and comparison of different tools available in the industry in Table 3.2 and Table 3.3

Table 3.1: Compact comparison of optimization approaches.

Approach category	**Main control mechanism**	**Typical strength**	**Main trade-off relevant to this dissertation**
Optimizing compilers	Compiler heuristics and optimization flags	Automatic optimization with little user effort	Transformation decisions are usually hidden from the developer
OpenMP/OpenACC directives	Source-level directives and pragmas	Portable way to express parallelism and selected transformations	Limited to constructs supported by the standard and compiler implementation
Annotation-based empirical tuning	Structured source annotations and search parameters	Can generate and evaluate many optimized variants	Often centered on empirical search and specific input languages
Domain-specific languages	Separate algorithm and schedule in a DSL	Strong optimization model for selected domains	Existing Fortran applications may need to be rewritten
Pattern/rewrite tools	Pattern rules, directives, or rewrite recipes	Can encode domain-specific transformations	Limited by recognized patterns and available legality checks
Scriptable source-to-source tools	External transformation scripts	Transparent, reproducible, and composable optimization steps	Requires accurate program representation and careful legality checking

Table 3.2: Detailed comparison of source-level and directive/script-based tools (part 1 of 2).

Tool or approach	**Input language**	**Transformation control mechanism**	**Supported loop transformations**	**Fortran support**	**Scriptability**
OpenMP 5.x loop transformations	C, C++, Fortran	Standard directives such as `tile` and `unroll`	Tiling and unrolling in the standard loop transformation model	Yes	Limited; directives are embedded in source
OpenACC	C, C++, Fortran	Directives for accelerator programming	Mainly parallelism and accelerator mapping rather than general loop rewriting	Yes	Limited; directive-based
Orio	Primarily C; future Fortran support discussed	Structured annotations and empirical search	Unrolling, tiling, permutation, unroll/jam, scalar replacement, register tiling	Limited in the described system	Yes, through annotations and tuning parameters
X Language	C/C++	Pragmas, transformation directives, pattern rewriting, macro language	Unrolling, strip-mining, scalarization, parameterized transformations	No direct Fortran focus	Yes, through transformation descriptions
Scout	C	Pragmas and configurable AST-based transformation	Unrolling, loop simplification, partial vectorization, SIMD-oriented rewrites	No	Configuration-based, not general transformation scripting
EPOD	C and Fortran	Pattern-oriented directives and EPOD scripts	Pattern-specific transformations including tiling, fission, peeling, layout changes	Yes	Yes, through EPOD scripts
OptiTrust	C and some C++ features	OCaml transformation scripts	Fission, fusion, unrolling, tiling, data-layout transformations	No	Yes
Clava	C, C++, OpenCL	LARA transformation scripts over a Clang-based AST	Interchange, tiling, instrumentation, and user-defined source transformations	No direct Fortran support	Yes
Xevolver	C and Fortran	XML AST and XSLT translation recipes	User-defined rewrites, including loop interchange examples	Yes	Yes, through XSLT recipes
Loopy	Kernel representation in Python; generates C/OpenCL C	Python API and embedded transformation language	Tiling, unrolling, fusion, permutation, vectorization, scheduling	Not direct Fortran source transformation	Yes
CHiLL	Primarily loop-oriented source representations	High-level transformation scripts over polyhedral representation	Permutation, tiling, unroll-and-jam, splitting, fusion, distribution	Related to Fortran loop transformation contexts	Yes
ASSIST	C, C++, Fortran	Directives or Lua scripts over ROSE AST	Interchange, unroll, strip mine, tile, specialization, vectorization support	Yes	Yes
POET	C, C++, Java, Fortran, DSLs	POET transformation scripts and syntax descriptions	Interchange, blocking, fusion/-fission, parallelization, scalar replacement, vectorization	Yes	Yes

Table 3.3: Detailed comparison of source-level and directive/script-based tools (part 2 of 2).

Tool or approach	**Source-to-source output**	**Transparency of transformations**	**Manual parameter control**	**Suitability for existing Fortran codebases**	**Main strength**	**Limitation relevant to this dissertation**
OpenMP 5.x loop transformations	Compiler-dependent generated code, not mainly source-to-source	Medium; intent is visible, final compiler transformation may not be	Yes, for directive parameters such as tile sizes or unroll factors	High when compiler support is available	Standardized and portable mechanism for selected loop transformations	Less flexible for custom multi-step transformation pipelines
OpenACC	Compiler-dependent generated code	Medium	Some parameter control through clauses	High for accelerator-oriented projects	Practical path to accelerator execution without full rewrite	Not primarily a general loop transformation framework
Orio	Yes	High for annotated regions and generated variants	High; explores parameter values empirically	Limited for direct Fortran transformation	Strong empirical tuning and variant generation	Primarily C-focused and search-oriented
X Language	Yes	Medium to high	High; supports empirical parameter search	Low	Compact representation of parameterized optimized programs	Focused mainly on C/C++ and selected optimization styles
Scout	Yes	Medium; vectorization is explicitly requested	Some, through configuration	Low	Strong source-to-source SIMD vectorization for C	Narrower focus than general loop transformation
EPOD	Yes	Medium to high	Yes, depending on pattern scripts	Medium to high	Encodes domain-specific optimization knowledge	More pattern-oriented than general user-selected loop transformation
OptiTrust	Yes	High; supports step-by-step diffs	High	Low for direct Fortran use	Very strong transparency and reviewability of optimization scripts	Does not target Fortran
Clava	Yes	High; transformations are expressed through LARA scripts and join points	High	Low for direct Fortran use	Strong LARA-based source-to-source framework with AST rewriting and code generation	Targets C/C++ and OpenCL rather than Fortran, but is conceptually close through LARA and script-controlled transformation
Xevolver	Yes	High, but through XML representation	Yes, if encoded in recipes	Medium	Separates platform-specific translations from application code	XML/XSLT can be verbose for complex transformations
Loopy	Yes, generates C/OpenCL C	High within Loopy's kernel model	High	Low to medium; requires representing kernels in Loopy	Powerful programmable control of generated kernels	Existing Fortran code must be adapted to another model
CHiLL	Yes	High at transformation-script level	High	Medium	Strong composition of high-level loop transformations	Requires polyhedral-style representation and constraints
ASSIST	Yes	Medium to high	Yes	High	Combines source transformation with profiling information	Broader performance-engineering tool, not only focused on loop transformation artifact design
POET	Yes	High for scripted transformations	High	Medium to high	Language-neutral programmable transformation framework	Requires syntax descriptions and general transformation infrastructure

Table 3.4: Comparison of DSL-based systems.

Tool	**Host or input model**	**Main control mechanism**	**Supported optimization style**	**Source-to-source relation**	**Main strength**	**Main trade-off for existing Fortran projects**
Halide	C++ embedded DSL for image-processing pipelines	Separation of algorithm and schedule	Tiling, fusion, vectorization, parallelization, storage/recomputation decisions	Generates optimized lower-level code from DSL representation	Very strong schedule control for image-processing pipelines	Existing Fortran code must be rewritten into Halide's model
Tiramisu	C++ embedded DSL with polyhedral representation	Scheduling commands over iteration domains and buffers	Tiling, splitting, fusion, shifting, vectorization, parallelization, GPU mapping, communication	Generates optimized backend code, including LLVM IR and CUDA paths	Fine-grained control for complex data-parallel computations	Existing Fortran code must be expressed in Tiramisu's representation

Chapter 4

Methodology

This chapter describes how the proposed source-to-source loop transformation support was designed, validated, and evaluated. It focuses on three aspects: the selection of transformation operations, the criteria used to reject unsafe cases, and the evaluation of transformed Fortran programs on benchmarks.

4.1 Research Approach

The research is structured around the development and assessment of controlled source-to-source loop transformations for Fortran. The goal is not to replace optimizing compilers or to automatically search for the best optimization sequence. Instead, the work studies whether explicit transformation control can support a practical workflow for applying, validating, and comparing loop transformations.

The scope is limited to Fortran `DO` loops. The evaluation focuses on loop structures present in the selected PolyBench/Fortran benchmarks.

The general research workflow consisted of the following steps:

- Identify the Fortran loop structures that must be represented in the METAFOR AST.
- Extend the dumping and AST construction stages where required, so that these structures are available to the transformation layer.
- Expose the required loop nodes and properties to the Fortran-JS interface.
- Design source-to-source algorithms for the selected loop transformations.
- Define conservative legality checks for transformation candidates.
- Generate transformed Fortran code from the modified AST.
- Validate the transformed programs on small benchmark datasets.
- Evaluate execution time and transformation success on larger benchmark datasets.

The artifact was considered successful when it could detect a candidate loop structure, check whether the requested transformation was safe according to the implemented rules, rewrite the AST, regenerate compilable Fortran code, and preserve the output of the original program. Performance improvement was also measured, but it was not the only criterion. In this dissertation, loop transformation is treated as a foundation for further optimization, including later parallelization and more advanced tuning.

4.2 Transformation Design

Loop transformations were selected because loops dominate the execution time of many scientific and high-performance computing programs. In Fortran programs, especially those based on arrays and numerical kernels, a large part of the computation is expressed as nested `DO` loops. Changing the structure of these loops can affect memory locality, loop overhead, compiler optimization opportunities, and the suitability of the code for later parallel execution.

The implemented transformations are loop fusion, loop fission, loop interchange, loop tiling, and loop unrolling. These transformations were chosen because they are well-known, practical, and representative of different optimization goals. They also cover different types of AST rewriting. Some transformations merge or split loops, while others change loop nesting, introduce new loop levels, or duplicate loop bodies.

The transformations are exposed as user-controlled operations in the Fortran-JS layer. A user selects a region of the program and requests a transformation. The framework then searches for matching `DO` loop structures inside that region. If a loop satisfies the required preconditions and legality checks, the AST is rewritten. If the loop is unsafe or unsupported, the transformation is not applied and the original code is preserved.

4.2.1 Selected transformations and their goals

The following table summarizes the transformations considered in this work and the main reason for including each of them.

Table 4.1: Loop transformations and their main purpose in the methodology.

Transformation	**Main purpose in the methodology**
Loop fusion	Merge adjacent loops with compatible iteration spaces to reduce loop overhead and possibly improve locality between producer and consumer statements.
Loop fission	Split a loop into several loops to separate independent computation stages and expose simpler loop bodies for later optimization.
Loop interchange	Swap the order of nested loops to improve memory access order or expose a more suitable loop structure.
Loop tiling	Introduce block-based traversal of nested loops to improve cache reuse and enable comparison with OpenMP-based tiling.
Loop unrolling	Duplicate the loop body by a fixed factor to reduce loop-control overhead and increase the amount of computation performed per iteration.

The transformations were not combined in the experimental evaluation. Each transformation was evaluated separately. However, the framework design does not prevent users from building transformation pipelines. For example, a user could apply loop fission first and then apply tiling to one of the resulting loops. This was not treated as an automatic search problem in this dissertation. Choosing the best transformation sequence remains the responsibility of the user.

The transformation parameters were also selected manually. The tile size used for loop tiling was `32`, and the unrolling factor used for loop unrolling was `4`. These values were selected to demonstrate that the framework can support parameterized transformations. They were not the result of an automatic tuning process.

4.2.2 General transformation procedure

The transformation procedure follows the same general structure for all implemented transformations. The user specifies the transformation kind and the target program region. The framework then traverses the AST, searches for candidate loops, checks preconditions, applies the transformation when legal, and regenerates source code.

Algorithm 1 Controlled source-to-source loop transformation.

Require:
ast — AST of the original Fortran program
target_region — program, subroutine, function, or loop selected by the user
transformation — requested transformation kind
parameters — transformation parameters, such as tile size or unroll factor
Ensure:
transformed_ast — AST after applying the requested transformation where legal
report — information about applied and rejected transformations

procedure APPLYTRANSFORMATION(*ast*, *target_region*, *transformation*, *parameters*)
  candidates ← FINDDOLOOPS(*target_region*, *transformation*)
  for each *loop_candidate* **in** *candidates* **do**
    if ¬ MATCHESREQUIREDSHAPE(*loop_candidate*, *transformation*) **then**
      mark *loop_candidate* as rejected
      continue
    end if
    if ¬ PASSESLEGALITYCHECKS(*loop_candidate*, *transformation*) **then**
      mark *loop_candidate* as rejected
      continue
    end if
    new_subtree ← REWRITEAST(*loop_candidate*, *transformation*, *parameters*)
    REPLACESUBTREE(*ast*, *loop_candidate*, *new_subtree*)
    mark *loop_candidate* as transformed
  end for
  return *ast* and *report*
end procedure

This procedure reflects the main methodological decision of the work. The framework does not silently change the program in uncertain cases. If the required loop shape is not present, or

if the legality checks cannot prove that the rewrite is safe, the transformation is rejected. In the user-facing result, this is reported as a transformation that was not applied.

4.2.3 Legality checking strategy

Correctness is the main risk in loop transformation. A transformation that changes the order of reads and writes can produce a program that still compiles but computes a different result. For this reason, each transformation is guarded by a set of legality checks.

The implemented legality checking strategy is conservative and rule-based. It is not a complete data dependency analysis. A full dependence analysis would need to reason about all array subscripts, scalar variables, procedure calls, aliases, pointers, and possible side effects. Such an analysis is outside the scope of this dissertation. Instead, the artifact implements checks that were required by the benchmark analysis and by the unsafe patterns encountered during development.

Several kinds of checks are used:

- **Syntactic loop-shape checks.** These checks verify whether the loop structure has the form required by the transformation. For example, loop interchange requires a perfectly nested pair of loops, loop fusion requires two adjacent loops with compatible bounds, and loop unrolling is applied only to innermost loops.
- **Array read/write comparison.** The transformation inspects statements in the candidate region and records which arrays are written and which arrays are read. If a transformation would move a write before a read that originally occurred earlier, or would cause a read to observe a partial value, the candidate is rejected.
- **Iterator-based subscript checks.** These checks compare how loop iterators appear in array subscripts. They are used to detect patterns such as transposed accesses, writes to the same element across all iterations, and reads that span a range of elements that may be updated by another loop.
- **Scalar dependency checks.** These checks detect scalar variables that are written in one statement and read by a later statement in a way that depends on the original loop order. This is especially important for loop fission, where scalar temporaries can lose their per-iteration meaning after the loop is split.
- **Reduction rejection.** Reductions were treated as unsafe in this work. Although reductions can sometimes be transformed legally, they require special handling to preserve numerical and semantic correctness. The current methodology therefore rejects them for safety.

The checks are intentionally stricter than a complete compiler dependence analysis would be. Some transformations that are legal in theory may be rejected by the framework. This is an acceptable trade-off for the current artifact because preserving program semantics is more important than maximizing the number of transformed loops.

The current checks operate only inside the selected loop nest. For loop fusion, the framework considers adjacent loops, because fusion is only meaningful when the loops can be merged without moving unrelated code across the new loop boundary. Procedure calls, I/O statements, and pointer-like constructs were not handled with specific legality rules because they did not appear in the benchmark cases used for this work. In future work, these constructs should be treated conservatively as possible side effects unless more precise information is available.

The following pseudocode summarizes the conservative legality strategy.

Algorithm 2 Conservative legality checking.

Require:
candidate — loop or loop nest selected for transformation
transformation — requested transformation kind
Ensure:
true if the candidate is considered safe, **false** otherwise

```
1:  function PassesLegalityChecks(candidate, transformation)
2:      if candidate contains an unsupported loop shape then
3:          return false
4:      end if
5:      reads, writes ← CollectArrayReadsAndWrites(candidate)
6:      scalars ← CollectScalarDefinitionsAndUses(candidate)
7:      if DetectsReductionPattern(candidate, reads, writes) then
8:          return false
9:      end if
10:     if DetectsUnsafeArrayDependence(candidate, reads, writes) then
11:         return false
12:     end if
13:     if DetectsUnsafeScalarDependence(candidate, scalars) then
14:         return false
15:     end if
16:     if transformation requires rectangular bounds and candidate contains triangular or
        variable-dependent bounds then
17:         return false
18:     end if
19:     return true
20: end function
```

This rule-based design matches the practical goal of the dissertation. The objective is not to prove every possible legal transformation. The objective is to provide a safe and understandable transformation mechanism that can be used, inspected, and extended.

Chapter 5

Implementation

This chapter describes the implementation of the source-to-source transformation support developed for Fortran DO loops. The implementation was built inside the METAFOR framework, which uses the LARA framework as its metaprogramming engine. The main goal of the implementation was not to build a complete optimizing compiler, but to make loop transformations available as explicit and controllable source-level operations. In this form, a user can select the part of the program to transform, choose the transformation, define parameters such as tile size or unrolling factor, and inspect the generated Fortran code afterwards.

The chapter is organized around three implementation aspects. First, it presents the architecture of the transformation pipeline and explains how Fortran code is parsed, represented, exposed to scripts, modified, and generated again. Second, it describes the five loop transformations implemented in the project: loop interchange, loop fusion, loop fission, loop unrolling, and loop tiling. Third, it explains how the implementation was improved through several correctness iterations, where unsafe transformations were identified on benchmark programs and then guarded by conservative legality checks.

The implementation focuses on Fortran DO loops because these loops are the central structure in many scientific kernels. The transformations were implemented as source-to-source rewrites over the program representation, and the generated output remains Fortran code. This makes the approach suitable for experimentation with existing Fortran programs, while still keeping the transformed code separate from the original source.

5.1 Architecture Overview

The implementation was developed as an extension of the METAFOR framework. METAFOR provides the infrastructure needed to parse Fortran programs, represent them as an Abstract Syntax Tree (AST), expose the program structure through join points, and manipulate these join points from transformation scripts. The framework uses the LARA framework under the hood, which provides the metaprogramming model used to navigate and rewrite the program.

The transformation pipeline consists of four main stages:

1. **Flang-22 and the `flang-dumper` plugin** parse the input Fortran source code and dump the internal representation as JSON.
2. **The Java AST module** reads the dumped JSON and builds the Fortran-transpiler AST.
3. **The Java Weaver** exposes AST nodes through a join-point model, so that they can be inspected and modified from scripts.
4. **Fortran-JS**, implemented with Node.js and TypeScript, provides the scripting interface used to write transformation passes.

This separation makes the architecture easier to extend. Each layer has a well-defined responsibility. Flang-22 and the dumper are responsible for extracting source code information. The Java AST module is responsible for building the tree representation. The Weaver is responsible for exposing this tree through join points. Fortran-JS is responsible for making these join points usable from scripts. Finally, the code generation stage traverses the modified AST and emits Fortran source code again.

A simplified view of the pipeline is shown below.

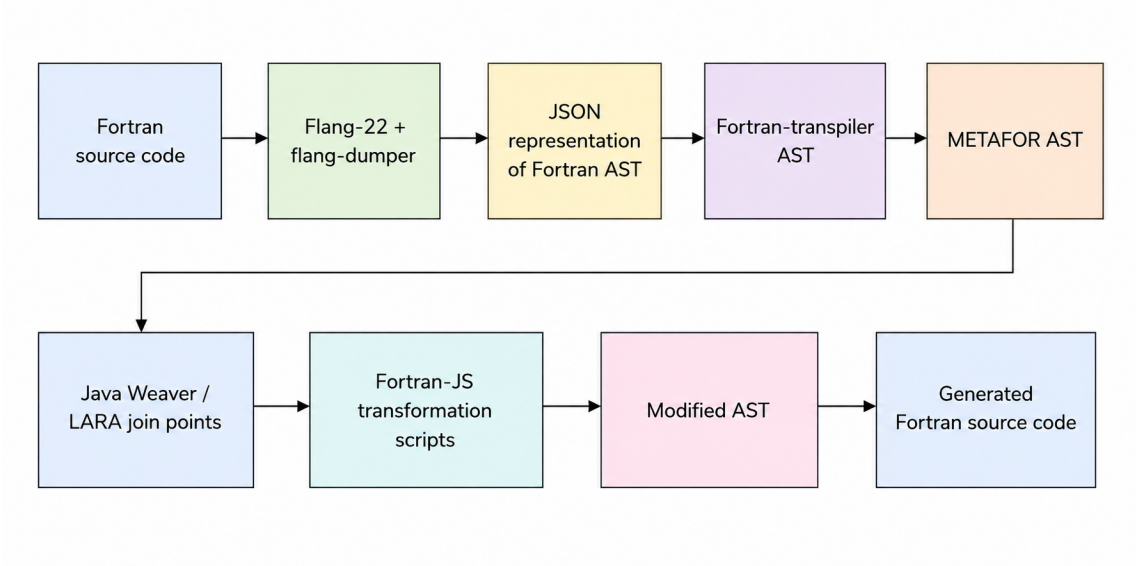

Figure 5.1: Metafor transformation pipeline overview.

The initial METAFOR infrastructure already provided the main components of this pipeline. However, not all Fortran language structures needed for loop transformations were fully supported. A large part of the implementation therefore consisted of extending the available node information and exposing it to the scripting layer. In particular, the `flang-dumper` plugin was extended to expose loop controls, array subscripts, scalar variables, and expression trees. These properties were required to recognize transformable loops and to implement legality checks.

This architecture also helped during debugging. When a transformed benchmark produced an incorrect result, the source of the problem could usually be located in one of the pipeline layers.

Some issues came from missing data in the dumper output. Others came from incomplete AST exposure through the Weaver. Several correctness failures were caused by transformation scripts that did not yet reject unsafe loop structures. In a smaller number of cases, the problem was in the code emitter, where the generated Fortran did not preserve the intended expression structure.

5.2 Parsing, AST Representation, Weaving, and Code Generation

The implementation starts by parsing the original Fortran program with LLVM Flang-22. Flang-22 provides a modern Fortran frontend and is used as the basis for obtaining a structured representation of the source program. However, the raw compiler representation is not directly convenient for writing source transformations. For this reason, a custom `flang-dumper` plugin is used to export the relevant program information in JSON format.

The dumper is an important part of the pipeline because it decides which parts of the compiler representation become available to the rest of METAFOR. During the implementation, the dumper had to be adjusted to correctly extract information from Fortran nodes with more complex internal structure. This was especially important for loop transformation, because the transformation scripts need access to loop header, loop variables, array references, and expressions. Without this information, the scripting layer can see that a loop exists, but it cannot decide whether it is safe to transform it.

After the JSON representation is produced, the Java AST module builds the Fortran-transpiler AST. Each supported node has its own representation and its own generation rules. For example, a `DO` loop node stores information about its control variable, initial value, terminal value, step, and body. Assignment nodes store the left-hand side and right-hand side expressions. Array access nodes store the array name and its subscripts. This structure allows transformations to work at the level of program elements instead of raw text.

To make AST manipulation available to users, the supported nodes and their methods must be exposed through the Weaver. This exposure is done through the join-point model, an abstraction layer over the AST that presents program elements as scriptable objects instead of raw AST nodes. A join point represents a relevant program element, such as a subroutine, a statement, an assignment, or a loop. The user script can search for these join points, inspect their properties, and apply transformations to them.

The exposure process contains several steps. The properties that must be visible at the scripting level are first described in XML files. After the XML structure is complete, FortranWeaver automatically generates TypeScript interfaces and abstract classes. These generated elements define what the TypeScript side can see and call. However, the connection to the real Java AST nodes still requires concrete classes. For each abstract join-point class, a concrete implementation must manually link the TypeScript interface to the underlying AST representation.

Once the required join points are available, transformations can be implemented in Fortran-JS. A typical transformation follows the same general workflow:

Algorithm 3 General transformation flow.

```
for each selected program region do
    search for candidate DO loops
    for each candidate loop do
        check whether the loop satisfies the transformation preconditions
        if the checks succeed then
            rewrite the corresponding AST subtree
        else
            preserve the original loop unchanged
        end if
    end for
end for
generate Fortran code from the final AST
```

This workflow is intentionally conservative. If a transformation cannot prove that a loop structure is safe according to the implemented rules, it does not apply the transformation. This does not mean that the transformation would be illegal in all possible cases. It only means that the current implementation cannot safely handle that case. This design choice was important because the goal was to produce correct transformed programs, even if that required rejecting some potentially valid transformations.

The final stage is code generation. After the scripts modify the AST, the tree is traversed and each node generates its corresponding Fortran source code. Since the project uses source-to-source compilation, the output is still Fortran. The generated code can then be compiled with Flang-22 and compared against the original program.

5.3 Implemented Loop Transformations

Five loop transformations were implemented: loop interchange, loop fusion, loop fission, loop unrolling, and loop tiling. These transformations were selected because they are common in compiler optimization and because they cover different forms of loop restructuring. Some transformations change the order of iteration, some change loop granularity, and some split or merge loop bodies.

The transformations are implemented in the Fortran-JS layer. They operate on join points that represent Fortran ‘DO‘ loops and related statements. Each transformation has its own implementation file and, where appropriate, a pass class that applies the transformation over a selected program region. The implemented transformations are summarized below in the Table 5.1.

The generic transformation scripts target benchmark kernel subroutines. In the PolyBench/-Fortran programs, these subroutines follow a naming pattern where the subroutine name starts with `kernel_`. This made it possible to apply transformations to relevant computation regions using queries such as:

```
1 Query.search(Subroutine, $jp => $jp.moduleName.startsWith("kernel_"))
```

Table 5.1: Summary of the implemented loop transformations.

Transformation	**Implementation**	**Main action**
Loop interchange	`LoopInterchange.ts`, `LoopInterchangePass`	Swaps the headers of a two-level perfect loop nest while keeping the body unchanged.
Loop fusion	`LoopFusion.ts`, `LoopFusionPass`	Merges two adjacent loops with compatible bounds into one loop.
Loop fission	`LoopFission.ts`, `LoopFissionPass`	Splits a loop with multiple statements into several loops.
Loop tiling	`LoopTiling.ts`, `LoopTilingPass`	Adds strip-mine loops and point loops around a two-level nest, using tile size 32.
Loop unrolling	`LoopUnroll.ts`, `LoopUnrollPass`	Duplicates the body of an innermost loop by factor 4 and adds a cleanup loop.

This search strategy keeps the transformation focused on computational kernels instead of initialization, printing, or validation code. It also reflects the general design of the framework: the user selects a region of interest, and the transformation is then applied only inside that region.

The framework does not automatically choose the best transformation sequence or parameter values. For example, the tile size and unrolling factor are selected by the user or by the experiment script. In this implementation, the tile size was fixed at 32 and the unrolling factor was fixed at 4. This was sufficient for demonstrating the feasibility of the transformation mechanism and for evaluating correctness across benchmark programs. More advanced automatic tuning can be added later as a separate layer on top of the current transformation support.

5.4 Loop Interchange

Loop interchange changes the order of two nested loops. In this implementation, the transformation is applied to two-level perfectly nested `DO` loops. The body of the loop nest is preserved, while the headers of the outer and inner loops are swapped. This transformation can improve memory access order or expose more useful loop structures for later transformations, but it is only safe when the new iteration order preserves the semantics of the original program.

The implementation is located in `LoopInterchange.ts`. Unlike the other transformations, it does not use a dedicated pass class in the same way. It is implemented through direct AST manipulation. The transformation identifies a candidate loop nest, checks its preconditions, and then rewrites the loop structure.

5.4.1 Preconditions

The implemented loop interchange requires a perfect two-level loop nest. This means that the outer loop body must contain the inner loop directly, without additional statements before or after it. This restriction keeps the transformation simple and avoids cases where statements outside the inner loop would need to be moved or duplicated.

The second precondition is that the inner loop header must not depend on the outer loop variable. Such a structure is often called a triangular loop nest. If the inner loop header references the outer variable, swapping the loops can move that reference to a position where the variable is not yet defined.

For example, consider the following simplified pattern:

```
! Original
do j = 1, maxgrid
   do i = j, maxgrid
     do cnt = 1, length
       diff(cnt, i, j) = sumTang(i, j)
     end do
   end do
end do
```

The inner initial value depends on `j`, which is the outer loop variable. If the loop headers are swapped directly, the generated code becomes unsafe:

```
! Incorrect after interchange
do i = j, maxgrid
  do j = 1, maxgrid
    do cnt = 1, length
      diff(cnt, i, j) = sumTang(i, j)
    end do
  end do
end do
```

Here, `j` is used in the initial value of the new outer loop, but it is not defined at that point. Depending on the compiler and runtime state, this can lead to incorrect output or a crash.

The third precondition is that nested loops inside the body must not use the original outer loop variable in their loop header. This is more subtle. The two swapped loops may have rectangular bounds, but a third loop inside the body may still depend on the old outer variable. After interchange, the expression may still be syntactically valid, but the evaluation order changes.

The `trmm` benchmark exposed this case. A simplified version is shown below.

```
do i = 2, ni
  do j = 1, ni
    do k = 1, i - 1
      b(j, i) = b(j, i) + alpha * a(k, i) * b(k, j)
    end do
  end do
end do
```

The bounds of `i` and `j` are rectangular, but the inner `k` loop depends on `i`. If `i` and `j` are interchanged, the order in which elements of `b` are read and written changes. Some values may be read before the corresponding earlier update has occurred. For this reason, the transformation rejects such cases.

5.4.2 Transformation algorithm

Algorithm 4 Loop interchange: legality check and rewrite.

Require: candidate outer `DO` loop
if outer loop does not contain exactly one nested `DO` loop as its body **then**
    reject and keep the loop unchanged
end if
$v_{\text{outer}} \leftarrow$ outer loop variable
$v_{\text{inner}} \leftarrow$ inner loop variable
if inner loop bounds reference v_{outer} **then**
    reject and keep the loop unchanged
end if
for all descendant `DO` loops inside the inner body **do**
    if any bound references v_{outer} **then**
        reject and keep the loop unchanged
    end if
end for
Swap the outer and inner loop headers.
Preserve the original loop body inside the new nesting order.
Regenerate the Fortran code.

5.4.3 Transformation Example

A safe interchange has the following general form.

```
! Before
do i = lo_i, hi_i
  do j = lo_j, hi_j
    body
  end do
end do
```

After transformation, the loop headers are swapped and the body remains inside the innermost loop.

```
! After
do j = lo_j, hi_j
```

```
    do i = lo_i, hi_i
      body
    end do
  end do
```

The transformation does not change the statements in `body`. It only changes the order in which the iteration space is traversed. Therefore, the legality checks focus on whether this new traversal order preserves the dependencies of the original program.

5.5 Loop Fusion

Loop fusion merges two adjacent loops into a single loop. The main purpose of fusion is to reduce loop overhead and, in some cases, improve locality by executing operations over the same iteration close together. However, fusion can also change the time at which values are produced and consumed. For this reason, the implementation applies fusion only when the two loops have compatible headers and when no unsafe dependency pattern is detected.

The implementation is located in `LoopFusion.ts` and is applied through `LoopFusionPass`. The pass searches for adjacent loops, checks whether their scopes are compatible, and then attempts to merge them. The function `_canFusePair()` performs the main legality checks.

5.5.1 Preconditions

The first precondition is structural. The loops must be adjacent, and they must have identical iteration boundaries. In practice, this means that the loops must have the same iterator, initial value, terminal value, and step. If the iteration spaces do not match, directly merging the loops would either skip iterations or execute some statements too many times.

The second group of preconditions concerns dependencies between the two loops. Three unsafe patterns were identified during correctness testing.

The first unsafe pattern is an incomplete reduction. In this case, one loop accumulates a value across its full iteration range, and the next loop reads that value. After fusion, the reading statement may see only a partial accumulation.

```
  ! Fusion variable: j
  do j = 1, n
    tmp(i) = tmp(i) + a(j,i) * x(j)
  end do

  do j = 1, n
    y(j) = y(j) + a(j,i) * tmp(i)
  end do
```

If these loops are fused, `tmp(i)` is read during the same `j` iteration in which it is still being built. The original program uses the final value of `tmp(i)`, while the fused version uses an intermediate value. The transformation therefore rejects this pattern.

The second unsafe pattern involves transposed subscripts. One loop writes an array as `X(inner_var, fusion_var)`, while the other loop reads it as `X(fusion_var, inner_var)`. This may look like a local access pattern, but it can create dependencies across different outer iterations.

```
! Fusion variable: i
do i = 1, n
  do j = 1, n
    a(j, i) = a(j, i) + u1(i) * v1(j)
  end do
end do

do i = 1, n
  do j = 1, n
    x(i) = x(i) + beta * a(i, j)
  end do
end do
```

The first loop writes a column of `a` for each `i`, while the second loop reads a row of `a` for each `i`. If the loops are fused, some row elements may be read before they have been updated by later column-writing iterations. The implementation rejects this pattern.

The third unsafe pattern is write-back before all reads are complete. In this case, one loop reads an array across a full inner range, while the second loop writes one element of the same array at the outer level. After fusion, the write in an earlier iteration may change a value that the first loop still needs in a later iteration.

```
! Fusion variable: p
do p = 1, np
  do s = 1, np
    sumA(p) = sumA(p) + a(s) * cFour(p,s)
  end do
end do

do p = 1, np
  a(p) = sumA(p)
end do
```

After fusion, `a(1)` may be overwritten during `p = 1`, but the first loop still needs to read `a(1)` when `p = 2`. This changes the meaning of the original program.

5.5.2 Transformation algorithm

Algorithm 5 Loop fusion: legality check and rewrite.

Require: selected program region
1: *pairs* ← pairs of adjacent `DO` loops in the region
2: **for all** $(L_1, L_2) \in$ *pairs* **do**
3: **if** L_1 and L_2 differ in iterator, lower bound, upper bound, or step **then**
4: **reject** this pair; **continue**
5: **end if**
6: analyze array reads and writes in the bodies of L_1 and L_2
7: **if** an implemented unsafe dependency pattern is found **then**
8: **reject** this pair; **continue**
9: **end if**
10: create a new loop L_f using the shared loop header
11: place the body of L_1 followed by the body of L_2 inside L_f
12: replace L_1 and L_2 with the fused loop L_f
13: **end for**
14: regenerate the Fortran code

The dependency checks are not a complete data dependence analysis. They are rule-based checks that cover the unsafe patterns encountered during benchmark analysis. This is consistent with the general implementation strategy: the framework should apply a transformation when the implemented rules can justify it, and it should leave the source unchanged otherwise.

5.5.3 Transformation Example

A simple safe fusion is shown below.

```
1  ! Before
2  do i = 1, n
3    a(i) = b(i)
4  end do
5
6  do i = 1, n
7    c(i) = d(i)
8  end do
```

The two loops have the same loop header and no dependency between the two statements. They can be merged into one loop.

```
1  ! After
2  do i = 1, n
3    a(i) = b(i)
4    c(i) = d(i)
```

```
5     end do
```

This transformation keeps the relative order of the two statements inside each iteration. The important difference is that all work for `i = 1` is now performed before the program moves to `i = 2`.

5.6 Loop Fission

Loop fission performs the opposite operation to loop fusion. It splits a loop body into several loops. This can be useful when different statements inside a loop would benefit from different transformations, or when separating statements makes later optimization easier. However, fission can be unsafe when statements communicate through scalar temporaries or when the original order between array reads and writes must be preserved across iterations.

The implementation is located in `LoopFission.ts` and is applied through `LoopFissionPass`. The pass searches for loops whose body contains at least two statements or statement groups. The function `canFission()` checks whether the loop can be safely split.

5.6.1 Preconditions

The first precondition is that the loop must contain multiple statements. A loop with a single statement cannot be split in a meaningful way.

The second precondition is that splitting must not break scalar temporary dependencies. If a scalar variable is written in one statement and then read in a later statement, the scalar value is often local to the current iteration. After fission, all iterations of the producer loop are executed before any iteration of the consumer loop. This changes the value seen by the consumer.

The `gramschmidt` benchmark exposed this pattern. A simplified version is shown below.

```
1     ! Original
2     do k = 1, nj
3       nrm = 0.0D0
4       do i = 1, ni
5         nrm = nrm + a(k, i) * a(k, i)
6       end do
7       r(k, k) = sqrt(nrm)
8     end do
```

Here, `nrm` is initialized, accumulated, and then consumed inside the same `k` iteration. If the loop is split into separate loops, the final use of `nrm` may see only the value left by the last iteration of the accumulation loop.

```
1     ! Incorrect after fission
```

```
  do k = 1, nj
    nrm = 0.0D0
  end do

  do k = 1, nj
    do i = 1, ni
      nrm = nrm + a(k, i) * a(k, i)
    end do
  end do

  do k = 1, nj
    r(k, k) = sqrt(nrm)
  end do
```

The third precondition concerns arrays that are read by an earlier statement and written by a later statement in the same loop. The later write produces the values that the earlier read needs on the next iteration. After fission, every earlier read runs before any later write, so the reads observe stale values.

```
  ! Original
  do t = 1, tmax
    ! S1: reads hz
    do i = 1, n
      e(i) = e(i) - 0.5 * (hz(i) - hz(i - 1))
    end do

    ! S2: writes hz
    do i = 1, n
      hz(i) = hz(i) - 0.7 * (e(i + 1) - e(i))
    end do
  end do
```

Within one time step, `S1` reads the `hz` values that `S2` produced in the previous time step. If the loop is split so that all `S1` iterations run before any `S2` iteration, `S1` reads `hz` before `S2` ever updates it, and every time step after the first sees stale values. Fission would therefore break the required order.

```
! After fission (wrong)
do t = 1, tmax ! all reads first
  do i = 1, n
    e(i) = e(i) - 0.5 * (hz(i) - hz(i - 1))
  end do
end do
```

```
  do t = 1, tmax ! all writes later - too late for the reads above
    do i = 1, n
      hz(i) = hz(i) - 0.7 * (e(i + 1) - e(i))
    end do
  end do
```

5.6.2 Transformation algorithm

The loop fission algorithm follows the steps below.

Algorithm 6 Loop fission: legality check and rewrite.

Require: candidate `DO` loop

if the loop body contains fewer than two statements or statement groups **then**
    reject and keep the loop unchanged
end if
for all statement groups in the loop body **do**
    collect scalar writes, array reads, and array writes
end for
if a scalar written in an earlier group appears in a later group **then**
    reject and keep the loop unchanged
end if
if a later group writes an array that an earlier group reads **then**
    reject and keep the loop unchanged
end if
for all statement groups G in the loop body **do**
    create a new loop L_G
    copy the original loop iterator, bounds, and step into L_G
    move the statements of G into L_G
end for
replace the original loop with the sequence of generated loops
regenerate the Fortran code

5.6.3 Transformation Example

A simple safe fission is shown below.

```
  ! Before
  do i = 1, n
    a(i) = b(i)
    c(i) = d(i)
  end do
```

The two statements are independent. They can be separated into two loops with the same iteration space.

```
! After
do i = 1, n
  a(i) = b(i)
end do

do i = 1, n
  c(i) = d(i)
end do
```

The transformation changes the order of execution. In the original program, both assignments for `i = 1` are completed before moving to `i = 2`. In the transformed program, all assignments to `a` are completed before any assignment to `c`. This change is safe only when the two statement groups are independent according to the implemented checks.

5.7 Loop Unrolling

Loop unrolling reduces loop-control overhead by duplicating the loop body several times inside a single iteration of a new loop. It can also expose more operations to the compiler, which may help instruction scheduling or vectorization. In this implementation, unrolling is applied to innermost `DO` loops and uses a fixed factor of 4.

The implementation is located in `LoopUnroll.ts` and is applied through `LoopUnrollPass`. The transformation creates a main loop with step 4 and a cleanup loop that handles remaining iterations when the total number of iterations is not divisible by 4.

5.7.1 Preconditions

The main precondition is that the target loop must be innermost. In other words, the loop body must not contain another `DO` loop. This restriction simplifies the transformation because only statements in the loop body need to be duplicated. It also avoids changing the structure of nested loop nests in ways that would require more complex dependence checks.

The transformation also assumes that the loop uses a simple `DO` control structure with a clear initial value, terminal value, and step. These values are needed to build the unrolled main loop and the cleanup loop.

5.7.2 Transformation algorithm

The unrolling algorithm is shown below.

Algorithm 7 Loop unrolling: legality check and rewrite.

Require: candidate innermost `DO` loop; unrolling factor F

if the loop body contains a nested `DO` loop **then**
    reject and keep the loop unchanged
end if
read the loop iterator v, initial value, terminal value, and step
generate a new main loop with step F
for $k \leftarrow 0$ **to** $F-1$ **do**
    duplicate the original loop body inside the main loop
    in the copy, replace each use of v with $v+k$
end for
generate a cleanup loop for the remaining iterations
replace the original loop with the main loop followed by the cleanup loop
regenerate the Fortran code

The cleanup loop is important because the number of iterations may not be a multiple of the unrolling factor. Without the cleanup loop, the transformed program would skip the last few iterations in such cases.

5.7.3 Transformation Example

For an unrolling factor of 4, the original loop has the following form.

```
! Before
do i = 1, n
  a(i) = b(i)
end do
```

After unrolling, the main loop executes four iterations at a time.

```
! After
do i = 1, n - 3, 4
  a(i) = b(i)
  a(i+1) = b(i+1)
  a(i+2) = b(i+2)
  a(i+3) = b(i+3)
end do

do i = n + 1 - MOD(n - (1) + 1, 4), n
  a(i) = b(i)
end do
```

The second loop handles the remaining iterations. The emitter fix mentioned earlier was important for this transformation because generated expressions must preserve the intended arithmetic grouping when index expressions are rewritten.

5.8 Loop Tiling

Loop tiling, also called blocking, restructures a loop nest so that the iteration space is processed in smaller blocks. This can improve cache locality when nearby iterations access nearby memory locations. In this implementation, tiling is applied to two-level nested ‘DO‘ loops and uses a fixed tile size of 32.

The implementation is located in `LoopTiling.ts` and is applied through `LoopTilingPass`. The transformation introduces two outer tile loops and two inner point loops. The original loop body is moved into the innermost point loop.

5.8.1 Preconditions

The first precondition is that the target structure must contain two nested loops. The implementation expects an ideal two-level nest where the body of the outer loop is the inner loop.

The second precondition is that the iteration space must be rectangular for the selected loop pair. The inner loop header must not depend on the outer loop variable. If the inner loop starts at the outer variable, tiling the two loops directly may lose the original triangular constraint.

For example:

```
! Original
do j = 1, maxgrid
  do i = j, maxgrid
    do cnt = 1, length
      diff(cnt, i, j) = sumTang(i, j)
    end do
  end do
end do
```

The condition `i >= j` is part of the original iteration space. A naive tiling transformation can replace the inner start with the tile-strip origin and fail to preserve this condition. It can also generate a tile loop header that references a variable not yet defined.

```
! Incorrect naive tiling
do jj = 1, maxgrid, 32
  do ii = j, maxgrid, 32
    do j = jj, MIN(jj + 32 - 1, maxgrid)
      do i = ii, MIN(ii + 32 - 1, maxgrid)
```

```
6           do cnt = 1, length
7             diff(cnt, i, j) = sumTang(i, j)
8           end do
9         end do
10      end do
11    end do
12  end do
```

Here, `j` appears in the initial value of `ii`, but `j` is not defined at that point. Also, the point loop does not enforce `i >= j`. The implementation therefore rejects triangular loop pairs.

The third precondition is similar to the one used for interchange. A nested `DO` loop inside the body must not use the outer tiled variable in its loop header. This case can preserve syntactic validity but change the required evaluation order. The `trmm` benchmark showed this pattern.

```
1   do i = 2, ni
2     do j = 1, ni
3       do k = 1, i - 1
4         b(j, i) = b(j, i) + alpha * a(k, i) * b(k, j)
5       end do
6     end do
7   end do
```

The `k` loop depends on `i`. If the pair `(i, j)` is tiled directly, tile ordering can make some reads of `b(k, j)` happen before the updates that the original order would have completed. For this reason, such cases are rejected for the selected loop pair.

5.8.2 Transformation algorithm

The tiling algorithm is implemented as follows.

Algorithm 8 Loop tiling: legality check and rewrite.

Require: candidate two-level `DO` loop nest; tile size T

detect the outer and inner loops; let their variables be v_{outer} and v_{inner}
if the inner loop bound references v_{outer} **then**
    reject and keep the loop nest unchanged
end if
for all descendant `DO` loops inside the inner body **do**
    if any bound references v_{outer} **then**
        reject and keep the loop nest unchanged
    end if
end for
generate the outer tile loop for the original outer dimension (step T)
generate the outer tile loop for the original inner dimension (step T)
generate the inner point loop for v_{outer}
generate the inner point loop for v_{inner}
bound the point loops with MIN expressions to handle boundary tiles
move the original loop body into the innermost point loop
replace the original loop nest with the tiled loop nest
regenerate the Fortran code

The checks use word-boundary regular expressions when searching for loop variables in loop header expressions. This avoids rejecting safe loops because a loop variable name appears as part of a larger identifier. For example, the variable ‘i‘ should not match the dimension name `ni`.

5.8.3 Transformation Example

A general two-level loop nest has the following form.

```
   ! Before
   do i = lo_i, hi_i
     do j = lo_j, hi_j
       body
     end do
   end do
```

After tiling with tile size `TILE`, the transformed code becomes:

```
   ! After
do ii = lo_i, hi_i, TILE
  do jj = lo_j, hi_j, TILE
    do i = ii, MIN(ii + TILE - 1, hi_i)
      do j = jj, MIN(jj + TILE - 1, hi_j)
        body
      end do
```

```
8     end do
9   end do
10 end do
```

The outer loops `ii` and `jj` iterate over tiles. The inner loops `i` and `j` iterate over points inside each tile. The `MIN` expressions ensure that the final tiles do not exceed the original loop header.

5.9 Iterative Correctness Engineering

Correctness was improved through three implementation iterations. Each transformation was applied to the PolyBench/Fortran benchmark suite using the small dataset configuration. A transformed benchmark was accepted only when it compiled successfully and produced array dumps matching the original program.

The first complete run showed that syntactically valid Fortran code can still be semantically unsafe. Some transformed programs compiled and executed, but changed the original order in which values were produced and consumed. For this reason, the failures were grouped by root cause and then converted into conservative legality checks.

5.9.1 Correctness Workflow

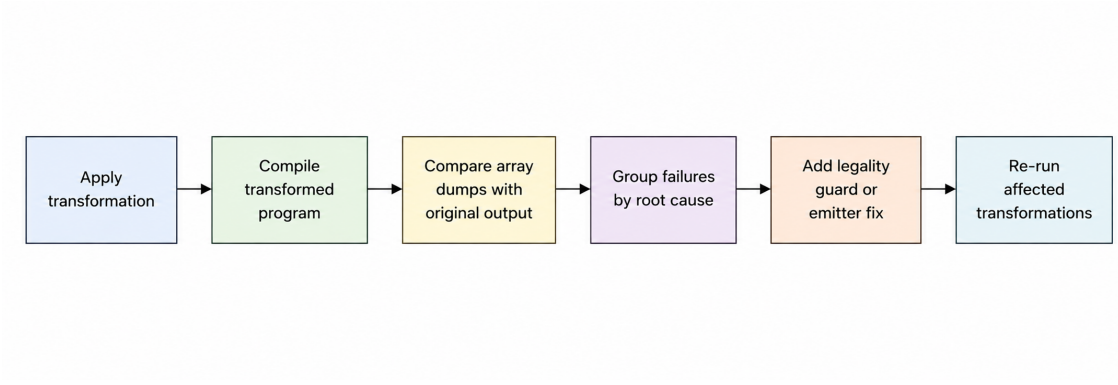

Figure 5.2: Correctness workflow

This workflow was used not only to find bugs, but also to turn observed benchmark failures into explicit rules that prevent unsafe transformations from being applied again.

5.9.2 Iteration Overview

Table 5.2: Overview of the three correctness iterations.

Iteration	Main focus	Main outcome
1	Baseline failure analysis	Unsafe dependency patterns were identified for fission, fusion, interchange, and tiling.
2	Interchange and fusion guards	Both transformations reached `30/30` matching benchmarks.
3	Tiling and fission fixes	All five transformations reached `30/30` matching benchmarks.

5.9.3 Correctness Progress

Table 5.3: Correctness progress across the three iterations (matching benchmarks out of 30).

Transformation	Iter. 1	Iter. 2	Iter. 3	Key change
Loop unrolling	`30/30`	—	—	Fixed expression parenthesization in the code emitter.
Loop interchange	`27/30`	**`30/30`**	—	Added `canInterchange()` checks for triangular bounds and nested loop bounds that depend on the outer variable.
Loop fusion	`27/30`	**`30/30`**	—	Added `_canFusePair()` checks for incomplete reductions, transposed accesses, and write-back-before-read patterns.
Loop tiling	`28/30`	`28/30`	**`30/30`**	Added `canTile()` checks and replaced plain substring matching with word-boundary regular expressions.
Loop fission	`21/30`	`21/30`	**`30/30`**	Fixed `Query.searchFrom` usage with `Query.searchFromInclusive`; added scalar and array dependency checks.

5.9.4 Main Failure Patterns and Fixes

Table 5.4: Main failure patterns and the implemented responses.

Finding	Affected transformation(s)	Examples	Implemented response
Producer-consumer order can be broken.	Fission	`gramschmidt`, `symm`, `adi`, `fdtd-2d`	Reject scalar threading and array write-before-read cases.
A consumer may read an incomplete value.	Fusion	`atax`	Reject reductions that are still being accumulated when the fused body would read them.
Array access direction matters.	Fusion	`gemver`	Reject transposed access patterns such as writing `A(j,i)` and reading `A(i,j)`.
Bounds can become invalid after reordering.	Interchange, tiling	`reg_detect`, `covariance`	Reject loop nests where the inner bound depends on the original outer variable.
Nested loop bounds can hide dependencies.	Interchange, tiling	`trmm`	Reject cases where descendant loop bounds depend on the original outer variable.
Naive string checks can reject safe code.	Tiling	`syrk`	Use word-boundary regular expressions, so loop variable `i` is not matched inside names such as `ni`.
AST queries must include the starting node when needed.	Fission	Direct assignment statements	Replace `Query.searchFrom` with `Query.searchFromInclusive` in dependency helper functions.

5.9.5 Lessons Learned

The correctness evaluation confirmed that structural loop rewriting alone is insufficient. Correct transformations also require legality checks that preserve data dependencies.

The correctness engineering process also showed that each loop transformation requires different legality checks. Loop fission was mainly limited by scalar and array dependencies, loop fusion by producer-consumer dependencies, and loop interchange and tiling by loop-bound constraints and nested-loop dependencies.

The evaluation supports the design decision to use conservative legality checks, as the implemented rules were sufficient to preserve correctness on the evaluated benchmark suite.

By the end of the third correctness iteration, all five transformations produced matching outputs for all thirty PolyBench/Fortran benchmarks on the small dataset. This demonstrates that the implemented legality checks were sufficient for the evaluated benchmark patterns, although not for arbitrary Fortran programs.

5.10 Implementation Summary

The implementation extended METAFOR with source-to-source support for five Fortran loop transformations. The work required changes across several layers of the framework. The `flang`

`-dumper` plugin was extended to expose the information needed by loop transformations. The Java AST and Weaver layers were used to represent and expose Fortran program elements. The Fortran-JS layer was used to implement transformation scripts, legality checks, and transformation passes. The code generation layer then emitted the modified AST back into Fortran source code.

A central implementation decision was to use conservative legality checks. The framework does not try to perform complete dependence analysis. Instead, it checks for concrete unsafe patterns that were observed during benchmark testing. When a transformation is not proven safe by these checks, the original loop is preserved. This makes the implementation practical for an artifact-oriented dissertation, where correctness and transparency are more important than applying every theoretically legal transformation.

The iterative correctness process was an important part of the implementation. The first version of the transformations revealed several unsafe patterns. These patterns were then converted into explicit checks for interchange, fusion, fission, and tiling. After three iterations, all five transformations produced matching results on all thirty PolyBench/Fortran benchmarks using the small dataset.

The final implementation demonstrates that loop transformations can be added to a Fortran source-to-source pipeline in a way that is visible, scriptable, and adjustable. The user can select the program region, choose the transformation, set relevant parameters, and inspect the resulting Fortran code. At the same time, the implementation remains a foundation rather than a complete production compiler. More advanced dependence analysis, broader Fortran language coverage, additional transformations, and automatic tuning can be added in future work.

Chapter 6

Evaluation

The previous chapter described the implementation and the iterative process used to develop conservative legality checks. This chapter evaluates the final artifact using the PolyBench/Fortran benchmark suite. The evaluation measures correctness, transformation applicability, and execution-time performance of the completed implementation.

6.1 Experimental Setup

The experiments were performed on 30 PolyBench/Fortran kernels. These benchmarks cover several common classes of scientific computation, including linear algebra kernels, linear algebra solvers, stencil codes, datamining kernels, and medley programs. Each benchmark contains one computational subroutine following the `kernel_*` naming pattern and includes a `!\$pragma scop / endscop` region around the main computation. This structure makes the suite suitable for evaluating source-to-source transformations on loop-heavy Fortran code.

All programs were compiled with:

```
1 $ flang-22 -O3 -fopenmp
```

The `-fopenmp` flag was kept active for both transformed and non-transformed programs, even when no parallel code was used. This choice keeps the compiler flags uniform across all experiments. The same optimization level was used for the original and transformed versions, so performance differences are caused by the generated source structure rather than by different compiler settings.

Correctness and performance were evaluated with different dataset sizes. Correctness was checked using `SMALL_DATASET`, which corresponds to `128 x 128` for matrix kernels. In this mode, `POLYBENCH_DUMP_ARRAYS` was enabled and the numerical output of the transformed program was compared against the original output using `compare.sh`.

Performance was measured using `LARGE_DATASET`, which corresponds to `2000 x 2000` for matrix kernels. In this mode, `POLYBENCH_TIME` was enabled and `POLYBENCH_DUMP_ARRAYS` was disabled. Array dumping was disabled to avoid measuring I/O overhead instead of kernel execution time.

It is important to distinguish the meaning of `MATCH` in the correctness and performance experiments. On `SMALL_DATASET`, `MATCH` means that the numerical array dump of the transformed program is equal to the original output. This is the main correctness guarantee. On `LARGE_DATASET`, array dumps are disabled, so `MATCH` only means that the program compiled, linked, executed without crashing, and produced the expected timing-output path after the timer string was stripped before comparison.

6.1.1 Hardware environment

The performance experiments were executed on a laboratory machine with the hardware configuration shown in Table 6.1.

Table 6.1: Hardware configuration used for performance experiments.

Component	Configuration
Processor	Intel Xeon E5-2630 v3 (Haswell-EP)
Sockets × cores	2 × 8 (16 physical cores total)
Base frequency	2.40 GHz (Turbo Boost disabled)
L1 cache	64 KiB per core
L2 cache	256 KiB per core
L3 cache	40 MiB total (2 × 20 MiB shared per socket)
NUMA configuration	2 nodes
Operating system	Ubuntu 26.04

Turbo Boost was disabled to remove one major source of variation in single-run measurements. However, the experiment did not control all possible sources of timing noise. Other power-management mechanisms, such as C-states and thermal behavior, may still have affected execution time. The machine was also not guaranteed to be fully idle. For this reason, the reported performance results should be interpreted as indicative measurements, not as a full statistical performance study.

6.1.2 Software environment

The software environment consisted of the METAFOR infrastructure, the Flang-22 compiler, and the runtime components needed by the AST and scripting layers. Table 6.2 summarizes the main software components.

Table 6.2: Software components used in the evaluation.

Software component	**Role**
Flang-22 (`-O3 -fopenmp`)	Compiles original and transformed Fortran programs and supports the dumping stage.
`flang-dumper` plugin	Extracts Fortran node information into a JSON-based intermediate representation.
METAFOR — Java AST + weaver	Builds the program AST and provides traversal and modification infrastructure.
Fortran-JS / Node.js	Exposes loop transformations as user-level scripts.
Java 21	Runtime for the AST and weaver components.
Node.js 22	Runtime for the transformation scripts.

Both original and transformed benchmark versions were compiled with the same Flang-22 configuration. This keeps the comparison focused on the generated source code and avoids introducing differences caused by compiler flags.

6.1.3 Experimental procedure

The same basic procedure was followed for each benchmark and transformation.

1. Preprocess the source code to remove macro interference.
2. Compile and run the original program on the small dataset.
3. Save the original array dump for correctness validation.
4. Select a target region for transformation.
5. Apply one requested loop transformation through METAFOR.
6. Generate the transformed Fortran source code.
7. Compile the transformed program with Flang22 and `-O3`.
8. Run the transformed program on the small dataset.
9. Compare the transformed array dump with the original array dump.
10. If validation succeeds, run the original and transformed versions on the large dataset.
11. Record the PolyBench kernel execution time.
12. Compute speedup and transformation success information.

Each transformation was evaluated independently. The framework is able to support transformation pipelines, but automatic pipelines were not part of the experiment. This choice made the results easier to interpret, because the effect of each transformation could be considered separately.

The reported value for each benchmark configuration corresponds to the measured kernel execution time from a single execution. Repeated execution, warm-up removal, and outlier filtering were outside the scope of this evaluation. The performance results therefore indicate the effect of the implemented transformations but should not be interpreted as a comprehensive statistical performance analysis.

Only kernel execution time was measured. This timing was already available through the PolyBench infrastructure. Full program execution time was not used because it would include setup, allocation, input/output, and other costs that are less directly related to loop transformation.

6.1.4 Performance Evaluation

Performance evaluation was used to study the practical effect of the generated transformations. The evaluation was not designed to prove that every transformation improves execution time. In fact, not every legal loop transformation is profitable. A transformation may preserve correctness and still lead to worse performance because of cache behavior, register pressure, code size, or interactions with compiler optimizations.

The main metrics were execution time, speedup, transformation success rate, and comparison with an OpenMP-based tiling alternative where applicable.

Execution time was measured using the kernel timing already included in the benchmark infrastructure. The original program and the transformed program were both compiled with Flang22 using `-O3`. The same large dataset was used for the original and transformed versions.

Speedup was computed as:

$$\text{Speedup} = \frac{T_{\text{original}}}{T_{\text{transformed}}} \tag{6.1}$$

where `T_original` is the execution time of the original benchmark kernel and `T_transformed` is the execution time of the transformed benchmark kernel. A speed-p greater than `1.0` means that the transformed version was faster.

Transformation success rate was used to describe how often the framework could apply a requested transformation and produce a valid result. It can be expressed as:

$$\text{Success Rate} = \frac{N_{\text{successful}}}{N_{\text{requested}}} \tag{6.2}$$

where `N_requested` is the number of transformation attempts and `N_successful` is the number of attempts that produced transformed code that compiled and passed correctness validation.

The results were intended to be analyzed in two ways. First, they can be grouped by transformation type. This shows which transformations were more often applicable and which ones produced better performance behavior. Second, they can be grouped by benchmark.

6.2 Correctness Results

Correctness was evaluated by comparing the numerical output of the transformed programs against the output of the original programs. The `SMALL_DATASET` configuration was used for this purpose because it allows the full array output to be dumped and compared without producing excessive files.

All five transformations achieved `30/30 MATCH` on `SMALL_DATASET`, confirming correct output for all evaluated benchmarks. During the performance experiments, all transformed benchmarks also compiled and executed without crashing on `LARGE_DATASET`, indicating that the transformations remained applicable under the performance configuration.

Table 6.3 summarizes the number of benchmarks transformed and rejected by each pass.

Table 6.3: Correctness and applicability of the five loop transformations.

Transform	**Transformed**	**Rejected**	**Result**
Loop unrolling	30	0	30/30 MATCH
Loop tiling	21	9	30/30 MATCH
Loop interchange	17	13	30/30 MATCH
Loop fission	10	20	30/30 MATCH
Loop fusion	8	22	30/30 MATCH

The results show that applicability varies strongly between transformations. Loop unrolling applies to all 30 benchmarks because every kernel contains at least one innermost simple ‘DO‘ loop. Loop fusion is the most selective transformation, with only 8 transformed benchmarks, because most kernels do not contain adjacent loops with compatible bounds or fail the conservative dependency checks. Tiling, interchange, and fission fall between these extremes. Their applicability depends on the presence of suitable two-level loop nests or multi-statement loop bodies.

6.3 Performance Results

Because each configuration was measured only once, small differences should be interpreted with caution. Speedups close to 1× may fall within measurement variability and should be regarded as indicative rather than statistically significant.

6.3.1 Loop Tiling

The strongest speedups were observed in memory-bound kernels where tiling improved cache locality. The best result was obtained for `mvt`, which reached a speedup of `4.52x`. This benchmark contains matrix-vector transpose access patterns that are not cache-friendly in the original traversal order. Tiling changes the traversal into smaller blocks, allowing a working set to remain resident in the lower cache levels for longer.

The second strongest result was `gemver`, with a speedup of `2.04x`. This kernel performs several matrix-vector operations. Tiling helps because two passes over a 2000-row matrix can be reorganized into blocked accesses. `floyd-warshall` also improved significantly, reaching `1.51x`. In this benchmark, the 2000 x 2000 adjacency matrix is around 32 MB and does not fit fully in cache. Tiling reduces cache misses by improving reuse around the `k` dimension.

Several dense linear algebra kernels also improved, but more modestly. The speedups were `1.22x` for `syrk`, `1.14x` for `syr2k`, `1.05x` for `3mm`, `1.04x` for `gemm`, `1.04x` for `2mm`, and `1.03x` for `symm`. These results indicate that tile size 32 may benefit some BLAS-like kernels, although the smaller speedups should be interpreted cautiously due to the measurement methodology.

Some benchmarks were nearly neutral. These include `correlation`, `covariance`, `seidel-2d`, `fdtd-apml`, `doitgen`, and `dynprog`, all of which stayed close to `1.00x`. In these cases, the selected tile size and the transformed loop structure did not substantially change the execution behavior.

The main regressions appeared in stencil and solver kernels. `jacobi-2d-imper` slowed down to `0.35x`, and `fdtd-2d` slowed down to `0.45x`. These stencil programs have dependencies between adjacent cells, and simple rectangular tiling of the outermost loops does not match the data reuse pattern well. Instead, it adds tile-boundary overhead without improving the relevant reuse distance. `lu` slowed down to `0.41x`. The applied tiling was limited to the perfectly nested rectangular loops inside the outer sequential loop. However, because the overall computation follows a triangular iteration pattern across successive outer-loop iterations, this locality optimization was ineffective and introduced additional tile-management overhead. Simple rectangular tiling cannot handle this non-rectangular iteration space efficiently without more advanced techniques such as skewed tiling. `trmm` reached `0.66x`; in this case, `canTile()` blocked the outermost `(i, j)` pair because the body contained `do k = 1, i-1`, so the tool tiled the inner `(j, k)` pair instead. This was safe, but it did not provide a locality benefit on the laboratory machine. `adi` and `reg_detect` had smaller regressions, with `0.89x` and `0.86x` respectively. The `reg_detect` benchmark is very short, around 39 ms in the baseline, so tile overhead can dominate the measured time.

The key result for tiling is that tile size `32` is a good fit for several BLAS-2 and BLAS-3 style kernels. A `32 x 32` block of 8-byte values occupies 8 KB, which fits well in L1 cache. Tiling is also particularly helpful for matrix-transpose access patterns such as those in `mvt` and `gemver`. However, the same transformation is not suitable for all loop structures. Stencil kernels require wavefront or time-skewing approaches, and triangular solvers require support for non-rectangular iteration spaces.

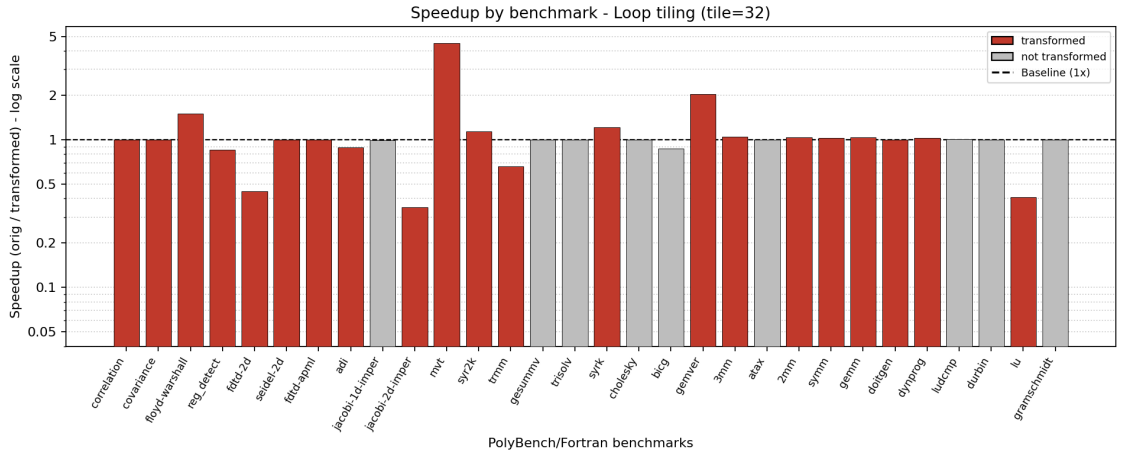

Figure 6.1: Loop Tiling Speedups

6.3.2 Loop Unrolling

Loop unrolling was applied with factor `4`. Unlike the other transformations, unrolling was applied to all 30 benchmarks, because each benchmark contains an innermost simple `DO` loop. All 30 benchmarks reported `MATCH` and are shown in Figure 6.2

The performance gains from unrolling were modest. This is expected because many large PolyBench kernels are limited by memory bandwidth rather than loop-control overhead. Removing three out of four loop-control operations can help instruction-throughput-limited kernels, but it does not substantially reduce the amount of data moved from memory. These speedups fall within the expected measurement variability and should be interpreted as indicative rather than statistically significant.

The main regressions occurred in kernels with recurrence-like behavior or short execution time. `adi` slowed down to `0.80x`, and `lu` slowed down to `0.81x`. These kernels contain loop-carried dependency chains, and unrolling four iterations can increase register pressure without exposing enough independent work. `dynprog` showed a similar pattern and reached `0.83x`. The `atax` and `reg_detect` benchmarks also regressed to `0.83x` and `0.79x`, respectively. Both are relatively short-running kernels, so cleanup-loop overhead is large compared with useful computation. `jacobi-2d -imper` slowed down to `0.88x` after loop unrolling.

Compared with tiling, unrolling is much weaker on memory-bound kernels. For example, `mvt` reached `0.99x` with unrolling but `4.52x` with tiling. `gemver` reached `0.98x` with unrolling but `2.04x` with tiling. `floyd-warshall` improved with both transformations, but tiling was stronger: `1.51x` compared with `1.05x`. On the other hand, unrolling is less risky than tiling in some cases. For `lu`, unrolling reached `0.81x`, while tiling reached only `0.41x`. This suggests that unrolling is best understood as a low-risk complementary transformation rather than a strong standalone optimization for this benchmark suite.

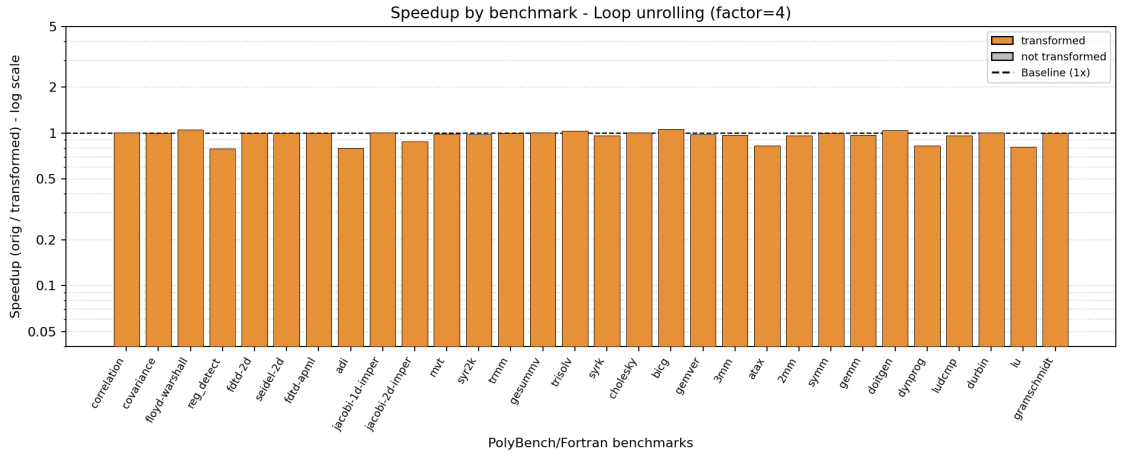

Figure 6.2: Loop Unrolling Speedups

6.3.3 Loop Fusion

Loop fusion was the most selective transformation in the evaluation. Only 8 of the 30 benchmarks were transformed, while 22 were left unchanged. A benchmark was not fused when it did not contain adjacent loops with compatible bounds, or when the legality checks in `_canFusePair()` blocked the merge. All 30 benchmarks reported `MATCH` and are shown in Figure 6.3

The only observed performance improvement from fusion was for `jacobi-2d-imper`, which reached `1.64x`. The transformation fused two loops over the two-dimensional stencil space, removing a full pass over the `ex` and `hz` arrays between two loop iterations, which is consistent with improved L2 reuse and reduced memory traffic. However, because some non-transformed benchmarks also exhibited similarly large apparent speedups, this result should be interpreted cautiously and not as conclusive evidence of the performance benefit of loop fusion.

The other results fall within the expected measurement variability and should be interpreted as indicative rather than statistically significant.

For benchmarks that were not transformed, the generated program is semantically identical to the baseline. Therefore, timing variation around `1.00x` should be interpreted as measurement noise rather than an effect of fusion.

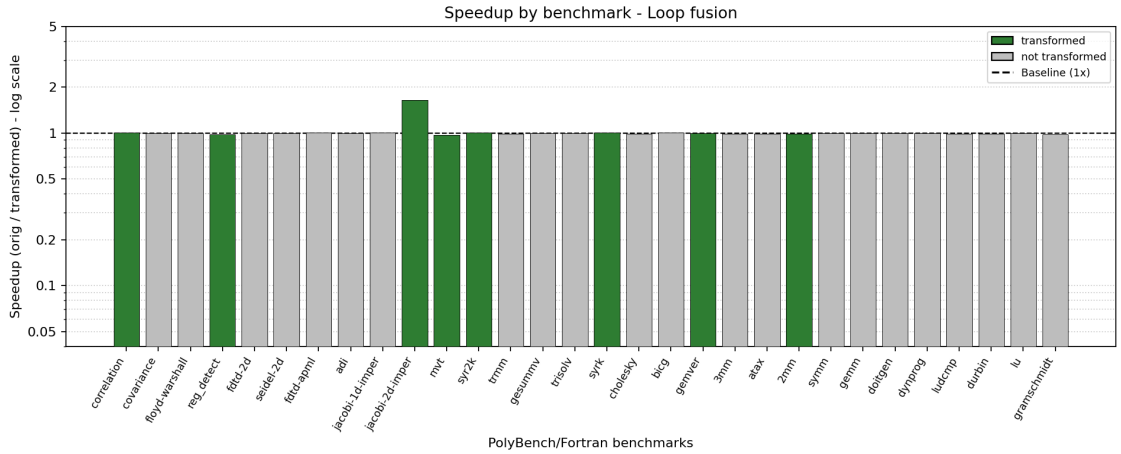

Figure 6.3: Loop Fusion Speedups

6.3.4 Loop Fission

Loop fission was applied to 10 of the 30 benchmarks. The other 20 benchmarks were left unchanged because no suitable fissionable loop was found, or because the legality checks rejected the split. All 30 benchmarks reported `MATCH` and are shown in Figure 6.4.

Although `reg_detect` showed small improvement `1.05x`, given the single-run measurement methodology, improvements of this magnitude should be considered statistically inconclusive rather than evidence of a performance benefit. Several transformed benchmarks were neutral: `2mm`, `3mm`, `doitgen`, `dynprog`, and `correlation` all stayed close to `1.00x`.

The main regressions occurred in accumulation kernels. `gesummv` and `bicg` both slowed down to `0.70x`. In both cases, fission split loops that originally shared data while it was still hot in cache. After splitting, each loop independently traversed the full array, increasing effective memory traffic. Similar behavior was observed in `symm`, which reached `0.80x`, and `atax`, which reached `0.81x`.

Untransformed benchmarks stayed near `1.00x`, as expected. One notable outlier was `jacobi-1d-imper`, which showed a speedup of `1.23x` even though it was marked as `Transform?=NO`. Since the fission pass did not find a fissionable loop in this benchmark, the transformed binary is identical to the baseline. This value should therefore be treated as timing noise on a short-running kernel, not as a fission improvement.

The key result for fission is that it did not act as a strong direct performance optimization in this suite. In fact, it often hurt memory-bound kernels by splitting loops that shared live data in cache. Its value is more structural: it can simplify loop bodies and expose opportunities for later transformations, such as vectorization or parallelization. In this sense, fission should be viewed as a preparatory transformation rather than a transformation that is expected to improve runtime by itself.

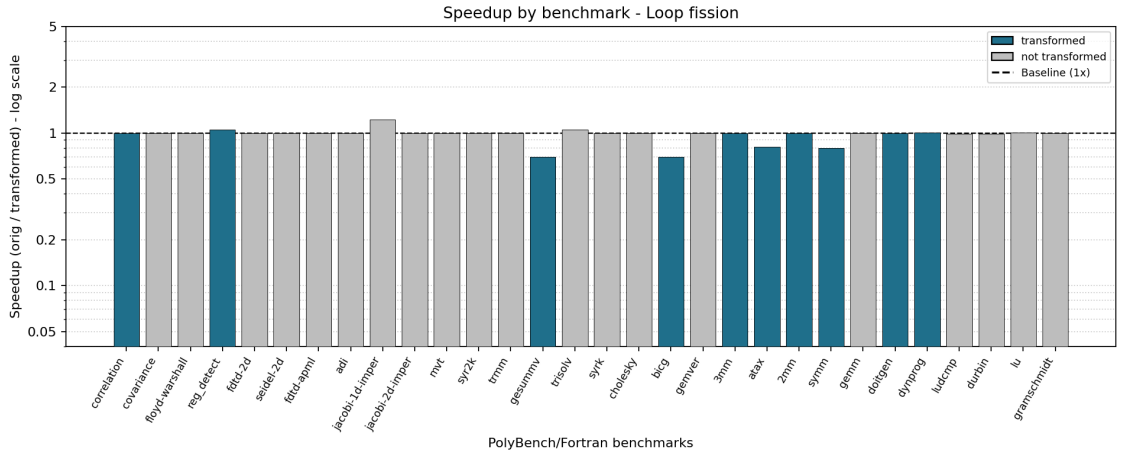

Figure 6.4: Loop Fission Speedups

6.3.5 Loop Interchange

Loop interchange was applied to 17 of the 30 benchmarks. The remaining 13 were not transformed because no perfect two-deep loop nest was found, or because `canInterchange()` blocked the swap. All 30 benchmarks reported `MATCH` and are shown in Figure 6.5.

A key caveat is that `interchangeGeneric` uses `canInterchange()` to guard semantic correctness, but this does not mean that every approved interchange is profitable. Some loop swaps are correct but harmful for cache locality.

Small performance differences were observed across several transformed benchmarks. However, gains between `1.01x` and `1.04x` fall within the expected measurement variability of the single-run methodology and should not be interpreted as evidence of a performance benefit. Benchmarks such as `correlation`, `covariance`, `seidel-2d`, `doitgen`, and `dynprog` therefore remained effectively neutral.

Some non-transformed benchmarks also exhibited apparent speedups, for example `reg_detect` (`1.38x`) and `jacobi-1d-imper` (`1.22x`). Since loop interchange was not applied, these differences are attributed to measurement variability rather than the transformation.

The largest regressions were observed in stencil and solver benchmarks. `fdtd-2d` slowed down to `0.06x`, `lu` to `0.05x`, `jacobi-2d-imper` to `0.07x`, and `adi` to `0.13x`. These are large slowdowns, but they are not correctness failures. All four benchmarks were transformed, passed the implemented legality checks, and matched the original output on the small dataset. The issue is performance. The interchange changed the loop order in a way that turned cache-friendly contiguous access into strided access over large arrays. On a `2000 x 2000` matrix, this can dramatically increase cache misses. `gemver` also suffered from this mechanism and reached `0.54x`, because the transformed matrix-update loop traversed rows instead of columns.

The main conclusion for interchange is that it has the highest variance among the five transformations. It can be slightly beneficial for dense linear algebra kernels when the new order improves column-major access, but it can be severely harmful for stencils and solvers when it

inverts a cache-friendly traversal. The implemented legality guard correctly checks whether the transformation is semantically safe, but it does not include a profitability model. Distinguishing cache-friendly from cache-hostile interchanges requires array-layout awareness and memory-access analysis beyond the current syntactic checks.

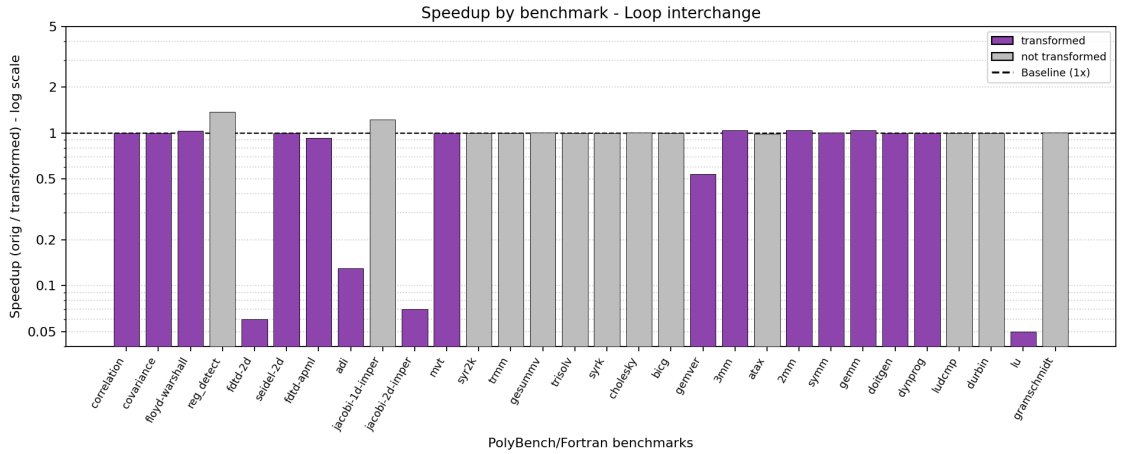

Figure 6.5: Loop Interchange Speedups

6.3.6 Cross-Transform Comparison

This section compares the five transformations across representative benchmarks. All numbers come from the `iteration-6-no-turbo` run on the laboratory machine with `LARGE_DATASET` and Turbo Boost disabled. A dash means that the benchmark was not transformed by that pass because it was not applicable or was blocked by the legality checks. Untransformed benchmarks usually remain close to `1.00x`, so those cells are omitted for clarity. Table 6.4 reports the representative speedups and Figure 6.6 shows visualization of all transformations performance results.

Table 6.4: Representative speedups across transformations.

Benchmark	**Tiling**	**Unrolling**	**Fusion**	**Fission**	**Interchange**
mvt	**4.52x**	0.99x	0.97x	—	1.00x
gemver	**2.04x**	0.98x	1.00x	—	0.54x
floyd-warshall	**1.51x**	1.05x	1.00x	—	1.03x
syrk	**1.22x**	0.96x	1.01x	—	—
syr2k	1.14x	0.98x	1.01x	—	—
bicg	—	**1.06x**	—	0.70x	—
doitgen	1.00x	1.04x	—	1.00x	1.00x
jacobi-2d-imper	0.35x	0.88x	**1.64x**	—	0.07x
2mm	1.04x	0.96x	0.99x	1.00x	**1.04x**
gemm	1.04x	0.97x	—	—	**1.04x**
lu	0.41x	0.81x	—	—	0.05x
fdtd-2d	0.45x	1.00x	—	—	0.06x
gesummv	—	1.01x	—	0.70x	—
adi	0.89x	0.80x	—	—	0.13x

Among the evaluated transformations, tiling produced the largest observed speedups, particularly for memory-bound kernels with transpose or blocked access patterns. The largest observed improvements were `mvt` (`4.52x`), `gemver` (`2.04x`), and `floyd-warshall` (`1.51x`). It also produced regressions on several stencil and solver kernels, indicating that its effectiveness depends on the loop structure and access pattern.

Unrolling was applicable to all 30 benchmarks and generally produced execution times close to the baseline. No substantial performance improvements or regressions were observed, suggesting that unrolling had a limited effect when applied as a standalone transformation in this evaluation.

Fusion was applicable to eight benchmarks. The largest observed speedup was `1.64x` for `jacobi-2d-imper`, although this result should be interpreted cautiously because similar gains were also observed for some non-transformed benchmarks.

Fission did not improve performance for most kernels in this suite. It regressed on accumulation-heavy kernels such as `bicg` and `gesummv`, both at `0.70x`, because it separated loops that originally shared hot data in cache.

Loop interchange exhibited the widest range of observed performance results. Small speedups fell within the expected measurement variability, while several stencil and solver kernels showed substantial slowdowns, including `fdtd-2d`, `lu`, `jacobi-2d-imper`, and `adi`. These results illustrate that a legal transformation is not necessarily a profitable one.

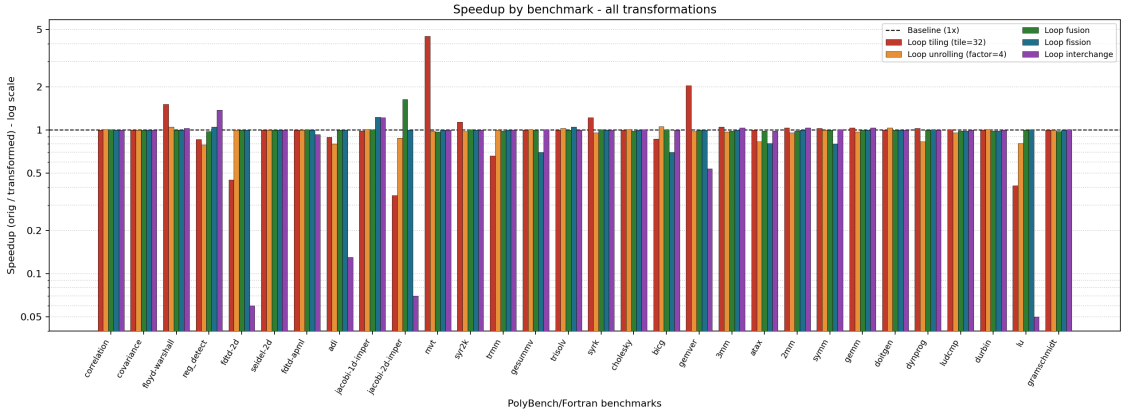

Figure 6.6: Comparison of all Loop Transformations' Speedups

6.4 Transformation Composition — Fission Followed by Tiling

The single-transformation evaluation showed that loop tiling had the strongest performance potential, but it was not applicable to all benchmarks. Tiling alone transformed 21 out of 30 benchmarks and skipped 9. This limitation comes from the current tiling implementation, which requires a suitable two-level loop nest. When an outer loop contains several independent statements, the loop is not directly tileable even if some of its internal computations contain loop nests that could benefit from tiling.

Loop fission was selected as a preprocessing step because its main value is structural. Although fission did not act as a strong standalone performance optimization, it can split a loop with multiple independent statements into separate loops. This may expose simpler, perfectly nested loop structures that the tiling pass can process. The evaluated composition was therefore:

```
1    Loop Fission -> Loop Tiling
```

The hypothesis was that applying fission before tiling would increase the number of successful tiling transformations without introducing correctness regressions.

Correctness was checked with `SMALL_DATASET` and `POLYBENCH_DUMP_ARRAYS` enabled. The output of each transformed benchmark was compared with the output of the original benchmark.

Two configurations were compared:

- **Tiling alone:** `LoopTilingPass`
- **Fission followed by tiling:** `LoopFissionPass -> LoopTilingPass`

Table 6.5 shows the full benchmark-level result.

Table 6.5: Effect of composing loop fission with loop tiling, per benchmark.

Benchmark	Tiling alone	Correct	Fission → Tiling	Correct	Effect of composition
correlation	TILED	MATCH	FISSIONED+TILED	MATCH	Still tiled
covariance	TILED	MATCH	TILED	MATCH	Still tiled
2mm	TILED	MATCH	FISSIONED+TILED	MATCH	Still tiled
3mm	TILED	MATCH	FISSIONED+TILED	MATCH	Still tiled
atax	SKIPPED	—	FISSIONED+TILED	MATCH	**Newly tiled**
bicg	SKIPPED	—	FISSIONED+TILED	MATCH	**Newly tiled**
cholesky	SKIPPED	—	SKIPPED	—	Still skipped
doitgen	TILED	MATCH	FISSIONED+TILED	MATCH	Still tiled
gemm	TILED	MATCH	TILED	MATCH	Still tiled
gemver	TILED	MATCH	TILED	MATCH	Still tiled
gesummv	SKIPPED	—	FISSIONED_ONLY	MATCH	Partial success
mvt	TILED	MATCH	TILED	MATCH	Still tiled
symm	TILED	MATCH	FISSIONED+TILED	MATCH	Still tiled
syr2k	TILED	MATCH	TILED	MATCH	Still tiled
syrk	TILED	MATCH	TILED	MATCH	Still tiled
trisolv	SKIPPED	—	SKIPPED	—	Still skipped
trmm	TILED	MATCH	TILED	MATCH	Still tiled
durbin	SKIPPED	—	SKIPPED	—	Still skipped
dynprog	TILED	MATCH	FISSIONED+TILED	MATCH	Still tiled
gramschmidt	SKIPPED	—	SKIPPED	—	Still skipped
lu	TILED	MATCH	TILED	MATCH	Still tiled
ludcmp	SKIPPED	—	SKIPPED	—	Still skipped
floyd-warshall	TILED	MATCH	TILED	MATCH	Still tiled
reg_detect	TILED	MATCH	FISSIONED+TILED	MATCH	Still tiled
adi	TILED	MATCH	TILED	MATCH	Still tiled
fdtd-2d	TILED	MATCH	TILED	MATCH	Still tiled
fdtd-apml	TILED	MATCH	TILED	MATCH	Still tiled
jacobi-1d-imper	SKIPPED	—	SKIPPED	—	Still skipped
jacobi-2d-imper	TILED	MATCH	TILED	MATCH	Still tiled
seidel-2d	TILED	MATCH	TILED	MATCH	Still tiled

The pipeline increased the number of tiled benchmarks from 21 to 23. This means that tiling applicability increased from 70.0% to 76.7% of the benchmark suite. No correctness regressions were observed. Table 6.6 summarizes the overall result.

Table 6.6: Effect of the fission-then-tiling pipeline on tiling applicability.

Configuration	Tiled benchmarks	Skipped benchmarks	Other result	Correctness result
Tiling alone	21/30	9/30	0/30	30/30 MATCH
Fission → Tiling	23/30	7/30	1/30 fissioned only	30/30 MATCH

The two newly tiled benchmarks were `atax` and `bicg`.

In `atax`, the original outer loop contains initialization of `tmp(i)`, accumulation of `tmp(i)`, and accumulation of `y(j)` inside the same loop body. Tiling alone skips this structure because the

outer loop body is not a single inner loop. After fission, the accumulation stages are separated into simpler two-level loop nests, which are then tiled successfully.

In `bicg`, the original computation combines updates to `s(j)` and `q(i)` in a structure that prevents direct tiling. Fission separates these independent updates and exposes suitable `(i, j)` loop nests. The tiling pass can then process the resulting loops.

Overall, the experiment confirms that transformation composition can improve transformation success rate. Fission is useful here not because it directly improves runtime, but because it reshapes the program and exposes additional opportunities for tiling. This supports the view of the implemented transformations as reusable passes that can be composed into optimization pipelines.

6.5 Threats to Validity

Several limitations affect the interpretation of the evaluation results.

First, numerical correctness at `LARGE_DATASET` was not directly verified. The large-dataset experiments disabled `POLYBENCH_DUMP_ARRAYS`, so `MATCH` in that configuration means that the binary executed successfully and produced the expected timing output path. It does not mean that the full numerical arrays were compared. The `SMALL_DATASET 30/30` array-dump comparison is therefore the only numerical correctness guarantee. A correctness bug that appears only at larger dataset sizes, such as an off-by-one error in a tile boundary formula, would not be detected by the large-dataset experiment.

Second, each benchmark was executed only once. No averaging, median calculation, or statistical analysis was performed. This makes speedup values for short-running kernels less reliable. Benchmarks such as `reg_detect`, `trisolv`, `gesummv`, and `jacobi-1d-imper` run in tens or hundreds of milliseconds, so small system disturbances can appear as large relative changes. This explains anomalies such as the fission result for `jacobi-1d-imper`, where an untransformed benchmark showed an apparent `1.23x` speedup.

Third, Turbo Boost was disabled, but other frequency-scaling effects were not fully controlled. C-states, voltage and frequency scaling under thermal load, and background system activity could still influence measurements. This is especially relevant for very long-running benchmarks such as `gramschmidt` and `3mm`, where execution lasts several minutes.

Fourth, the tile size and unroll factor were fixed and not autotuned. The tile size was `32`, and the unroll factor was `4`. These values demonstrate that the transformations support parameters, but they are not necessarily optimal. Larger tile sizes such as `64` or `128` might improve results for dense matrix kernels such as `gemm` and `symm`. A smaller unroll factor such as `2` might reduce register-pressure regressions in kernels such as `adi` and `lu`.

Fifth, the evaluation used one compiler and one CPU architecture. All results were obtained with `flang-22 -O3` on one `x86_64` machine. Other compilers, such as `gfortran` or Intel `ifx`, may interact differently with the transformed code. Other architectures, such as ARM or Power, may also show different behavior, especially for vectorization and memory-layout effects.

Finally, the conservative legality checks reduce the transformation yield. For example, `canTile()` and `canInterchange()` use word-boundary regular expressions on bound-expression strings. This can reject safe loops if a variable name appears in an unrelated expression. Such false negatives are acceptable for this work because preserving correctness is more important than maximizing the number of transformations. However, it also means that some potentially beneficial transformations may be skipped.

Chapter 7

Conclusion

This dissertation investigated source-to-source loop transformation for Fortran programs in the context of the METAFOR infrastructure. The work was motivated by a practical problem in scientific software engineering: performance-oriented code restructuring can improve execution time, but it can also make the original algorithm harder to understand, maintain, and adapt. The approach followed in this dissertation keeps the original Fortran program as the reference version and generates transformed variants through explicit transformation scripts.

The main result is an artifact integrated into METAFOR. The artifact supports five classical loop transformations over Fortran DO loops: loop fusion, loop fission, loop interchange, loop tiling, and loop unrolling. These transformations are exposed as script-controlled operations and applied through the METAFOR transformation pipeline, which parses Fortran code, represents it as an AST, exposes program elements through join points, rewrites selected loop structures, and regenerates Fortran source code.

Correctness was the main constraint throughout the work. Each transformation was guarded by conservative legality checks that reject unsupported or potentially unsafe cases. These checks include loop-shape validation, array read/write analysis, scalar dependency checks, iterator-based subscript checks, reduction rejection, and handling of triangular or variable-dependent bounds. The checks are intentionally conservative: when the tool cannot justify a transformation under the implemented rules, it preserves the original code.

The evaluation showed that the implemented transformations can be applied safely to the tested PolyBench/Fortran benchmarks. The transformed programs matched the original outputs in the correctness experiments. The performance results were mixed, but informative. Loop tiling produced the strongest improvements in selected memory-bound kernels. Loop fusion produced a clear gain in one stencil case, but applied to fewer benchmarks. Loop unrolling was broadly applicable, but its gains were modest. Loop interchange showed that a semantically correct transformation may still be inefficient if it damages memory locality. Loop fission was most useful as a preparatory transformation rather than as a direct source of speedup.

The transformation-composition experiment strengthened the main argument of the dissertation. Applying loop fission before loop tiling increased tiling applicability from 21 to 23 out of

30 benchmarks, without correctness regressions in the tested configuration. This result shows that transformation pipelines can expose opportunities that isolated transformations miss. In particular, fission can simplify loop bodies and make additional loop nests suitable for tiling.

7.1 Analysis of objectives

The first objective was extend METAFOR with reusable source-to-source operations for classical Fortran loop transformations. The dissertation implemented five transformations and made them available through the METAFOR scripting workflow. The result demonstrates that METAFOR can support structured loop rewriting and regeneration of Fortran source code. We can consider first objective achieved.

The second objective was to implement conservative legality checks to prevent unsafe transformations while preserving benchmark outputs for supported cases. The evaluation part proves this. The implemented checks rejected unsafe patterns found during development, and the final transformed programs matched the original outputs in the correctness experiments. This does not mean that all legal transformations can be detected, but it shows that a conservative safety strategy is practical for the supported loop structures.

The third objective to provide a practical workflow in which users select program regions, parameters, and transformation order through script-controlled transformations. The artifact supports this workflow. Transformations are invoked through scripts, parameters such as tile size and unrolling factor are explicit, and the generated Fortran source can be inspected before compilation.

The fourth objective was to increase transformation applicability by applying transformation composition, specifically by using loop fission to enable more successful loop tiling transformations than tiling alone. The fission-then-tiling experiment confirmed this for the evaluated benchmark suite. Fission exposed additional tileable structures in `atax` and `bicg`, increasing the number of successfully tiled benchmarks from 21 to 23.

7.2 New Paths Opened by the Work

The dissertation opens several directions for further work on Fortran source-to-source transformation.

The first path is the development of richer transformation pipelines. The implemented transformations can already be applied as reusable passes, and the fission-then-tiling experiment shows that composition can improve applicability. This creates a basis for studying other combinations, such as tiling followed by unrolling, interchange followed by tiling, or fission followed by parallelization.

The second path is the separation of legality analysis from profitability analysis. The current artifact mainly answers whether a transformation can be applied safely. The evaluation shows that this is not enough to guarantee good performance. Future work can build profitability models on top of the existing legality checks, especially for transformations such as interchange and tiling.

The third path is the use of METAFOR as an experimental platform for Fortran transformation research. Since the output remains Fortran source code, transformed programs can be inspected, compiled with different compilers, tested on different platforms, and compared with the original version. This makes the framework useful for studying both transformation correctness and performance behavior.

7.3 Future Work

The evaluation suggests several directions for future improvement.

One important direction is to add a cache-layout profitability heuristic to loop interchange. The current `canInterchange()` function checks whether the swap is semantically safe, but it does not model array access patterns or Fortran's column-major layout. A lightweight heuristic could check whether the innermost array subscript after interchange becomes a non-unit-stride dimension. This could prevent the large stencil and solver regressions, such as `fdtd-2d` at `0.06x` and `lu` at `0.05x`, while preserving the small gains observed in dense linear algebra kernels.

A second direction is autotuning tile size. The fixed tile size of `32` worked well for some kernels, but it may be too small or too large for others. Kernels such as `gemm`, `symm`, and large dense solvers may benefit from tile sizes such as `64` or `128`, while short kernels such as `reg_detect` may require smaller tiles. Choosing tile size based on array dimensions and cache topology could reduce both neutral and negative outcomes.

A third direction is stencil-aware tiling. Simple rectangular tiling was not effective for `fdtd-2d`, `jacobi-2d-imper`, and `adi`. Techniques such as wavefront tiling or time skewing are more appropriate for these patterns, but they require more advanced legality analysis than the current syntactic checks.

Fission should also be studied as a preprocessing step for parallelization. Although fission alone often did not improve runtime, it can separate independent loop bodies and make them easier to parallelize. Combining fission with a future OpenMP parallelization pass could expose performance benefits that are not visible when fission is evaluated in isolation.

In addition to these directions, the artifact should be extended to cover a broader range of Fortran constructs. The current work focuses on `DO` loops and the structures present in the benchmark suite. Real scientific applications may include procedure calls, modules, array syntax, derived types, pointer-like constructs, I/O operations, and more complex control flow. Supporting these features would make the framework more useful for production Fortran codebases.

A stronger dependence analysis would also improve the framework. The current rule-based checks are clear and safe, but they reject some cases that may be legal. More precise analysis would allow the tool to transform more programs while still preserving correctness, especially in the presence of reductions, triangular loops, and complex array subscripts.

Another direction is to extend the pipeline to support source-to-source adaptation for specialized Fortran compilers. For example, the framework could transform modern Fortran constructs

into equivalent forms compatible with compiler-specific programming models, such as OpenACC, enabling optimization by compilers that support a more restricted subset of the language.

Finally, future versions should provide clearer transformation reports. Since the goal is to make optimization decisions inspectable, the framework should explain why each candidate loop was transformed or rejected. This would help users understand the relation between loop structure, legality checks, and generated code.

7.4 Final Remarks

The dissertation achieved its main goal: it demonstrated that METAFOR can be extended with reusable, script-controlled loop transformation support for Fortran programs. The implemented artifact applies five classical transformations, guards them with conservative legality checks, regenerates Fortran source code, and validates the transformed programs against the originals.

The results also show that loop transformation should be treated as a workflow rather than as a single isolated optimization step. Some transformations are directly beneficial, some mainly prepare code for later transformations, and some require additional profitability analysis before they should be applied. The fission-then-tiling experiment illustrates this point clearly: transformation composition can increase the number of successful transformation opportunities.

Overall, the work provides a practical foundation for further research on explicit Fortran optimization pipelines. It shows that source-to-source transformation can make optimization steps visible, repeatable, and adjustable while preserving the original program as the maintainable reference version.

References

[1] Riyadh Baghdadi et al. “Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code”. In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 0015. Washington, DC, USA: IEEE, Feb. 2019, pp. 193–205. ISBN: 978-1-7281-1436-1. DOI: 10.1109/CGO.2019.8661197. URL: https://ieeexplore.ieee.org/document/8661197/ (visited on 11/18/2025).

[2] João Bispo and João M.P. Cardoso. “Clava: C/C++ source-to-source compilation using LARA”. In: *SoftwareX* 12 (July 2020). 0001, p. 100565. ISSN: 23527110. DOI: 10.1016/j.softx.2020.100565. URL: https://linkinghub.elsevier.com/retrieve/pii/S2352711019302122 (visited on 11/11/2025).

[3] Arthur Charguéraud et al. “OptiTrust: an Interactive Framework for Source-to-Source Transformations”. In: (2022). 0014.

[4] Chun Chen, Jacqueline Chame, and Mary Hall. *CHiLL: A framework for composing high-level loop transformations*. Tech. rep. Citeseer, 2008.

[5] Huimin Cui et al. “Extendable pattern-oriented optimization directives”. In: *ACM Trans. Archit. Code Optim.* 9.3 (Oct. 2012). ISSN: 1544-3566. DOI: 10.1145/2355585.2355587. URL: https://doi.org/10.1145/2355585.2355587.

[6] Sebastien Donadio et al. “A Language for the Compact Representation of Multiple Program Versions”. In: *Languages and Compilers for Parallel Computing*. Ed. by Eduard Ayguadé et al. Red. by David Hutchison et al. Vol. 4339. 0012 X Language. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 136–151. ISBN: 978-3-540-69329-1 978-3-540-69330-7. URL: http://link.springer.com/10.1007/978-3-540-69330-7_10 (visited on 12/13/2025).

[7] Albert Hartono, Boyana Norris, and P. Sadayappan. “Annotation-based empirical performance tuning using Orio”. In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. Distributed Processing (IPDPS). 0007. Rome, Italy: IEEE, May 2009, pp. 1–11. ISBN: 978-1-4244-3751-1. DOI: 10.1109/IPDPS.2009.5161004. URL: http://ieeexplore.ieee.org/document/5161004/ (visited on 11/11/2025).

[8] Olaf Krzikalla et al. “Scout: A Source-to-Source Transformator for SIMD-Optimizations”. In: *Euro-Par 2011: Parallel Processing Workshops*. Ed. by Michael Alexander et al. Red. by David Hutchison et al. Vol. 7156. 0017. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 137–145. ISBN: 978-3-642-29739-7 978-3-642-29740-3. URL: http://link.springer.com/10.1007/978-3-642-29740-3_17 (visited on 12/07/2025).

[9] Youenn Lebras et al. “ASSIST: An FDO source-to-source transformation tool for HPC applications”. In: *International Workshop on Parallel Tools for High Performance Computing*. Springer. 2017, pp. 39–56.

[10] Kedar S. Namjoshi and Nimit Singhania. "Loopy: Programmable and Formally Verified Loop Transformations". In: *Static Analysis*. Ed. by Xavier Rival. Vol. 9837. 0008. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 383–402. ISBN: 978-3-662-53412-0 978-3-662-53413-7. URL: http://link.springer.com/10.1007/978-3-662-53413-7_19 (visited on 11/18/2025).

[11] Juseong Park et al. "HYPERF: End-to-End Autotuning Framework for High-Performance Computing". In: *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '25: 34th International Symposium on High-Performance Parallel and Distributed Computing. 0013. University of Notre Dame Conference Facilities Notre Dame IN USA: ACM, July 20, 2025, pp. 1–14. ISBN: 979-8-4007-1869-4. DOI: 10.1145/3731545.3731588. URL: https://dl.acm.org/doi/10.1145/3731545.3731588 (visited on 11/18/2025).

[12] Jonathan Ragan-Kelley et al. "Decoupling algorithms from schedules for easy optimization of image processing pipelines". In: *ACM Transactions on Graphics* 31.4 (Aug. 5, 2012). 0021, pp. 1–12. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/2185520.2185528. URL: https://dl.acm.org/doi/10.1145/2185520.2185528 (visited on 12/07/2025).

[13] Hiroyuki Takizawa et al. "Xevolver: An XML-based code translation framework for supporting HPC application migration". In: *2014 21st International Conference on High Performance Computing (HiPC)*. 2014 21st International Conference on High Performance Computing (HiPC). 0016. Goa, India: IEEE, Dec. 2014, pp. 1–11. ISBN: 978-1-4799-5976-1. DOI: 10.1109/HiPC.2014.7116902. URL: http://ieeexplore.ieee.org/document/7116902/ (visited on 11/18/2025).

[14] Qing Yi. "POET: a scripting language for applying parameterized source-to-source program transformations". In: *Software: Practice and Experience* 42.6 (June 2012). 0011, pp. 675–706. ISSN: 0038-0644, 1097-024X. DOI: 10.1002/spe.1089. URL: https://onlinelibrary.wiley.com/doi/10.1002/spe.1089 (visited on 11/20/2025).