POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Recurrence-Aware Value Range Analysis for Precision Tuning

Tesi di Laurea Magistrale in
Computer Science Engineering - Ingegneria Informatica

Author: **Luca Achini**

Student ID: 101035
Advisor: Prof. Giovanni Agosta
Co-advisors: Gabriele Magnani, Niccolò Nicolosi
Academic Year: 2025-26

Abstract

Should one invest in numerical precision or sacrifice it in favor of execution speed? This trade-off naturally arises in many areas of computer science and is particularly relevant during the code compilation phase. Precision tuning addresses this problem by applying numerical analysis and conversion techniques to tolerate a limited loss of accuracy in exchange for improved performance. TAFFO is an LLVM-based framework designed to automate this process. A fundamental, yet non-trivial, aspect of precision tuning is the analysis of the ranges that program values can assume, as these ranges directly influence the correctness and effectiveness of numerical conversions. Several techniques have been proposed to address this problem, and the topic remains an active area of research. The aim of this thesis is to show how a pattern- recurrence-based approach to static range analysis can help to detect valid bounds, improve the analysis control and, as a consequence, enhance the performance benefits obtained through precision tuning.

Keywords: TAFFO, LLVM, Compilers, VRA, Scalar Evolution, Precision Tuning

Abstract in lingua italiana

Conviene investire nella precisione numerica o sacrificarla a favore della velocità di esecuzione? Questo compromesso si presenta in molte aree dell'informatica ed è particolarmente rilevante durante la fase di compilazione del codice. Il precision tuning affronta questo problema applicando tecniche di analisi numerica e di conversione tra rappresentazioni, tollerando una perdita di accuratezza limitata in cambio di un miglioramento delle prestazioni. TAFFO è un framework basato su LLVM progettato per automatizzare questo processo. Un aspetto fondamentale, ma tutt'altro che banale, del precision tuning è l'analisi dei range che i valori del programma possono assumere, poiché tali informazioni influenzano direttamente la correttezza e l'efficacia delle conversioni numeriche. Diverse tecniche sono state proposte per affrontare questo problema e il tema rappresenta tuttora un ambito di ricerca attivo. L'obiettivo di questa tesi è mostrare come un approccio pattern- recurrence-based applicato all'analisi statica dei range possa aiutare ad individuare bound validi, migliorare il controllo dell'analisi e, di conseguenza, incrementare i benefici prestazionali ottenibili tramite il precision tuning.

Parole chiave: TAFFO, LLVM, Compilatori, Analisi dei Range, Scalar Evolution, Precision Tuning

Contents

1 | Introduction

Precision tuning is a technique aimed at improving performance and resource efficiency by acting on computational precision. It falls within the broader paradigm of Approximate Computing and is gaining increasing relevance due to the rapid development of embedded systems [22]. Such systems are characterized by limited memory and computational resources. Many of them also rely on small batteries and must nevertheless guarantee long operational lifetimes.

One of these techniques is based on increasing control over the management of real numerical representations in programs. In particular, it operates on values that are typically represented using floating-point types, introducing a form of "discretization" of their representation.

Such discretization consists in explicitly fixing the position of the fixed-point, statically determining the subdivision of the available bits between the integer and the fractional parts. Given the same total number of bits, this choice makes the maximum number of digits representable before and after the fixed-point deterministic, enabling a more conscious management of the trade-off between representable range and numerical precision.

The TAFFO [5, 8] framework fits within this context as an LLVM-based tool [17] designed to systematically implement the precision tuning process. It provides a complete workflow that integrates static analysis and code transformation, enabling automatic intervention on the numerical representations used within a program.

The pipeline is structured into multiple stages. Starting from the parsing of the source code, the system identifies instructions eligible for conversion, analyzes the propagation of values, and determines, for each involved variable, the most appropriate fixed-point configuration. The process culminates in the effective transformation of numerical types and in an evaluation phase that allows the impact of the adopted choices to be measured in terms of both accuracy and performance.

Within this pipeline, value range analysis plays a central role. The correct allocation of bits between the integer and fractional parts depends on the knowledge of the minimum

and maximum bounds that each value may assume during program execution. Only with such information is it possible to ensure that the chosen representation is both safe—preventing overflow phenomena—and efficient, maximizing the achievable numerical precision given a fixed total number of bits.

In this perspective, the quality of range analysis directly affects the effectiveness of the entire precision tuning process: overly conservative bounds may lead to oversized representations and reduced precision, whereas overly optimistic estimates risk compromising program correctness. For this reason, improving Value Range Analysis techniques constitutes a key element of the approach proposed in this work.

1.1. Objectives and motivation

This thesis addresses Value Range Analysis within the context of precision tuning.

The current version of TAFFO performs Value Range Analysis (VRA) by propagating user-provided input ranges through the program using interval algebra rules. In each instruction, the operand bounds are combined to compute the range of results. While effective in linear contexts, this approach shows limitations in the presence of loops.

In iterative constructs, the analysis must either simulate variable evolution for a number of iterations equal to the loop trip count—potentially increasing compilation time—or rely on manual annotations to constrain propagation and prevent excessive or non-terminating analysis. Although functional, both solutions reveal the structural limitations of a purely iterative strategy.

This work overcomes such limitations by identifying structurally predictable behaviors within loops through LLVM-IR analysis and variable dependency inspection. These behaviors are classified according to mathematical models—such as affine, geometric, and polynomial recurrences—described by closed-form expressions. The integration of these models into VRA enables direct computation of loop-final ranges without iterative simulation or additional annotations.

The proposed approach improves scalability, reduces programmer intervention, and enhances range precision, contributing to a more robust and automated precision tuning process. Throughout the work, soundness has remained a fundamental requirement: computed intervals are guaranteed to safely over-approximate all possible execution values.

Moreover, the method has been designed to preserve compilation-time comparability with the existing framework, avoiding excessive analysis overhead. These objectives have been

achieved through the development of *Recurrence-Aware VRA (RA-VRA)*, an algorithm that recognizes, classifies, and resolves recurrence patterns in closed form within the value range analysis process.

1.2. Methodology overview

The proposed method introduces the algorithm *Recurrence-Aware VRA (RA-VRA)*, whose name reflects the central role assigned to recurrences in the value range analysis process. The algorithm explicitly targets recurrences arising within loop constructs, treating them as the main abstraction for range computation. It is bounded on a lattice-based approach [7], where the abstract domain consists of intervals associated with program values. Starting from initial ranges, the information is progressively propagated through monotonic transfer operators until a fixed point is reached, guaranteeing convergence and preserving soundness.

A second theoretical foundation is the study of scalar evolution over time [1], traditionally defined for precise scalar values and here extended to the interval domain. This extension enables the evolution of minimum and maximum bounds to be modeled through formal recursive relations.

Recurrences enrich the lattice with structured information, improving the precision of representable value coverage. Section 5.1 classifies common recurrence classes and derives their closed-form expressions. Before introducing the full algorithm, the two fundamental phases are discussed: the *inspection* phase (Section 5.2), devoted to identifying potential recurrences in iterative constructs, and the *classification and assembling* phase (Section 5.3), where identified elements are organized and modeled.

Recurrence recognition relies on detecting structural patterns in LLVM-IR. The combination of pattern-based structural analysis and explicit recurrence modeling motivates the definition of the method as a *pattern-recurrence-based approach*.

Not all patterns correspond to directly solvable recurrences; some emerge only from combining multiple elements, and resolution order is crucial due to inter-recurrence dependencies. For clarity, techniques are first presented intuitively, while the full formalization is provided in Section 5.4.

The RA-VRA algorithm consists of five phases: *preseed*, *inspection*, *assembling*, *maximum iteration bound computation*, and *propagation*. The first two are executed once, whereas the remaining phases are iterated due to recurrence dependencies, progressively enriching the lattice.

The *preseed* phase establishes a dominance-based ordering of LLVM-IR basic blocks and initializes the lattice state. The same ordering is reused during propagation to ensure consistency. During *inspection*, candidate recurrence instructions are collected; in *assembling*, they are classified and formally modeled once their dependencies and parameters are validated within the program context.

Recurrence is resolved with respect to an iteration bound N, representing the maximum number of loop iterations, enabling direct computation of the final variable range.

Finally, the *propagation* phase distributes resolved recurrence information across the lattice, potentially unlocking additional recurrences or leading to stabilization.

Compared to traditional iterative simulation, this approach reduces updates by relating analysis complexity to the number of recognized recurrences rather than to the actual loop iterations. As a result, it achieves a more favorable scalability–precision trade-off while preserving soundness.

1.3. Summary of results

The validation of the proposed method is organized into three complementary phases.

The first phase validates the closed-form expressions derived from the identified recurrences, verifying their soundness. Only the input range of a vector is defined; the vector is initialized with randomly generated values within that interval, and the recurrences are applied. The experiment is repeated multiple times to ensure that all observed concrete values are contained within the interval computed by the recurrence formula, thus empirically confirming the soundness property.

The second phase evaluates the algorithm as a whole, assessing its ability to recognize, classify, construct, and resolve recurrences. Numerical errors, achieved speed-up through precision tuning, and compilation times are compared with those of the previous VRA implementation under equivalent conditions. The results show correct identification and modeling of recurrences, comparable compilation times and speed-ups, and a significant reduction in required annotations and manual intervention.

The analysis highlights that the greatest benefits arise for linearly evolving recurrences, where interval growth remains controlled and supports effective bit allocation between integer and fractional parts. Conversely, exponentially growing recurrences lead to rapid interval expansion, wider bounds, and reduced fractional precision due to increased integer bit requirements.

Finally, the approach is evaluated on benchmarks from the PolyBench suite [21], which includes kernels with representative recurrence patterns. Results confirm previous observations and enable discussion of cases with larger errors, mainly due to unimplemented recurrence types, strongly exponential behaviors, or the need for additional heuristics to estimate maximum iteration bounds in certain loops.

The goal of this work is not to exhaustively cover every recurrence pattern, but to provide a reliable, scalable, and extensible framework that can be adaptable to different application contexts.

1.4. Thesis organization

In addition to the description of the proposed method and the experimental results obtained, this work includes additional chapters closely related to the topic under investigation, aimed at providing the theoretical and technical background necessary for a full understanding of the presented contribution.

Chapter 2, introduces the theoretical background. It begins with a description of floating-point and fixed-point numerical representations, analyzing their advantages and disadvantages, and illustrating how the ability to regulate such representations is closely related to the precision tuning process and to the role of Value Range Analysis in this context. The chapter then recalls the fundamental concepts of compilation and its main phases, followed by an introduction to LLVM, the compilation framework on which this work is based. The chapter concludes with a discussion of iterative constructs in intermediate representations, presenting their definitions, canonical structures, and properties relevant to analysis.

Chapter 3 reviews the state of the art in the field of analysis at the intermediate representation level. The differences between static and dynamic analysis are discussed, and existing approaches that have influenced the design of the proposed method are examined in detail. The chapter concludes with an overview of parallel and alternative methodologies available in the literature.

The subsequent chapters provide a complete formalization of the proposed method, describe its implementation (chapter 5), and present the experimental validation (chapter 6). The thesis concludes with a final discussion (chapter 7) of the results obtained and outlines possible directions for future development.

2 | Background

2.1. Numerical precision and range-aware optimization

In nature, phenomena span vastly different scales, ranging from extremely small to extremely large, and numbers provide a natural means to represent such variability. Although the human mind is not naturally suited to reason about these extremes simultaneously, it is highly effective when operating on limited ranges of values and can rapidly adapt its scale of reasoning as needed. Similarly, computers process information through numerical representations and are inherently constrained by the finite quantity and precision of the numbers they can represent. This section introduces how numbers are represented within computing systems and how such representations can be scaled to describe both micro- and macro-level quantities. Then it discusses the conversion between different numerical representations and the role of precision tuning in this process. Finally, it highlights how knowledge of value ranges plays a central role in guiding appropriate precision choices.

2.1.1. Computer representation of numerical values

Before entering into the details of precision tuning, it is useful to provide a brief introduction to how numbers are represented within a computer. Unlike human beings, who are accustomed to expressing numbers and performing computations in base 10, machines operate in base 2, using bits. A bit represents the smallest unit of information in computing systems and, at a logical level, encodes a binary state. At the physical level, this state is realized through electrical, optical, or radio signals, which typically represent two possible values, 0 and 1. From this elementary representation, it is possible to build more complex structures to represent integer numbers, real numbers, and other types of data. By considering a sequence of bits, it is possible to represent integer numbers whose set of representable values grows exponentially with the number of bits used.

For example, using 8 bits, equivalent to 1 byte, it is possible to represent $2^8 = 256$ distinct

values. In many programming languages, this number of bits is commonly associated with the *char* type, which represents an 8-bit integer value, and can be used, depending on the adopted encoding, for the representation of textual characters. Starting from 16 bits, larger integer types are introduced, such as *short integers*, while with 32 bits one obtains the so-called *integer* or *int*, which represents the "classical" integer type in most modern architectures. By using a larger number of bits, additional integer types can be defined, such as *long integers* capable of representing even wider ranges of values. Bits are considered as an ordered sequence, in which each bit assumes a different weight depending on its position, in a manner analogous to the decimal system. A fundamental aspect in the representation of integer numbers is the handling of the sign. Depending on the adopted choice, a sequence of bits may represent exclusively non-negative values, in the case of unsigned integers, or also include negative values, in the case of signed integers. Typically, the sign is represented by reserving the most significant bit of the sequence: by convention, the value 0 denotes a positive number, whereas the value 1 denotes a negative number. These representations constitute the foundation upon which numerical formats for the representation of real numbers are built, which will be analyzed in the following.

2.1.2. Fixed-Point and Floating-Point representations

In the context of this work, the representations of the affected numerical values are fixed-point and floating-point [10]. A fixed-point representation of a real number is defined by the tuple

$$(s, I, D)$$

where s is the sign bit, I is the number of bits dedicated to the integer part, and D is the number of bits dedicated to the fractional part. The higher the number of bits I, the wider the range of representable values; instead, increasing the number of bits D, improves the precision of the number. Between the last bit of I and the first bit of D there is a logical position representing the decimal point. Since the representation is split into two integer parts, arithmetic operations can be executed directly by the processor's ALUs. The position of the decimal point is typically chosen according to the target architecture, enabling particularly efficient implementations.

Instead, a floating-point representation is defined by the tuple

$$(s, e, m)$$

where s is the sign bit, e is the exponent and m is the mantissa. To clarify this representation, some examples are considered in base 10 using the exponential: value

$70.46 \rightarrow 7.046 \times 10^1$, with exponent equal to 1 and mantissa equal to 7.046; value $4025 \rightarrow 4.025 \times 10^3$, with exponent equal to 3 and mantissa equal to 4.025. In computers, the same principle applies, but using base–2 numbers. The exponent allows scaling the absolute value of the mantissa, enabling the representation of very large or very small numbers with a limited number of significant digits. Unlike fixed-point representations, however, most architectures adopt the IEEE 754 standard [15], which enforces a predefined number of bits for sign, exponent, and mantissa. Although the effective range and precision vary as a function of the exponent [13], the overall structure of the format remains fixed and does not allow dynamic modification of the distribution of bits between range and precision.

Since computers can operate only on representations with a finite number of bits, while real numbers are not limited in either magnitude or precision, performing arithmetic operations inevitably introduces representation errors. Two main types of error can be identified: overflow and quantization.

An overflow error occurs when the result of an operation, such as an addition, exceeds the range of values that can be represented by the adopted numerical format. In such cases, the result cannot be correctly represented and the behavior depends on the numerical model and the architecture, making the obtained value unreliable for further computations.

The quantization error, on the contrary, is related to the need to approximate a value when its representation would require a higher precision than the one available. In these situations, the number is forcibly rounded or truncated to the nearest representable value, introducing a loss of information that may propagate and accumulate in subsequent operations.

The following two lists compare fixed-point and floating-point numbers with respect to the errors described above, as well as other notable aspects.

Fixed-point:

- **Overflow**: it may occur easily if the range is not properly sized; the behavior is often undefined or implementation-dependent.
- **Quantization**: it depends on the number of bits dedicated to the fractional part and results in a constant and predictable error.
- **Precision**: uniform across the entire representable range.
- **Error propagation**: limited and deterministic, provided that the format is well

chosen.

- **Configurability**: highly configurable through the explicit choice of I and D.
- **Computational cost**: generally lower, as operations are executed via ALUs.
- **Behavior predictability**: high, if the value range is known *a priori*.

Floating-point:

- **Overflow**: handled by the IEEE 754 standard through special values such as $\pm\infty$ and NaN, but still resulting in loss of numerical validity.
- **Quantization**: it depends on the exponent value; the relative error varies with the magnitude of the number.
- **Precision**: higher for small values and lower for large values.
- **Error propagation**: it may accumulate over time due to rounding effects (drift), especially in loops.
- **Configurability**: fixed by the IEEE 754 standard.
- **Computational cost**: higher, as dedicated FPUs are required.
- **Behavior predictability**: lower, due to rounding and dynamic scaling.

2.1.3. Precision tuning and value ranges

Once the characteristics of the two main numerical representations have been established, it becomes natural to investigate techniques capable of automating, at compile time, the assignment of the most appropriate representation to each numerical value. Such techniques fit within the broader context of precision tuning.

Precision tuning belongs to the Computer Approximating paradigm and encompasses the set of strategies aimed at regulating computational precision and, when appropriate, reducing it in favor of benefits such as increased execution speed, lower energy consumption, or a more efficient use of hardware resources.

Since the early days of computing, this need has always been present: initially, it was almost mandatory due to the severe limitations in terms of available memory, registers, and computing units, so that every optimization produced significant benefits. As automatic tools such as TAFFO were developed only later, the earliest applications of precision tuning relied on manual adjustments performed directly by programmers.

The current scenario is characterized by constantly increasing computational power, which might suggest that precision tuning has lost its relevance. However, its role has by no means diminished. On the one hand, the wide availability of hardware resources and the high performance achievable today have encouraged the development of applications that often pay little attention to avoiding unnecessary resource consumption.

Moreover, in recent years, there has been a significant diffusion of hardware systems that can be integrated into a wide variety of everyday devices, commonly referred to as *embedded systems*. Due to strict constraints in terms of size, weight, and energy consumption, such systems provide extremely limited resources, making precision tuning a fundamental aspect.

Among the most widespread precision tuning techniques is the controlled assignment of numerical types, in particular, the selection between fixed-point and floating-point representations. The underlying principle consists in favoring, whenever possible, more tailored fixed-point representations over floating-point ones, in order to obtain more controllable accuracy and a more predictable error propagation.

TAFFO specifically addresses this technique and follows a multi-stage pipeline. The process begins with the analysis of the source code, aimed at identifying the involved variables and collecting the associated type information. At this stage, additional information or annotations provided by the programmer may also be acquired in order to facilitate the compilation process. A detailed description of TAFFO is provided in chapter 4.

Another well-known technique, which will only be briefly mentioned here, is profiling: it consists of repeatedly executing the program in order to collect statistical information that can guide precision tuning decisions. More details are provided in Section 3.4.1.

Impact of value ranges on precision tuning Ranges represent the bounds of the values that a variable may assume during the execution of a program and constitute a fundamental piece of information for the application of precision tuning techniques. When the minimum and maximum values that a variable may assume are known, it is possible to allocate in a targeted manner the number of bits devoted to the integer part of the numerical representation, reserving the remaining ones for the fractional part. This allocation allows to reduce resource waste while preserving the level of accuracy required by the computation. However, at compile time, this information is not directly available and must be inferred without executing the program. The deduction of such properties falls within the scope of *static analysis*, whose goal is to derive characteristics of program behavior through code inspection rather than runtime evaluation.

For completeness, the formal definition of ranges and the most common operations on intervals used throughout this work are provided in Appendix A.

2.2. Basic notions about compilers

Compilation is the translation process that enables the transformation of code from a form understandable by humans to a form that can be executed by a machine. This process is carried out by specific programs, called *compilers*. Compilation is structured into multiple phases, each of which may produce as output an intermediate representation of the program. These representations become progressively less human-readable and increasingly closer to machine language, enabling the application of targeted analyzes and transformations throughout the entire compilation pipeline. In the following overview section, a general description of the compilation pipeline is provided, illustrating the main steps, the output produced by each phase, and the role of intermediate representations. Subsequently, the three macro-phases of a compiler-front-end, middle-end, and back-end are discussed in detail. Finally, the LLVM compilation framework [17] is briefly introduced as a representative example of a mature compiler infrastructure widely adopted in both academic research and industrial contexts.

2.2.1. Compilation steps

In the introduction, it was described how a compiler operates through a sequence of operations organized as a pipeline, and how these operations are typically divided into three main macro-phases. Figure 2.1 illustrates these three macro-phases—front-end, middle-end, and back-end—highlighting the different stages of the generated code.

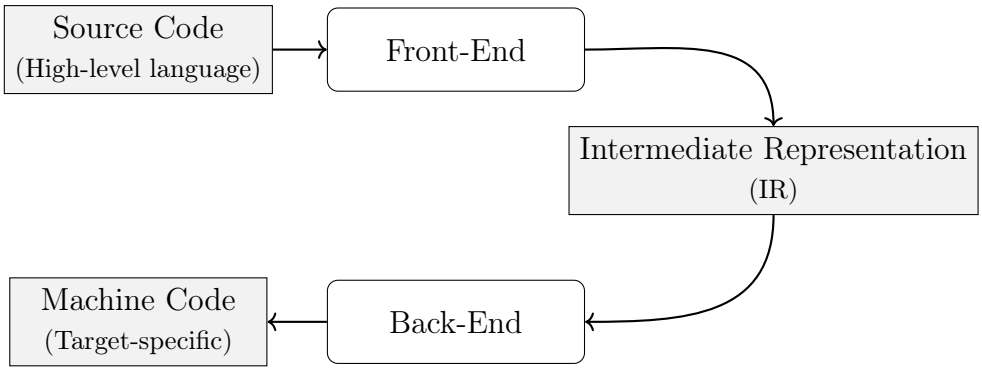


Figure 2.1: Diagram block view of the compilation pipeline. The front-end translates source code into an intermediate representation (IR), while the back-end generates target-specific machine code from the IR.

It is important to emphasize that the intermediate representation, commonly referred to as *IR* (Intermediate Representation), is designed to be independent of both the source programming language and the target machine architecture. This organization enables two fundamental properties:

- **Portability**: thanks to the use of a common intermediate representation, it is possible to support multiple programming languages and multiple target architectures without the need to implement a direct translation for every possible language–architecture combination. A single front-end can translate source code into IR, while different back-ends can generate code for various platforms starting from the same IR.
- **Modularity**: the separation between front-end, middle-end, and back-end allows each component to be designed and developed independently. In particular, a new front-end can be introduced without requiring knowledge of target architecture details, just as a new back-end can be designed without modifying existing front-ends.

2.2.2. Front-end and Back-end

The *front-end* is the phase of the compiler that reads high-level source code and transforms it into an internal representation more suitable for subsequent processing, ultimately producing an *intermediate representation* (IR). To achieve this goal, the front-end performs two main tasks: *language recognition* and *program translation*.

Furthermore, language recognition is divided into three steps:

1. **Lexical analysis (tokenization)**: the source code is converted into a sequence of tokens, each belonging to a specific category (keyword, identifier, operator, numeric literal, separator, etc.).
2. **Syntactic analysis (parsing)**: the tokens are organized into a tree structure according to the formal grammatical rules of the language. During this phase, the syntactic correctness of the program is verified and a structured representation of its syntax is built, often distinguishing between a *parse tree* (concrete syntax) and an *abstract syntax tree* (AST), which abstracts away purely syntactic elements.
3. **Semantic analysis**: starting from the AST, semantic correctness and logical consistency properties of the program are checked (e.g., symbol resolution, type compatibility, correct usage of variables and functions, and context-dependent language constraints).

Figure 2.2 illustrates all these phases in sequence together with the artifacts produced at each step. Once all phases are completed, the *intermediate representation* (IR) is obtained, ready to be processed by the *middle-end* of the compilation pipeline.

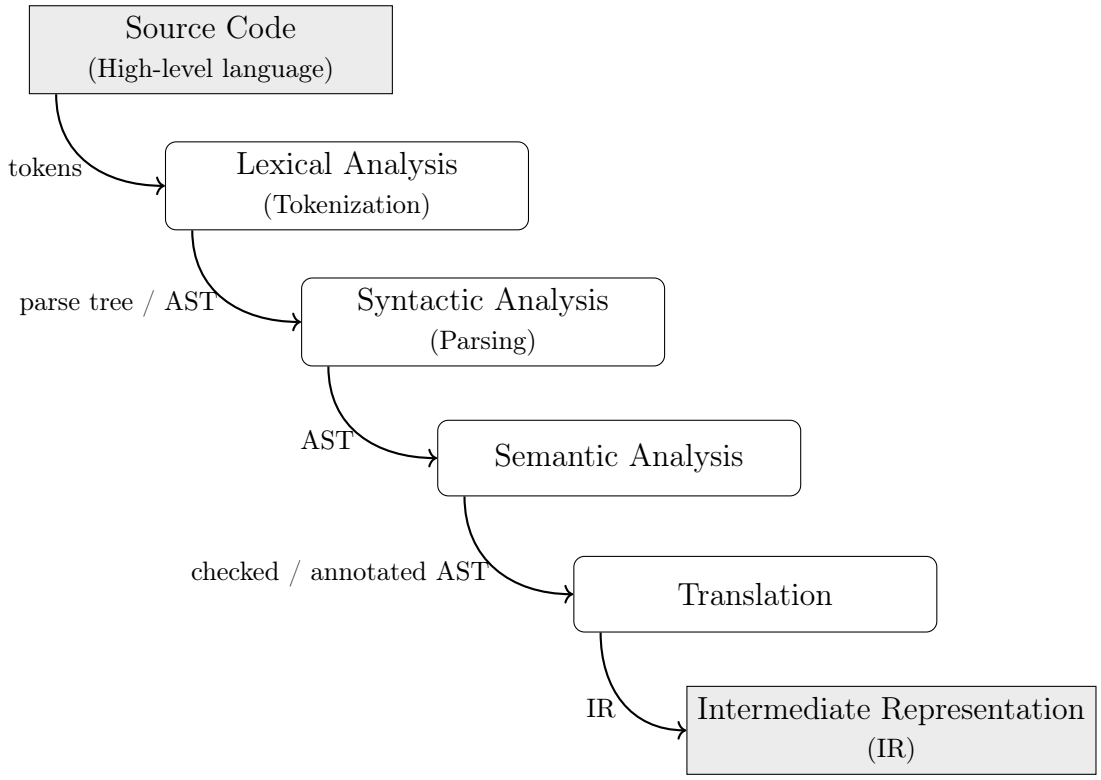


Figure 2.2: Front-end pipeline. The main phases are shown as blocks, while the intermediate artifacts produced at each step are reported as labels on the transitions.

The *back-end* encompasses the phases that are most closely tied to the target architecture and is generally more complex than the front-end. As mentioned previously, its main task is to translate the *intermediate representation* (IR) into a form that is compatible with a specific *target architecture*.

Each architecture exhibits different characteristics and constraints, ranging from traditional general-purpose architectures to those optimized for high performance or reduced energy consumption. These differences also depend on the context of the application, which may vary from desktop and notebook systems to mobile devices or embedded systems.

For each architecture to be supported, a dedicated *back-end module* must be developed, taking into account several architecture-specific factors. Among the most relevant are:

- *memory layout and management*;

- *instruction selection* from the IR (including addressing modes);
- exploitation of the *memory hierarchy*;
- support for *hardware parallelism*;
- *register allocation*.

In general, the back-end pipeline can be divided into two main phases. The first is *code generation*, in which the IR is transformed into target-specific code while accounting for the architectural aspects listed above. The second phase is *linking*, during which the generated object files are combined with each other and with the required external libraries to produce the final executable. Figure 2.3 completes the graphical overview by illustrating the back-end pipeline.

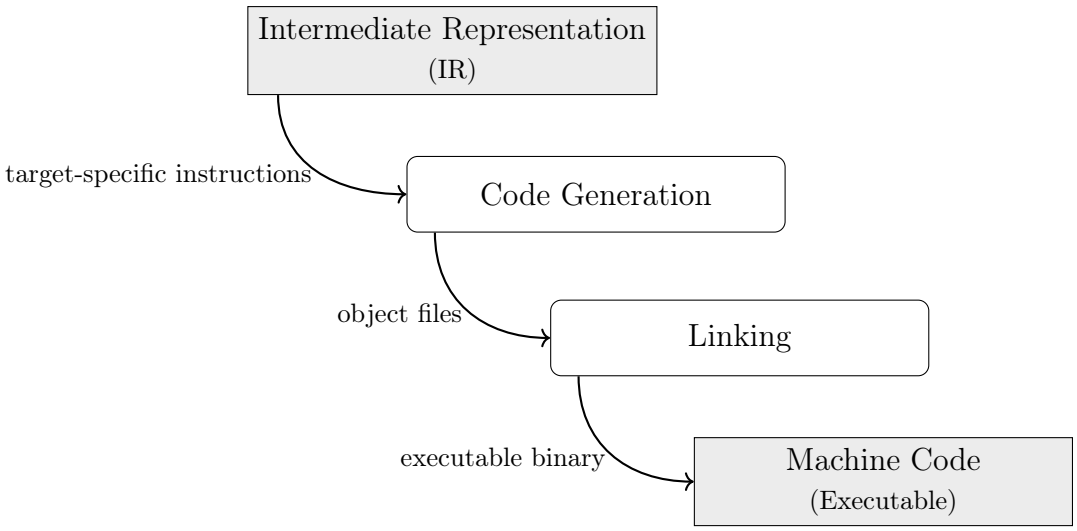


Figure 2.3: Back-end pipeline. Starting from the intermediate representation (IR), the back-end performs code generation and linking to produce the final machine code executable.

Between these two phases lies the *middle-end*, which operates on the intermediate representation and represents the core stage for applying architecture-independent program analyses and transformations. Since this thesis focuses on such analyses and optimizations, the middle-end is discussed in detail in the following section.

2.2.3. Middle-end

Between the front-end and back-end phases lies the *middle-end*. This part of the compiler is responsible for performing *static analyses* and *code transformations* on the intermediate representation, with the goal of improving the efficiency of the program or collecting useful

information for the subsequent stages of the compilation pipeline.

All transformations applied in the middle-end must preserve the *semantics* of the original program, ensuring that the observable behavior of the code remains unchanged.

Middle-end analyses and transformations are typically organized as a sequence of *passes*, i.e., modular units of code that operate on the intermediate representation at different levels of granularity. This organization allows each pass to remain focused on a single concern, promoting modularity, reusability, and composability of optimizations.

The following is a non-exhaustive list of some of the main analyses and transformations commonly applied in the middle-end:

- **Dead Code Elimination**: removal of instructions or code regions that do not affect the program output.
- **Constant Folding and Constant Propagation**: evaluation and propagation of constant expressions to reduce the number of runtime operations.
- **Loop Optimizations**: transformations applied to iterative constructs, such as loop-invariant code motion, unrolling, and fusion.
- **Canonicalization**: rewriting code into a standardized form to simplify subsequent analyses.
- **Strength Reduction**: replacement of computationally expensive operations with equivalent but more efficient ones.
- **Data-flow Analyses**: analyses based on data flow information, such as live variable analysis or reaching definitions.
- **Value and Range Analyses**: analyses aimed at determining information about the values assumed by variables during program execution.

As stated previously, the work presented in this thesis aims to improve techniques for *value range analysis*, with the objective of supporting and enhancing the *precision tuning* process. Limiting value ranges enables the conversion of variables represented in floating-point format into fixed-point representations by appropriately selecting the number of bits allocated to the integer and fractional parts.

For an overview of existing approaches, the reader is referred to the chapter on the *state of the art*, while the *methodology* chapter describes in detail how these techniques have been extended and improved within the context of this work.

For a comprehensive and systematic treatment of compiler design principles, analyses, and optimizations, the reader is referred to the standard reference text by Aho et al. [2].

2.2.4. LLVM

LLVM [17] is a compilation framework designed to support and optimize all stages of a program lifecycle, from compilation and linking to execution. It is written in C++ and supports a wide range of front-ends for high-level programming languages, such as C, C++, Rust, and Fortran, as well as a broad set of target architectures, including x86 and ARM. LLVM is composed of numerous modular and reusable components, a characteristic that makes it suitable for both small-scale projects and industrial-grade systems.

The intermediate representation used by LLVM is called LLVM Intermediate Representation (LLVM-IR) [16]. It organizes the program code in a hierarchical structure and can be represented in textual form through files with the `.ll` extension. The basic hierarchy consists of modules, functions, basic blocks, and instructions. A module can be seen as a collection of functions sharing the same global scope. Functions correspond to those defined in the original high-level source code. Basic blocks group lists of instructions that are executed sequentially and represent the nodes of the control flow graph described in the previous sections. Each instruction represents a single elementary operation or statement.

LLVM-IR adopts the Static Single Assignment (SSA) form, in which each value is assigned exactly once, and follows a three-address code model. This representation makes data dependencies explicit and simplifies the implementation of static analyses and code transformations, making LLVM particularly suitable for the development of advanced optimizations, such as those required by precision tuning.

LLVM adopts a pass-based architecture. The framework provides a wide range of composable passes, as well as the possibility of defining new ones, which can be classified into two main categories:

- **Analysis passes**: they collect information about the code without modifying the intermediate representation.
- **Transformation passes**: they exploit the information produced by analysis passes to rewrite or optimize the IR according to specific criteria.

Passes can operate at different scope levels, including module, function, loop, and basic block. This architecture promotes modularity, reusability, composability, and enables the construction of complex compilation pipelines.

2.3. Loop structures and properties into IR

Loops play a central role in the code optimization process. Any improvement applied to the instructions contained within a loop is propagated across all its iterations, amplifying its overall impact on program performance and numerical precision.

In this work, loops are of fundamental importance, as the optimization of value range analysis is based on a thorough study of the iterative behavior of variables. This section describes how loops are identified and managed in the intermediate representation and introduces the main structural properties and terminology required to understand the methodology proposed in this thesis.

2.3.1. Definition and identification of loops

A loop is a portion of a program that is executed iteratively, generally until a certain condition is satisfied. In the intermediate representation, code is modeled as a set of elementary regions, organized into nodes, and connected through control-flow relationships.

This graphical representation, known as the Control Flow Graph (CFG), makes it possible to describe the behavior of a program by means of a directed graph. A directed edge from node A to node B indicates that the execution of node A precedes, at least for the first time, the execution of node B. Some nodes may have multiple outgoing edges, indicating the presence of control-flow branches, while others may have multiple incoming edges, representing join points of previous branches.

Backward edges are also allowed, introducing cycles in the graph whenever the control flow permits the re-execution of certain portions of code. Therefore, a loop can be defined as a cycle within the CFG (Figure 2.4).

There is a second definition, more informal but significantly more useful for analysis purposes. Before introducing it, it is necessary to discuss the role of *Strongly Connected Components* (SCCs) within Control Flow Graphs and to define the concept of dominance. From a purely graphical perspective, a loop can be associated with a *maximal SCC* of the CFG. Figure 2.5 illustrates the meaning of *maximality*: although multiple SCCs can be identified within the graph, only the one composed of nodes $\{2, 3, 4, 5\}$ is maximal and can be considered a coherent representation of a loop.

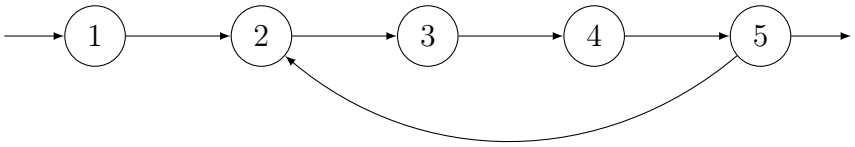


Figure 2.4: Loop inside CFG: cycle between nodes {2,3,4,5}

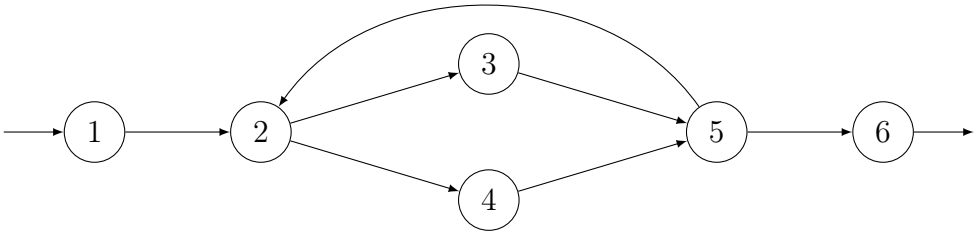


Figure 2.5: Loop and SCC view: three SCC are present: {2,3,4,5}, {2,3,5}, {2,4,5}

However, restricting the definition of loops to SCCs alone leads to the identification of numerous cyclic structures, making the analysis complex and less effective. For this reason, more restrictive criteria are required in order to reduce the number of loops under consideration and to simplify their analysis.

To this end, the concept of *dominance* in a Control Flow Graph is introduced. Given the start node s_0 without predecessors, a node d is said to *dominate* a node n if d is traversed along any possible directed path from s_0 to n. By definition, every node dominates itself.

The dominance relation satisfies the following properties:

- *reflexivity*: for every node a, a dom a;
- *transitivity*: if a dom b and b dom c, then a dom c;
- *anti-symmetry*: if a dom b and b dom a, then $a = b$.

These properties make the dominance relation a *partial order* over the nodes of the CFG.

Using the concept of dominance, a more structured and restrictive definition of loops can be introduced, commonly referred to as a *natural loop*. This definition refines the purely graph-based notion of strongly connected components by incorporating dominance relationships, which are essential for compiler analyses.

Given a CFG, a *back-edge* is defined as a directed edge $(n \to h)$ such that the destination node h dominates the source node n. The node h is called the *loop header* and represents the unique entry point of the loop.

The *natural loop* induced by a back-edge $(n \to h)$ is defined as the set of nodes consisting of h, n, and all nodes that can reach n through directed paths that do not pass through h. By construction, all nodes in the natural loop are dominated by the header h, ensuring the *single-entry* property of the loop.

From this perspective, a natural loop can be seen as a structured subset of a strongly connected component, characterized by a unique header node that dominates all other nodes in the loop. While every natural loop is contained within a strongly connected component of the CFG, the converse does not necessarily hold, as SCCs may represent cyclic structures that lack a single dominating entry point.

2.3.2. Loop structure and canonical form

The *natural loop* defined in the previous section represents the starting point for many compiler analyses and optimizations, and it is precisely this type of loop that is considered in this work. Throughout the discussion, several terms associated with the structure of a loop have been introduced, whose understanding is essential to follow the proposed methodology.

In the previous section, two key concepts have already been introduced, namely the *loop header* and the *back-edge*, which are illustrated in Figure 2.6 to facilitate their interpretation. In addition to these, other relevant nodes can be identified within the structure of a loop:

- *loop latch*: the node that contains the outgoing edge leading back to the loop header, thus closing the cycle;
- *loop body*: the set of nodes belonging to the loop, excluding the loop header and the loop latch;
- *loop preheader*: a node that does not belong to the loop and has a single outgoing edge pointing to the loop header;
- *loop exit*: one or more nodes outside the loop, representing the control-flow destinations reached once the loop exit condition is satisfied.

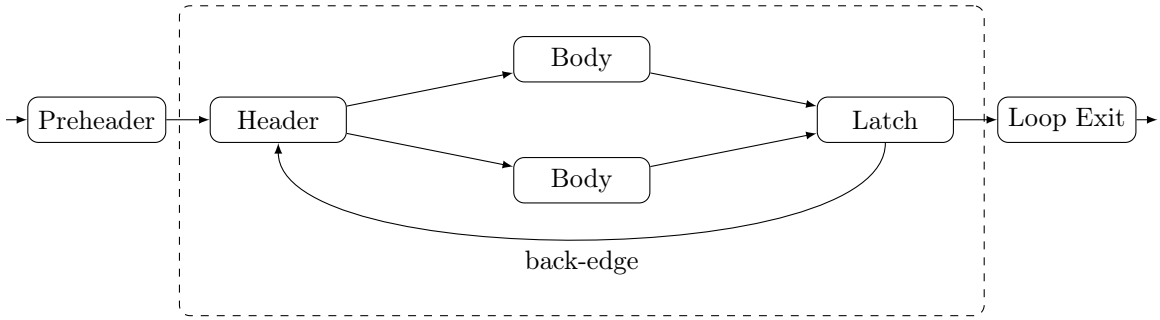


Figure 2.6: Graph representation of loop structure: dashed area includes all loop nodes

As a constraints of this work, loops must be transformed into a structurally normalized form in which these elements are explicitly identifiable. LLVM provides specific passes to easily satisfy this obligation. Such a normalization, referred to as the *loop canonical form*, represents a fundamental prerequisite for the recurrence and value range analyses developed in the following chapters.

A loop in canonical form typically satisfies the following properties:

- a unique *preheader*, serving as the single entry point into the loop;
- a unique *header*, which dominates all other blocks in the loop;
- a unique *latch* block, containing the back-edge to the header;
- a single back-edge, making the iterative structure explicit;
- a normalized exit condition, evaluated in a well-defined control-flow location;
- canonical *PHI nodes* for induction variables, explicitly representing loop initialization and update.

Modern compilers, including LLVM, provide dedicated transformations to bring loops into canonical form. Further details on induction variables and PHI nodes will be discussed in the following section, while LLVM-specific aspects will be addressed after the loop-related discussion.

A final aspect to be considered concerns *nested loops*. Given two natural loops a and b, identified by their respective back-edges and with headers h_1 and h_2, if h_1 dominates h_2 then all nodes belonging to loop b are included within loop a. In this case, loop b is referred to as the *inner loop* of a, while a represents the *outer loop*.

2.3.3. Loop properties

There are three fundamental aspects that are worth introducing in order to understand the methodology adopted in this work. The first concerns the so-called *loop-invariant* expressions, the second addresses *induction variables*, and the last one focuses on the *trip count* of a loop.

An instruction is said to be *loop-invariant* when, once executed in the first iteration of the loop, it produces a value that remains unchanged for the entire duration of the loop execution. More formally, an instruction d of the form

$$t := a_1 \text{ op } a_2$$

is loop-invariant if at least one of the following conditions holds:

- each operand a_i is a constant;
- all definitions of a_i that reach d are outside the loop;
- there exists exactly one definition of a_i reaching d, and this definition itself is loop-invariant.

Loop-invariant instructions may be explicitly introduced by the programmer or, more frequently, generated as a result of compiler transformations. Whenever it can be guaranteed that moving such an instruction does not alter the program semantics, it can be hoisted to the *loop preheader* and executed only once, instead of being recomputed at each loop iteration.

The correctness of this transformation is established through specific data-flow analyzes; a detailed treatment can be found in the text referenced by Aho et al. [2].

Before introducing the concept of *induction variables*, it is useful to briefly describe the *Static Single Assignment* (SSA) form and the so-called *PHI functions*. Many compiler optimizations rely on *def–use* chains of variables, and the adoption of the SSA form represents a significant improvement over this structure.

As previously introduced in the context of LLVM, the SSA form is an intermediate representation in which each variable is associated with a single definition. When a variable may be defined at multiple program points in the original code, the SSA representation introduces distinct versions of that variable to preserve this property.

PHI functions are special constructs that are used to merge values coming from different control-flow paths. The value produced by a PHI function is an SSA variable that serves as

a join point in a node with multiple predecessors, and an argument is associated with each incoming edge. Such functions typically appear at join nodes generated by conditional branches, as well as in loop headers, where they are used to represent variables that are reassigned at each loop iteration.

The analysis of PHI functions plays a particularly important role in recurrence detection, as they make explicit all updates applied to variables during the previous loop iteration.

Once the SSA form and PHI functions have been introduced, it is possible to define *induction variables* within a loop. These variables exhibit regular and predictable behavior, as their values evolve systematically at each iteration.

A *basic induction variable* is defined as a variable v whose evolution between two successive loop iterations is described by the relation

$$v_{i+1} = v_i + c$$

where i denotes the iteration index and c is a constant or, more generally, a *loop-invariant* value. In other words, a variable v is said to be an induction variable if its value changes by a constant amount at each loop iteration.

Starting from a basic induction variable, it is also possible to define *derived induction variables*. An instruction of the form

$$j = a \cdot v_i + b$$

where a and b are constants or loop-invariant values, defines a derived induction variable that belongs to the same family as the variable v. Expressions of this kind are described by *affine functions*, as they represent linear transformations with a constant offset applied to an induction variable.

In the chapter dedicated to the methodology, the concept of an *affine function* will be revisited and renamed, without changing its semantics, as an *affine recurrence*, in order to emphasize the iterative nature of the underlying mechanism.

To conclude the section dedicated to loops, it is necessary to introduce the *trip count*. The trip count is the number of iterations executed by a loop during its execution and represents a key piece of information for many analyses and optimizations applied within loops.

A loop transformed into canonical form typically relies on an induction variable that is updated with a constant step until the exit condition is satisfied. When this informa-

tion is available, the trip count can be determined directly. An example is shown in Algorithm 2.1.

Algorithm 2.1 Example of a loop with a constant trip count

```
1: x ← 0
2: while x < 10 do
3:     do_something()
4:     x ← x + 1
5: end while
```

Since the induction variable x is initialized to 0 and incremented by 1 at each iteration, and the exit condition is $x < 10$, the resulting trip count is equal to 10.

In more general scenarios, where no recognizable induction variable is present or the exit condition depends on multiple factors (possibly non-linear), determining the trip count can become complex and, in the absence of additional information, may require simulating the evolution of the loop. In this work, and in the test suite used for validation, it is assumed that the trip count is derived exclusively through static analysis, without relying on dynamic or profiling-based information.

3 | State of the art

The analysis of value ranges in programs has been addressed through a wide variety of techniques and remains a broad and active research area. Numerous approaches have been proposed in the literature, each characterized by different assumptions, strengths, and limitations, and no single solution can be considered universally superior. The effectiveness of a given Value Range Analysis (VRA) technique strongly depends on factors such as the application domain, the structure of the code analyzed, and the level of precision required.

In practice, VRA can be viewed as a trade-off between two competing aspects: on one side, the accuracy of the inferred ranges, and on the other, the computational complexity and scalability of the analysis. Increasing precision often comes at the cost of longer analysis time and more complexity of the implementation. For this reason, it is essential to carefully evaluate the characteristics of the target problem and identify an appropriate balance between precision and cost.

Among the most significant contributions providing an overview of the state of the art in precision tuning is the survey by Cherubin and Agosta [6]. In this chapter, references to this work are limited to the aspects related to Value Range Analysis, with the aim of highlighting the main challenges, limitations, and open research directions associated with existing approaches.

This chapter first introduces the distinction between static and dynamic approaches to value range analysis. Then it reviews the main techniques proposed in the literature, highlighting their underlying principles and discussing their respective advantages and limitations.

3.1. Static and dynamic value range analysis

In the fourth section of the Cherubin and Agosta survey, the analysis of the value range is identified as one of the fundamental components of the precision tuning process [6]. Knowing the set of values that each variable may assume during program execution helps

prevent overflow errors and allows the reduction of the space of candidate numerical representations during precision reduction. The survey presents several approaches that have been developed over the years, and can be broadly grouped into two main categories: static analysis and dynamic analysis.

3.1.1. Static range analysis

Static analysis refers to analyses performed at compile-time, based solely on the program source code, its structure, and any annotations or metadata provided by the programmer. In this case, no executions of the program are performed and no information is collected about its runtime behavior. As a consequence, static approaches must guarantee the correctness of the estimated ranges for all possible inputs, remaining conservative whenever there is insufficient information available to further restrict the ranges.

The most widely adopted techniques for Value Range Analysis include interval arithmetic, affine arithmetic, and control-flow analysis, especially in the presence of branches and loops. More precise, but computationally more expensive, methods rely on SMT solvers or geometric approaches, such as polyhedral representations. As discussed in the survey, these techniques tend to produce tighter range estimates but suffer from scalability limitations [6]. In general, static approaches are known to infer wider and more conservative ranges; nevertheless, thanks to their input independence and suitability for integration into compilation pipelines, they remain the preferred choice in many application contexts.

3.1.2. Dynamic range analysis

In contrast to static approaches, dynamic range analysis is based on observing and profiling the program behavior during execution. By executing the program one or more times, the collected range estimates are typically more precise, as they are derived from real input data. However, this increased precision comes at the cost of reduced generality, since the results are tightly dependent on the chosen inputs and do not guarantee coverage of all possible execution scenarios. Moreover, the overhead introduced by instrumentation and repeated program execution represents an additional factor to be taken into account.

3.1.3. Trade-offs between soundness, precision and scalability

The literature clearly shows that no family of approaches to value range analysis can be considered dominant in an absolute sense. Each technique is positioned along a trade-off among three fundamental aspects: *soundness*, precision of the inferred ranges, and

scalability. Improving one or two of these aspects generally leads to a degradation of the remaining ones.

- **Soundness**: an essential requirement in static analyses, as it guarantees that no possible program behavior is excluded. In the presence of complex control structures and loops, it is necessary to rely on over-approximations, which can drastically reduce precision.
- **Precision**: a fundamental requirement for precision tuning, since, as discussed in the previous chapter, tighter ranges allow higher accuracy in the computation of fixed-points. However, more accurate estimations introduce high computational complexity.
- **Scalability**: increasing the complexity of abstract domains and analyses improves precision and broadens the range of applicable cases, but inevitably degrades efficiency. As a result, the prevailing trend is to adopt simple abstract domains.

At present, precision is the requirement that is most often sacrificed in order to preserve the other two. However, low precision leads to ranges that are sound but poorly informative, a condition that is frequently identified as a practical limitation of Value Range Analysis. Conversely, no theoretical limitations have been identified, leaving significant room for improvement. Recent studies, including the work presented in this thesis, tend to avoid increasing domain expressiveness and instead focus on improving precision by exploiting structural information of the program.

3.2. Lattice-Based approaches

3.2.1. Abstract interpretation framework

Lattice-based approaches represent the theoretical foundation of a large portion of static program analysis. This formal methodology was introduced in the late 1970s by Patrick Cousot and Radhia Cousot through the framework of *abstract interpretation* [7]. The core idea of this paradigm is to approximate the concrete behavior of a program by means of an abstract representation organized as a lattice, whose abstract domain includes all possible executions of the program with respect to the set of inputs. The lattice is equipped with a partial ordering that reflects the degree of precision of the represented information, allowing different abstract states to be compared and combined.

The analysis proceeds by computing fix-points of the abstract transformations associated with program operations, with the goal of reaching a stable representation that charac-

terizes the program behavior. In this context, the concept of *over-approximation* plays a central role: in order to guaranty *soundness*, the analysis must never exclude any possible program behavior. When information is missing or incomplete, the results must therefore remain sufficiently broad to include all possible executions, accepting a loss of precision as an unavoidable consequence.

In programs that contain iterative constructs, such as loops, the fix-point computation may fail to converge naturally. For this reason, the *widening* and *narrowing* operators are introduced. Widening is used to enforce convergence of the analysis by accelerating the computation of a fix-point, even at the cost of losing precision, while narrowing is subsequently applied to recover part of the lost information by exploiting additional constraints present in the code.

In the context of *Value Range Analysis*, the abstract domain of the lattice is typically represented by the set of ranges associated with the program variables. Each interval includes all the values that a variable may assume during execution. Program operations induce transformations on the lattice elements, propagating information along the control flow. The choice of this domain guaranties computational simplicity and high efficiency through the use of interval arithmetic. An extension of this model, affine arithmetic, increases computational complexity but aims to capture relationships between variables to improve result precision. Additionally, the analysis of constraints present in the code, typically found in branch conditions and loop structures, further contributes to this goal.

3.2.2. Key contributions

Thanks to the strong theoretical foundations of the lattice-based approach, several works have attempted both to extend the abstract interpretation paradigm and to instantiate it within real-world compilers. The former direction is represented by the work of Oh et al. [20], which constitutes one of the first systematic attempts to address the issue of scalability without abandoning the lattice-based abstract interpretation formalism. The core idea is to make the fixpoint computation sparse, by analyzing semantic data dependencies rather than relying solely on syntactic def-use chains. This allows the analysis to avoid propagating irrelevant components of the abstract state, thereby simplifying and accelerating the fixpoint computation while preserving soundness.

Campos et al. [4] propose a more practically oriented approach, which introduce a concrete Value Range Analysis integrated into LLVM. In this work, range analysis is improved through a fixpoint management strategy based on the identification of strongly connected components (SCCs) in the control-flow graph, as well as through extensions of the in-

termediate representation via annotations derived from the information available in the code. The resulting analysis is a lattice-based VRA that is more efficient and better suited for real compilation pipelines.

A second practical example is represented by the TAFFO framework, which develops its own Value Range Analysis as a static analysis pass. Unlike the previous approach, TAFFO explicitly exploits affine algebra and trip count estimation to handle loop constructs more precisely. The analysis models iterative numerical behavior through affine recurrences and subsequently instantiates the resulting bounds using trip count information, with the goal of deriving tighter value ranges in iterative contexts that are particularly relevant for precision tuning applications, as described in Section 3.2.2 of the original TAFFO thesis [8].

3.2.3. Limits

More expressive abstract domains also exist, such as polyhedral domains. These extensions allow for more accurate estimations, but exhibit evident scalability limitations.

The literature highlights that lattice-based approaches constitute a solid and theoretically well-founded basis for static range analysis, while still presenting practical limitations mainly related to the handling of iterative constructs. In particular, for such constructs, the combination of over-approximation and widening tends to progressively enlarge the inferred ranges, reducing the effectiveness of subsequent precision tuning phases. However, these limitations do not represent an intrinsic weakness of the paradigm, but rather a consequence of the design choices required to guarantee soundness and scalability.

In light of these considerations, several recent works aim to improve the precision of lattice-based analyses without abandoning the Abstract Interpretation framework. The approach proposed in this thesis is positioned within this context and exploits structural information of the program, such as computational patterns and loop recurrences, to tighten the inferred ranges while preserving soundness and maintaining integration within a lattice-based static analysis.

3.3. From affine to pattern-recurrence approach

The study of interval arithmetic and affine algebra has represented a fundamental first step in the analysis of relationships among variables, enabling the extraction of additional information without extending or abandoning the abstract domain adopted by lattice-based approaches. However, these tools are inherently limited and prove insufficient

when trying to accurately characterize the behavior of complex iterative constructs. In this section, the main limitations of such approaches and their impact on range analysis are first highlighted. The discussion then introduces a different paradigm based on the recognition and modeling of recurrences, which are employed as means to improve the precision of the analysis.

3.3.1. Limitations of interval and affine arithmetic

The affine algebra provides a simple and effective abstraction for reasoning about linear relationships among variables. However, in the presence of non-linear relations or complex iterative constructs, such simplicity becomes a limitation. In the former case, the difficulty arises from the fact that affine algebra typically assumes linear relationships with fixed coefficients, making it unsuitable for modeling non-linear growth or iteration-dependent behavior.

In the case of loops, variables frequently follow recurrence relations that cannot be faithfully represented by a single affine form, but instead require expressing structured dependencies among multiple variables and across successive iterations. Even the combination with loop unrolling techniques or widening heuristics does not resolve this issue, as also shown by the TAFFO case study (see Chapter 4).

3.3.2. Impact of non-linear operations and loops

The inability of affine algebra to handle non-linear operations leads to the need for conservative over-approximations. When such over-approximations are applied within iterative constructs, imprecision tends to accumulate from one iteration to the next, rapidly degrading the quality of the estimated ranges. Moreover, in the absence of a reliable estimation of the number of iterations and in order to guarantee termination of the analysis, it often becomes necessary to apply widening techniques, which further enlarge the computed ranges.

A possible direction to mitigate these limitations has been investigated by Darulova and Kuncak [9], who propose the use of satisfiability solvers to iteratively refine estimated ranges and correct under-approximations introduced by initial abstractions. These ideas have been subsequently concretized in the *ROSA* tool, which combines affine analysis with constraint solving to improve the estimation of numerical errors in programs involving non-linear operations.

3.3.3. Scalar Evolution

To improve the precision of range estimation in iterative contexts, approaches have been proposed that aim to identify relationships among variables by exploiting loop-oriented analyses, that is, by leveraging the structural characteristics of loops. Such analyses make it possible to derive representations that are richer in information than those obtainable through affine algebra alone.

One of the central aspects of this line of work is the recognition and exploitation of chains of recurrences, whose recursive nature enables a more accurate description of value evolution across loop iterations. A concrete realization of this approach is provided by the LLVM module known as Scalar Evolution (SCEV), discussed by J. Absar during the EuroLLVM Developers' Meeting in 2018 [1]. SCEV constitutes one of the main starting points for several loop optimizations in LLVM and represents a significant reference for pattern-based approaches.

3.4. Other approaches

The following section considers some additional approaches, already mentioned earlier, in order to complete the overview of the state of the art and to motivate the decision not to adopt them in this work. In particular, *profile-driven* approaches for dynamic range analysis are firstly discussed, then *polyhedral domains*, and finally methods based on *Satisfiability Modulo Theories* (SMT).

3.4.1. Profile-driving approaches

Dynamic analyses base their results on repeated executions of the program and on the observation of values produced at runtime. Such analyses range from techniques aimed at estimating dynamic value ranges and deriving the required data bit width to the creation of *shadow execution* environments capable of tracking the evolution of values during program execution. Among the most representative works are *Autoscaler for C* [14] and the approach proposed by Menard *et al.* [18], both widely discussed in the precision tuning literature. A more exhaustive and detailed overview of these techniques is provided in Section 3.2.2 of the survey by Agosta and Cherubin [6].

Profile-driven approaches can also be combined in a hybrid fashion with static analyses in order to refine their results. However, they are strongly dependent on the input data used during the profiling phase. In the context of the present work, which aims at improving Value Range Analysis within a compiler pipeline oriented toward automatic

precision tuning, this input dependency represents a significant limitation and discourages the adoption of dynamic analyses.

3.4.2. Polyhedral domains

Another approach to static analysis consists in enriching the abstract domain by including relations among multiple variables expressed as systems of linear inequalities. The conjunction of such constraints defines regions of the state space that approximate the set of possible program behaviors. Value Range Analysis can clearly benefit from this increased expressiveness, as it enables the computation of tighter ranges and a more accurate description of nested loops governed by multiple induction variables.

An in-depth study of this approach is provided by Bygde *et al.* [3], which shows that, in the context of VRA, the risk of losing soundness is concrete, but can be mitigated at the cost of a significant increase in the complexity of the analyses. In a setting focused on automatic precision tuning, this limited scalability makes polyhedral approaches unsuitable for application to real-world code.

3.4.3. SMT

The last class of static analysis techniques worth mentioning is represented by SMT solvers, which can also be applied in the context of Value Range Analysis. Although these techniques can provide highly precise and formally verified error bounds, they mainly operate as constraint-solving and verification backends rather than as automatic range inference mechanisms. As such, they are complementary to, but not a replacement for, scalable Value Range Analysis in a general-purpose compiler pipeline, and are therefore outside the scope of this work.

4 | TAFFO

TAFFO (Tuning Assistant for Floating Point To Fixed Point Optimization) [5, 8] is a precision tuning framework whose goal is to replace, whenever possible, floating point operations with fixed point representations. The framework is based on LLVM and integrates into its compilation ecosystem. In this chapter, the pipeline of passes to enable the *precision tuning* process is presented first. The main features of the annotation system, which are also relevant for the value range analysis, are then discussed. Finally, the chapter concludes with an overview of the state of the art of the current VRA, highlighting its main limitations. The complete documentation and source code of the TAFFO framework are available at [23].

4.1. Pipeline

TAFFO implements a pipeline composed of six LLVM passes, each of which plays a well-defined role within the precision tuning process. In addition, a feedback estimator is included to evaluate the conversion outcome. All the details concerning the TAFFO framework and its components are extensively described in [8]. In this work, these topics are instead introduced in a simplified manner, with the goal of providing the reader with the essential background required to understand the context and the techniques developed. Figure 4.1 shows these passes in their order of execution, while the explanation is given below.

The TAFFO pipeline is composed of the following LLVM passes, each playing a specific role within the precision tuning process:

- **Data Type Deducer Pass**: the knowledge and correct distinction of the types associated with numerical variables represent a fundamental prerequisite for determining which of them can be subject to analysis and subsequent conversion. Before the introduction of *opaque pointers* in the more recent versions of LLVM, such information was explicitly encoded, and therefore directly accessible within the LLVM-IR. This design choice was later revised with the aim of reducing the size

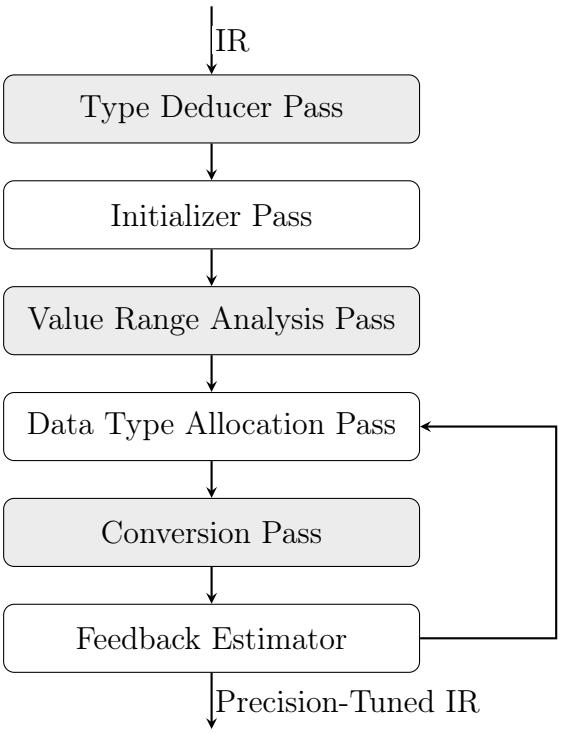


Figure 4.1: TAFFO precision tuning pipeline implemented as a sequence of LLVM passes.

of the IR and improving the compilation times. However, the removal of explicit information about the pointed-to type has affected all those static analyses that rely on precise knowledge of the underlying types. The work presented in Nicolosi et al. [19] explicitly addresses this issue by proposing a methodology for reconstructing pointed-to types directly from the LLVM-IR in the presence of *opaque pointers*. This study led to the development of the *Type Deducer Analysis* pass (TDA), capable of systematically inferring the type information required by subsequent analyses. Within TAFFO, the TDA is integrated as a wrapper, enabling the framework to recover the type information that is essential for the Value Range Analysis and for *precision tuning* transformations.

- **Initializer Pass**: this pass is responsible for identifying and collecting all code regions that are relevant to precision tuning. During this phase, annotations present in the source code are read and the metadata that will be propagated along the pipeline are initialized. In addition, for each function call appearing in the control flow, a specialized copy of the function is created. This approach allows the analysis to be customized according to the invocation context, thus preserving a higher level of precision.

- **Value Range Analysis Pass**: this pass reconstructs, through control flow analysis,

the minimum and maximum values that each variable may assume during program execution. Particular attention is devoted to variables that are recursively updated within loops, which require more sophisticated analysis techniques. As this pass represents the core of this work, the Value Range Analysis is discussed both in the state-of-the-art (chapter 3) and in detail in the methodology (chapter 5).

- **Data Type Allocation Pass**: based on the type and range information produced by the previous passes, this pass assigns a potential new numerical type to each variable as a function of the estimated range. Since an excessive heterogeneity of fixed-point representations may negatively affect performance, the pass attempts to merge similar fixed-point types whenever possible. In addition, duplicated functions whose arguments are identical are removed, reducing the overall code size.
- **Conversion Pass**: after the analysis passes have been executed, this pass applies the actual code transformation. In this phase, floating-point variables identified as convertible are replaced with the corresponding fixed-point representations determined by the Data Type Allocation Pass, while variables that do not meet the conversion criteria remain unchanged.
- **Feedback Estimation Pass**: this final pass evaluates the benefits obtained from the conversion in terms of numerical precision and performance. If the result is not deemed satisfactory, the Data Type Allocation Pass may be re-executed with different parameters, in order to explore alternative solutions and achieve a better trade-off between accuracy and performance.

4.2. Annotation

An additional important feature of TAFFO is the possibility to insert annotations directly into the source code, with the purpose of enabling explicit interaction between the programmer and the precision tuning process. This mechanism is inherited from the LLVM infrastructure and preserves all of its functionalities.

A key aspect of annotations is that their usage does not require any modification to the compiler front-end. Moreover, compilers that do not support these annotations simply ignore them, without altering the standard compilation workflow. This makes annotations a flexible and non-intrusive mechanism.

An example of an annotation is provided below. It specifies that the associated variable should be treated as a scalar and that its value range is bounded between -5 and 5. The term *scalar* refers to the category of numerical data types and indicates that the variable

represents a single value, even when such a value is used within vector-like structures or operations.

```
__attribute__((annotate("scalar(range(-5,5))")))
```

4.2.1. Annotations and Value Range Analysis

Annotations play a particularly important role during value range analysis, even within the method proposed in this thesis. In particular, two main cases can be identified in which their presence provides essential support to the VRA.

The first case concerns the initialization of values that are unknown at compile time, as they depend on external inputs or data that cannot be determined statically. In the absence of additional information, ensuring the soundness of the analysis becomes difficult, since an incorrect or excessively over-approximated initialization of ranges may compromise the correctness of the results from the very early stages of propagation.

The second case is related to the management of the fallback mechanisms at the end of the analysis. When it is not possible to determine a sufficiently precise range for a given value, the presence of an annotation on the result allows avoiding the assignment of an overly conservative interval, typically corresponding to the widest bounds of the representation domain. In this way, annotations help to preserve a minimum level of precision even in cases where the analysis is not fully conclusive.

4.3. Current TAFFO Value Range Analysis pass

4.3.1. Value categories

At the current state, *TAFFO* focuses its efforts on the recognition and management of a well-defined set of internal representations, specifically selected to be compatible with the *precision tuning* process. In particular, the Value Range Analysis (VRA) supports the following categories of values:

- **Scalars**, including all numerical representations, both floating-point and integers. This category also encompasses vectors of arbitrary dimensionality, which are treated as extensions of scalar types.
- **Structures**, namely aggregates of heterogeneous data that share a common semantic meaning. For each field of a structure, the same conditions defined for scalar representations apply.

- **Pointers**, intended as memory addresses used to access scalar values or structures allocated in memory.
- **GEP instructions (GetElementPtr)**, which compute the address of a specific element within an aggregate in memory starting from a base pointer and one or more indices.

The current version of the VRA handles value ranges differently depending on the specific category. For **scalar values**, the range is defined as the minimum and maximum interval of values that a variable may assume at run time. In the case of **vectors**, the associated range represents a *summary*, namely the minimum and maximum value that can be observed across all elements and indices.

From a theoretical perspective, it would be possible to associate a distinct range with each individual element of a vector, enabling a more precise *fine-grained* analysis. However, such an approach would introduce a significantly higher computational cost. The method presented in this work does not introduce changes in this direction, and therefore continues to operate on summary-based ranges.

Structures are handled similarly to scalar values, with the distinction that ranges are maintained separately for each field, reflecting the different semantic roles of the individual properties.

Finally, **pointers and GEP instructions** do not possess an intrinsic range, since associating a numerical interval to a memory address would be semantically meaningless. Nevertheless, these values are exploited as auxiliary mechanisms to retrieve the range of the data referenced. In the specific case of GEP instructions, the associated range corresponds to the summary range of the vector or aggregate to which the access refers.

4.3.2. Scope management and range storage

During the analysis, the ranges associated with each value are stored in dedicated containers, which effectively act as *scopes*. These structures explicitly model the visibility and validity of ranges along the program control flow.

At the global level, the `VRAGlobalStore` represents the root scope of the analysis. This storage is initialized with all global values that can be statically collected from the code. Moreover, at the end of the analysis pass, it serves as the final collection point for the results, containing the ranges associated with all analyzed instructions.

During the execution of the analysis, additional local storages are created. Among

them, `VRAFunctionStore` instances are associated with individual functions and preserve the ranges of formal arguments, as well as those representing function return values. Each `VRAFunctionStore` also acts as a container for the storages related to the LLVM `llvm::BasicBlock`s belonging to the function.

The storages associated with `llvm::BasicBlock`s, referred to as `VRAAnalyzer`, maintain the ranges of the instructions contained in the block, representing the state resulting from the last iteration of the analysis. This aspect is particularly relevant in the presence of iterative constructs, where such states reflect the convergence point, or an approximation thereof, reached during loop analysis.

4.3.3. Limitations of the current VRA

In the current implementation of the VRA, two main limitations can be identified, both of which are addressed in this work.

The first limitation concerns loop handling. When the analysis reaches basic blocks belonging to a loop, it re-executes the analysis of all loop blocks for a fixed number of iterations, effectively simulating a controlled loop unrolling. This approach imposes a trade-off between the unrolling depth and the actual number of loop iterations, and therefore between the temporal and spatial complexity of the analysis.

When loops have a high trip count, two additional issues arise. On the one hand, many instructions are recomputed repeatedly, leading to a progressive over-approximation of the resulting intervals. However, in order to prevent excessive compilation times, it becomes necessary to introduce a limit on the number of analyzed iterations, which further degrades the overall precision.

Currently, these issues are mitigated by limiting loop unrolling and by making extensive use of annotations, both on input data and on intermediate and final compilation steps.

The second limitation is directly related to the use of such annotations. The need to explicitly control and annotate ranges across different stages of the code increases the required manual effort and introduces a large number of tunable control points whenever high precision is desired. Besides increasing management complexity, this approach reduces the generality and reusability of the analysis.

By exploiting recurrence information, it is possible to overcome this compromise: instructions inside loops are analyzed as a function of the number of detected recurrences, rather than of the actual number of loop iterations. As a result, the main control parameter remains associated with the input data only, significantly simplifying the overall

management of the analysis.

A further aspect, of a more theoretical nature, concerns the loss of temporal information within loops. In the current model, it is not possible to determine the interval assumed by a variable at a specific iteration without explicitly introducing intermediate checkpoint states into the analysis. Recurrences, on the other hand, provide a parametric representation that allows recovering the interval associated with a given iteration, thus enabling more refined and structured analyses whenever required.

5 | Methodology: a pattern-recurrence-based VRA

This chapter proposes a method for identifying recurring behaviors within loops, based on the analysis of patterns present in the intermediate representation. The chapter begins by classifying several known types of recurrences, ranging from the simplest forms to more advanced ones, thus providing the necessary context for the subsequent methodology. It then proceeds with the description of the inspection and assembly phases of the recurrences, and finally concludes with the complete presentation of the *RA-VRA algorithm*, illustrated in all its phases and further detailed in its most relevant steps through pseudocode.

5.1. Classes of recurrences

The proposed methodology is based on the identification and exploitation of multiple classes of *recurrences*, directly recognized within the intermediate representation of the program, with the goal of computing value ranges by leveraging their intrinsic structural and mathematical properties. This work introduces and formalizes a set of recurrence types that are considered sufficiently expressive to cover a wide class of loop patterns. Nevertheless, the approach remains extensible: additional recurrence classes can be integrated whenever it becomes necessary to broaden the application domain or improve the precision of the analysis.

From a *semantic* perspective, a recurrence can be interpreted as a function

$$f(k)$$

that describes the value—or, in the context of Value Range Analysis (VRA), the range of values—assumed by the modeled variable at iteration k of the loop. This functional interpretation provides a unified formal foundation for describing the evolution of values over time, independently of the specific recurrence class being employed.

Each recurrence is characterized by a set of coefficients that determine its evolution. In classical models, these coefficients are typically scalar constants; in the context of Value Range Analysis, however, this model must be generalized by allowing coefficients to be *intervals* themselves. In this sense, a constant value V represents a special case and is equivalent to a degenerate (point) interval $[V, V]$.

In order to make usable by the analysis, a recurrence must expose a common interface that allows querying the range assumed at a given iteration. In particular, each recurrence class is required to provide the method:

$$\texttt{at}(k)$$

which returns the range of values assumed by the recurrence at iteration k (or an over-approximation thereof). This abstraction enables a clear separation between the recognition and construction of recurrences and the subsequent range propagation phase, making the analysis modular, sound, and easily extensible.

A fundamental aspect of recurrence analysis is determining whether a recurrence admits a *closed-form representation*, since, when available, it allows the `at(k)` method to be evaluated with constant time complexity $O(1)$, significantly reducing the computational cost of the analysis. Instead, when such a representation cannot be derived, soundness must be preserved by resorting to accumulation operators—such as summation, product, or convex hull constructions—whose evaluation typically incurs a linear cost $O(N)$, with N denoting the number of loop iterations, as they safely over-approximate the set of values assumed across iterations.

The following sections present the recurrence classes analyzed and implemented in this work, highlighting their defining characteristics and the strategy used to compute the `at`(k) method, including whether or not a closed-form expression is available. Throughout the remainder of this work, recurrences are described using a mathematical notation $R(k)$, which denotes the semantic value (or range of values) assumed by the recurrence at iteration k. In the implementation of the analysis, this quantity is computed through the method `at(k)` exposed by each recurrence class. The two notations are semantically equivalent, and therefore, throughout the following sections, the relation

$$\texttt{at}(k) \equiv R(k)$$

is assumed without further clarification.

The recurrences analyzed in this section are grounded, from a mathematical perspective, in

the well-established theoretical framework of difference equations, adopting as a primary reference the textbook *Concrete Mathematics* by Graham, Knuth, and Patashnik, which provides a systematic treatment of affine, geometric and linear recurrences [11].

5.1.1. Affine recurrence

The simplest and most common form of recurrence encountered in loop-based programs is the *affine recurrence*. It models a value that increases or decreases by a constant amount across loop iterations, thus exhibiting a linear evolution with respect to the iteration index.

Let r_0 denote the initial value of the recurrence. It can be defined by the following system:

$$\begin{cases} R(0) = r_0 \\ R(k) = R(k-1) + c \quad \text{for } k > 0 \end{cases}$$

where c represents a constant increase.

By recursively expanding the definition, a closed-form expression of the recurrence can be derived:

$$R(k) = r_0 + k \cdot c$$

The availability of a closed-form representation allows the value assumed by the recurrence at a generic iteration k to be computed directly, with constant time complexity.

In the context of Value Range Analysis, affine recurrences can be generalized to the case in which the increment is not a scalar constant but an interval of values. Denoting by

$$C = [c_{\min}, c_{\max}]$$

the interval associated with the increment, the system becomes:

$$\begin{cases} R(0) = r_0 \\ R(k) = R(k-1) + C \quad \text{for } k > 0 \end{cases}$$

In this case, the closed-form expression is interpreted using interval arithmetic, yielding:

$$R(k) = r_0 + k \cdot C = r_0 + [k \cdot c_{\min},\ k \cdot c_{\max}]$$

This formulation preserves the soundness of the analysis and enables the direct computation of the range assumed by the variable at a specific iteration, while maintaining constant computational cost even in the presence of non-pointwise coefficients.

Example of an affine recurrence in code The following code snippet illustrates a typical affine recurrence commonly found in loop-based programs, where a variable is updated by a constant increment at each iteration:

```
int x = 10;
for (int i = 0; i < N; i++) {
  x = x + 2;
}
```

Listing 5.1: Affine recurrence example

In the example shown in 5.1, the variable `x` is initialized to a constant value before entering the loop and is subsequently incremented by a fixed amount at each iteration. This behavior can be modeled as an affine recurrence by identifying the initial value and the constant increment:

$$r_0 = 10, \qquad c = 2$$

The corresponding recurrence is therefore defined as:

$$\begin{cases} R(0) = 10 \\ R(k) = R(k-1) + 2 \quad \text{for } k > 0 \end{cases}$$

By unfolding the recurrence, the following closed-form expression is obtained:

$$R(k) = 10 + 2k$$

Table 5.1 reports the evolution of the value across the first iterations of the loop.

Iteration	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
Value $R(k)$	10	12	14	16	18	20

Table 5.1: Evolution of an affine recurrence with $r_0 = 10$ and $c = 2$.

Setting $k = 5$ is immediately computed $R(5) = 10 + 2 * 5 = 20$.

This representation makes explicit the linear evolution of the variable x across loop iterations and allows the range assumed at a given iteration, as well as over the entire loop execution, to be computed efficiently once a bound on the trip count is available.

Example of an affine recurrences with composed increments In addition to the introductory example, it is useful to present further cases that frequently occur in real-world loops. In particular, it is important to emphasize that an affine recurrence does not necessarily require a single addition or subtraction, but it may also arise from a concatenation of multiple linear arithmetic operations along the instruction chain involved in the loop.

Consider the following example:

```
int c = 2;
for (int i = 0; i < N; i++) {
    float b = c + 2;
    c = b + 3;
}
```

Listing 5.2: Affine recurrence with composed increments

In this loop, variable c represents the state of the recurrence, while variable b is a temporary variable whose value is not carried across successive iterations. Although b is reassigned at each iteration, it does not form a recurrence, since it does not propagate state information through the loop, but merely contributes a constant offset to the update of c.

By inspecting the loop body, it can be observed that the effective increment applied to c at each iteration is given by the algebraic sum of the constants appearing in the instruction chain, namely $2 + 3 = 5$. As a result, the recurrence on c can be modeled as a first-order affine recurrence, with initial value equal to 2 and constant step equal to 5.

Table 5.2 reports the evolution of the values across the first iterations of the loop.

Iteration	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
Value $b(k)$	–	4	9	14	19	24
Value $c(k)$	2	7	12	17	22	27

Table 5.2: Evolution of the affine recurrence induced by a composed increment with $c_0 = 2$ and step 5.

The observed behavior is consistent with the closed-form representation of the recurrence:

$$R(k) = 2 + 5k$$

from which it follows that, after N iterations, the value of `c` is given by $c_N = 2 + 5N$. For the case considered with $N = 5$, this yields $c_5 = 27$, confirming the final range:

$$c \in [2,\ 27]$$

This example shows how chains of linear operations, even when distributed across multiple intermediate instructions, can be reduced to a single affine recurrence through instruction-chain analysis, without loss of precision and without requiring iterative loop interpretation.

Affine recurrences on arrays and loop-carried dependencies When working with indexed data structures such as arrays, affine recurrences may arise in the presence of *loop-carried dependencies*, that is, dependencies between successive iterations of a loop involving memory locations.

Consider the following example:

```
const int N = 5;
float arr[N] = {3, 5, -1, 2, 6};   // range of arr = [-1, 6]

for (int i = 1; i < N; i++) {
    arr[i] = arr[i - 1] + 5;
}
```

Listing 5.3: Array affine recurrence example

In this case, each element of the array is updated using the value computed at the previous iteration, introducing a direct dependence between `arr[i]` and `arr[i-1]`. This pattern represents a first-order affine recurrence on memory, with a constant step equal to 5.

If all intermediate values of the array were available at compile time, the exact evolution of the recurrence could be reconstructed. By explicitly iterating over the indices, the following progression is observed:

- iteration 1: `arr[1]` $=$ `arr[0]` $+ 5 = 3 + 5 = 8$
- iteration 2: `arr[2]` $=$ `arr[1]` $+ 5 = 8 + 5 = 13$

- iteration 3: `arr[3]` $=$ `arr[2]` $+ 5 = 13 + 5 = 18$
- iteration 4: `arr[4]` $=$ `arr[3]` $+ 5 = 18 + 5 = 23$

As a result, the values effectively assigned during the loop execution belong to the interval $[8, 23]$.

However, in a static analysis context, the initial values of the array are typically known only in terms of a range, in this case $R_{\texttt{arr}}(k) = [-1, 6]$. Applying the closed-form of the affine recurrence with step 5, after $N - 1 = 4$ iterations the resulting range is:

$$R(5) = R_{\texttt{arr}}(k) + 4 \cdot 5 = [-1, 6] + 20 = [-1, 26]$$

The resulting interval is wider than the one observed at runtime, since the first value of the recurrence does not necessarily coincide with one of the range extremes. Nevertheless, this approximation is correct and fully sound, as it safely includes all the possible values that the array elements may assume in the absence of more precise information about indices and intermediate values.

This example highlights how, in the case of affine recurrences on memory, some loss of precision is a natural consequence of abstraction, but represents a necessary trade-off to preserve the soundness of the analysis.

Affine recurrences in reductions A particular case of affine recurrence, yet extremely common in practice, is represented by *reductions*. In this pattern, a variable progressively accumulates values coming from a data structure, typically an array, through an associative operation such as addition.

Consider the following example, where the values of the array are known at compile time only in terms of a range:

```
1 // range of arr = [-2, 3];
2 float sum = 0;
3 for (int i = 1; i < N; i++) {
4     sum += arr[i];
5 }
```

Listing 5.4: Affine recurrence in reductions

In this case, the variable `sum` is updated at each iteration by adding a value read from the array. Since `arr[i]` is not a constant but belongs to an interval, the increment applied

to `sum` is not constant, but *interval-based*.

Assuming $R_{\texttt{arr}}(k) = [-2, 3]$, the evolution of the range of `sum` across the iterations of the loop is reported in Table 5.3.

Iteration	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
Range $R_{\texttt{sum}}(k)$	0	$[-2, 3]$	$[-4, 6]$	$[-6, 9]$	$[-8, 12]$	$[-10, 15]$

Table 5.3: Evolution of a reduction modeled as an affine recurrence with interval-based increment.

In this scenario, unlike classical affine recurrences with a constant step, both the minimum and the maximum of the range evolve with the number of iterations, since the increment may also assume negative values.

The recurrence can be described through the following closed-form expression:

$$R(k) = 0 + k \cdot [-2, 3]$$

From which, for $k = 5$, it follows that:

$$R(5) = [-10, 15]$$

This example shows how reductions can be modeled as affine recurrences with interval-based increments, preserving the soundness of the analysis even in the absence of precise information about the intermediate values of the array.

Reductions from matrices to vectors Applying the same principle, reductions operating on two-dimensional data structures such as matrices and producing one-dimensional vectors exhibit a behavior analogous to array reductions.

In the case shown in listing 5.5, for each fixed value of `i`, the inner loop performs an additive reduction over the elements of the matrix row `mat[i][j]`, accumulating the result into `cum[i]`. This update introduces a cumulative recurrence on the inner loop, fully analogous to the one observed in the one-dimensional case.

```
// range of mat = [-1, 4];
const int N = 4;
const int M = 5;

for (int i = 0; i < N; i++) {
    cum[i] = 0;
    for (int j = 0; j < M; j++) {
        cum[i] += mat[i][j];
    }
}
```

Listing 5.5: Reductions from matrices to vectors example

Assuming that the matrix values are known at compile time only in terms of a range $R_{\texttt{mat}}(k) = [-1, 4]$, the evolution of the range of `cum[i]` across the iterations of the inner loop is reported in Table 5.4.

Iteration	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
Range $R_{\texttt{cum[i]}}(k)$	0	$[-1, 4]$	$[-2, 8]$	$[-3, 12]$	$[-4, 16]$	$[-5, 20]$

Table 5.4: Evolution of a matrix-to-vector reduction modeled as a cumulative affine recurrence.

At the end of the inner loop, for each index `i`, the value of `cum[i]` therefore belongs to the interval:

$$R_{\texttt{cum[i]}}(k) = [-5, 20]$$

Since this estimate holds for every value of `i`, it follows that the overall range of the vector `cum` is:

$$R_{\texttt{cum}}(5) = [-5, 20]$$

In general, this rule applies to any additive reduction that maps an N-dimensional data structure to an $(N-1)$-dimensional one, preserving the soundness of the analysis even in the presence of partial information about intermediate values.

A similar situation arises when an accumulation is present and the entire update expression is repeated across an additional induction variable that is not used for indexing the variable involved in the recurrence.

Consider the following example:

```
// range of res_mat = [0, 0];
// range of src_mat = [2, 5];
const int T = 5;
const int N = 5;
const int M = 5;

for (int t = 0; t < T; t++) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            res_mat[i][j] += 3 * src_mat[i][j];
        }
    }
}
```

Listing 5.6: Repeated affine accumulation across an extra loop dimension

In this case, the update

$$\texttt{res\_mat[i][j]} \mathrel{+}= 3 \cdot \texttt{src\_mat[i][j]}$$

is repeated T times for each matrix element.

Given

$$R_{\texttt{src\_mat}}(k) = [2, 5],$$

we obtain

$$R_{\texttt{res\_mat}}(k) = 3\dot{R}_{\texttt{src\_mat}}(k) = [6, 15].$$

Since `res_mat[i][j]` is initially in $[0, 0]$, the outer loop over `t` induces an affine recurrence:

$$R_{\texttt{res\_mat}}(k) = R_{\texttt{res\_mat}}(k-1) + [6, 15].$$

The evolution of the range across the $T = 5$ iterations is reported in Table 5.5.

Iteration	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
Range $R_{\texttt{res\_mat}}(k)$	$[0,0]$	$[6,15]$	$[12,30]$	$[18,45]$	$[24,60]$	$[30,75]$

Table 5.5: Evolution of a repeated affine accumulation governed by an additional induction variable.

After completion of the outer loop, the final range is therefore:

$$R_{\texttt{res\_mat}}(5) = [30, 75].$$

Although the induction variable `t` does not appear in the indexing expression of `res_mat`, it controls how many times the affine increment is applied. Consequently, the behavior is again modeled as a affine recurrence whose closed form is

$$R_x(k) = R_x(0) + k \cdot R(\delta),$$

with $R_x(0) = [0, 0]$, $R(\delta) = [6, 15]$, and $k = T$.

It is worth noting the structural difference between the two examples. In the first case, the recurrence evolves along a *spatial* dimension (the index j of the matrix row), whereas in the second case it evolves along a *temporal* dimension (the additional induction variable t). Despite this difference, both situations are formally modeled by the same affine recurrence, confirming the generality of the proposed abstraction. Within the context of this work, and in particular during the validation phase, a clear distinction was made between the cases of *affine flattened on S* (spatial) and *affine flattened on T* (temporal).

5.1.2. Delta affine recurrence

When an affine recurrence spans multiple nested loops, it is necessary to account for how each loop trip count contributes differently to the accumulation of the final value. In such cases, the increment observed at the outer loop level does not directly correspond to the increment applied in the inner loop, but rather depends on the complete result of the nested recurrence.

Consider the following example:

```
1 float sum = 0;
2 for (int i = 0; i < 4; i++) {
3     for (int j = 0; j < 3; j++) {
4         sum += 2;
5     }
6 }
```

Listing 5.7: Delta affine recurrence example

Within the inner loop, the variable `sum` follows a classical affine recurrence with constant

increment equal to 2. Since the inner loop executes 3 iterations, the total contribution produced by a single iteration of the outer loop is:

$$\Delta_{\text{in}} = 2 \cdot 3 = 6$$

From the perspective of the outer loop, `sum` is therefore not updated with step 2, but with an increment equal to the result of the inner affine recurrence. The recurrence observed at the outer loop level can be expressed as:

$$R_{\text{out}}(k) = R(0) + k \cdot \Delta_{\text{in}}$$

Assuming $R(0) = 0$ and an outer-loop trip count equal to 4, the final value becomes:

$$R(4) = 0 + 4 \cdot (2 \cdot 3) = 24$$

The evolution of `sum` across the iterations of the outer loop is reported in Table 5.6.

Outer iteration	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
Inner increment	–	$2 \cdot 3 = 6$	$2 \cdot 3 = 6$	$2 \cdot 3 = 6$	$2 \cdot 3 = 6$
Value `sum`	0	6	12	18	24

Table 5.6: Evolution of an affine recurrence across nested loops (AffineDelta).

Based on this observation, in this work, such recurrences are referred to as *AffineDelta*, in order to distinguish them from classical affine recurrences and to enable the application of the appropriate closed-form expression, which explicitly accounts for the composition of loop trip counts.

Direct reductions from matrices to scalars as AffineDelta A further relevant case arises when a reduction operates directly on a multidimensional data structure and produces a single scalar value. In this scenario, the dimensionality gap between the source and the destination is greater than one, and the accumulation is performed through nested loops.

Consider the following example:

```
// range of mat = [2, 5];
const int N = 4;
const int M = 3;

float grand_total = 0;
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        grand_total += mat[i][j];
    }
}
```

Listing 5.8: Direct matrix to scalar affine example

In this case, the inner loop performs an additive reduction over the matrix elements for a fixed value of `i`. The instruction `grand_total += mat[i][j]` introduces a cumulative recurrence on the inner loop, where the increment is not constant but belongs to the interval $R_{\texttt{mat}}(k) = [2, 5]$.

Assuming an inner-loop trip count equal to $M = 3$, the overall contribution produced by the inner loop can be summarized into a single interval-based increment:

$$\Delta_{\text{in}} = M \cdot R_{\texttt{mat}}(K) = 3 \cdot [2, 5] = [6, 15]$$

From the perspective of the outer loop, the variable `grand_total` is therefore not updated iteratively with individual elementary contributions, but with the already resolved result of the inner reduction. The recurrence observed at the outer-loop level can thus be modeled as an affine recurrence whose increment depends on the result of another recurrence.

In this work, such a pattern is classified as *AffineDelta*, since the increment applied at each iteration of the outer loop corresponds to the *delta* produced by the inner loop. Formally, the recurrence on the outer loop can be expressed as:

$$R_{\text{out}}(k) = R(0) + k \cdot \Delta_{\text{in}}$$

Assuming $R(0) = 0$ and an outer-loop trip count equal to $N = 4$, the final value of the recurrence is:

$$R(4) = 4 \cdot [6, 15] = [24, 60]$$

The evolution of the range of `grand_total` across the iterations of the outer loop is reported in Table 5.7.

Outer iteration	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
Range $R_{\texttt{grand\_total}}(k)$	0	$[6, 15]$	$[12, 30]$	$[18, 45]$	$[24, 60]$

Table 5.7: Evolution of a direct matrix-to-scalar reduction modeled as an AffineDelta recurrence.

This example shows how direct reductions from multidimensional data structures to scalars can be treated as AffineDelta recurrences, where the composition of nested-loop trip counts is explicitly modeled. Such a representation enables the application of a correct and sound closed-form expression, preserving precision within the limits imposed by compile-time information.

Although not addressed in this work, the same principles could be extended, under the assumption of independence between dimensions and affine update semantics, to reductions involving multiple spatial dimensions by recursively composing delta operators across each dimension. Likewise, in the presence of multiple temporal dimensions, successive delta compositions could model the cumulative effect of nested iterative dependencies along the temporal axes.

5.1.3. Crossing affine recurrence

A final affine recurrence pattern addressed in this work involves multiple variables that, when considered individually, do not appear to produce a linear recurrence. However, when analyzed jointly, they can be modeled as *crossing affine recurrences*.

Consider the following example:

```
float a = -3;
float b = 2;
for (int i = 0; i < 5; i++) {
    a = b + 5;
    b = a - 1;
}
```

Listing 5.9: Crossing affine recurrence example

The evolution of the variable values across the loop iterations is reported below:

Iteration	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
Value a_k	-3	7	11	15	19	23
Value b_k	2	6	10	14	18	22

Table 5.8: Evolution of a crossing affine recurrence with coupled variables.

After the first iteration, a stable linear behavior emerges, where both variables increase linearly and maintain a constant relationship between each other.

By inspecting the loop body, the variable updates can be expressed as the following system of coupled recurrences:

$$\begin{cases} a_k = b_{k-1} + 5 \\ b_k = a_k - 1 \end{cases}$$

Substituting the first equation into the second yields:

$$b_k = (b_{k-1} + 5) - 1 = b_{k-1} + 4$$

Thus, variable `b` follows a classical affine recurrence with constant increment equal to 4. Analogously, substituting the expression for b_{k-1} into the first equation leads to:

$$a_k = a_{k-1} + 4$$

Therefore, starting from the first iteration, both variables follow an affine recurrence with step equal to 4. However, these recurrences are *coupled*, since each variable depends (directly or indirectly) on the other.

Assuming the stable initial state given by the values at the first iteration:

$$a_1 = 7 \qquad b_1 = 6$$

the closed-form expressions of the two recurrences are:

$$\begin{aligned} a_k &= 7 + (k-1) \cdot 4 \\ b_k &= 6 + (k-1) \cdot 4 \end{aligned} \quad \text{for } k \geq 1$$

This confirms that both variables are affine with the same increment, but exhibit a *cross-dependency*, which requires them to be analyzed as a coupled system rather than as independent recurrences.

This example shows that some affine recurrences are not immediately recognizable when variables are analyzed in isolation. A joint dependency analysis, however, makes it possible to identify a stable affine behavior. In this work, such patterns are referred to as *crossing affine recurrences*, highlighting the need for a system-based recurrence analysis.

5.1.4. Geometric recurrence

When the accumulation is not generated by a linear variation but by a *multiplicative coefficient*, the recurrence falls into the class of *geometric recurrences*. In this case, the evolution of the value is no longer linear, but *exponential*.

Let r_0 denote the initial value of the recurrence. A geometric recurrence can be defined by the following system:

$$\begin{cases} R(0) = r_0 \\ R(k) = a \cdot R(k-1) \quad \text{for } k > 0 \end{cases}$$

where a is a constant multiplicative coefficient.

By recursively expanding the definition, the following *closed-form expression* is directly obtained:

$$R(k) = a^k \cdot r_0.$$

In the context of Value Range Analysis (VRA), it is necessary to extend this model by considering that the multiplicative coefficient may not be pointwise, but instead represented

by an interval of values of the form:

$$A = [a_{\min}, a_{\max}].$$

In this case, the system becomes:

$$\begin{cases} R(0) = r_0 \\ R(k) = A \cdot R(k-1) \quad \text{for } k > 0 \end{cases}$$

and the evolution of the recurrence is described by the expression:

$$R(k) = A^k \cdot r_0,$$

interpreted using interval arithmetic.

The interval associated with the multiplicative coefficient can make the closed-form evaluation *not always the most suitable solution* from a computational and precision standpoint. It is therefore important to analyze the properties of the interval A in order to distinguish different behaviors:

- if $a_{\min} > 1$, the recurrence describes a monotonic exponential growth;
- if $0 < A < 1$, the value decreases and converges to zero;
- if the interval A contains the value 1, the recurrence may degenerate into an affine behavior;
- if A assumes negative values, the powers A^k alternate in sign, introducing oscillations that may significantly reduce the precision of the analysis.

Example of a geometric recurrence in code The following code snippet illustrates a simple geometric recurrence, where a variable is updated by repeatedly applying a multiplicative factor at each loop iteration:

```
1 int x = 2;
2 for (int i = 0; i < N; i++) {
3   x = 3 * x;
4 }
```

Listing 5.10: Geometric recurrence example

In this example, the variable `x` is initialized before entering the loop and is multiplied by a

constant factor at each iteration. This behavior can be modeled as a geometric recurrence by identifying:

$$r_0 = 2, \qquad a = 3.$$

The corresponding recurrence is therefore defined as:

$$\begin{cases} R(0) = 2 \\ R(k) = 3 \cdot R(k-1) \quad \text{for } k > 0 \end{cases}$$

By recursively expanding the definition, the following closed-form expression is obtained:

$$R(k) = 3^k \cdot 2.$$

Table 5.9 reports the evolution of the value across the first iterations of the loop.

Iteration	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
Value $R(k)$	2	6	18	54	162	486

Table 5.9: Evolution of a geometric recurrence with $r_0 = 2$ and $a = 3$.

Setting $k = 5$ is immediately computed $R(5) = 3^5 * 2 = 486$.

Since the multiplicative coefficient is pointwise and strictly greater than one, the recurrence exhibits a monotonic exponential growth and admits an operationally usable closed-form representation. As a result, the `at(k)` method can be evaluated in constant time for any iteration k.

All cases addressed for affine recurrences, provided that the involved intervals do not include negative values, can be reformulated in a geometric form whenever the reduction operations are multiplicative.

Geometric recurrences in reductions A closely related yet conceptually distinct pattern is represented by *multiplicative reductions*, which naturally give rise to *geometric recurrences*. In this setting, a variable progressively accumulates values coming from a data structure through an associative multiplication operation.

Consider the following example, where the values of the array are known at compile time only in terms of an interval:

```
// range of arr = [2, 3];
float acc = 1;
for (int i = 1; i < N; i++) {
    acc *= arr[i];
}
```

Listing 5.11: Geometric recurrence in reductions example

In this case, the variable `acc` is updated at each iteration by multiplying its current value by an element of the array. Since `arr[i]` is not a constant but belongs to an interval, the multiplicative factor is *interval-based*, inducing an exponential growth of both the lower and the upper bounds of the accumulated range.

Assuming $R_{\mathsf{arr}}(k) = [2, 3]$ and $acc_0 = 1$, the evolution of the range of `acc` across the iterations of the loop is reported in Table 5.10.

Iteration	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
Range $R_{\mathsf{acc}}(k)$	$[1, 1]$	$[2, 3]$	$[4, 9]$	$[8, 27]$	$[16, 81]$

Table 5.10: Evolution of a reduction modeled as a geometric recurrence with interval-based ratio.

In this scenario, both bounds of the interval grow exponentially with the number of iterations. The *lower bound* reflects the repeated multiplication of the minimum possible array value, while the *upper bound* analogously follows the maximum one. This behavior clearly distinguishes geometric recurrences from affine recurrences, in which the growth of the bounds is linear with respect to the iteration count.

The recurrence can be described by the following expression:

$$\mathsf{acc}_k = \prod_{t=1}^{k} \mathsf{arr}[t]$$

which, in terms of intervals, becomes:

$$R(k) = [2, 3]^k = [2^k,\ 3^k]$$

Since the loop iterates for $i = 1, \ldots, N-1$, the corresponding trip count is $TC = N - 1$. Therefore, the final range of the accumulator variable is:

$$R\mathsf{acc}_{\text{final}}(k) = [2^{N-1},\ 3^{N-1}]$$

This example shows how multiplicative reductions can be naturally modeled as *geometric recurrences with interval-based ratios*, in which both bounds evolve exponentially. Once the trip count of the loop is known, the availability of a closed-form representation allows the final range to be computed directly, avoiding iterative evaluation or explicit loop unrolling, while preserving the soundness of the analysis even in the presence of incomplete information about the intermediate array values.

Crossing geometric recurrence The exponential nature of geometric recurrences makes *delta geometric recurrences* of limited practical interest, although they are formally admissible when the involved intervals do not include negative values and therefore do not introduce oscillatory behaviors. More interesting, instead, are *crossing recurrences with multiplicative operations*, in which multiple variables are mutually updated within the loop.

Consider the following example:

```
1 float a = 2;
2 float b = 4;
3 for (int i = 0; i < 5; i++) {
4     a = b * 3;
5     b = a * 2;
6 }
```

Listing 5.12: Crossing recurrence example

In this case, the variables `a` and `b` form a *crossing recurrence*, where each update depends on the freshly computed value of the other variable. Both assignments are multiplicative and introduce constant growth factors, resulting in an overall geometric behavior.

Let k denote the number of completed iterations and let (a_k, b_k) be the values at the beginning of iteration k. The recurrence relations can be expressed as:

$$\begin{cases} a_{k+1} = 3 \cdot b_k \\ b_{k+1} = 2 \cdot a_{k+1} \end{cases}$$

Assuming the initial values $a_0 = 2$ and $b_0 = 4$, the evolution of the variables across the iterations of the loop is reported in Table 5.11.

Iteration	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
a_k	2	12	72	432	2592
b_k	4	24	144	864	5184

Table 5.11: Evolution of a geometric crossing recurrence with multiplicative updates.

It can be observed that both variables grow exponentially while maintaining a *fixed structural ratio* between them. In particular, the system can be reduced to a single geometric recurrence, since:

$$b_{k+1} = 2 \cdot a_{k+1} = 6 \cdot b_k$$

from which it follows that:

$$b_k = 4 \cdot 6^k \qquad \text{and} \qquad a_k = 2 \cdot 6^k$$

This example highlights how multiplicative crossing recurrences, despite involving multiple variables, exhibit a highly regular structure and can be reduced to a global geometric growth. Once such a structure is identified, the availability of a closed-form representation allows the final values (or ranges) to be derived directly, avoiding iterative simulation while preserving the soundness of the analysis.

5.1.5. Linear recurrence

A generalization of the previously introduced recurrence classes is represented by *linear recurrences*. In this model, the evolution of a value still depends linearly on the state assumed at previous iterations, but through a *linear combination* of past values rather than a simple constant increment. Linear recurrences with constant coefficients form a well-known class, typically solved through the characteristic polynomial approach [11, 25].

Let r_0 denote the initial value of the recurrence. A first-order linear recurrence can be defined by the following system:

$$\begin{cases} R(0) = r_0 \\ R(k) = a \cdot R(k-1) + b \quad \text{for } k > 0 \end{cases}$$

where a and b are constant coefficients.

By recursively expanding the definition, the recurrence admits a *closed-form representation*. For the case $a \neq 1$, the cumulative form is given by:

$$R(k) = a^k \cdot r_0 + b \cdot \sum_{i=0}^{k-1} a^i$$

which can be leading the closed form by this simplification:

$$R(k) = a^k \cdot r_0 + b \cdot \frac{a^k - 1}{a - 1}$$

In the special case $a = 1$, the recurrence reduces to an affine recurrence:

$$R(k) = r_0 + k \cdot b$$

while when $b = 0$, the recurrence reduces to a geometric recurrence:

$$R(k) = a^k * r_0$$

The availability of a closed-form expression allows the value (or, in the context of Value Range Analysis, the range of values) assumed by the recurrence at a generic iteration k to be computed directly, making the `at(k)` method evaluable in constant time.

The formulation previously introduced for first-order linear recurrences can be extended to the *interval case*. Considering the recurrence

$$\begin{cases} R(0) = r_0 \\ R(k) = A \cdot R(k-1) + B \quad \text{for } k > 0 \end{cases}$$

where both A and B are intervals, the evolution of the recurrence can be described by the following *cumulative form*:

$$R(k) = A^k \cdot r_0 + B \cdot \sum_{i=0}^{k-1} A^i,$$

with all operations interpreted using interval arithmetic.

When the additive term B is interval-valued, no additional precautions are required, since its contribution is linear and can be safely handled through a final interval multiplication. In contrast, the multiplicative coefficient A requires more careful treatment, as certain

configurations can introduce critical issues in the evaluation of the recurrence.

In particular, whenever possible, the geometric summation can be further *simplified* as:

$$R(k) = A^k \cdot r_0 + B \cdot \frac{A^k - 1}{A - 1}.$$

This simplification is *operationally valid and stable* when the interval A satisfies one of the following conditions: $a_{\min} > 1$, i.e., the interval is entirely greater than one, or $0 < A < 1$, i.e., the interval is entirely contained between zero and one. In these cases, the geometric summation exhibits a monotonic behavior and the division by $A - 1$ is well defined, allowing the `at(k)` method to be evaluated in constant time without resorting to accumulation operators.

Conversely, when the interval A *contains the value one*, the simplification becomes problematic, as the denominator $A - 1$ may include zero, making interval division undefined or excessively conservative. Moreover, when A assumes *negative values*, the powers A^k alternate in sign, introducing oscillations that break the monotonicity of the geometric summation. In such situations, the direct evaluation of the simplified form tends to produce *excessive over-approximations*, significantly reducing the precision of the analysis.

To mitigate this effect and preserve soundness, it is therefore preferable to avoid algebraic simplification and treat the recurrence using a *cumulative form*, based on accumulation operators (such as interval summations or convex hull constructions), which provide a more stable control over over-approximation errors.

Example of a linear recurrence in code The following code snippet illustrates a simple first-order linear recurrence commonly found in loop-based programs, where a variable is updated through a linear combination of its previous value:

```
1 int x = 1;
2 for (int i = 0; i < N; i++) {
3   x = 2 * x + 3;
4 }
```

Listing 5.13: Linear recurrence example

In this example, the variable `x` is initialized before entering the loop and is updated at each iteration by applying a multiplicative and an additive constant. The update can be modeled as a first-order linear recurrence by identifying:

$$r_0 = 1, \qquad a = 2, \qquad b = 3.$$

The corresponding recurrence is therefore defined as:

$$\begin{cases} R(0) = 1 \\ R(k) = 2 \cdot R(k-1) + 3 \quad \text{for } k > 0 \end{cases}$$

By expanding the recurrence, the following closed-form expression is obtained:

$$R(k) = 2^k \cdot 1 + 3 \cdot \frac{2^k - 1}{2 - 1} = 2^k + 3 \cdot (2^k - 1).$$

Table 5.12 reports the evolution of the value across the first iterations of the loop.

Iteration	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
Value $R(k)$	1	5	13	29	61	125

Table 5.12: Evolution of a first-order linear recurrence with $r_0 = 1$, $a = 2$, and $b = 3$.

Since the multiplicative coefficient is pointwise and strictly greater than one, the recurrence admits an operationally usable closed-form representation, allowing the `at(k)` method to be evaluated in constant time. Therefore, setting $N = 5$ is immediately calculated $R(5) = 2^5 + 3(2^5 - 1) = 125$.

Notes on higher-order linear recurrences This class of recurrences can be extended to *higher orders*, depending on how many previous states the value at iteration k depends on. In particular, a second-order linear recurrence can be expressed by the following system:

$$\begin{cases} R(0) = r_0 \\ R(1) = r_1 \\ R(k) = a_1 \cdot R(k-1) + a_2 \cdot R(k-2) + b \quad \text{for } k \geq 2 \end{cases}$$

where a_1, a_2, and b are constant coefficients.

In the *pointwise case*, such a recurrence admits a closed-form solution obtained by solving the associated characteristic equation. The solution can be expressed as a linear combination of exponential terms of the form:

$$R(k) = \alpha \cdot \lambda_1^k + \beta \cdot \lambda_2^k + \gamma,$$

where λ_1 and λ_2 are the roots of the characteristic equation, and α, β, and γ are constants determined by the initial conditions. From a computational standpoint, this closed form

can be evaluated in constant time $O(1)$, similarly to first-order linear recurrences.

When the characteristic roots are not real, the pointwise formulation still admits a general solution that can be expressed in real form and evaluated in constant time $O(1)$. Let the complex conjugate roots be defined as $\lambda_{1,2} = \rho e^{\pm i\theta}$; the general solution can be rewritten as:

$$R(k) = \rho^k \big(A\cos(k\theta) + B\sin(k\theta)\big) + \gamma$$

where ρ denotes the modulus of the roots, θ their argument, and A and B are constants determined by the initial conditions $R(0) = r_0$ and $R(1) = r_1$. The presence of trigonometric terms introduces an oscillatory behavior that may be damped, stable, or divergent depending on the value of ρ (respectively $\rho < 1$, $\rho = 1$, or $\rho > 1$).

The situation becomes more delicate in the interval setting. In this case, the characteristic roots may become intervals of complex numbers, and all the involved parameters assume interval values. Evaluating the closed form therefore requires computing interval powers and trigonometric functions over intervals, which rapidly leads to a loss of correlation between modulus and phase. This typically results in highly conservative over-approximations, even when the underlying pointwise system is stable.

A possible strategy consists in considering only the dominant modulus and ignoring the oscillatory component, thus obtaining a sound but generally looser bound. Alternatively, a cumulative or iterative approach can be adopted for a finite number of steps, extracting the minimum and maximum values effectively reached and achieving a more favorable trade-off between precision and computational cost.

5.1.6. Fake recurrences

A final case, very common in practice, is worth mentioning, since it can easily be mistaken for a recurrence, while in reality it represents a simple assignment.

Consider the following example:

```
1 // range of arr = [-3, 8]
2 for (int i = 0; i < N; i++) {
3     arr[i] = arr[i] + 5;
4 }
```

Listing 5.14: Fake recurrence example

At first glance, the fact that the assignment operates on the same memory object (`arr`) might suggest the presence of a recurrence. However, a closer inspection reveals that each iteration operates on a distinct index of the array, without introducing any dependency between successive iterations. In other words, no *loop-carried dependence* is present.

In a static analysis context where only the range of the array is known, the behavior of the loop can be resolved by applying the addition operation *only once* to the initial range. Starting from $R_{\mathsf{arr}}(k) = [-3, 8]$, a single iteration produces the interval $[2, 13]$. As shown in Table 5.13, subsequent iterations do not lead to any further expansion of the range, which remains stable throughout the execution of the loop.

Iteration	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
Range $R_arr(k)$	$[2, 13]$	$[2, 13]$	$[2, 13]$	$[2, 13]$	$[2, 13]$

Table 5.13: Evolution of an apparent recurrence classified as FakeRecurrence.

As a consequence, the final range of the array never exceeds the interval $[2, 13]$. By joining this interval with the initial one, the overall range becomes:

$$R_{\mathsf{arr}}(k) = [-3, 13]$$

which is fully sound, without the need to execute more than one iteration of the loop.

In the methodology proposed in this work, cases of this kind are still handled within the recurrence framework, but are classified as *FakeRecurrence*. A fake recurrence is a recurrence whose initial value is the range before entering the loop and that, when evaluated at any iteration greater than zero, returns a constant interval corresponding to the result of a single iteration.

The decision not to introduce a separate data structure for these cases is motivated by the fact that, with this strategy, they can participate in the dependency system described in Section 5.3, contributing to the correct resolution order of instructions without compromising either soundness or simplicity of the analysis.

5.1.7. Final remarks and summary

Several additional recurrences can be identified and studied with the aim of future extensions of the proposed method. Due to their limited frequency of occurrence in the benchmark suites used for the validation of this thesis, they have not been implemented, in favor of a deeper focus on the inspection and assembly techniques, which represent the

core contribution of this work and constitute a natural starting point for the introduction of new recurrence patterns.

Among the recurrences that admit a closed-form representation, polynomial recurrences can be mentioned, in which the growth step is itself recurrent (still detailed in [11]).

To further increase the level of precision, if one were to operate not on summary intervals of data structures but on individual elements or their aggregates, it is also worth considering cumulative recurrences. In such cases, a closed-form expression is generally not available; however, it is possible to estimate the values at each iteration and derive a sound interval through a controlled iteration over the involved instructions.

To conclude the section devoted to the classification of recurrences, and with the aim of summarizing and schematizing the analyzed cases, two summary tables are reported. Table 5.14 presents the main recurrence classes assuming constant growth parameters, whereas Table 5.15 shows the corresponding interval-based formulations, in which the growth factors are expressed as intervals.

Recurrence class	Recurrence def.	Closed-form expression
Standard affine	$x_{k+1} = x_k + c$	$x_k = x_0 + kc$
Delta affine	$\begin{cases} x_{k+1} = x_k + \delta_k \\ \delta_k = \delta_0 + sk \end{cases}$	$x_k = x_0 + k\delta_0 + s\dfrac{k(k-1)}{2}$
Crossing affine	$\begin{cases} x_{k+1} = y_k + c_1 \\ y_{k+1} = x_k + c_2 \end{cases}$	Linear system reducible to affine form
Geometric	$x_{k+1} = rx_k$	$x_k = x_0 r^k$
Delta geometric	$\begin{cases} x_{k+1} = x_k + \delta_k \\ \delta_k = \delta_0 r^k \end{cases}$	$x_k = x_0 + \delta_0 \dfrac{r^k - 1}{r - 1}$
Crossing geometric	$\begin{cases} x_{k+1} = ay_k \\ y_{k+1} = bx_k \end{cases}$	$x_k = x_0(ab)^{k/2},\ y_k = y_0(ab)^{k/2}$
Linear (order 1)	$x_{k+1} = ax_k + b$	$x_k = a^k x_0 + b\dfrac{a^k - 1}{a - 1}$
Linear (order 2)	$x_{k+2} = ax_{k+1} + bx_k$	Solution via characteristic polynomial

Table 5.14: Classification of recurrence patterns and their corresponding closed-form expressions.

Recurrence class	Recurrence def.	Closed-form expr.
Standard affine	$x_{k+1} = x_k + C,$ $C \in [c_{\min}, c_{\max}]$	$R(x_k) = R(x_0) + kC$
Delta affine	$\begin{cases} x_{k+1} = x_k + \Delta_k \\ \Delta_k = \Delta_0 + Sk \end{cases}$ $\Delta_0 \in [d_{0\,\min}, d_{0\,\max}]$ $S \in [s_{\min}, s_{\max}]$	$R(x_k) = R(x_0) + k\Delta_0$ $+\frac{k(k-1)}{2}S$
Crossing affine	$\begin{cases} x_{k+1} = y_k + C_1 \\ y_{k+1} = x_k + C_2 \end{cases}$ $C_1 \in [c_{1\,\min}, c_{1\,\max}]$ $C_2 \in [c_{2\,\min}, c_{2\,\max}]$	Interval affine system reducible to closed form
Geometric	$x_{k+1} = x_k \cdot R,$ $R \in [r_{\min}, r_{\max}]$	$R(x_k) = R(x_0) \cdot R^k$
Delta geometric	$\begin{cases} x_{k+1} = x_k + \Delta_k \\ \Delta_k = \Delta_0 R^k \end{cases}$ $\Delta_0 \in [d_{0\,\min}, d_{0\,\max}]$ $R \in [r_{\min}, r_{\max}]$	$R(x_k) = R(x_0) + \Delta_0 \frac{R^k - 1}{R - 1}$
Crossing geometric	$\begin{cases} x_{k+1} = A \cdot y_k \\ y_{k+1} = B \cdot x_k \end{cases}$ $A \in [a_{\min}, a_{\max}]$ $B \in [b_{\min}, b_{\max}]$	$R(x_k) = R(x_0)(AB)^{k/2}$ $R(y_k) = R(y_0)(AB)^{k/2}$
Linear (order 1)	$x_{k+1} = A \cdot x_k + B,$ $A \in [a_{\min}, a_{\max}]$ $B \in [b_{\min}, b_{\max}]$	$R(x_k) = A^k R(x_0)$ $+B\frac{A^k - 1}{A - 1}$
Linear (order 2)	$x_{k+2} = A \cdot x_{k+1} + B \cdot x_k,$ $A, B \in \mathbb{I}$	Solution via interval-based characteristic polynomial

Table 5.15: Classification of recurrence patterns with interval-based formulations and corresponding interval closed-form expressions.

5.2. Potential recurrences inspection

The first step in exploiting recurrences for value range analysis consists in identifying, within the intermediate representation, the instructions involved in their construction. A recurrence is characterized by the presence of a *root node*, which is responsible for accumulating the value computed across the loop iterations, and by a chain of `llvm::Value` instances that describe the operations applied to such value at each iteration.

In LLVM-IR, two main categories of instructions can be identified as candidates for acting as recurrence roots:

- **PHI nodes**, typically located in the *loop header*, which represent in SSA form the evolution of a variable across loop iterations;
- **`store` instructions**, which model the update of a memory location, for instance in the case of arrays or indexed data structures, and allow recurrences to be expressed through memory accesses rather than scalar variables.

Both categories enable the representation of values that depend on the result of the previous iteration, making them suitable starting points for the identification and formal construction of recurrence relations.

A first code fragment that illustrates a recurrence on the variable `sum` is shown below.

```
1 float sum = 0.0;
2 for (int i = 0; i < 5; i++) {
3     sum += 3;
4 }
```

Listing 5.15: SSA recurrence example

The recurrence on `sum` is trivially affine, since at each iteration it is incremented by the constant value 3. Additional examples involving simple scalar variables are presented in the section dedicated to the different classes of recurrences 5.1.

The intermediate representation corresponding to the recurrence shown above is presented in the listing 5.16.

```
for.cond:                                         ; preds = %for.inc, %entry
  %sum.0 = phi float [ 0.000000e+00, %entry ], [ %add, %for.inc ]
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i32 %i.0, 5
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %add = fadd float %sum.0, 3.000000e+00
  br label %for.inc

for.inc:                                          ; preds = %for.body
  %inc = add nsw i32 %i.0, 1
  br label %for.cond
```

Listing 5.16: LLVM-IR of the recurrence shown in listing 5.15

From the IR code it can be observed that the recurrence is accumulated on the PHI node named `%sum.0`. In particular, the initial value of the recurrence is provided by the first *incoming block*, highlighted in blue, which associates the PHI node with the constant value `0.0`. This value represents the *start* of the recurrence.

The second *incoming block*, highlighted in red, identifies the source of the accumulation. In this case, the value `%sum.0` is read as the left operand of the binary operation `add`, also highlighted in red, which implements the increment of the accumulated value.

Following the dependency chain, the `add` instruction is in turn used as the second *incoming value* of the PHI node, thus generating a closed path. It is therefore possible to clearly identify a recurrence whose root is the PHI node, and whose dependency chain forms a closed circuit: starting from the root and following the *def-use chain*, the flow returns to the root itself.

This recurrence chain can therefore be summarized as:

$$\text{PHI}_{\%\text{sum.0}} \rightarrow \text{add} \rightarrow \text{PHI}_{\%\text{sum.0}}$$

For the moment, the specific type of recurrence is intentionally ignored, in order to focus exclusively on the process of identifying all possible recurrences in the code. Once identified, these recurrences can be collected and passed as input to a subsequent phase, which is responsible for managing their analysis and scheduling. The details of this phase are

discussed in section 5.3.

The following code fragment illustrates a recurrence that involves an array.

```
for (let i = 0; i < 5; i++) {
    dst[i] = src[i];                                    // (a)
    res[i] = res[i - 1] + 4;                            // (b)
}
```

Listing 5.17: Arrays potential recurrences

The first aspect to observe is that, for an operation to be considered a recurrence, it must load elements from an array whose base pointer is the same as the one to which the result of the iterated computation is subsequently stored.

In the considered example, it is therefore possible to distinguish that only operation *(b)* constitutes a recurrence, since it loads a value from the `res` array and stores it back into the same array. Operation *(a)*, on the other hand, simply represents an initialization of the `dst` vector, obtained by copying data from the `src` array.

The LLVM-IR representation of the code considered is reported below.

```
entry:
  %0 = call ptr @llvm.stacksave.p0()
  %vla = alloca float, i64 5, align 16
  %vla1 = alloca float, i64 5, align 16
  ...
  ...
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i32 %i.0, 5
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %idxprom = sext i32 %i.0 to i64
  %arrayidx = getelementptr inbounds [15 x float], ptr @src, i64 0, i64
    %idxprom
  %1 = load float, ptr %arrayidx, align 4
  %idxprom2 = sext i32 %i.0 to i64
  %arrayidx3 = getelementptr inbounds float, ptr %vla, i64 %idxprom2
```

```
20    store float %1, ptr %arrayidx3, align 4
21    %sub = sub nsw i32 %i.0, 1
22    %idxprom4 = sext i32 %sub to i64
23    %arrayidx5 = getelementptr inbounds float, ptr %vla1, i64 \%idxprom4
24    %2 = load float, ptr %arrayidx5, align 4
25    %add = fadd float \%2, 4.000000e+00
26    %idxprom6 = sext i32 %i.0 to i64
27    %arrayidx7 = getelementptr inbounds float, ptr %vla1, i64 \%idxprom6
28    store float %add, ptr %arrayidx7, align 4
29    br label %for.inc
30
31  for.inc:                                          ; preds = %for.body
32    %inc = add nsw i32 %i.0, 1
33    br label %for.cond
```

In this case, the starting point, and thus the root of the recurrence, is represented by the `store` instruction. From this instruction, it is necessary to trace back, through the corresponding `getelementptr` instruction, to the base of the array on which the value is stored. In the example, the final `store` (in blue) writes to the address `%arrayidx7`, computed by the GEP `%arrayidx7 = getelementptr . . . ptr %vla1, . . .` ; the base of the array involved is therefore `%vla1` (allocated in memory on the first instruction in blue into entry block. Instructions in green, help focusing the chain to reach the base pointer of the array used as its main identificator.

From the same `store` instruction, it is also possible to identify the value that will be written into the array, namely the "value" operand of the `store`. In the shown fragment, this value is `%add`. Starting from `%add`, it is therefore possible to trace back the entire *def-use* chain of instructions that contribute to the computation of the value to be stored. This chain is indicated in red.

The key element that suggests the presence of a recurrence is the occurrence, along this chain, of a `load` instruction. In the considered case, the accumulated value depends on `%2 = load float, ptr %arrayidx5`, where `%arrayidx5` is in turn computed through the GEP `%arrayidx5 = getelementptr . . . ptr %vla1, . . .` . As for the `store`, it is therefore necessary to trace back from the `load` to the base of the array.

If the base identified from the `load` coincides with the one identified from the `store` (in this example both equal to `%vla1`), then the update can be marked as an *analyzable* recurrence, since the value being written depends on a value read from the same data structure.

The recurrence chain can therefore be summarized as:

$$\texttt{store}_{\text{vla1}} \rightarrow \texttt{add} \rightarrow \texttt{load}_{\text{vla1}}$$

The chains, both those with a PHI-based root and those with a store-based root, may have variable length depending on the number of instructions involved before returning to the starting point (in the case of PHI nodes) or before identifying a `load` instruction accessing the same memory base as the `store` (in the case of memory-based variables).

Tracking memory initializations To improve the overall efficiency of the proposed method, it is also useful to keep track of certain initialization patterns, in particular those involving memory objects. The previous example illustrates one such case: instruction (a) stores into the array `dst` a copy of the values loaded from the array `src`.

Such instructions may represent a recurrence if considered in an aggregated form, jointly forming the two *crossing* recurrences illustrated in Sections 5.1.3 and 5.1.4. However, when taken individually, they do not constitute a recurrence since the value written to memory does not depend on the previous state of the same memory location. Consequently, estimating the range of `dst` in the first or in the last loop iteration yields the same result. In this context, the analysis therefore considers an aggregated range for the array, independent of the specific iteration.

In principle, if the ranges of all index expressions were available, it would be possible to perform an element-wise analysis of the accessed values in order to preserve a higher level of precision. However, such an approach would entail a significantly higher computational cost and is not required in the scope of this work.

Since dependencies may exist among different recurrences, and these dependencies must be resolved to ensure a correct interpretation, keeping track of initialization instructions allows such dependencies to be handled in an orderly manner. The details of this process will be discussed in the assembly phase 5.3. At this stage, it is therefore sufficient to annotate these types of instructions, leveraging the information already collected during the inspection analysis.

Role of the chain's instruction typology At this stage, it is important to recognize and annotate the roots of potential recurrences, as well as the instructions that are effectively involved in their evolution. However, it is not necessary to store all the instructions traversed by the chain, but only those that have a direct impact on the recurrence. Some operations, such as casts, do not in fact need to be applied at every loop iteration: if

handled correctly, their effect can be absorbed into the definition of the recurrence itself, while still preserving the soundness of the analysis.

To this end, consider the following simple example in which a recurrence on an integer variable depends on a decimal value that is converted to an integer before the accumulation operation:

```
int val = 0;
for (int i = 0; i < 5; i++) {
val += (int)2.3f;
}
```

Listing 5.18: Recurrence with casting example

In the C/C++ language, the cast from `float` to `int` truncates the value toward zero. As a consequence, the expression `(int)2.3f` always evaluates to `2`. The actual final value of the variable `val` at the end of the loop is therefore:

$$val = 0 + 5 \cdot 2 = 10.$$

From a soundness perspective, it is essential to respect the semantics of the cast: for positive values, truncation corresponds to rounding toward negative infinity, whereas for negative values, it corresponds to rounding toward positive infinity. In this case, since the value is constant and positive, the increment applied at each iteration is deterministic and equal to 2.

The recurrence can therefore be modeled as an affine recurrence with

$$\text{start} = 0, \qquad \text{step} = 2,$$

which produces the interval ([0,10]) when considered throughout the entire execution of the loop, and the exact value (10) at the end of the last iteration.

This example shows how cast operations can be made implicit in the dependency chain and therefore do not need to be explicitly annotated or iterated. Absorbing such operations into the recurrence modeling makes it possible to simplify the analysis and reduce the number of instructions to be considered, without losing precision or correctness.

Below, a non-exhaustive list of instructions that must be included in the chain of a potential recurrence is reported, limited to those encountered in this work:

- **Binary operations**, such as `add`, `sub`, `mul`, `div`, and their floating-point or integer

variants;

- **Unary operations**, such as `neg` or equivalent instructions;
- **Function calls** (`call`) that produce a return value or update arguments passed by reference;
- **PHI nodes belonging to other recurrences**, in the case of composed or nested recurrences (recurrence of a recurrence);
- **`load` and `store` instructions**, when they involve the same memory base, as discussed in the previous paragraphs.

It is instead possible to exclude from the recurrence chain some instructions that do not directly influence the iterative evolution of the value, including:

- **Casts**, both explicit and implicit, provided that they are incorporated in a sound manner into the recurrence definition;
- **Allocation instructions**, such as `alloca`;
- **Purely representational extension or truncation instructions**, such as `zext`, `sext`, and `trunc`, when they do not alter the semantic meaning of the value with respect to the recurrence;
- **Debug instructions**, such as `dbg.value` and `dbg.declare`.

The implementation of the inspection phase will be described in the algorithmic section, under the subsection *Inspecting phase* (see Section 5.4.3). The focus now shifts to the recognition of the identified potential recurrences and to the assembly of such recurrences.

5.3. Recurrence assembly

Given a collection of instructions identified as potential *recurrence roots*, it is possible to analyze the chain of operations involved and attempt to assemble the corresponding recurrence. Within the methodology presented in this work, this stage is referred to as the *detection and assembly phase*.

This section first introduces the criteria that must be evaluated in order to select or discard a specific recurrence class with respect to others, and then outlines how the construction of the recurrence is carried out once a suitable classification has been identified.

5.3.1. Tree-based representation of recurrences

A recurrence can be represented by means of a *tree*. Figure 5.1 shows two illustrative examples. Chain A, depicted on the left, represents a recurrence whose root is a *PHI node*. In this case, the PHI node appearing at one of the leaves coincides with the root itself, highlighting the cyclic nature of the recurrence. Chain B, on the right, has the same features, but the root is the Store Instruction and the rightmost leaf is the Load which gathers from the memory of the same object of the store.

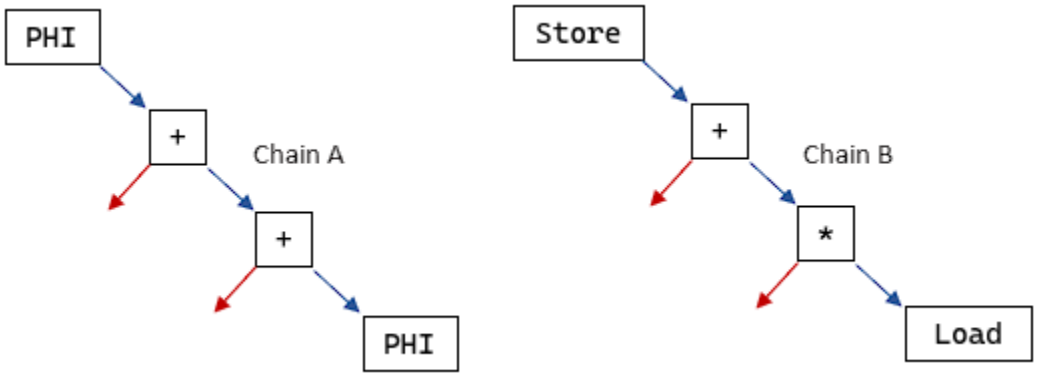


Figure 5.1: Two recurrence representation: PHI based on the left, Store-Load on the right

Two different types of *edges* can be identified. On the one hand, the edges belonging to the *recurrence chain* guide the traversal required to return to the root; these are highlighted in blue in the figure. On the other hand, the remaining edges are generated by *binary operations* that introduce dependencies outside the main recurrence chain; these are shown in red.

In the graphical representation, the portion of the tree connected to these latter edges is omitted. Although it contributes to the overall definition of the recurrence, it will be shown that such a contribution can be *computed in aggregate* and summarized by a single node representing its *final range*.

In addition to the binary operations along the main path, other kinds of operations may also appear and must be carefully evaluated. Figure 5.2 illustrates two additional recurrences. Chain C represents a recurrence that includes a *trigonometric function* along the main path (blue edges). This presence must be taken into account, as it may require a more in-depth analysis of the recurrence behavior.

Chain D, instead, shows how recurrences may include *nested recurrences*. When such nested recurrences appear along the main path, they require further investigation in order to properly classify and analyze the resulting recurrence type.

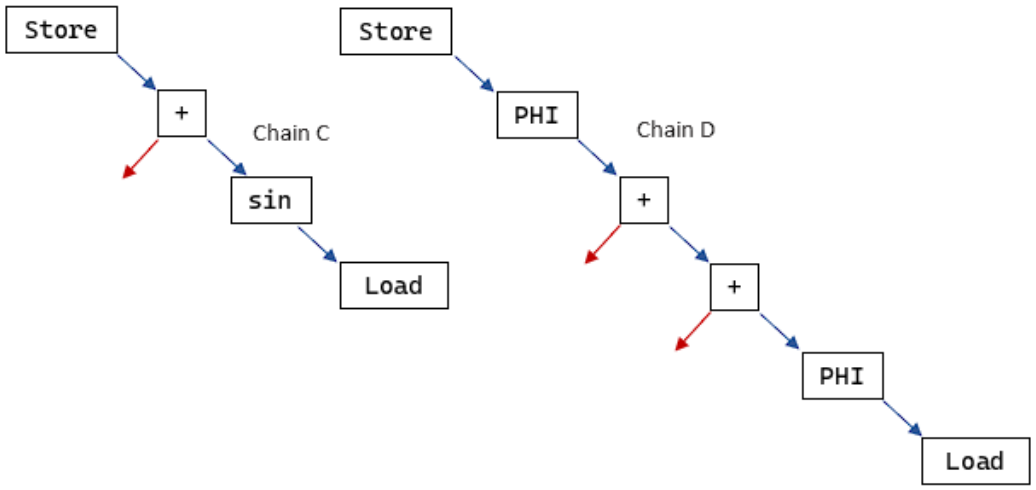


Figure 5.2: Other two recurrence representation: PHI based with non linear operation on the left, Store-Load which includes another recurrence on the right

For both cases shown in Figure 5.2, and more generally, the critical situation arises when these particular features occur along the *main recurrence chain*. The presence of nested recurrences or non-linear functions in secondary branches (red edges) may certainly require additional care, but is generally manageable in a *sound* manner.

For instance, if a function such as `sin` appears in the left branch of an addition, within an interval-based analysis it is sufficient to degrade its contribution to the constant interval $[-1, 1]$ in order to preserve soundness.

5.3.2. Typology classification

The first aspect to be evaluated is whether the *operations involved in the recurrence* are compatible with the *recurrence class* under consideration. If even a single operation is not valid, the analyzed class is discarded and the analysis proceeds by considering an alternative classification.

At this preliminary stage, it is sufficient to analyze only the operations belonging to the *main recurrence chain*, namely the binary operations that, in Figure 5.1, are represented along the path identified by the *blue edges*. By comparing these operations with the structural properties of the different recurrence classes, a coarse-grained classification can be performed in a straightforward manner.

- **Affine recurrences.** The main chain may include only addition or subtraction operations. The presence of any other operation, or of mathematical functions that

are intrinsically non-linear (e.g., trigonometric functions like figure 5.2, chain C), is sufficient to exclude this class.

- **Geometric recurrences.** The main chain may include exclusively multiplication and division operations. The presence of even a single addition or subtraction, or of mathematical functions that are intrinsically non-linear (e.g., trigonometric functions like figure 5.2, chain C), is enough to rule out this recurrence class.
- **Linear recurrences.** In the presence of a mixture of additions, subtractions, multiplications, and divisions, it is in some simple cases possible to attempt the recognition of a linear recurrence.

Referring again to Figure 5.1, chain A can therefore be classified as an affine recurrence, while chain B can be identified as a linear recurrence.

5.3.3. Dependency-awareness and control

It has been implicitly stated that the nodes belonging to the main chain of the recurrence tree influence the classification of the recurrence type. However, once a recurrence has been identified and classified, it is not guaranteed that it can be immediately assembled. In order to preserve the *soundness* of the analysis, a recurrence must be constructed only when all the operands involved along secondary paths are themselves sound. This condition is satisfied when such operands can be represented by an interval that correctly encloses all the minimum and maximum values that can be computed by the program.

It therefore becomes necessary to analyze the dependencies present in all the nodes of every secondary branch. In this context, the notion of *loop-invariant values* once again becomes essential. A value is said to be loop-invariant with respect to a specific loop if it is not involved in any recurrence relative to that loop, and thus remains constant throughout all its iterations.

The code fragment shown in the listing 5.19, better illustrates this concept. This code computes the sum of the areas of N circles given their diameters.

Observing the body of the loop, it is possible to distinguish two different types of function calls. The function `getRadius` requires, as input, the diameter associated with the current iteration and therefore produces a result that varies across iterations. Conversely, the function `getPI` always returns the same value, regardless of the current loop iteration, and can thus be classified as loop-invariant.

```
float getRadius(float d) { return d / 2; }
float getPI() { return 3.14f; }

const float diameters[N];
float radius[N];
float areaSum = 0;

for (int i = 0; i < N; i++) {
    radius[i] = getRadius(diameters[i]);
    float pi = getPI();
    areaSum = areaSum + pi * radius[i];
}
```

Listing 5.19: Dependency-awareness example

If it is now represented the recurrence over the iteration variable `i` and the recurrence over `areaSum` using a tree-based representation (figure 5.3), explicitly including the secondary branches, it can be observed that the recurrence of `areaSum` involves both a loop-invariant call to `getPI` and a `load` from the array `radius`. While the former does not introduce any blocking dependency, the latter requires that the previous `store` to `radius` has been correctly resolved, as shown in the portion of the tree labeled as *branch of load(radius)*.

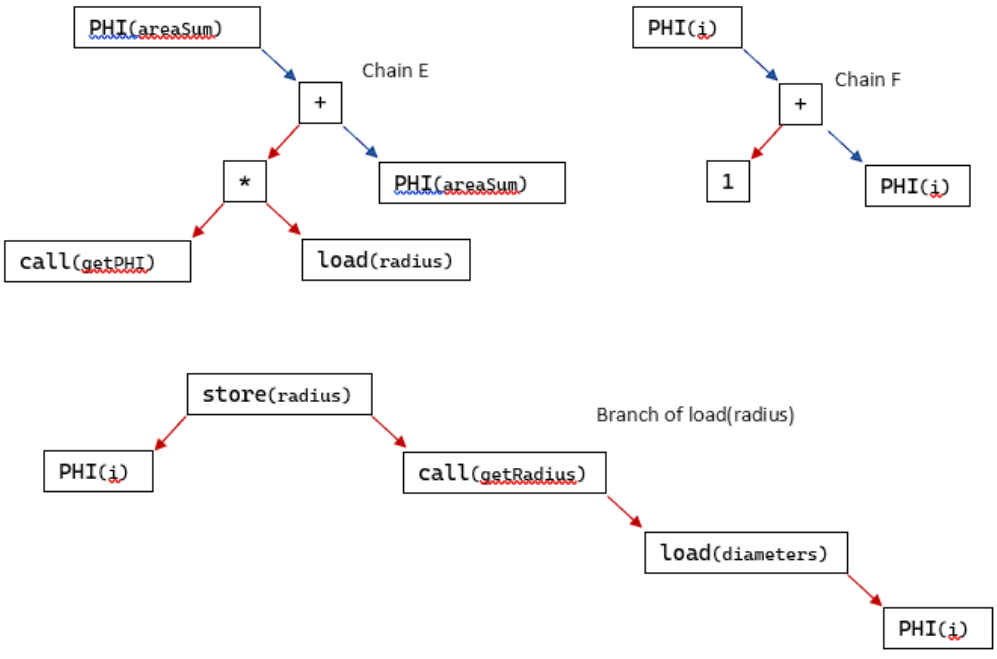
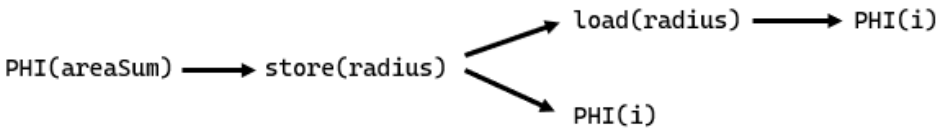


Figure 5.3: Tree-form representation of area sum computation example

The `store` to `radius` itself depends on the loop iterator `i`. Moreover, the call to `getRadius` involves, as its argument, a `load` operation that also depends on iterator `i`. Iterator `i` represents a trivial loop-variant value, modeled as a PHI recurrence with a unit increment at each iteration.

The resulting dependency chain can therefore be expressed in linear form as follows, where the direction of the arrows denotes a *depends-on* relationship:



This representation highlights how the assemblability of the `areaSum` recurrence depends not only on its main accumulation pattern, but also on the correct resolution of all secondary dependencies, including memory updates and loop-variant operands. Once a blocking dependency has been resolved, the corresponding node can be safely replaced by the resulting sound interval, thereby enabling the assembly of other recurrences that were previously blocked.

5.3.4. Final assembly

Once it has been verified that the analyzed recurrence is compatible with the class currently under consideration, and after all its dependencies have been successfully resolved, the recurrence can be assembled. Reference is again made to the previous example computing the sum of the areas of circles with varying diameters.

The recurrence on the iteration variable `i` involves a binary addition operator and, along its secondary branch, the constant value 1. From the source code, the initial value of `i` can be inferred to be zero. The constructed recurrence can therefore be expressed as:

```
Affine(start = 0, step = 1)
```

Resolving this recurrence unlocks both the `store(radius)` and the `load(diameters)` operations. As previously discussed, in this work arrays are conservatively modeled using their overall value ranges, so in this case it is not crucial known the value of N and compute the affine with its closed form. Let R_r denote the range of values assumed by the array `radius`, and let R_d denote the range of the array `diameters`. It is then possible to compute the final interval associated with the left operand of the addition operation in chain E as:

$$R_{left\_add}(k) = \texttt{getPI()} \times \texttt{getRadius}(R_d(k))$$

Finally, since the initial value of `areaSum` is zero, the resulting recurrence can be assembled as:

$$\texttt{Affine(start = 0, step = } R_{\texttt{left\_add}}(k)\texttt{)}$$

After constructing a recurrence, it is possible to estimate the interval assumed at iteration k by means of a closed-form (or equivalent) representation. This interval can then be used to resolve recurrences that depend on it. Consider the following example:

```
1 float x = 1;
2 int i = 0;
3 while (i < N) {
4     i++;
5     x *= i + 5;
6 }
```

Listing 5.20: Past solved recurrence dependency example

In this case, two recurrences are present. The first concerns the iteration variable i (the induction variable) and is an affine recurrence:

$$\texttt{affine(start = 0, step = 1)}$$

The recurrence on x depends on the recurrence on i, since the multiplicative factor $(i+5)$ varies at each iteration. By fixing $N = 5$, the index i assumes the values $1, 2, 3, 4, 5$ during the five loop iterations and then final range is [0,5].

A final consideration concerns the implications of this process in the context of *precision tuning*. In fixed-point optimization, it is necessary to ensure that all values assumed at run time by a variable are representable by the selected fixed-point format. This implies that considering only the final value of a recurrence is not sufficient; instead, the entire range of values covered by the variable from the first to the last iteration must be taken into account.

As an illustrative example, consider a variable initialized as `float z = 2.04` and subsequently multiplicatively updated for a fixed number of iterations. If only the final value were considered (e.g., $5 \times 2.04 = 10.2$), the analysis might suggest that a fixed-point format with a single decimal digit is sufficient. However, such a choice would fail to correctly represent the initial value 2.04, introducing an unacceptable loss of precision.

Conversely, by retaining the full interval $[2.04, 10.2]$ as the result of the analysis, the precision requirements across the entire execution can be accurately captured. This example highlights how interval-based reasoning, extended to the whole execution trajectory rather than limited to the final or asymptotic value, is essential to guarantee the *soundness* of precision tuning decisions.

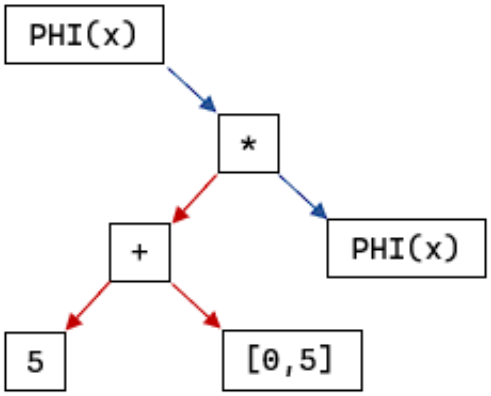


Figure 5.4: Tree-based representation of x geo example

Once this aspect has been clarified, the affine recurrence of i can be resolved, yielding the interval $[0, 5]$ as the post-loop value (figure 5.4). At this point, the secondary branch associated with $(i + 5)$ can be resolved as:

$$[0, 5] + 5 = [5, 10]$$

This results in a sound over-approximation of the growth of x, modeling the multiplicative factor through an interval:

$$\texttt{geometric(start = 1, ratio} \in [5, 10])$$

After 5 iterations, a sound bound is therefore obtained:

$$x \in [1, 10^5] = [1, 100000].$$

A higher degree of precision could be achieved by evaluating the product iteration by iteration, rather than summarizing the entire evolution through a single interval on the multiplier; however, this lies outside the scope of the present example.

5.4. Overall analysis algorithm

In this section, the *Recurrence-Aware VRA algorithm* (RA-VRA) implementing the proposed method is presented, providing a detailed analysis of all its phases and their execution order. Each step is illustrated by highlighting the involved procedures and methods through pseudocode, the output produced by each phase, and an estimate of the computational complexity. The underlying ideas of the *inspection* and *assembling* phases have been discussed in detail in Sections 5.2 and 5.3; here, their implementation is described.

5.4.1. Algorithm overview

In general, RA-VRA takes as input an intermediate representation (IR). In this work, as previously discussed, the representation is encoded using the LLVM Intermediate Representation (LLVM-IR). Before the analysis is executed, the IR is preprocessed by the `mem2reg` pass, whose purpose is to promote memory-allocated variables to registers in Static Single Assignment (SSA) form. This pass removes, whenever possible, redundant allocation, load, and store instructions, and introduces `PHI` nodes to explicitly represent value dependencies across control-flow joins. As a result, the data flow becomes explicit and easier to analyze, especially in the presence of loops and recurrence patterns.

A second requirement necessary for the correct execution of the proposed algorithm is loop canonicalization. Loops in canonical form (Section 2.3.2) simplify the analysis and ensure that the PHI nodes eligible to act as recurrence roots are located in the loop header. Moreover, in normalized loops, PHI nodes are structured to have exactly two incoming values: the first corresponds to the loop-invariant value coming from the preheader, while the second originates from the latch and is associated with the last instruction involved in the operation chain that produces the updated value at the end of each iteration.

In principle, it would be possible to handle recurrences also in non-normalized loops, where PHI nodes may have multiple incoming values, typically due to the presence of internal branches or multiple re-entry points into the header. In such cases, to preserve the soundness of interval analysis, the range associated with the PHI node should be computed as the convex hull of the ranges of all incoming instructions. However, this approach leads to increased imprecision—caused by the loss of correlation among different control-flow paths—and makes both the identification of the recurrence structure and the derivation of its closed form more computationally expensive. For these reasons, loop canonicalization has been imposed as a preliminary requirement, favoring a control-flow structure that enables a more direct and less ambiguous recognition of recurrences.

The Value Range Analysis (VRA) pass within the TAFFO framework is an analysis pass and therefore does not perform any modification to the intermediate representation. Instead, it collects and annotates the value ranges that can be statically inferred, that is, without executing the final program. These annotations are later exploited by the precision tuning passes, where the most suitable fixed-point representation is selected for each value and the corresponding transformation is applied.

RA-VRA is structured into five phases executed sequentially, complemented by an optional *fallback* mechanism. However, only the first two phases are executed once, while the remaining ones are iterated until range stability is reached, that is, until a fixpoint of the analysis is obtained.

The execution starts from one or more *entry-point* functions, typically `main`, and proceeds by following the function call graph. Upon entering a function, the current ranges of its arguments are recorded, while at the end of its execution the range of the return value is stored, if present. When a function is invoked again with argument ranges that have not undergone relevant changes capable of affecting the analysis outcome, the function evaluation can be skipped and the previously computed results can be safely reused.

Once convergence is achieved, the resulting ranges are stored and made available both to subsequent function calls (interprocedural analysis) and to the following stages of the precision tuning process.

Figure 5.5 illustrates all the phases of the algorithm and their execution order. Each phase is described in greater detail in the following sections.

Scope and store management Compared to the current version of TAFFO, the proposed method does not introduce any modifications to the mechanism used to store analyzed values. The notion of *scope* is implemented in TAFFO through concrete data structures called *Stores*, which are organized hierarchically according to three levels, from the most general to the most specific:

- **Global scope**, implemented by the `VRAGlobalStore` class, which is unique for the entire program;
- **Function scope**, represented by the `VRAFunctionStore` class, with one instance for each analyzed function;
- **Block-level scope**, implemented by the `VRAAnalyzer`, which encapsulates the store of a single basic block and is responsible for applying the algebraic rules required to compute the ranges of the instructions it contains.

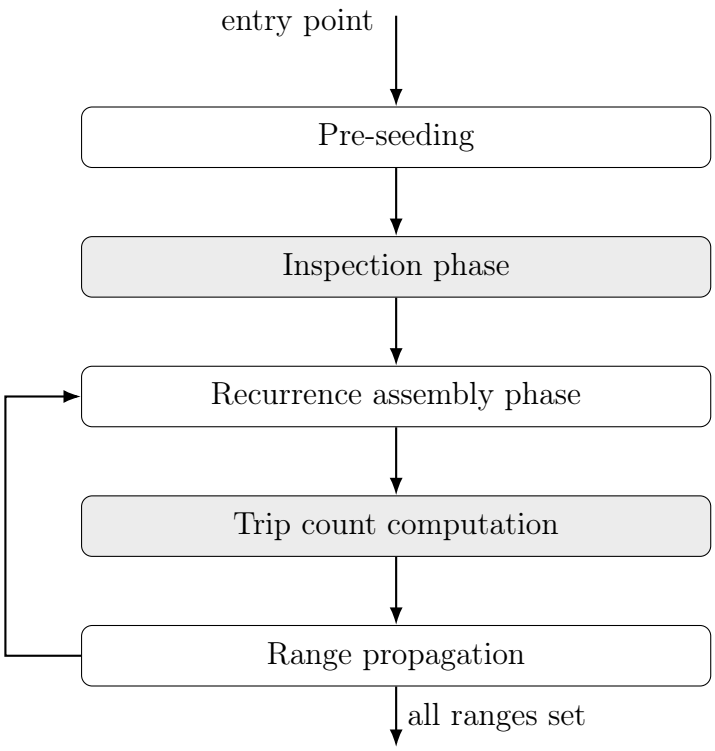


Figure 5.5: Block diagram of the RA-VRA phases executed for each interpreted function.

The global store is visible to all other scope levels and is initialized with constants and values defined at the `llvm::Module` level. It also preserves the ranges derived from annotations, both for function arguments and for annotated internal instructions, allowing initial values to be restricted or corrected when they cannot be precisely determined at compile time. At the end of the pass, all ranges computed for the analyzed instructions are consolidated into this store, which acts as a single collection point for the final result serialization.

The function-level store maintains the ranges of input arguments and return values, when present, and collects all the `VRAAnalyzer` instances associated with the basic blocks belonging to the same function. This level isolates the analysis of individual functions, enabling result reuse and supporting interprocedural caching mechanisms.

Block-level stores can access both the function-level and the global store, as well as the ranges produced by predecessor blocks within the same function. When control flows from one block to the next, the updates resulting from the analysis of predecessor blocks are propagated to the scope of the current block; in the presence of multiple predecessors, the corresponding values are combined through join operations, ensuring the soundness of the analysis. If the range of a value is requested and is not found in the local scope, the visibility hierarchy allows the analysis to progressively fall back to higher-level stores,

up to the global one, guaranteeing consistent information availability at every point.

Monotonicity and lattice-based convergence The `VRAFunctionStore` and `VRAAnalyzer` instances are never destroyed during the execution of the analysis, but persist until the final result-saving phase. Similarly, the ranges associated with instructions are never removed, but can only be updated across successive iterations. These updates can occur in two distinct forms:

- **temporary and incremental widenings** of one or more intervals, caused by local propagation of incomplete information;
- **definitive updates** resulting from the resolution of one or more recurrences, which allow a valid interval over the entire iteration domain to be computed directly, effectively performing an accelerated yet stable form of widening.

Once a range has received a definitive widening, the method does not allow further narrowing. This design choice ensures that the evolution of values is monotonic with respect to the partial order defined over the range domain. As a consequence, at each iteration of the algorithm, scopes may remain unchanged or expand, progressively approaching a stable state of the analysis.

The structure described above is fully compatible with a *lattice-based* approach, as it satisfies the requirements of monotonicity and convergence toward a fixpoint. However, unlike traditional methods that rely exclusively on heuristic widening strategies, the proposed method exploits recurrence recognition and resolution to guide range expansion, achieving faster convergence while preserving higher precision in the final computed values.

5.4.2. Pre-seeding phase

At the beginning of the pass, once all constants and annotations have been collected into the `VRAGlobalStore`, the *preseeding* phase is started. It begins with the interpretation of one or more *entry-point* functions and visits, by following the call graph, all functions reachable in the intermediate representation.

The main tasks of this phase are:

- to analyze all basic blocks of the functions and the instructions they contain exactly once, following an order consistent with the *dominator tree preorder*, and to record this order in order to simplify and make deterministic the iterations of the subsequent phases;

- to initialize the ranges of all instructions through an initial interpretation of the code, which effectively corresponds to a single iteration of the analysis, in which iterative constructs are traversed only once without resolving recurrences;
- to collect information about all the functions and loops identified, which is required by the subsequent phases of the analysis.

At the end of this phase, an ordered flow of basic blocks is obtained, which remains valid for the entire duration of the pass without the need for further structural analyses, together with a consistent hierarchy of stores containing the initial ranges associated with each instruction.

Control-flow algorithm To ensure that every reachable basic block is analyzed and that such analysis takes into account all the information collected in the previous phases, the control-flow management algorithm plays a crucial role. In LLVM-IR, each basic block explicitly records all immediately preceding blocks (*predecessors*) and terminates with a *branch* instruction to one or more successor blocks. In the presence of multi-way branches, the selection of the successor block is governed by the outcome of a comparison instruction. Once the analysis of a basic block is completed, each successor is evaluated by a *following* algorithm, which verifies the validity of the control-flow path and determines whether the successor block can be safely analyzed.

The main elements required to understand the algorithm are listed below:

- **bbVisited**: an ordered vector of already visited basic blocks. It is used to define the visitation order and to mark blocks that have already been analyzed, thus avoiding redundant analyses. All blocks belonging to a loop are visited before the loop is considered completely analyzed, thereby enabling the analysis of the successor blocks of that loop.
- **workload**: a LIFO vector containing the basic blocks that are still to be visited and that have been approved by the *following* algorithm.
- **loopCollection**: a stack of loops currently being processed. The loop at the top of the stack indicates that the current context lies within that loop; multiple loops in the stack represent nested loops. An empty collection indicates that the current block is outside of any loop.
- **succFollowed**: an auxiliary vector used to avoid following the same successor of a block multiple times, in particular in the presence of self-loops.
- **loops**: a dictionary that maps each loop (identified by its LLVM reference) to a

structure containing relevant loop-related information, such as the *trip count* (when available) and a loop-specific `bbVisited` vector.

Algorithm 5.1 reports the control-flow traversal and management algorithm adopted in this phase.

The first phase of the procedure initializes all the data structures described above and initializes the *workload* with the entry-point basic block of the function. At this stage, all instructions belonging to the visited basic blocks are analyzed.

During the analysis phase, for each current instruction the ranges of the instructions it depends on (the so-called *instruction users*) are retrieved. Based on the instruction *opcode*, the resulting range is then computed and stored in the current scope.

Some characteristic instruction cases are listed below:

- **Loop header `PHI` nodes**: they are initialized with the range of the incoming value from the loop preheader block.
- **Binary and unary operations**: the resulting range obtained by applying interval arithmetic is computed and stored.
- **Cast instructions**: the type of conversion is inspected and, when required, the range is conservatively extended to the immediately preceding and following representable values with respect to the current minimum and maximum bounds.
- **Call instructions**: the interpretation of the called function is triggered and, when applicable, the corresponding return range is awaited and collected.
- **Load instructions**: the range is initialized using the specific vector element when available; otherwise, the corresponding summary range is used.
- **Store instructions**: the summary range is created or updated in the memory storage associated with the scope.
- **Alloca instructions**: the *home* store in which the memory-resident vector will live is defined, leaving the associated range initially empty.

Algorithm 5.1 Basic Block Traversal and Loop-Aware Interpretation

```
1: bbVisited ← ∅
2: workload ← {entry_block}
3: succFollowed ← ∅
4: loopCollection ← ∅
5: loops ← map⟨loop_ref, VRALoopInfo⟩
6: while workload ≠ ∅ do
7:    curBlock ← pop(workload)
8:    for all instruction I in curBlock do
9:       analyze I and update scope
10:   end for
11:   if loopCollection is empty then
12:      add curBlock to bbVisited
13:   else
14:      curLoop ← top(loopCollection)
15:      retrieve bbVisited from loops[curLoop] and mark block
16:   end if
17:   succFollowed ← ∅
18:   for all successor S of curBlock do
19:      if S ∈ succFollowed then
20:         continue
21:      end if
22:      add S to succFollowed
23:      Result ← FOLLOWING(curBlock, S)
24:      if Result = ENQUE then
25:         push S into workload
26:      else if Result = LOOP_FORK then
27:         create new VRALoopInfo L
28:         push L into loopCollection
29:         push S into workload
30:      else if Result = LOOP_JOIN then
31:         for all loop exit block B in top(loopCollection) do
32:            push B into workload
33:         end for
34:         pop loopCollection
35:      end if
36:   end for
37: end while
```

Once the analysis of the current basic block is completed, the block is marked as visited and all its successor blocks are retrieved. The control-flow path between the current block and each successor is then evaluated by a dedicated *following* algorithm, which defines the enqueueing policy (Algorithms 5.2 and 5.3). The possible outcomes are the following:

- **ENQUE**: enqueueing is allowed and the successor block is added to the workload;
- **LOOP_FORK**: enqueueing is allowed and a new loop is entered. Before proceeding, the corresponding `VRALoopInfo` structure is initialized and the loop context is pushed onto the `loopCollection` stack;
- **LOOP_JOIN**: enqueueing is not allowed, but the current block represents the last loop latch to be visited. The loop context is therefore closed, and control returns to the parent loop context, if any, or directly to the function-level context. All exit blocks of the completed loop are then enqueued, starting from the most dominant;
- **NO_ENQUE**: enqueueing is not allowed and no action is performed.

The enqueueing policy performs two main checks. First, it verifies that all predecessors of the destination block have been marked as visited, in order to ensure that such blocks have already been analyzed and that the computed ranges are not based on stale operands. However, the predecessor check is suspended in two special cases.

The first case concerns predecessors that belong to an outer loop: during the analysis, such blocks may not have been visited yet. This behavior is consistent with the adopted policy, according to which an outer loop is never considered fully visited until all its inner loops have been completely analyzed.

The second case concerns the loop header: one of its predecessors is the corresponding loop latch, which, being reached through a back-edge, has not yet been visited at the time the loop is entered. Excluding these two special cases, the absence of even a single visited predecessor causes the check to terminate immediately with the result `NO_ENQUE`.

Once it has been verified that all relevant predecessors have been analyzed, the policy checks for the entry into a nested loop, namely the case in which the destination block is the loop header of a loop different from that of the source block. In this situation, enqueueing is allowed, but the entry into a new iterative context is signaled by returning the outcome `LOOP_FORK`.

The case of the back-edge between the loop latch and the loop header, previously excluded, is then handled. Only when all loop latches have been visited does the check return the outcome `LOOP_JOIN`, allowing the closure of the loop context.

If none of the previous conditions is met, the policy falls back to the standard case: if the successor block has never been visited, the outcome is `ENQUE`; otherwise, enqueueing is denied with `NO_ENQUE`.

Algorithm 5.2 FOLLOWING(src, dst): predecessor analysis

Input: source node src, dest node dst
Output: Following policy
$SrcLoop \leftarrow$ LOOPOF(src)
$DstLoop \leftarrow$ LOOPOF(dst)
if dst is an exit block of $DstLoop$ **then**
  return NO_ENQUE
end if
if $loopCollection$ is empty **then**
  $curFlow \leftarrow bbVisited$
else
  $LI \leftarrow$ top($loopCollection$)
  $curFlow \leftarrow LI.bbVisited$
end if
{Analyze all predecessors of dst}
for all predecessor blocks P of dst **do**
  $PredLoop \leftarrow$ LOOPOF(P)
  if P is the loop latch of $PredLoop$ **and** dst is the header block of $PredLoop$ **then**
    continue
  end if
  if $PredLoop = DstLoop.ParentLoop$ **and** dst is the header block of $DstLoop$ **then**
    continue
  end if
  if $PredLoop \in loopCollection$ **and** $DstLoop = PredLoop.ParentLoop$ **and** $PredLoop$ is not entirely visited **then**
    return NO_ENQUE
  end if
  if $P \notin curFlow$ **then**
    return NO_ENQUE
  end if
end for

Algorithm 5.3 FOLLOWING(src, dst) (continued): loop events and enqueue decision

```
{Continues from Algorithm 5.2; SrcLoop, DstLoop, and curFlow are already defined.}
{New loop found: entering a header from outside}
if dst is the header block of DstLoop and SrcLoop ≠ DstLoop and DstLoop ∉ loopCollection then
    return LOOP_FORK
end if
{Back-edge to the header: decide whether the loop can be closed}
if dst is the header block of DstLoop and SrcLoop = DstLoop and src is the loop latch of SrcLoop then
    if SrcLoop is entirely visited then
        return LOOP_JOIN
    else
        return NO_ENQUE
    end if
end if
if dst ∉ curFlow then
    return ENQUE
end if
return NO_ENQUE
end function
```

Termination and complexity considerations The tracking of already visited basic blocks and the protection against self-loops ensure that each basic block is analyzed exactly once, including those belonging to loops. This property guarantees the termination of the algorithm and results in a time complexity that is linear with respect to the number of basic blocks B in the LLVM-IR representation. Therefore, this phase of the analysis runs in $O(B)$ time.

From a space complexity perspective, each basic block contains one or more instructions, and each instruction is analyzed only once. At most, an instruction allocates a representation in the scope in the form of an SSA value or a memory summary. Denoting by N the total number of instructions in the intermediate representation, the space complexity is also linear, namely $O(N)$.

5.4.3. Inspecting phase

The inspection phase provides an implementation-oriented counterpart to the theoretical concepts introduced in the previous section 5.2. The goal of this phase of the Value Range Analysis (VRA) algorithm is to identify all the possible recurrences present in the program's intermediate representation.

Recurrences are recognized exclusively at the level of the involved instructions, without performing any numerical evaluation or range propagation at this stage. For this reason, the inspection phase can be executed only once for each analyzed function.

A recurrence may arise either within a loop or through recursive functions. In this work, the analysis is restricted to recurrences generated by loops, observing that many forms of recursion can be transformed into equivalent iterative constructs [2], thus allowing the same analysis principles to be applied.

The core idea consists in iterating over all the loops present in a function and, for each of them, analyzing all the basic blocks they comprise. When a `PHI` instruction is encountered in the loop header, it is analyzed by a dedicated handler. Similarly, `Store` instructions are delegated to a specific handler capable of reconstructing possible recurring dependencies through memory. For optimization purposes, it is preferable to traverse loops starting from the most deeply nested ones. In this way, inner recurrences are identified first, and if outer recurrences have operands that are roots of nested ones, they can simply reference those root operands without re-analyzing the internal chain.

Whenever one of the handlers identifies a potential recurrence, the analyzed instruction is marked as the *root* of the recurrence, and all the instructions participating in the recurring chain are recorded. The information collected during this phase constitutes the foundation for the subsequent stages of the algorithm, which are dedicated to recurrence classification and to the computation of the associated value ranges.

Before presenting further details on the handler algorithms, the structure of the procedure responsible for invoking them is first illustrated (Algorithm 5.4). In this algorithm, the `RecurrenceInfo` (RI) object appears for the first time. It is used throughout the entire pass to collect and maintain all the information associated with a single potential recurrence. In particular, it stores:

- the **root of the recurrence**, which can be either a *PHI node* or a `StoreInstr`;
- a **`chain` vector**, representing the direct instruction path that starts from the root and leads to the closure of the recurrence;

- the **type**, indicating whether the identified structure represents an actual recurrence or a simple initialization;
- a reference to the **recurrence to be constructed in the subsequent phase**, whose absence may indicate either that the recognition has not occurred or that it has occurred but construction is temporarily postponed due to unresolved dependencies;
- the **final range associated with the recurrence**, together with the iteration at which it has been computed, whose presence implies that the recurrence has been both recognized and resolved.

Algorithm 5.4 Recurrence Inspection Phase

```
1: RRs ← empty dictionary mapping Value to RecurrenceInfo
2: for all loops L in getLoopsInnermostFirst() do
3:   for all basic blocks BB in L do
4:     for all instructions I in BB do
5:       if I is neither a PHI instruction nor a Store instruction then
6:         continue
7:       end if
8:       IR ← new RecurrenceInfo
9:       if I is a PHI instruction and BB is the loop header of L then
10:        handlePHI(L, I, IR : inout)
11:      end if
12:      if I is a Store instruction then
13:        handleStore(L, I, IR : inout)
14:      end if
15:      if IR.isValid then
16:        RR ← new RecurrenceInfo(IR)
17:        insert RR into RRs
18:      end if
19:    end for
20:  end for
21: end for
```

The `handlePHI` and `handleStore` methods introduced in Algorithm 5.4 are solely responsible for initializing the root, the instruction chain, and the recurrence type. Once the inspection phase is completed, this information is considered immutable and serves as the

structural input for the subsequent assembling and resolution phases.

Finally, note that only `PHI` nodes located in the loop header should be considered. Other `PHI` nodes may appear in different blocks as the merge point of conditional branches; in such cases they do not represent loop-carried values and therefore cannot be treated as roots of any recurrence.

Inspection of PHI-node instruction chains The algorithm that inspects the instruction chain of PHI nodes follows the use–def chain in forward mode. In LLVM-IR, each instruction keeps track of the list of its users. The goal of this analysis is to traverse the users, and the users of users, in order to eventually return to the originating PHI node.

As specified in Section 5.4.1, loops must be normalized into canonical form before executing the proposed algorithm. In this context, the PHI nodes subject to analysis in LLVM-IR are located in the loop header and exhibit the following structure:

```
1 %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
```

Listing 5.21: PHI node LLVM-IR representation

Therefore, the PHI node is composed of a name (e.g., `%i.0`) and a set of incoming values, each enclosed in square brackets. Each incoming specifies both the value and the basic block from which the control flow originates. Typically, the first incoming value comes from outside the loop (i.e., it is loop-invariant) or, as in the example, is a constant (zero), and represents the initialization of the variable. The second incoming value is produced by an instruction that varies within the loop (e.g., `%inc`), and its corresponding basic block must belong to the loop itself (e.g., `%for.inc`).

Therefore, to ensure that a recurrence and its associated instruction chain have been correctly identified, it is sufficient to reach the instruction corresponding to the second incoming value. In loops written in canonical form, this instruction is typically located in the loop latch block.

Algorithm `5.5` illustrates how the analysis is performed in *forward mode*, by following the users of each instruction. The `isValidInstruction` function allows the analysis to advance only when the next instruction is meaningful from the perspective of recurrence detection, thereby filtering out constants and instructions that, by their nature, do not contribute to variations of the represented value.

A KeyValue data structure is employed to keep track of the predecessors visited during

the traversal phase; it is subsequently exploited to reconstruct the instruction chain in an ordered manner. Within this chain, only the instructions relevant to the identification of the recurrence are retained, namely those listed in 5.2. The remaining instructions can be discarded in order to reduce the number of elements to be processed and to accelerate the subsequent assembling phase and dependency resolution.

As a final remark, it is worth emphasizing that a forward-mode traversal of the use–def chain is preferable for PHI nodes, since these instructions define values that are consumed by subsequent operations within the loop body. Following the users naturally reflects the evolution of the variable and allows the analysis to directly identify the loop-carried dependency by reaching the instruction associated with the back-edge.

A backward analysis, instead, would require reasoning about multiple incoming definitions and control-flow join points, with the risk of including or having to discard a larger number of irrelevant values, such as definitions originating outside the loop.

Inspection of Store instruction chains In contrast to the algorithm used to recognize PHI-based recurrences, a different strategy is required to handle store instructions. Unlike PHI nodes, which receive their updates through the loop back-edge, memory stores in LLVM-IR are placed after all the instructions that compute the updated value. For this reason, the most suitable solution in this case is to perform a *backward* analysis, following the operands starting from the store instruction and moving upwards along the def–use chain.

The following example shows a store instruction operating on an array, together with its auxiliary instructions:

```
1 %idxprom  = sext i32 %i.0 to i64
2 %arrayidx = getelementptr inbounds [5 x float],
3                             ptr %res, i64 0, i64 %idxprom
4 store float %call, ptr %arrayidx, align 4
```

Listing 5.22: Store instruction LLVM-IR representation

Each store instruction targeting an array contains a reference to the instruction producing the value to be stored (in this example, `%call`) and a pointer to the memory location where the value will be written. The memory location is computed by the `getelementptr` (GEP) instruction (in this example, `%arrayidx`), which is responsible for determining the exact position within the array by computing the offset with respect to the base memory address of the object, using the current index (here `%idxprom`).

In the case of store-based recurrences, identifying a potential recurrence requires traversing the instruction chain backward, starting from the instruction that produces the value being stored, until a load instruction accessing the same memory location is encountered. A load instruction has a structure similar to that of a store, with the difference that the value is retrieved from the memory cell and made available to the instructions that consume it.

Algorithm `5.6` performs the analysis of store instructions. The procedure is similar to the one described for PHI-node handling, with the key difference that the termination condition is evaluated when a load instruction is encountered. When this happens, the analysis compares the underlying base memory objects and, if they match, the access can be classified as a recurrence candidate.

It is important to note that the recurrences identified in this phase as *store-based* are only *potential* recurrences, since equality of the base memory object does not always imply the presence of a recurrence. In particular, it is necessary to analyze the index *delta* between the store and load accesses in order to determine whether a loop-carried dependency actually exists. Further details, including the index analysis, are discussed in the assembling phase 5.4.4.

If the analysis terminates without recognizing any recurrence, the store can be marked as an initialization.

Complexity considerations From a time-complexity perspective, given I instructions to be inspected, if an instruction is recognized as part of a recurrence, the analysis may traverse at most the entire set of S instructions belonging to the loop. Both the forward and backward analyses avoid reprocessing instructions that have already been visited; therefore, in the worst case, the total number of iterations is bounded by $O(I \cdot S)$. For each execution of the forward or backward analysis, the auxiliary data structures (work-lists, visited set, predecessor map, and temporary chains) store at most the instructions belonging to the loop, resulting in a space complexity of $O(S)$ per run. If the recurrence information is retained for all inspected instructions, the overall memory usage may grow to $O(I \cdot S)$ in the worst case, where I is the number of analyzed roots; this corresponds to $O(S^2)$ when I is proportional to the number of loop instructions.

Algorithm 5.5 HandlePHI: forward use–def traversal for PHI recurrences

```
1: Input: Value Root
2: Input/Output: current RecurrenceInfo RI
3: incoming ← Root.getIncomingValue(1)
4: initialize stack worklist
5: push Root into worklist
6: initialize empty map pred
7: pred[Root] ← NULL
8: while worklist is not empty do
9:    cur ← pop from worklist
10:   if cur = incoming then
11:      initialize empty list reverseChain
12:      v ← cur
13:      while v ≠ NULL do
14:         append v to reverseChain
15:         if v ∉ pred then
16:            break
17:         end if
18:         v ← pred[v]
19:      end while
20:      initialize empty list RI.chain
21:      for all v ∈ reverseChain in reverse order do
22:         if isRelevantInstruction(v) then
23:            append v to RI.chain
24:         end if
25:      end for
26:      RI.kind ← RECURRENCE
27:      return
28:   end if
29:   for all u ∈ cur.users() do
30:      if isValidInstruction(u) and u ∉ pred then
31:         pred[u] ← cur
32:         push u into worklist
33:      end if
34:   end for
35: end while
```

Algorithm 5.6 HandleStore: backward traversal for store-based recurrences

Input: Value *Store*
Input/Output: current RecurrenceInfo *RI*
$StoreValue \leftarrow Store.getValueOperand()$, $StoreBase \leftarrow Store.getBaseMemory()$
init stack *worklist* $\leftarrow$ [$StoreValue$], map *pred* $\leftarrow$ {$StoreValue \mapsto$ NULL}, set *visited* $\leftarrow \emptyset$
while *worklist* not empty **do**
  cur $\leftarrow$ pop(*worklist*)
  if *cur* $\in$ *visited* **then**
    continue
  end if
  visited $\leftarrow$ *visited* $\cup$ {*cur*}
  if isLoadInst(*cur*) **then**
    if $StoreBase = getBaseMemoryObject(cur.getPointerOperand())$ **then**
      init empty list *RI.chain*, *v* $\leftarrow$ *cur*
      while *v* $\neq$ NULL **do**
        if isRelevantInstruction(*v*) **then**
          prepend *v* to *RI.chain*
        end if
        if *v* $\notin$ *pred* **then**
          break
        end if
        v $\leftarrow$ *pred*[*v*]
      end while
      RI.kind $\leftarrow$ RECURRENCE; **return**
    end if
  end if
  if isValidInstruction(*cur*) **then**
    for all *op* $\in$ *cur.operands*() **do**
      if *op* $\notin$ *pred* **and** *op* $\notin$ *visited* **then**
        pred[*op*] $\leftarrow$ *cur*; push(*worklist*, *op*)
      end if
    end for
  end if
end while
RI.kind $\leftarrow$ INIT

5.4.4. Assembly phase

In the previous phase, potential recurrences were identified and the initialization instructions that may depend directly or indirectly on other recurrences were collected. From this point onward, the analysis enters the *lattice solving* phase, which may be executed iteratively in order to reach a fixed point.

As discussed in Section 5.3.3, dependencies may exist among recurrences that prevent their immediate construction. For this reason, at each iteration of the algorithm it is verified whether the previous step has resolved at least one dependency; if so, the algorithm attempts to assemble new recurrences by exploiting the newly available information.

At the end of the analysis, the constructed recurrences are propagated to the subsequent phases. In the first phase, described in the following section, it is verified whether some loops can infer a previously unknown *trip count*. In the second phase, all scopes are updated by taking into account both the newly assembled recurrences and the computed trip counts.

This section focuses exclusively on the assembling algorithm, while the underlying conceptual approach has already been introduced in Section 5.3.

This phase consists of several steps, some of which are based on the same principles introduced during the inspection phase. For this reason, only the characteristic parts of the algorithm are described in detail, while the remaining aspects are discussed more concisely.

Algorithm 5.7 illustrates the parsing procedure applied to each identified recurrence. The example shown refers to the standard affine recurrence; however, the structure of the algorithm is general and can be applied to all the other recurrence types considered. For clarity, the overall workflow is described step by step below.

Algorithm 5.7 Affine Recurrence Assembling

Input/Output: RecurrenceInfo RI (passed by reference)
Output: **true** if the recurrence is recognized, **false** otherwise
if $RI.root$ is a `PHI` node **then**
  $isSolvable \leftarrow (|RI.chain| > 0)$
  for all $node \in RI.chain$ **do**
    if $node$ is not an affine instruction **then**
      return false
    end if
    $isSolvable \leftarrow isSolvable \wedge$
    $isSolvableDependenceTreeBackward(node.left)$
    $isSolvableDependenceTreeBackward(node.right)$
  end for
  if not $isSolvable$ **then**
    return true {Recognized but not solved}
  end if
  $Start \leftarrow fetchCurrentRange(RI.root)$
  $Step \leftarrow sub(fetchCurrentRange(RI.root_{latch}))$
  $RI.RR \leftarrow makeAffine(Start, Step)$
  return true
else if $RI.root$ is a `Store` instruction **then**
  $isSolvable \leftarrow (|RI.chain| > 0)$
  for all $node \in RI.chain$ **do**
    if $node$ is not an affine instruction **then**
      return false
    end if
    $isSolvable \leftarrow isSolvable \wedge$
    $isSolvableDependenceTreeBackward(node.left)$
    $isSolvableDependenceTreeBackward(node.right)$
  end for
  if not $isSolvable$ **then**
    return true {Recognized but not solved}
  end if
  $Start \leftarrow fetchCurrentRange(RI.loadJunction)$
  $Step \leftarrow sub(fetchCurrentRange(RI.root_{latch}))$
  $RI.RR \leftarrow makeAffine(Start, Step)$
  return true
end if
return false

- The algorithm iterates over all recurrences identified during the *inspection* phase.
- Recurrences that have already been recognized and successfully constructed are skipped.
- For each recurrence, the parsing of the different recurrence types is attempted sequentially using Algorithm 5.7. If the algorithm returns `false`, the next type is tested; if it returns `true`, the current type is considered correct and no further types need to be analyzed. More complex classes, such as *delta* and *crossing recurrences*, are analyzed before simpler ones in order to avoid approximate recognitions.
- Algorithm 5.7 iterates over the instructions belonging to the main chain and, for each of them:
  1. checks whether the instruction is compatible with the recurrence type currently under analysis; in case of incompatibility, the parsing is interrupted and the next recurrence type is considered;
  2. verifies that all instructions belonging to the secondary branch are loop-invariant or that the recurrence they depend on has already been solved (Algorithm 5.8); otherwise, the recurrence is considered recognized, but its construction is deferred to the next iteration.
- If all the conditions are satisfied, only the *start* and increment ranges are retrieved. The former is obtained from the current range of the root, while the latter is derived from the range of its *incoming* instruction, exploiting the most recent scope computation performed either by the *preseed* phase (during the first iteration) or by the *propagate* phase (during subsequent iterations). Since the incoming value already accounts for one iteration starting from the current start value, the start must be subtracted from it in order to obtain the correct increment.
- If no recurrence type is compatible, a *fallback* recurrence is assigned; in the absence of more precise information, it is handled as a *Top* range.

It is necessary to keep track of all recurrences that are recognized and assembled during each iteration. The algorithm schedules a new iteration at the end of the *propagate* phase only if at least one new piece of information has been introduced.

If no additional recurrences can be solved, two situations may arise: either all recurrences have been successfully resolved, or it is no longer possible to derive further useful information for their construction. In the former case, the analysis terminates after the last propagation step. In the latter case, a final iteration is scheduled, in which a *fallback*

mechanism is applied to all remaining recurrences and to the instructions for which no additional information can be inferred.

Algorithm 5.8 Solvability Check for a Value

```
1: Input: Value cur, Loop L, set of RecurrenceInfo RRs
2: Output: true if cur is solvable in the current iteration
3: if not isInvariant(cur, L) then
4:   if cur is a function call then
5:     // All non-invariant arguments must already be solved
6:     for arg ∈ cur.args do
7:       if not isInvariant(arg, L) and not RRs.isSolved(arg) then
8:         return false
9:       end if
10:    end for
11:  else
12:    if cur is a load then
13:      // Induction variable and dominating stores must be solved
14:      IV ← getInductionFromLoad(cur, L)
15:      if not RRs.isSolved(IV) then
16:        return false
17:      end if
18:      for storeRoot ∈ RRs do
19:        if storeRoot dominates cur and not RRs.isSolved(storeRoot) then
20:          return false
21:        end if
22:      end for
23:    else
24:      // Fallback: recurrence root must be solved
25:      root ← getRRRootByInstruction(cur)
26:      if not RRs.isSolved(root) then
27:        return false
28:      end if
29:    end if
30:  end if
31: end if
32: return true
```

Further details on dependency analysis Algorithm 5.8 represents the core of the dependency analysis and is applied to each instruction belonging to the secondary branches.

In contrast with the inspection phase, at this stage it is preferable to always traverse dependencies by following the operands of the instructions, even for recurrences whose root is a `PHI` node. During inspection, the objective is to recognize the recurrence pattern, and it may be convenient to follow the users of instructions in order to reconstruct the closure of the recurrence. In the solvability verification phase, however, the goal is no longer to identify the structure of the recurrence, but rather to ensure that all of its dependencies have already been resolved.

Following the users of a `PHI` node through subsequent binary operations may easily lead to cycles in the SSA graph, causing already analyzed instructions to be revisited and unnecessarily increasing the complexity of the analysis. For this reason, even in the case of `PHI`-based recurrences, it is sufficient to start from the incoming instruction representing the value coming from the loop latch. By construction, this instruction corresponds to the last operation of the potential recurrence. From this point, the analysis proceeds backward along the operands, allowing the dependency chain to be checked in a controlled and acyclic manner.

For `Store`-based recurrences, the behavior is analogous to the one adopted during the inspection phase in terms of traversal mechanism (i.e., following operands backward). However, while in the inspection phase such exploration is aimed at recognizing the recurrence pattern, in this stage it is exclusively devoted to verifying the solvability of dependencies.

The checks performed may vary depending on the type of instruction. In the case of a function call, it is not necessary to analyze the instructions inside the callee; it is sufficient to verify that all arguments, including parameters passed by reference, are either loop-invariant or already solved. Both contribute to modifying the return scope only if their values have changed.

For `Load` instructions, the base memory object must be inspected and the presence of `Store` instructions referring to the same base must be verified. If there exists a recurrence or an initialization that has not yet been resolved, the analysis can proceed only if such operations occur after the recurrence under consideration. This condition is checked by exploiting dominance relations.

LLVM provides this information through the Dominator Tree analysis. It is important to note that dominance is evaluated at the basic block level; when dominance between

instructions within the same basic block is required, it is determined according to their order of appearance, where an instruction dominates all subsequent ones. In the context of this pass, this approximation is sufficient, since instructions are processed following the same order as in the original code.

Complexity considerations From a temporal perspective, the algorithm is applied to the entire function over R candidate recurrences, which represent a subset of the N total instructions. For each recurrence, at most T parsing strategies are attempted, where T is the number of implemented recurrence classes. In the worst case, parsing a single strategy may require traversing a chain of length $O(N)$. As a result, the worst-case time complexity is $O(R \cdot T \cdot N)$ per iteration of the algorithm. The use of caching structures prevents redundant re-analyses, significantly improving the average-case behavior, while leaving the asymptotic worst-case bound unchanged.

From a spatial perspective, the analysis maintains an information structure for each candidate recurrence ($O(R)$), together with the associated instruction chains and dependency information. In the worst case, each chain may have length $O(N)$, leading to an overall memory consumption of $O(R \cdot N)$. Additional memory is required for caches used to memoize intermediate results, such as instruction compatibility and solvability checks; this overhead grows at most linearly with the number of instructions in the function ($O(N)$) and therefore does not affect the overall asymptotic bound.

5.4.5. Trip count computation

The *trip count* of a loop represents a key piece of information for determining the ranges associated with the identified recurrences. Since the growth of ranges is a direct function of the number of iterations, an accurate estimation of the trip count becomes crucial in order to obtain meaningful bounds. Before describing the techniques adopted to compute this value, it is therefore appropriate to clarify the reasons underlying its central role in the analysis.

Once recurrences inside loops have been identified, it becomes possible to compute the final range of each variable by applying the previously defined *at* operation. All recurrence classes implemented in this work produce sequences of ranges whose width grows monotonically with respect to the iteration index.

Formally, let $at(k)$ denote the value (or range) of a recurrence at iteration k, and let $W(\cdot)$ be the function that returns the width of a range. The following monotonicity property

holds:

$$W\big(\text{Range}\left(at(k-2),\, at(k-1)\right)\big) \;\leq\; W\big(\text{Range}\left(at(k-1),\, at(k)\right)\big)\,, \qquad \forall k \geq 2.$$

This property implies that, as the loop iterations progress, the set of possible values cannot shrink, but can only remain unchanged or expand.

Figure 5.6 shows that an affine recurrence with $start = 0$ and $step = 1$ produces a constant monotonic growth, with the range width always equal to the step. This horizontal growth is an intrinsic property of affine recurrences, since the contribution between successive iterations remains unchanged.

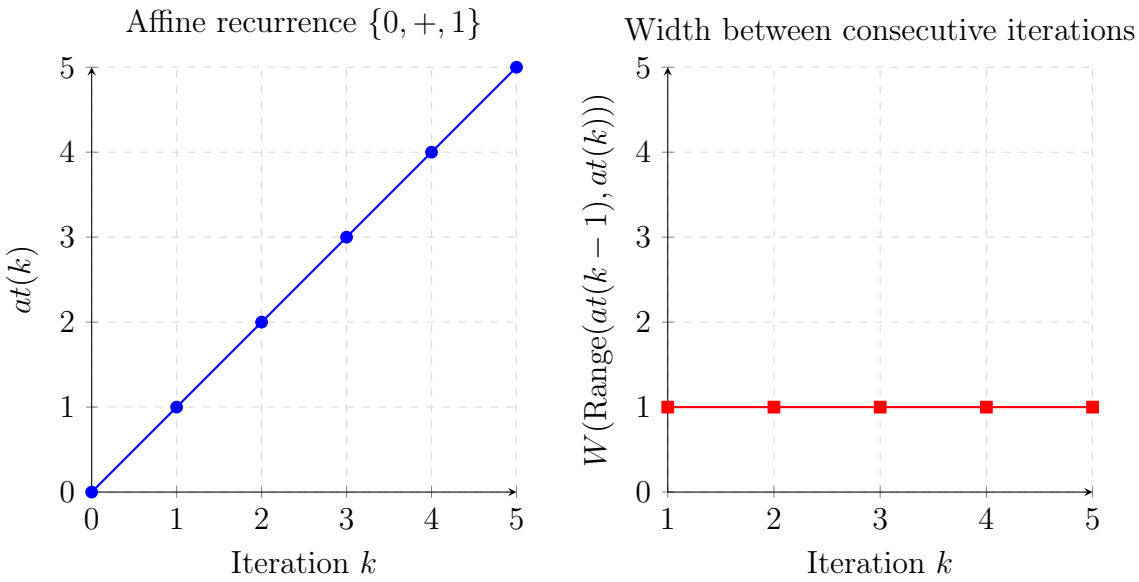


Figure 5.6: Affine recurrence $\{0, +, 1\}$ over six iterations (left) and constant width between consecutive iterations (right).

In the case of *geometric recurrences*, or more generally of all recurrences that introduce repeated multiplications, the observed behavior is analogous to the affine case, although the growth is exponential in nature. Figure 5.7 illustrates this scenario for a recurrence with $start = 1$ and $step = 2$, showing how both the recurrence value and the range width increase rapidly as the number of iterations grows.

When the *step* is negative, the recurrence exhibits an oscillatory behavior, alternating the sign of the values across successive iterations. However, despite the non-monotonicity of the values themselves, the range width remains monotonically increasing. This behavior is shown in Figure 5.8, where the growth of the range width continues to be governed by the absolute value of the multiplicative factor.

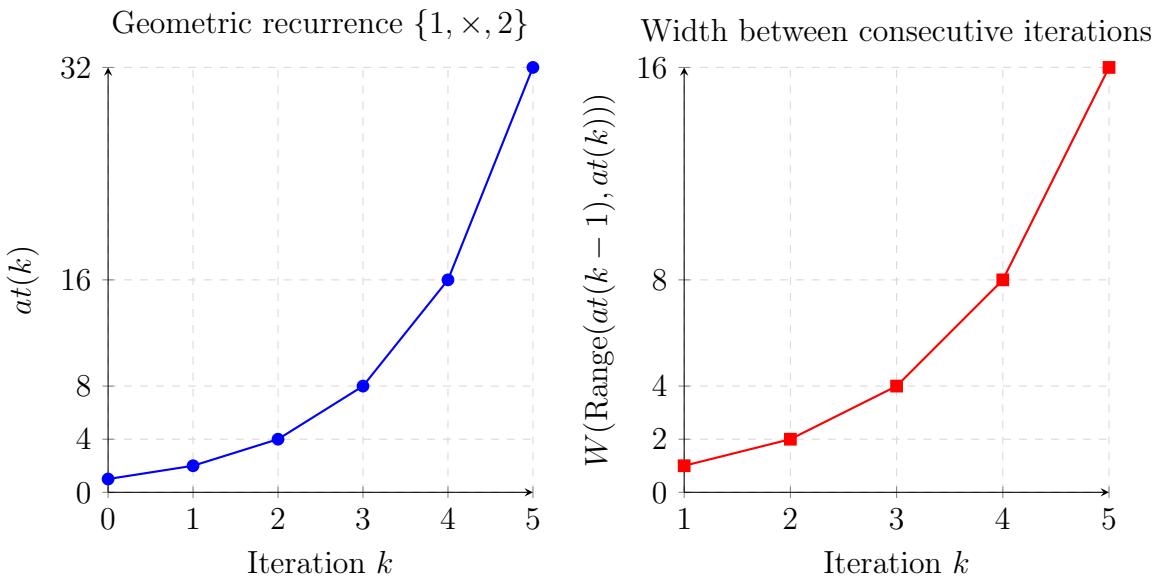


Figure 5.7: Geometric recurrence $\{1, \times, 2\}$ over six iterations (left) and increasing width between consecutive iterations (right).

At this point, it is possible to state that a recurrence, when evaluated at the initial iteration and at the final iteration of the loop, always produces a range that guarantees the *soundness* of the analysis. Such a range safely encloses all the values that the variable may assume during the execution of the loop.

A second relevant question concerns the *precision* of this range, namely whether it represents the tightest possible bound. When the recurrence inside the loop originates from a control-flow path that is executed at every iteration, each iteration contributes to the expansion of the variable range, and in this case the resulting range can be considered tight. Conversely, if the loop contains branches that may prevent the recurrence from being activated in some iterations, the computed range could be further tightened, since not all iterations contribute uniformly to its growth.

In this work, a conservative assumption is adopted, treating recurrences as potentially active at every iteration of the loop in order to preserve soundness.

It is also worth noting that the LLVM framework provides, through the *Scalar Evolution* (SCEV) module [1], several mechanisms to support the computation of the *trip count*. However, these mechanisms are effective only in a subset of cases, in particular when loop bounds are expressed using constants or simple symbolic forms.

For this reason, this work adopts an *ad hoc* method, specifically designed to increase the number of cases in which the trip count can be successfully extracted. In particular, the

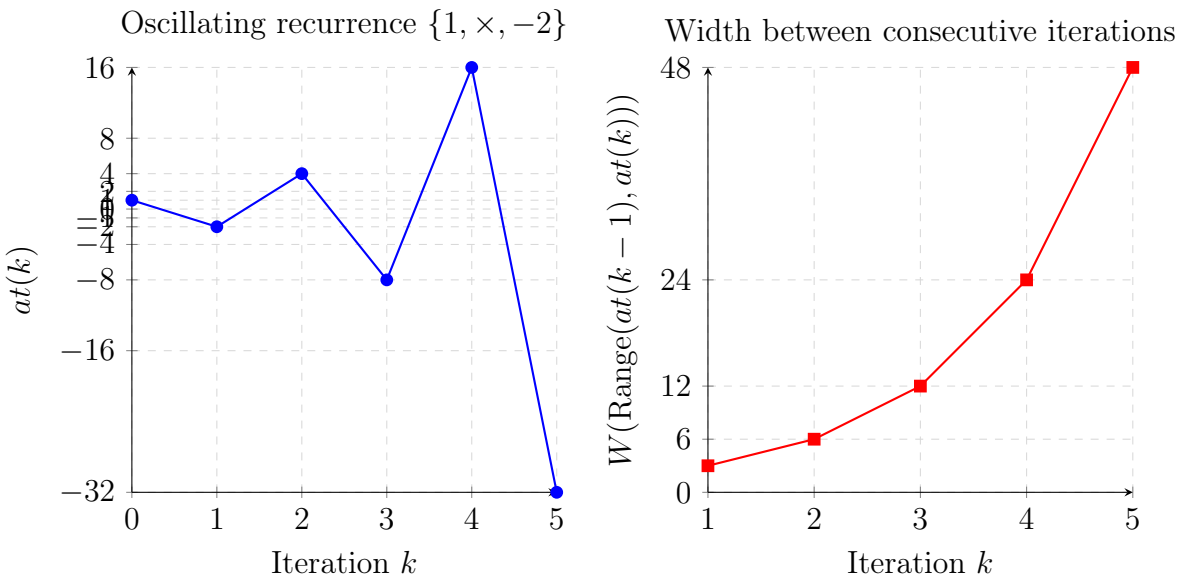


Figure 5.8: Oscillating geometric recurrence $\{1, \times, -2\}$ over six iterations (left). Although the sequence alternates in sign and is not monotone, the width between consecutive iterations is monotone increasing (right).

proposed approach extends the computation to loops characterized by *interval bounds* rather than constant ones and allows all the analyzed recurrences to be exploited for this computation, instead of being limited to the affine case alone. This enables a more informative and, in many cases, exact estimation, even in contexts where standard SCEV-based techniques prove insufficient.

For illustrative purposes, consider the following loop written in the C language:

```
1 int x = 5;
2 for (int i = 0; i < 10; i++) {
3     x = x + 2;
4 }
```

Listing 5.23: Loop C example

Its translation into LLVM-IR is reported below:

```
1  for.preheader:
2      br label %for.cond
3
4  for.cond:                         ; preds = %for.inc, %entry
5      %x.0 = phi i32 [ 5, %entry ], [ %add, %for.inc ]
6      %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
7      %cmp = icmp slt i32 %i.0, 10
8      br i1 %cmp, label %for.body, label %for.end
9
10 for.body:                         ; preds = %for.cond
11     %add = add nsw i32 %x.0, 2
12     br label %for.inc
13
14 for.inc:                          ; preds = %for.body
15     %inc = add nsw i32 %i.0, 1
16     br label %for.cond
17
18 for.end:                          ; preds = %for.cond
19     ret i32 0
```

Listing 5.24: Loop LLVM-IR representation of Loop C example

To compute the *trip count* using the method proposed in this work, it is necessary to identify the comparison instruction that may cause the loop to exit, namely the one that leads to a *loop exit block* (in the example, the `for.end` block), and to analyze it in order to understand the conditions under which such an exit is triggered. The required steps are summarized below:

1. Identify the *branch* instruction for which one of the destination blocks is a loop exit block (in the considered example, the branch at line 8).
2. From this branch instruction, trace back to the corresponding comparison instruction (`cmp`) on which it depends (line 7).
3. Verify the validity of the comparison instruction: **exactly one operand** must be *loop invariant*, while the other operand must correspond either to the root or to a node of a previously recognized recurrence.
4. Check the existence of a valid bound by analyzing the comparison operator and the monotonicity of the recurrence:

- for operators < or <=, the recurrence must be monotonically increasing;
- for operators > or >=, the recurrence must be monotonically decreasing.

5. If the recurrence admits a closed-form expression, the inverse formula can be used to directly derive the number of iterations k; otherwise, the recurrence is iteratively evaluated until the comparison condition is invalidated.
6. The resulting value k represents the *trip count* of the loop.

Algorithm 5.9 provides a formal description of the trip count computation steps discussed above.

Handling unknown trip counts When the conditions required for computing the trip count are not satisfied and its value is classified as `UNKNOWN`, a fundamental piece of information for applying the recurrence system is lost. In particular, it is no longer possible to determine the argument k of the characteristic function $at(k)$, thus preventing a direct evaluation of the closed-form expression.

A first conservative strategy consists in considering the asymptotic behavior of the recurrence for $k \to +\infty$, in order to guarantee the soundness of the analysis. In the absence of additional structural information, this leads to a maximal over-approximation, namely the `TOP` element of the interval domain, i.e., $(-\infty, +\infty)$. However, when additional semantic or structural properties can be inferred, more informative bounds may still be derived. If the recurrence is recognized as monotonically increasing with initial value r_0, the resulting range can be restricted to $[r_0, +\infty)$; analogously, in the case of a monotonically decreasing recurrence, it can be restricted to $(-\infty, r_0]$.

In other situations, the bound may be intrinsically determined by the semantics of the involved operation, independently of the number of iterations. A representative example is the absolute value operator (`abs`). Given an operand x with range $R_x = [x_{\min}, x_{\max}]$, the result necessarily satisfies

$$R_{\mathsf{abs}} = [0, \max(|x_{\min}|, |x_{\max}|)],$$

thus ensuring a finite and sound bound even when the trip count remains unknown.

At the current stage of the implementation, whenever these refinement strategies are not applicable, the most robust approach remains the use of the previous VRA version, possibly supported by explicit range annotations, in order to preserve soundness while avoiding excessive loss of precision.

Algorithm 5.9 Trip Count Computation (Proposed Method)

Require: Loop L with a list of recognized recurrences RRs
Ensure: Trip count T if computable, otherwise UNKNOWN

Identify a branch instruction br such that one successor is a loop exit block
if br not found **then**
  return UNKNOWN
end if
Retrieve the comparison instruction cmp associated with br
if cmp is invalid **then**
  return UNKNOWN
end if
Let (op_1, op_2) be the operands of cmp
{Exactly one operand must be loop invariant}
if isLoopInvariant(op_1) == isLoopInvariant(op_2) **then**
  return UNKNOWN
end if
Let b be the loop-invariant operand
Let r be the recurrence-driven operand get from RRs
Determine the comparison operator $\mathcal{O}$ used by cmp
if $\mathcal{O} \in \{<, \leq\}$ **then**
  if R is not monotonically increasing **then**
    return UNKNOWN
  end if
else if $\mathcal{O} \in \{>, \geq\}$ **then**
  if R is not monotonically decreasing **then**
    return UNKNOWN
  end if
else
  return UNKNOWN
end if
if R admits a closed-form expression **then**
  Compute k using the inverse recurrence formula
else
  $k \leftarrow 0$
  while cmp holds for $R(k)$ **do**
    $k \leftarrow k + 1$
  end while
end if
return k

Complexity considerations The proposed algorithm is applied to each loop contained in the function being interpreted. Let N denote the number of loops in the function and M the number of instructions belonging to a single loop. In the worst case, identifying the exit branch instruction (i.e., the branch leading to a loop exit block) requires a linear scan with cost $O(M)$. Once such a branch is found, the corresponding comparison instruction (`cmp`) can be retrieved immediately through def-use dependencies.

Similarly, identifying the iteration variable and the root of the recurrence requires at most an additional linear scan or traversal of the dependency chain, again with cost $O(M)$. Therefore, when the recurrence admits a closed-form expression and its inversion is available, the overall time complexity is $O(N \cdot M)$.

When a closed-form expression is not available and the algorithm resorts to iterative evaluation until the comparison condition is invalidated, the cost per loop becomes $O(M + T)$, where T denotes the trip count (or the conservative bound adopted). In this case, the overall time complexity is $O(N \cdot (M + T))$.

From a spatial complexity perspective, the impact is negligible: for each loop, a constant amount of information is stored (in particular, the trip count and associated metadata), resulting in an overall space complexity of $O(N)$.

5.4.6. Propagation phase

The final step of the RA-VRA algorithm is responsible for updating the abstract state of the ranges associated with the program instructions, moving the lattice from the current state S_i to the next state S_{i+1} by exploiting the information obtained during the recurrence analysis. The state S, represented by the set of ranges associated with the instructions across the different scopes, is an element of the product lattice induced by the individual interval domains.

Thanks to the availability of loop trip counts and the detected recurrences, it is possible to apply a form of *semantically guided widening acceleration*, which allows the analysis to reach an advanced lattice state without explicitly simulating each loop iteration. Instead of incrementally propagating the effects of every iteration, the analysis evaluates the recurrence at its final iteration, obtaining a sound over-approximation of the range assumed by the variable throughout the entire loop execution. This approach preserves soundness while significantly accelerating convergence to the fixpoint.

Instructions that do not directly depend on already resolved recurrences, or for which no new information is currently available, are not subject to widening in this phase and retain

their current range while awaiting further updates from propagation. When no additional useful constraints can be derived after applying all available information, the analysis resorts to a fallback mechanism that forces widening towards the top of the domain or towards a known bound, thus ensuring fixpoint reachability and algorithm termination.

The validity of the lattice-based approach is guaranteed by the monotonic evolution of the abstract states with respect to the partial order of the domain. When transitioning from S_i to S_{i+1}, ranges may only grow (in a controlled manner), progressively approaching the fixpoint S_f, at which further applications of the transfer functions no longer produce any variation.

The propagation phase is formalized in Algorithm 5.10. The algorithm exploits the block ordering computed during the preseed phase to avoid re-analyzing the main execution path (*best path*). For each instruction that is not a recurrence root, the corresponding range is updated by applying algebraic interval computations to its operands. Conversely, for recurrence root instructions, the range is computed directly by evaluating the closed-form representation of the recurrence at the loop trip count, through the operator *at*(*TripCount*). The resulting ranges are then propagated to subsequent instructions, contributing to the global update of the abstract state.

Algorithm 5.10 Propagation phase

```
1: Input: Entry function, preseedBlocksFlow, Recurrence set RRs
2: Output: Updated abstract state of instruction ranges
3:
4: for all basic blocks B in preseedBlocksFlow do
5:   if B belongs to a loop L then
6:     TripCount ← LoopInfo[L].TC
7:   end if
8:   for all instructions I in B do
9:     if I ∈ RRs and RRs[I].builtRR is defined then
10:      Range ← RRs[I].builtRR.at(TripCount)
11:      setScope(I, Range)
12:    else if I is a call instruction then
13:      Propagate ranges inside the called function
14:    else
15:      Retrieve the current ranges of all operands of I
16:      Range ← handleRangeOperation(I)
17:      setScope(I, Range)
18:    end if
19:  end for
20: end for
```

Complexity considerations The propagation phase visits each of the N LLVM-IR instructions exactly once, following the block order established during the preseed phase. Each instruction is processed in constant time, either by applying algebraic interval computations, by directly evaluating a recurrence, or by performing interprocedural propagation in the case of function calls.

Since the algorithm does not introduce any additional data structures and only updates the already existing global abstract state of instruction ranges, the overall time complexity is $O(N)$. The additional space complexity is $O(1)$, as no auxiliary memory proportional to the program size is required.

5.4.7. Termination phase

The final stage of the algorithm is responsible for stopping the iterative process, collecting the computed results, and storing them so that they can be made available to the subsequent passes of the pipeline.

The termination of the algorithm may occur in three distinct cases, which are listed below.

1. **Complete resolution of recurrences.** All the identified recurrences have been successfully resolved, and their corresponding ranges have been correctly propagated to all existing scopes. In this case, the only remaining operation consists in transferring the final ranges of each instruction into the global store, in order to proceed with the saving of the results.

2. **Exceeding the maximum iteration threshold.** A threshold can be defined to represent the maximum number of iterations allowed. Although in most cases recurrences are either resolved within a few iterations or remain unsolved from the early stages, this preventive measure avoids pathological situations such as unexpected stalls or deadlocks. When the threshold is exceeded, the algorithm activates the range fallback mechanism and then terminates following the same saving procedure adopted in the first case.

3. **Lack of progress in recurrence or trip count resolution.** The algorithm also terminates when no further recurrences are resolved and no new trip counts are computed, even though at least one recurrence has not yet been assembled or recognized. Indeed, trip count evaluation may itself depend on previously resolved recurrences; conversely, newly computed trip counts may unlock the assembly and resolution of additional recurrences. Termination therefore occurs when a fixpoint is reached, that is, when neither recurrence resolution nor trip count computation

produces new information. This situation may arise in the presence of recurrence patterns not supported by the recognition algorithm or due to unhandled cross-iteration dependencies. In this scenario as well, the fallback process is activated before saving the results of the analysis.

In the last two situations listed before, it becomes necessary to adopt a *fallback mechanism*. The choice of the fallback strategy strongly depends on the application context and on the goals of the analysis. In general, a fallback must guarantee *termination* while preserving *soundness*, even at the cost of reduced precision.

In the context of TAFFO, where range annotations can be exploited to correctly initialize input values and to constrain output intervals, a conservative yet effective approach has been adopted. All recurrences that remain unresolved at the end of the assembling phase are explicitly removed, and the analysis proceeds by performing one final complete execution of the propagation phase. In this way, the ranges associated with the involved values are recomputed using only standard range operations, without relying on recurrence models.

This strategy presents several advantages:

- it guarantees termination without introducing uncontrolled iterations;
- it delegates to the programmer the possibility of supplying explicit range annotations, thus allowing an explicit trade-off between soundness and precision;
- it naturally integrates with the TAFFO workflow, leveraging annotations to prevent excessive precision loss on final outputs.

A possible alternative would be the application of an aggressive *widening* on unresolved recurrences, forcing rapid convergence toward very large intervals. However, such a strategy is not particularly meaningful in the considered context: although it ensures termination, it produces overly pessimistic ranges that significantly degrade the effectiveness of precision tuning and drastically reduce optimization benefits.

For these reasons, the removal of unresolved recurrences followed by a final propagation phase represents a suitable compromise between *analysis robustness*, *integration within TAFFO*, and *quality of the obtained results*.

5.5. Final considerations about methodology

All the steps of the algorithm have been designed with the goal of improving precision while maintaining *soundness* as a fundamental principle of the analysis. A more accurate

recognition of recurrences within loops makes it possible to avoid both the explicit execution of all loop iterations and techniques such as loop unrolling, thus drastically reducing analysis time while simultaneously improving the precision of the computed ranges.

A key role is played by the determination of the loop *trip count*, obtained either directly or indirectly, since it is essential for computing the final range associated with each recurrence. In cases where, due to code complexity, this information cannot be derived at compile time, it is advisable to rely on annotations, exploiting knowledge of the application domain and usage context.

The proposed approach inevitably introduces computational overheads related to recurrence analysis and the resolution of dependencies among them. These costs become negligible in the presence of loops with large trip counts, but they may negatively impact performance when the number of iterations is small.

It is important to emphasize that this approach maintains a high degree of scalability. Indeed, new types of recurrences can be studied when needed and added in a straightforward manner by acting solely on the assembling phase. Similarly, new strategies for computing the loop trip count can be implemented by extending or modifying the dedicated analysis step. Both types of extensions, provided that they preserve soundness, may lead to a further improvement in the overall precision of the analysis.

Finally, the ranges computed during the analysis always undergo a monotonic, yet controlled, expansion, avoiding in most cases the derivation of excessively wide intervals with respect to the original application use cases.

In the next chapter, the proposed method and the conclusions drawn in this chapter will be evaluated on several use cases.

6 | Validation

The validation process of the method proposed in the previous chapter has been organized into three main phases. The first phase, which is fundamental to guarantee the soundness of the approach, concerns the verification of the expressions introduced for the different types of recurrences analyzed. In this phase, the theoretical correctness and consistency of the adopted mathematical models have been carefully examined, ensuring that the proposed representations accurately reflect the actual behavior of the recurrences.

The second phase focuses on the evaluation of the RA-VRA algorithm within the input LLVM-IR, implemented in the new Value Range Analysis (VRA) pass of TAFFO. This section analyzes the effectiveness of the algorithm in identifying and classifying recurrences, assessing its ability to operate correctly on real-world code and representative test cases.

Finally, the third phase concerns the experimental evaluation of the method on a set of benchmark algorithms, with the aim of measuring its practical impact in terms of range precision, performance, and scalability. For each of the three sections, the adopted testing methodology will be described in detail, together with the implementation choices and a critical discussion of the results obtained.

6.1. Validation of ranges computed by recurrences

6.1.1. Objective

This first test aims at verifying that all the `at(k)` methods proposed for the different recurrence classes always return sound ranges, namely ranges that never underestimate the actual bounds that each variable can assume at run-time. Ensuring this soundness property is essential to guarantee that the range analysis does not produce unsafe results. An underestimated bound could lead to the selection of an insufficient numerical representation during the precision tuning phase, ultimately causing arithmetic overflow or undefined behavior during program execution.

Example of unsound computed range. Consider the following affine recurrence:

$$x_{k+1} = x_k + 10, \quad x_0 = 0$$

After 100 iterations, the exact value is:

$$x_{100} = 1000.$$

If the corresponding `at(k)` method incorrectly returns the range

$$R_x(k) = [0, 200],$$

the precision tuning phase might select an unsigned 8-bit representation for the integer part (range $[0, 255]$), assuming that it is sufficient. However, the real value exceeds this bound, leading to arithmetic overflow at run-time. In the context of fixed-point conversion, this may result in wrap-around effects or undefined behavior, thus compromising the correctness of the transformed program.

6.1.2. Setup and testing methodology

In order to reliably validate the closed-form formulas associated with the recurrences, it is necessary to test each implemented recurrence type. Every recurrence is validated according to the following procedure:

- input arrays are generated and initialized with random values within predefined intervals;
- the evolution of the recurrence is simulated by explicitly iterating over all array indices and performing the actual computation that would occur at run-time;
- starting from the same initial ranges, the corresponding recurrence instance is constructed, and its predicted range is directly computed using the `at(k)` method;
- the range obtained through the real simulation is compared with the range computed by the recurrence.

To ensure a sufficiently broad coverage of possible scenarios, the entire process is repeated several times, each time using new randomly generated values. The validation criterion is strictly conservative: if, even in a single case, the real range exhibits bounds that lie outside the range computed by the recurrence, the test is considered to have failed.

6.1.3. Validation results and analysis

Recurrence Type	Succ./Fail.	Widest Real Range	Computed Range
Aff.	20/0	[-21.4872, 17.523]	[-50, 50]
Aff. flattened	20/0	[-32.5444, 23.5486]	[-50, 50]
Aff. delta	20/0	[-57.338, 46.7253]	[-550, 550]
Aff. crossing (1st)	20/0	[-20.1394, 20.5617]	[-71, 71]
Aff. crossing (2nd)	20/0	[-22.1328, 19.3342]	[-73, 73]
Geo.	20/0	[0.295023, 2.20871]	[0.0563135, 9.31323]
Geo. flattened	20/0	[0.411685, 1.80728]	[0.167772, 4.29982]
Geo. delta	20/0	[0.0879051, 5.60153]	[1.00013e-08, 2.40341e+06]
Geo. crossing (1st)	20/0	[0.7891, 13.0793]	[0.00242644, 1674.8]
Geo. crossing (2nd)	20/0	[0.825694, 12.8169]	[0.00169851, 2177.24]
Lin. recurrence	20/0	[18.3075, 47.5333]	[1.26, 108]
Fake recurrence 1D	20/0	[-10.6157, 11.1544]	[-12, 12]
Fake recurrence 2D	20/0	[-11.2231, 11.8153]	[-12, 12]

Table 6.1: Validation results of closed-form recurrence formulas: comparison between real simulated ranges and ranges computed through the `at(k)` method.

From Table 6.1 it can be observed that all tests were successfully completed without any failures. The ranges computed through the `at(k)` methods always include the corresponding real ranges obtained via simulation, thus confirming the soundness property of the proposed approach. In a context where inputs are randomly generated and only their minimum and maximum bounds are known, the ranges produced by the recurrences represent a correct and conservative over-approximation of the values actually assumed at run-time.

The objective of this validation phase can therefore be considered fully achieved, since in no case did the real ranges exceed the computed ones.

As an additional remark, it can be noticed that in many cases the real ranges are significantly narrower than those estimated by the recurrences. This behavior is partly due to the fact that random values are generated according to a uniform distribution, so that, on average, the observed values tend to lie far from the theoretical interval extremes.

A possible future extension of the method could consist in introducing additional statistical information, such as the mean and variance of input vectors, in order to obtain more precise range estimations. However, such an approach would shift the analysis from

a strictly conservative framework to a probabilistic one, and might therefore no longer guarantee soundness in a formal sense. Nevertheless, this direction represents an interesting potential trade-off between analysis precision and reliability.

6.2. Validation of the algorithm and input

6.2.1. Objectives

The second part of the experimental analysis represents the core of the work carried out. It has been divided into two distinct phases. In the first phase, the ability of the RA-VRA algorithm to identify the recurrences present in the proposed codes is evaluated, together with its capability to correctly assemble them and exploit them to determine the ranges of the values involved in the computation. It is then compared with the previous version of Value Range Analysis Pass in order to highlight the differences in term of detected ranges.

In the second phase, which is more analytical in nature, several tests are performed in which the width of the input ranges is progressively varied, with the aim of assessing the impact of such variations on the overall performance of the precision tuning process.

For both phases, the same source codes are employed. Each of them has been designed to include one or more types of recurrences, in order to allow a systematic and comparable evaluation of the algorithm behavior across different scenarios.

6.2.2. Test environment and compilation parameters

In order to ensure reproducibility of the experiments and to allow a correct interpretation of the results, the hardware and software environments used for the execution of the tests are described in detail below.

The experimental setup was configured as follows:

- **Guest operating system:** Debian GNU/Linux 12 (Bookworm), running inside a Windows Subsystem for Linux (WSL) virtual environment;
- **Host machine:** Windows 11, equipped with an AMD Ryzen 5 2600 CPU (6 physical cores, 12 logical threads) and 64 GB of RAM;
- **Resources allocated to the virtual machine:** approximately 50 GB of RAM;
- **TAFFO version:** based on LLVM-18 [24].

All experimental tests were executed using CTest, the test driver program provided by the CMake build system [12].

- `-O3`: enables the highest level of compiler optimization, activating aggressive transformations aimed at maximizing the performance of the generated code. This ensures that the comparison with TAFFO is carried out against an already highly optimized baseline;
- `-fno-vectorize`: disables automatic loop vectorization performed by the compiler. This option was selected in order to prevent SIMD transformations from affecting performance in an uncontrolled manner and to make the comparison between compiled versions more interpretable;
- `-fno-slp-vectorize`: disables Superword Level Parallelism (SLP) vectorization, which operates at the basic block level by grouping independent scalar operations into vector instructions. This option further contributes to keeping the generated code strictly scalar, leading to more stable and deterministic measurements.

The combined use of these parameters allows for a consistent comparison between traditionally compiled code and code optimized through TAFFO, minimizing the impact of automatic compiler optimizations that could otherwise introduce variability not directly related to the precision tuning process.

6.2.3. Experimental setup

For the execution of the tests, dedicated source codes were specifically developed with the purpose of highlighting, in a controlled manner, the different types of recurrences analyzed in the previous chapter. Both scalar-based cases and vector-based scenarios were considered, to cover a representative set of application contexts.

Each test follows the procedure described below:

- Input data are generated randomly only once, using a uniform distribution within a predefined interval. These values are stored and reused unchanged in all subsequent executions, in order to ensure reproducibility of the results and to make performance measurements directly comparable;
- The recurrent operations defined by the specific test are applied, and the produced results are stored in dedicated destination variables;
- All output values obtained at the end of the computation are written to a file, so that they can be later analyzed and compared;

- In order to obtain stable and meaningful execution time measurements, all previous operations, with the exception of the input initialization phase, are repeated several times. This allows the measurement to focus exclusively on the computational kernel of the test, excluding any overhead related to data generation;
- The source code is first compiled and executed using a standard compiler, thus providing a baseline reference, and then compiled again using TAFFO and executed a second time, in order to evaluate the impact of precision tuning;
- Finally, the output files generated by the two versions of the program are compared, analyzing the differences both in terms of execution time and numerical precision of the obtained results.

Execution time measurements are performed at low level by directly reading the CPU cycle counter through the `RDTSC` and `RDTSCP` instructions. This approach allows a highly accurate estimation of the number of cycles actually required by the computational kernel, minimizing the influence of external factors such as operating system activity, system load, or the latency of system calls. The measured timings are recorded in dedicated files and subsequently analyzed in order to reliably compare the performance of the different compiled versions of the code.

Thanks to this experimental procedure, it is possible to carry out a direct and controlled comparison between the behavior of traditionally compiled code and code optimized through TAFFO, enabling a systematic evaluation of the impact of precision tuning techniques on both performance and numerical accuracy.

In the second part of the analysis, only the proposed RA-VRA was evaluated, as recurrence relations were not supported by the previous VRA implementation. For affine recurrences, the base configuration (scale factor equal to 1) was set to the range $[-0.5, 1]$. From this reference interval, the scaling factor was reduced by both 10^{-1} and 10^{-3}, and increased by 10, 10^3, 10^6, 10^9, and 10^{12}, to evaluate the stability of the model across multiple orders of magnitude.

For geometric recurrences, instead, the base range corresponding to scale factor 1 was fixed to [1, 2.5]. In this case, the scaling factor was only increased by 10, 10^2, and 10^3, with the explicit goal of triggering the divergence and analyzing the resulting numerical behavior.

6.2.4. Evaluation metrics

Starting from the values obtained in the experiments, it is possible to derive two main types of metrics, which are used to evaluate the effectiveness of the precision tuning process in the context of each specific test. The metrics adopted are the same originally introduced in the first implementation of TAFFO and described in detail in Section 4.2.2 of the thesis by Cattaneo and Di Bello [8]. They are briefly recalled here for the sake of clarity and convenience.

The first metric considered is the *speed-up*, which can be computed using the execution times recorded during the tests. The percentage speed-up is defined as:

$$S\% = \left(\frac{t_{fl}}{t_{fx}} - 1 \right) \cdot 100$$

where t_{fl} represents the execution time of the floating-point version, and t_{fx} denotes the execution time of the optimized fixed-point version. Values of $S\%$ greater than zero indicate an actual performance improvement achieved through the optimization process.

In order to quantify the errors introduced by the reduction in numerical precision, two additional metrics are employed. The first one is called *Average Absolute Error* (mean_abs_err) and is defined as:

$$E_a = \frac{1}{N} \sum_{i=1}^{N} |d_{fx}(i) - d_{fl}(i)|$$

where N is the total number of output values, $d_{fx}(i)$ represents the value at position i computed by the fixed-point version, and $d_{fl}(i)$ is the corresponding value produced by the floating-point version.

The second metric is the *Average Relative Error* (mean_rel_err), defined as:

$$E\% = 100 \cdot \frac{\sum_{i=1}^{N} |d_{fx}(i) - d_{fl}(i)|}{\sum_{i=1}^{N} |d_{fl}(i)|}$$

This metric allows the error to be evaluated in normalized form with respect to the magnitude of the reference values, providing a more meaningful measure when the involved quantities span different numerical scales.

The combination of these metrics therefore enables a quantitative analysis of the trade-off between performance improvement and loss of numerical precision introduced by the

precision tuning process.

6.2.5. Validation result and analysis

The following section presents the results obtained by comparing the old and the new VRA with the RA-VRA implemented, applying range annotations exclusively to input values while leaving the output vector ranges to be automatically inferred by the analysis algorithm. The complete result tables are reported in Appendix B.

From the scatter plot 6.1, it can be observed that the two VRA versions are overall comparable in terms of speed-up and mean relative error. No significant correlation emerges between performance improvements and loss of precision. The only exception is represented by the orange point located in the upper-left region of the plot, corresponding to the *geo_ delta* recurrence. In this case, the annotated range in the old VRA, although very wide, acts as a static bound and prevents the propagation of the recursive growth. In contrast, the closed-form expression adopted by the RA-VRA, due to its exponential nature, leads to a rapid expansion of the estimated interval.

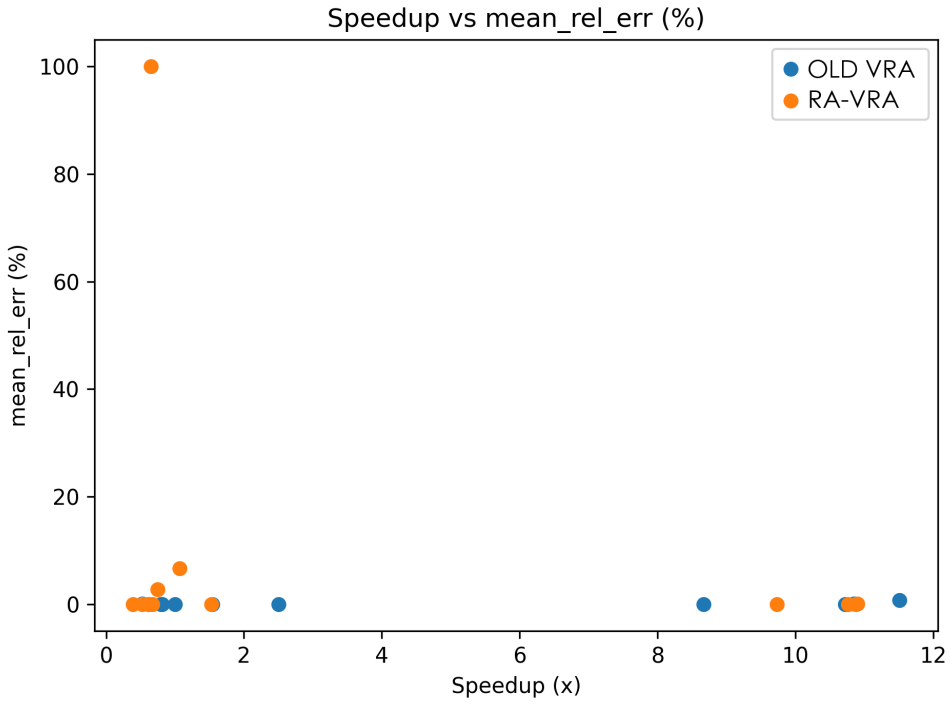


Figure 6.1: Scatter plot comparing speedup and mean relative error between the old VRA and the new RA-VRA.

A similar behavior can be observed in the second scatter plot 6.2, which relates speed-up to the mean absolute error. The outlier associated with *geo_ delta* dominates the distribution, whereas the remaining benchmarks exhibit a limited absolute error, consistent with the trend observed in the relative error analysis.

To provide a clearer interpretation of the overall behavior, the median of the mean absolute error was also reported for both approaches. This choice mitigates the influence of extreme cases and yields a more representative assessment of the typical behavior. It can be observed that RA-VRA reduces the median mean absolute error by approximately 2.2× compared to the previous version, indicating a systematic improvement in the typical error magnitude.

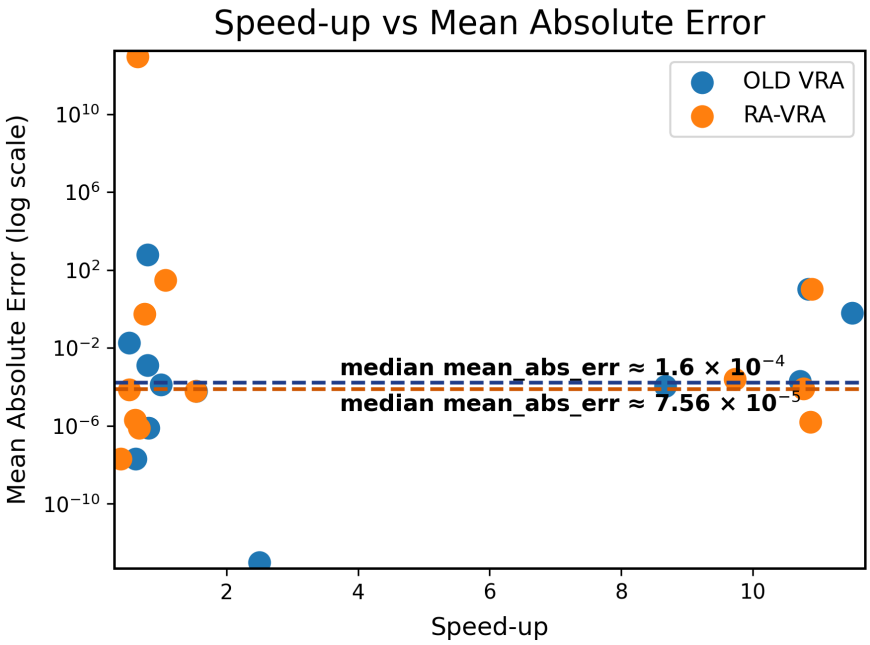


Figure 6.2: Scatter plot comparing speedup and mean absolute error between the old and the RA-VRA with median lines.

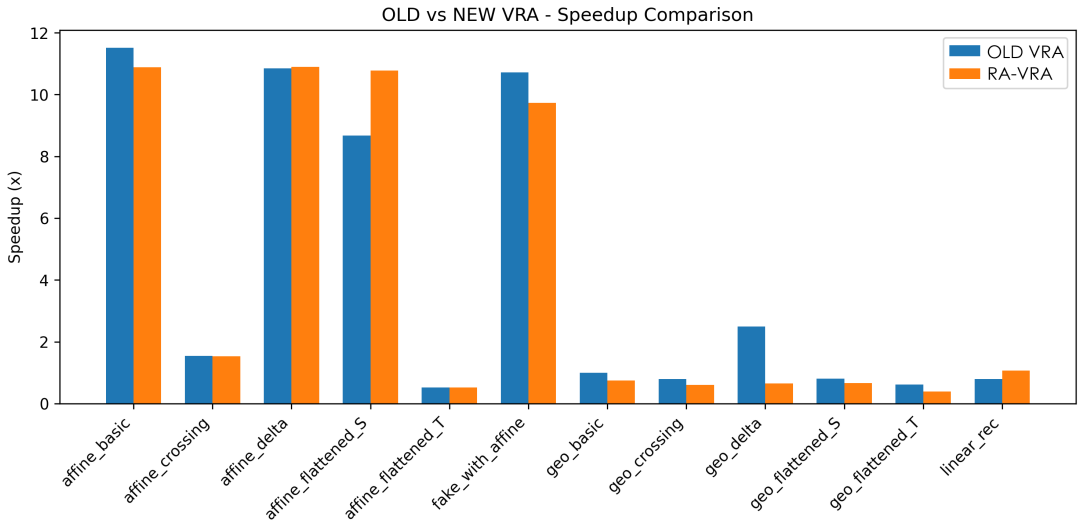


Figure 6.3: Speedup comparison per benchmark between the old VRA and the new RA-VRA.

In the second figure 6.3, it can be observed that the speedups achieved by the two VRA versions are overall comparable. This allows us to state that the proposed algorithm does not negatively impact the performance of the precision tuning process. In general, manually annotated ranges tend to be narrower and may therefore enable higher speedups; nevertheless, the observed differences are minimal, validating the proposed approach, which avoids the need for pervasive output range annotations. It is natural to observe that in benchmarks such as *affine_delta* and *affine_flattened* wider bounds tend to be produced. In these cases, the analysis propagates the extreme values of the interval across many iterations, resulting in a progressive accumulation of the upper and lower limits. This behavior is consistent with a conservative approach: extreme scenarios, although less representative of typical executions, must be considered in order to preserve the soundness of the method.

Nevertheless, having explicit knowledge of the bounds at the end of iterative constructs represents valuable structural information. The availability of well-defined extremal values enables the application of subsequent heuristics or refinement strategies aimed at narrowing the ranges in a more controlled and targeted manner, without compromising the correctness of the analysis.

Finally, the last bar chart 6.4 shows that compilation times are also very similar between the two versions. Considering the additional effort required by the algorithm to recognize, assemble, and propagate recurrences, this represents a particularly significant result.

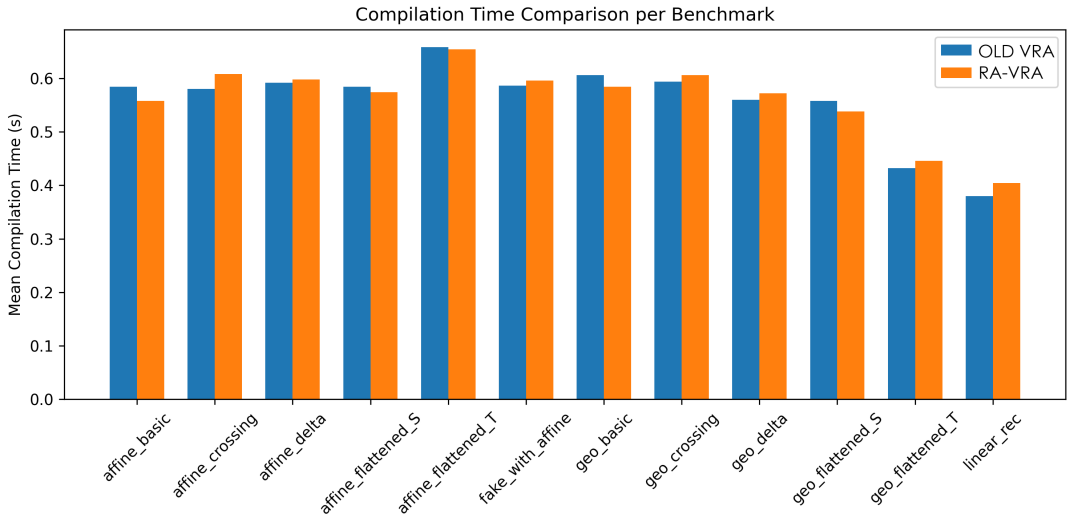


Figure 6.4: Compilation time comparison per benchmark between the old VRA and the new RA-VRA.

Stress test on recurrences The following part reports the results of the stress test performed on affine, geometric and linear recurrences.

Due to their linear growth behavior, it can be observed that the *mean_rel_err* on affine recurrences (Figure 6.5) remains essentially negligible even for very large initial ranges. Most affine variants remain stable with negligible relative error across several orders of magnitude, while `affine_delta` and `affine_flattened_T` exhibit loss of precision only at extremely large scales.

Similarly, the *mean_abs_err* (Figure 6.6), while practically zero for small bounds, increases in an approximately linear fashion and proportionally to the width of the initial range, maintaining a controlled and predictable trend. Standard affine forms show gradual and predictable error growth, whereas `affine_delta` and `affine_flattened_T` display super-linear behavior at very large magnitudes.

A different behavior is observed for geometric and linear recurrences. The corresponding results are reported in Figures 6.7 and 6.8. In these cases, even moderate increases of the initial range—by a few orders of magnitude—lead to a marked degradation of accuracy.

This effect can be attributed to the intrinsic multiplicative component of such recurrences, which induces a super-linear or exponential growth of the error, rapidly resulting in numerical instability.

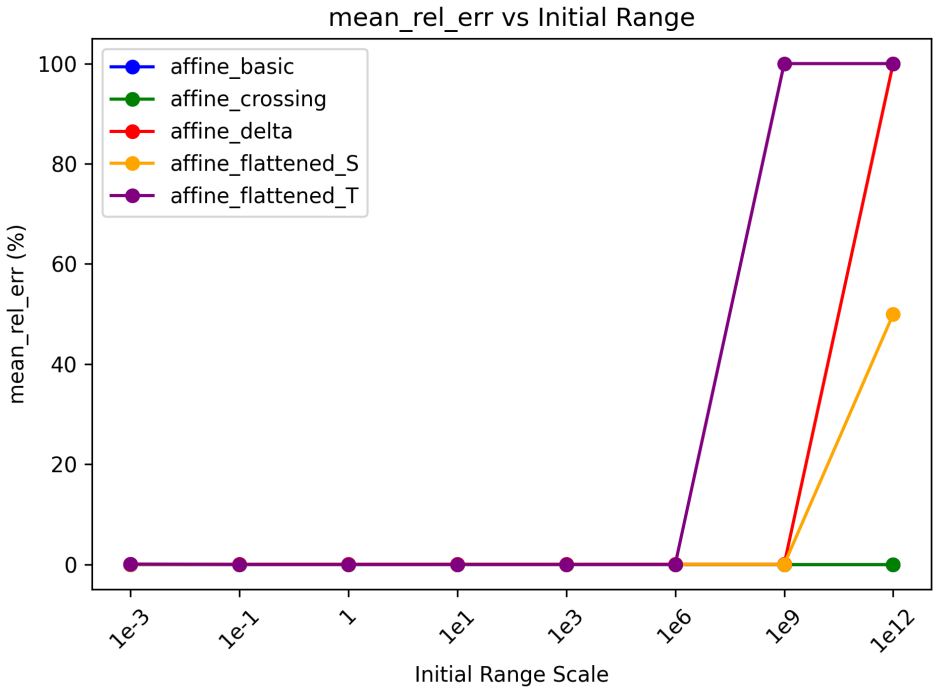


Figure 6.5: Trend of the mean relative error for affine recurrences as the initial input range progressively increases

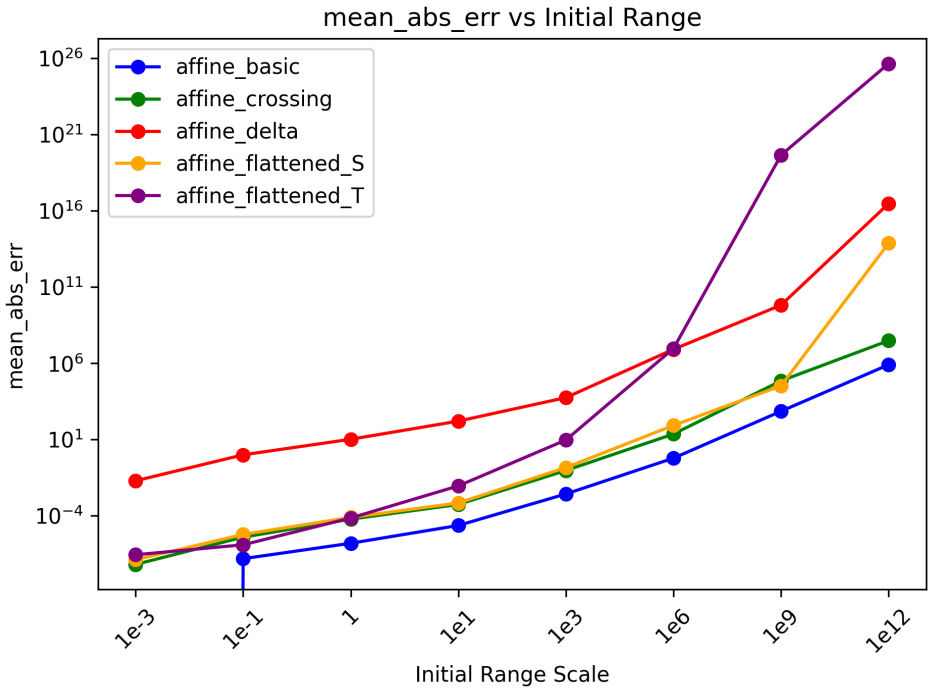


Figure 6.6: Trend of the mean absolute error for affine recurrences under progressively wider initial ranges (logarithmic scale on the y-axis).

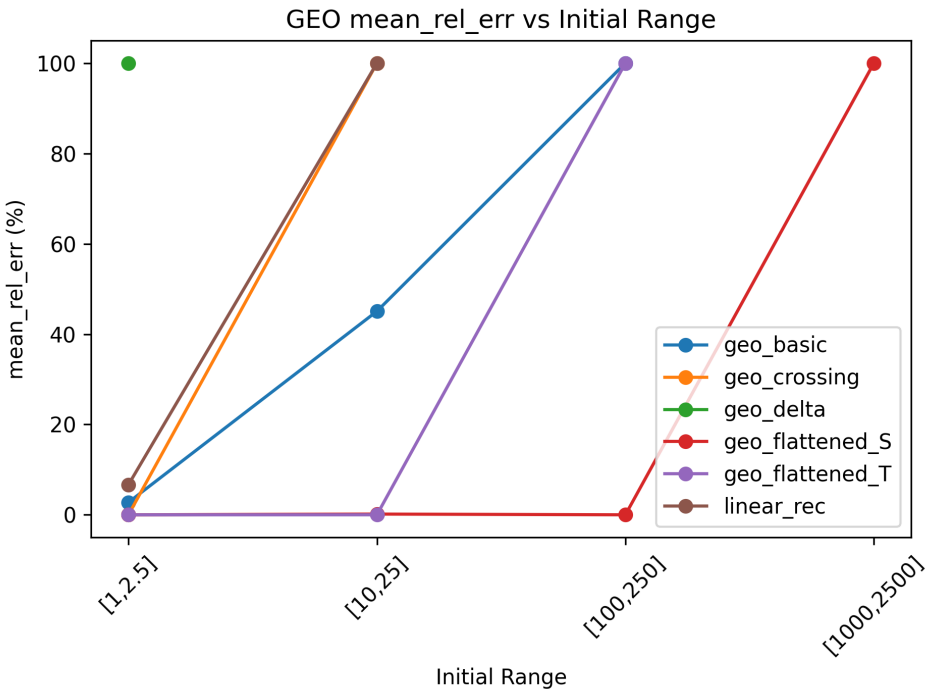


Figure 6.7: Trend of the mean relative error for geometric and linear recurrences as the multiplicative range increases.

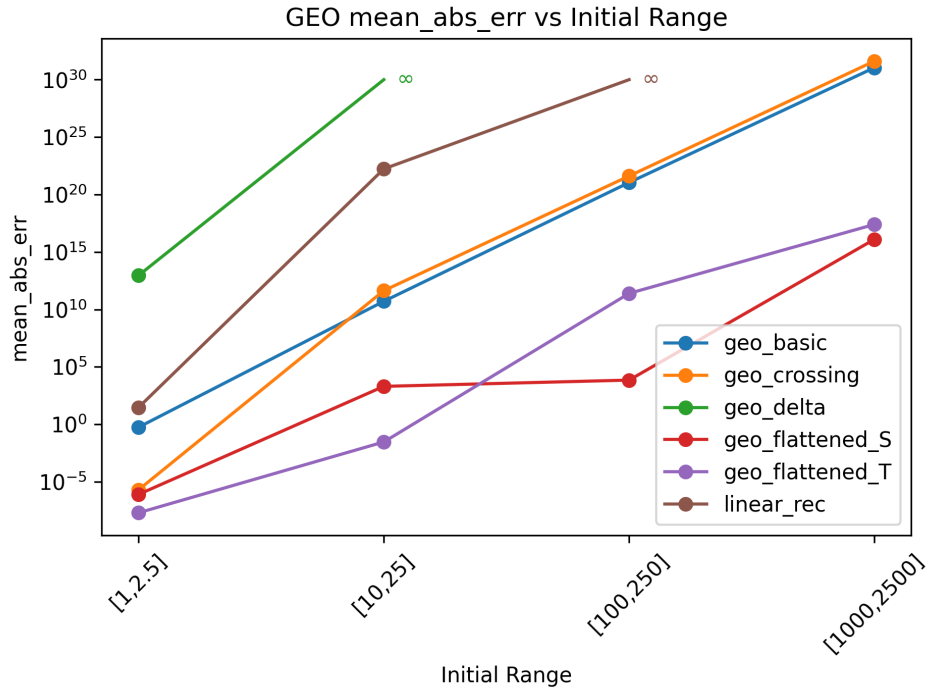


Figure 6.8: Trend of the mean absolute error for geometric and linear recurrences under increasing multiplicative ranges (logarithmic scale on the y-axis).

6.3. Validation on PolyBench

The final phase of the experimental evaluation applies the proposed method to the *PolyBench* benchmark suite [21]. This collection includes a wide range of numerical kernels characterized by static control flow and is extensively used to evaluate optimization techniques and static analysis methods. The benchmarks cover multiple application domains, including linear algebra, physical process simulations, and statistical analysis.

The choice of this suite is motivated by the presence, within its source code, of several forms of recurrence patterns analyzed in this work, applied to vectors and matrices of varying sizes. Not all available benchmarks were considered; the analysis was restricted to cases compatible with the recurrence types studied in this thesis.

6.3.1. Objectives

This validation phase pursued a twofold objective. First, the original version of the VRA was compared with the newly proposed implementation, following a methodology analogous to that adopted in the RA-VRA-specific test described in Section 6.2.

Second, the impact of automatically inferred ranges on the precision tuning process was evaluated for the selected benchmarks.

Finally, cases where the applicability of the proposed method is limited were analyzed and discussed, providing a technical justification for such limitations and a possible solution for future improvements.

6.3.2. Test setup

The testing environment and compilation parameters were kept unchanged with respect to the previous evaluations.

Unlike the earlier experiments, in this phase the original VRA was executed using its initial configuration, with range annotations applied to the most representative variables. The RA-VRA, instead, operates without any range annotations, leaving the inference of value ranges entirely to the proposed algorithm.

The procedure then follows the workflow described in Section 6.2.3. The results produced by the original VRA are compared with those obtained with the new implementation, evaluating execution speedup, compilation time, and mean error (relative and absolute).

Additionally, the inferred ranges of the most representative vectors in each benchmark-

typically those associated with the program outputs—are recorded separately. These data are subsequently used to support and motivate the interpretation of the experimental results.

6.3.3. Algorithm capability and range evaluation metrics

The three metrics described in Section 6.2.4 are also employed in this phase to compare the previous approach with the proposed one. In addition, further metrics are introduced to specifically assess the quality of the inferred ranges.

The first metric evaluates the ability of the algorithm to correctly detect the recurrence patterns implemented. For this purpose, the number and types of recurrences that are theoretically identifiable within the analyzed benchmarks were manually determined and compared against those actually detected by the algorithm.

The tool automatically produces statistics reporting both the number and the categories of recurrences identified.

The second metric evaluates the soundness property of the analysis.

Let

$$R_{\text{real}} = [r_{\min}, r_{\max}]$$

be the real range observed at runtime, and

$$R_{\text{calc}} = [c_{\min}, c_{\max}]$$

be the statically inferred range.

The analysis is considered sound if the following condition holds:

$$c_{\min} \leq r_{\min} \quad \text{and} \quad c_{\max} \geq r_{\max}.$$

That is, the real range must be entirely contained within the computed one.

As recalled in Appendix A, the width of a range is defined as:

$$W = r_{\max} - r_{\min}.$$

The *Over-Approximation Ratio* (OAR) is defined as:

$$\mathrm{OAR} = \frac{W_{\mathrm{calc}}}{W_{\mathrm{real}}}.$$

A value equal to 1 indicates perfect correspondence between computed and real ranges. Values greater than 1 quantify the degree of over-approximation.

The same concept can be expressed on a normalized scale between 0 and 1 by defining the *Tightness Index* (TI) as the inverse of the OAR:

$$\mathrm{TI} = \frac{W_{\mathrm{real}}}{W_{\mathrm{calc}}}.$$

A value equal to 1 indicates exact matching, whereas values approaching 0 indicate increasing imprecision.

Finally, a more fine-grained evaluation of the interval bounds is provided by the *Normalized Bound Error* (NBE):

$$\mathrm{NBE} = \frac{|r_{\min} - c_{\min}| + |r_{\max} - c_{\max}|}{W_{\mathrm{real}}}.$$

This metric assumes value 0 in the case of perfect bound matching. Values close to 1 indicate a total deviation comparable to the real range width, while values greater than 1 denote progressively higher imprecision. All the newly introduced metrics are evaluated exclusively on the RA-VRA version. The previous implementation relies on manually annotated ranges, which are not derived from an automatic inference procedure and therefore do not constitute a meaningful reference for evaluating bound accuracy.

6.3.4. Validation result and analysis

Most of the benchmarks in the suite were evaluated by comparing the previous version of the VRA with the proposed one. The cases not reported fall mainly into two categories: benchmarks requiring the implementation of additional recurrence patterns and cases presenting specific critical issues that are still under investigation. For clarity, these situations were not included in the direct comparison and are instead discussed separately.

The scatter plot in Figure 6.9 illustrates the relationship between speed-up and mean relative error for both configurations. In the majority of the benchmarks, the error remains close to zero for both versions. However, the Old VRA exhibits a few significant outliers in terms of relative error (notably in covariance, syrk, and fdtd-2d), whereas the proposed

version mitigates such deviations while maintaining comparable or, in several cases, higher speed-up values.

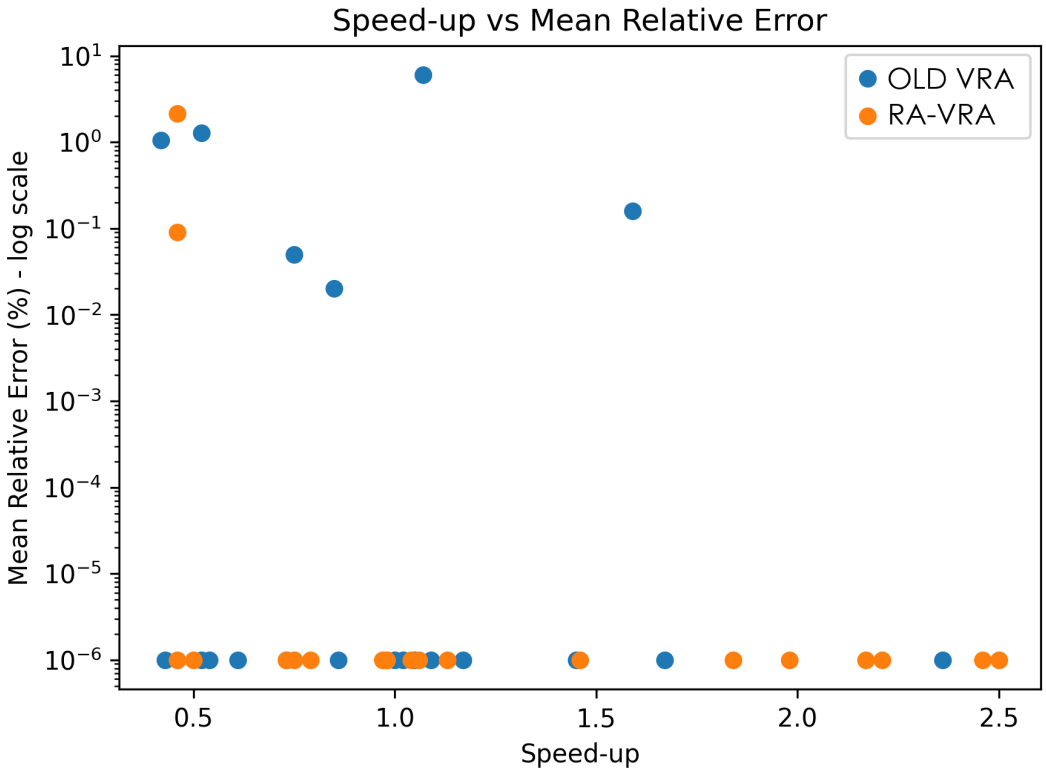


Figure 6.9: Scatter plot comparing speedup and mean relative error between the old VRA and the new RA-VRA.

An even clearer trend emerges from the scatter plot in Figure 6.10, where speed-up is compared against the mean absolute error. Also the median value for both configurations is reported to provide a robust indicator that is less sensitive to the presence of potential outliers.

The comparison between the two medians highlights a reduction of approximately one order of magnitude in the error: the median decreases from 5.6×10^{-4} for the Old VRA to 2.35×10^{-5} for the proposed version. This result confirms that the improvement introduced is not limited to isolated cases, but rather represents a structural benefit across the entire benchmark suite.

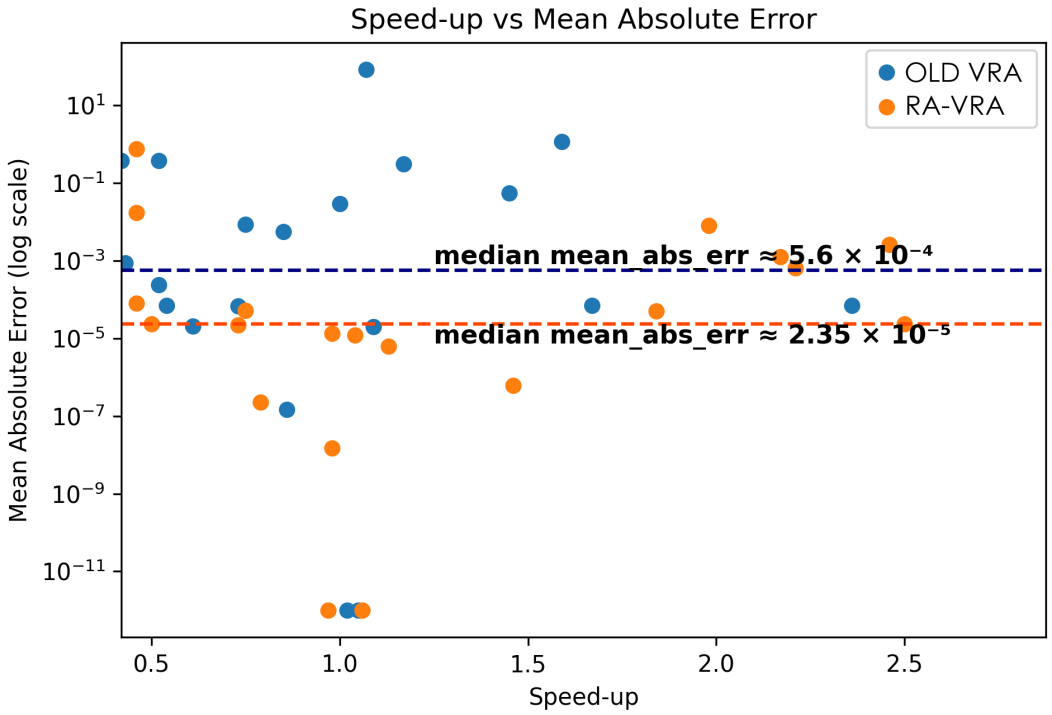


Figure 6.10: Scatter plot comparing speedup and mean absolute error between the old VRA and the new RA-VRA with median lines.

Figure 6.11 presents the direct comparison of speed-up for each benchmark. In several cases (such as 2mm, 3mm, atax, and doitgen), the RA-VRA shows a clear improvement over the previous version. In most of the remaining benchmarks, performance remains largely comparable, with only marginal variations. Overall, the median speed-up exhibits a slight increase, confirming that the proposed methodology does not introduce a performance penalty and, in fact, provides a modest but consistent improvement.

In a limited number of cases, a slight degradation can be observed. This behavior is consistent with the nature of the proposed algorithm: each detected recurrence progressively accumulates the interval extrema without considering the effective distribution of values at intermediate indices. Although this approach may produce more conservative bounds, it ensures increased robustness of the analysis.

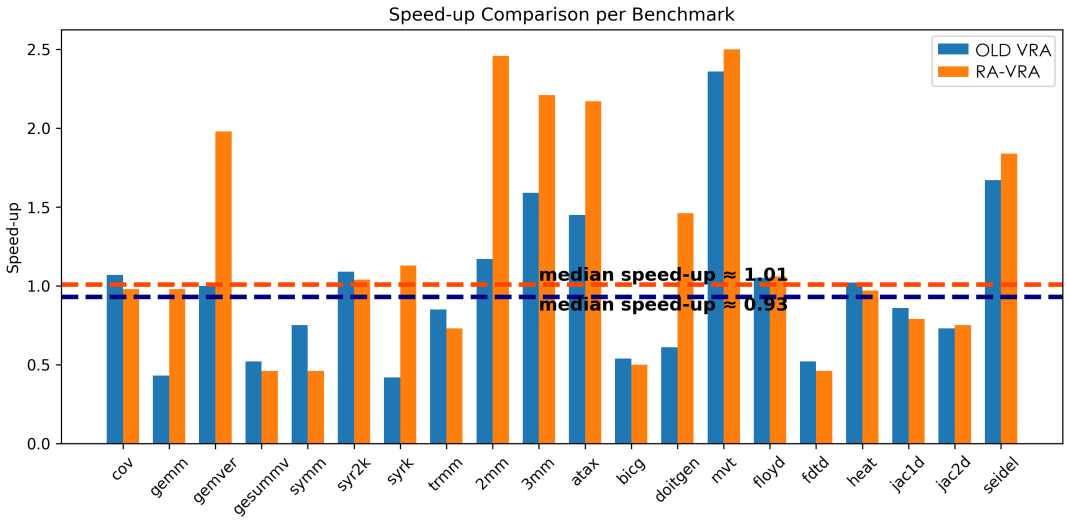


Figure 6.11: Speedup comparison per benchmark between the old VRA and the new RA-VRA with median lines.

It is worth emphasizing that, compared to arbitrary manual range annotations, the automatic determination of effective extrema at the end of program execution enables the future application of additional refinement heuristics or controlled stochastic strategies. This opens the possibility of explicitly balancing precision and soundness. A deeper investigation of such strategies is left for future work.

Finally, Figure 6.12 reports the comparison of compilation times. The introduction of the new algorithm leads to a slight increase in analysis time for some benchmarks, due to the additional computational cost associated with recurrence detection and propagation, as well as the management of dependencies among recurrences. However, this increase remains limited and does not translate into a significant growth in overall compilation time.

Overall, the introduced overhead is moderate and largely compensated by the improvements achieved in terms of range quality and, in several cases, execution speed-up.

Real and computed ranges for the most significant variables are reported in Appendix B.2.

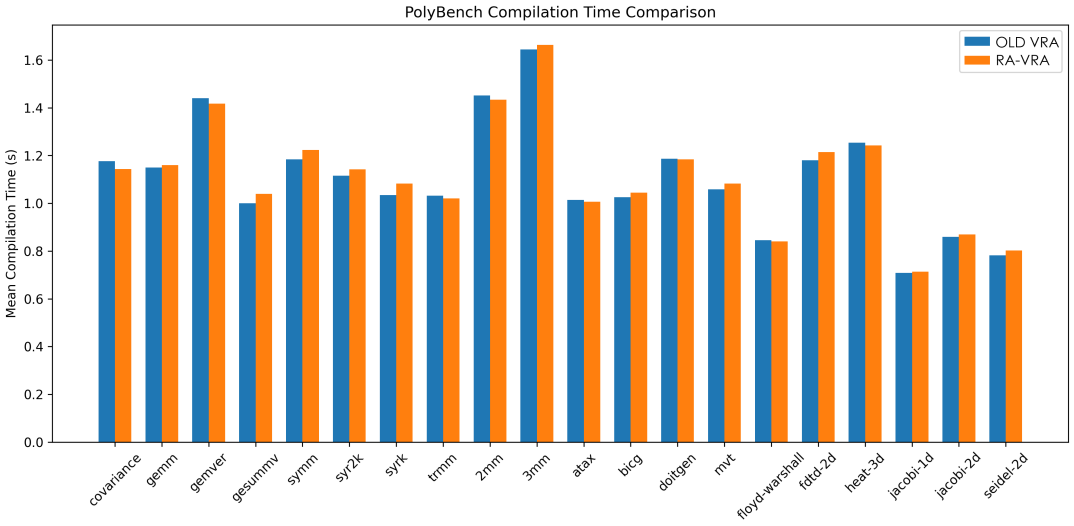


Figure 6.12: Compilation time comparison per benchmark between the old VRA and the new RA-VRA.

Table 6.2 reports the comparison between the computed ranges and the real ranges for the PolyBench benchmarks. It can be observed that soundness is guaranteed in all cases, as well as complete recognition of the implemented recurrences (100%).

In the majority of the benchmarks, an over-approximation of the computed ranges with respect to the real ones can be observed. This behavior is coherent with the primary objective of the proposed analysis, which prioritizes soundness over tight bound precision. The accumulation is modeled by conservatively assuming that, at each iteration, the extreme values of the interval may contribute to the result. Although this represents a worst-case scenario that is statistically unlikely in practice, it is necessary to ensure the formal correctness of the analysis.

The behavior is more controlled in the linear algebra benchmarks with predominantly affine recurrences (from *gemm* to *mvt*), where the growth of the bounds remains essentially linear and the Over-Approximation Ratio remains limited. In these cases, the analysis proves particularly suitable, as the expansion of the interval is proportional to the trip count and does not exhibit explosive growth.

Benchmark	OAR	TI	NBE	% det. rec.	Is Sound
covariance	56652.37	0.00	56651.59	100%	Y
gemm	1.71	0.77	0.71	100%	Y
gemver	4.52	0.44	3.52	100%	Y
gemsum	1.82	0.73	0.82	100%	Y
symm	5609.77	0.50	5608.77	100%	Y
syr2k	2.00	0.75	1.00	100%	Y
syrk	4.40	0.56	3.40	100%	Y
trmm	1.88	0.68	0.88	100%	Y
2mm	2.45	0.70	1.45	100%	Y
3mm	1.00	1.00	0.00	100%	Y
atax	18.94	0.52	17.94	100%	Y
bicg	1.64	0.98	0.79	100%	Y
doitgen	2.32	0.56	1.32	100%	Y
mvt	1.82	0.73	0.82	100%	Y
floyd-warshall	250.50	0.00	249.50	100%	Y
fdtd-2d	9879.38	0.02	9878.38	100%	Y
heat-3d	4.49	0.27	3.49	100%	Y
jacobi-1d	1.00	1.00	0.00	100%	Y
jacobi-2d	1.01	0.99	0.01	100%	Y
seidel-2d	1.01	1.00	0.01	100%	Y

Table 6.2: Comparison between Real and Computed Ranges on PolyBench Benchmarks

Among the linear algebra kernels, *symm* shows a significant over-approximation. However, it is not an isolated case: more pronounced phenomena can be observed in *covariance*, *fdtd-2d*, and *floyd-warshall*, where the Over-Approximation Ratio reaches very high values. In *symm*, the presence of extreme values along the main diagonal of matrix A contributes to amplifying the accumulation and consequently widening the bounds.

In the *atax* benchmark, a more pronounced over-approximation can be observed compared to other linear algebra kernels. It is plausible that this behavior is related to the presence of division operations depending on induction variables during initialization. While such

operations effectively reduce the magnitude of the computed values at runtime, the analysis conservatively preserves the interval of the dividend, contributing to the enlargement of the final result.

In *covariance*, a similar dynamic occurs due to division operations inside the kernel. Individual instructions are analyzed with limited intervals, but the combined effect of accumulation does not allow a progressive narrowing of the bounds, which remain significantly wider than the observed real values.

For *fdtd-2d*, the over-approximation is mainly due to the fact that the analysis does not capture the numerical convergence behavior of the iterative method. The accumulation is therefore modeled as potentially proportional to the number of iterations, without accounting for the stabilizing effect of the simulated physical process. Moreover, the initial ranges are already of higher order of magnitude compared to other benchmarks, further amplifying the phenomenon.

In *floyd-warshall*, the excessive over-approximation can be attributed to the presence of conditional constructs inside the accumulation. The recurrence is applied only in specific cases, but the analysis conservatively assumes that such updates may occur at every iteration. An extension capable of estimating the effective selection frequency of conditional branches could enable a more precise resolution of the recurrence.

Finally, the last benchmarks (*jacobi-1d*, *jacobi-2d*, and *seidel-2d*) exhibit Over-Approximation Ratios close to one. In these cases, only recurrences associated with induction variables are present, without significant accumulation behavior that would progressively widen the intervals. The analysis is therefore able to maintain bounds very close to the real ones.

Overall, the results confirm that the proposed approach based on RA-VRA robustly guarantees soundness and complete recurrence detection, while providing good precision in affine growth scenarios and highlighting the current limitations in contexts characterized by numerical convergence, progressive normalization, or conditionally applied accumulations.

6.3.5. Unsupported cases and limitations

Among the benchmarks that are not yet supported are *correlation*, *cholesky*, and *deriche*. These cases require an extension of the current model, particularly with respect to the handling of recurrences in the presence of specific mathematical functions. The aforementioned benchmarks include operations such as square roots and exponential functions. These functions are monotonic over the domain of interest and are, in principle, fully

compatible with interval analysis. However, their effective integration within the recurrence model requires a dedicated treatment capable of preserving both soundness and an adequate level of precision. The proposed method therefore maintains its applicability, but requires a targeted extension to properly cover such constructs.

The main limitation encountered in the context of recurrence-based static analysis concerns the presence of interval divisors that cross zero. This situation has been identified in the benchmarks *durbin*, *lu*, *gramschmidt*, *trisolv*, *ludcmp*, and *adi*.

When an interval used as a divisor includes zero, the division operation must account for values arbitrarily close to zero. From an interval arithmetic perspective, this leads to potentially unbounded results, since division by infinitesimal quantities tends toward infinity. Under such conditions, the analysis inevitably produces extremely wide bounds, significantly degrading precision while still preserving soundness.

This does not necessarily imply that the method is inapplicable, but rather highlights the need for more refined strategies to handle these situations, possibly depending on the application context. One possible solution consists in splitting the divisor interval into distinct sub-intervals, treating the negative and positive components separately, and optionally excluding a region of indeterminacy around zero when permitted by the application domain. Such an approach could reduce over-approximation while maintaining formal correctness.

At the current stage, for the cases described above, the most robust solution remains the use of the previous version of the VRA, integrated with explicit range annotations at critical program points. This approach preserves the soundness of the analysis by relying on programmer-provided additional information in contexts where the automatic modeling of recurrences or interval operations is not yet sufficiently refined, while simultaneously limiting excessive over-approximation.

Fallback and annotation hybrid strategy. A final evaluation was conducted by testing the proposed method while preserving the original annotated ranges for the critical benchmarks, thereby generating a hybrid configuration. The results are reported in Tables 6.3 and 6.4.

In these cases, the *fallback* mechanism was activated for unrecognized recurrences. Before elevating the bound to the TOP element of the abstract domain, the analysis prioritizes the original annotations, considering them sufficiently informative. When over-approximation cannot be avoided—such as in the presence of unsupported recurrence patterns or divisions by intervals crossing zero—the bound is instead set to TOP. Under these conditions,

TAFFO automatically prevents the type conversion, preserving the floating-point representation.

The results show that this hybrid configuration does not introduce a systematic degradation in either performance or accuracy. In the benchmarks *correlation* and *adi*, an increase in speed-up was observed together with a complete elimination of the error. In *durbin*, *gramschmidt*, and *lu*, the error remained unchanged or very similar compared to the previous configuration. In *trisolv*, the absolute error was reduced by four orders of magnitude, while in *deriche* and *nussinov* no significant variations were observed in either performance or accuracy.

The only genuinely critical cases were *cholesky* and *ludcmp*, where the RA-VRA, due to the activation of the fallback on problematic intervals, was unable to provide sufficiently informative bounds. In these circumstances, the previous VRA version, supported by explicit annotations, currently remains the most effective solution.

Benchmark	Speedup	C. Time(s)	Mean Rel. Err.(%)	Mean Abs. Err.
correlation	0.83x	16.63	1.32	1.32e-02
cholesky	0.14x	13.76	0.02	1.39e-04
durbin	0.46x	9.44	0.38	4.96e-05
gramschmidt	0.55x	14.69	0.69	3.44e-02
lu	0.23x	14.23	0.32	2.11e-03
ludcmp	0.71x	18.94	0.42	4.83e-05
trisolv	0.41x	8.08	0.00	2.02e-06
deriche	0.97x	22.55	0.00	9.43e-09
nussinov	1.01x	9.37	0.00	0.00e+00
adi	0.62x	19.02	60.02	6.00e-01

Table 6.3: PolyBench results (excluded from main analysis) obtained using Old VRA with annotated ranges

Benchmark	Speedup	C. Time(s)	Mean Rel. Err.(%)	Mean Abs. Err.
correlation	0.85x	14.96	0.00	5.03e-16
cholesky	1.12x	13.98	NaN	NaN
durbin	0.43x	8.68	0.38	4.96e-05
gramschmidt	0.98x	13.83	0.69	3.43e-02
lu	1.00x	13.37	0.71	4.78e-03
ludcmp	0.97x	18.18	24.40	2.82e-03
trisolv	0.91x	7.53	0.00	2.78e-10
deriche	0.98x	21.99	0.00	9.38e-09
nussinov	1.00x	8.99	0.00	0.00e+00
adi	0.87x	17.72	0.00	2.55e-06

Table 6.4: PolyBench results (excluded from main analysis) obtained using RA-VRA with fallback to unbounded intervals

7 | Conclusions and future developments

The present work originated from the need to improve the accuracy of the Value Range Analysis within the TAFFO framework, with particular attention to iterative constructs. The previously adopted range propagation techniques relied either on the complete iteration of each loop or on delegating the management of bounds to the programmer through explicit annotations.

The main objective of this thesis was therefore to design and integrate into TAFFO a new VRA methodology capable of explicitly recognizing and modeling different types of recurrences — affine, geometric, and linear — in order to automatically derive more informative bounds and reduce the dependence on output annotations. Considering the variety and complexity of recurrent structures appearing in numerical kernels, particular attention was devoted to the scalability of the approach, increasing its applicability and enabling future extensions.

Given the static nature of the analysis, an additional goal was to guarantee the soundness of the proposed method, while maintaining compilation times and speed-up levels comparable to those of the previous VRA version.

In light of the obtained results, it can be stated that the new *RA-VRA* algorithm correctly detects the implemented recurrences and constructs bounds that are consistent with their semantic evolution. The integration of the new algorithm does not introduce significant regressions in compilation time and preserves overall speed-up levels comparable to the previous implementation. The ranges derived from recurrences remain sound and suitable for the subsequent precision tuning process.

The scalability of the approach is ensured by the modular structure of the algorithm, which progressively analyzes possible recurrences, classifying them when recognized or delegating them to fallback mechanisms otherwise.

The main limitation observed concerns the rapid growth of bounds, a phenomenon in-

trinsically related to the repeated application of operations. This behavior becomes particularly evident when the initial ranges are wide and is especially critical in geometric recurrences due to their exponential nature. In contrast, affine recurrences exhibit a more controlled growth, although they may still produce conservative bounds.

In the most problematic cases, a hybrid approach combining explicit annotations with automated VRA could represent a promising direction for future work, allowing the integration of programmer-provided semantic knowledge with the compiler's automatic analysis capabilities.

A further natural extension involves the study of additional recurrence types, particularly in the presence of non-linear behaviors or division operations where ranges include zero. Further improvements may include the introduction of narrowing mechanisms, either applied directly to the implemented recurrences or through a post-fixpoint phase leveraging the knowledge of extreme bounds.

In programs manipulating vector or matrix data, the introduction of stochastic components based on statistical measures such as mean and variance could also be explored, considering the low probability that all elements simultaneously assume extreme values. Although such an approach would relax strict theoretical soundness, if carefully calibrated it could significantly improve precision.

Finally, further integrations with other analysis techniques — both static, such as polyhedral methods, and dynamic — may be considered in order to broaden the applicability of the proposed methodology.

Bibliography

[1] J. Absar. Scalar evolution demystified. EuroLLVM Developers' Meeting, 2018. URL `https://llvm.org/devmtg/2018-04/`. Presentation and video.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[3] S. Bygde, B. Lisper, and N. Holsti. Improved precision in polyhedral analysis with wrapping. *Science of Computer Programming*, 133:74–87, 2017. ISSN 0167-6423. doi: 10.1016/j.scico.2016.07.006.

[4] V. H. S. Campos, R. Rodrigues, and F. Q. Pereira. Speed and precision in range analysis. In *Proceedings of the 12th Brazilian Symposium on Programming Languages (SBLP)*, pages 42–56, 2012.

[5] D. Cattaneo, M. Chiari, G. Agosta, and S. Cherubin. TAFFO: The compiler-based precision tuner. *SoftwareX*, 20:101238, 2022. doi: 10.1016/j.softx.2022.101238.

[6] S. Cherubin and G. Agosta. Tools for reduced precision computation: A survey. *ACM Computing Surveys*, 53(2), Apr. 2020. doi: 10.1145/3381039.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, USA, 1977. ACM. doi: 10.1145/512950.512973.

[8] A. D. B. Daniele Cattaneo. Tuning assistant for floating point to fixed point optimization. Master's thesis, Politecnico di Milano, https://www.politesi.polimi.it/handle/10589/144730, 2018.

[9] E. Darulova and V. Kuncak. Towards a compiler for reals. *ACM Transactions on Programming Languages and Systems*, 39(2):1–28, Mar. 2017. doi: 10.1145/3014426. URL `https://doi.org/10.1145/3014426`.

[10] D. Goldberg. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. ACM Computing Surveys, 1991.

[11] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 2nd edition, 1994. ISBN 978-0201558029.

[12] Kitware, Inc. Ctest – cmake test driver program. `https://cmake.org/cmake/help/latest/manual/ctest.1.html`, 2026. Official documentation of the CTest executable in the CMake suite.

[13] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1997. ISBN 0201896842. doi: 10.5555/270146.

[14] I.-I. Kum, J. Kang, and W. Sung. AUTOSCALER for C: An optimizing floating-point to integer c program converter for fixed-point digital signal processors. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47 (9):840–848, 2000. ISSN 1057-7130. doi: 10.1109/82.868453.

[15] L. Lamport. *IEEE Standard for Floating-Point Arithmetic - 754-2019*. Pearson Education India, 1994.

[16] LLVM Project. Llvm language reference manual. `https://llvm.org/docs/LangRef.html`, . Accessed: 2025-12-22.

[17] LLVM Project. The llvm project. `https://llvm.org/`, . Accessed: 2025-12-22.

[18] D. Menard, D. Chillet, F. Charot, and O. Sentieys. Automatic floating-point to fixed-point conversion for dsp code generation. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, pages 270–276, Grenoble, France, 2002. ISBN 1-58113-575-0. doi: 10.1145/581630.581674.

[19] N. Nicolosi, G. Magnani, E. Corigliano, D. Baroffio, F. Reghenzani, and G. Agosta. Type deduction analysis: Reconstructing transparent pointer types in LLVM-IR. In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction (CC 2026)*, pages 194–204, Sydney, NSW, Australia, 2026. Association for Computing Machinery. ISBN 979-8-4007-2274-5. doi: 10.1145/3771775.3786268. URL `https://doi.org/10.1145/3771775.3786268`.

[20] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for c-like languages. In *Proceedings of the 33rd ACM SIGPLAN*

Conference on Programming Language Design and Implementation (PLDI), pages 229–238, Beijing, China, 2012. ACM. doi: 10.1145/2254064.2254092.

[21] L.-N. Pouchet. Polybench/c: The polyhedral benchmark suite. In *Proceedings of the International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2012. http://polybench.sourceforge.net.

[22] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 164–174. ACM, 2011. doi: 10.1145/1993498.1993518.

[23] TAFFO-org. Taffo documentation (doc directory). GitHub repository documentation. URL `https://github.com/TAFFO-org/TAFFO/tree/master/doc`. Accessed: 2026-01-31.

[24] TAFFO Project. Taffo: Tuning assistant for floating to fixed point optimization. `https://github.com/TAFFO-org/TAFFO/tree/llvm-18`, 2024. Version based on LLVM 18.

[25] Wikipedia contributors. Linear recurrence with constant coefficients. `https://en.wikipedia.org/wiki/Linear_recurrence_with_constant_coefficients`, 2025. Accessed: 2026-01-31.

A | Appendix A: ranges

A.1. Definitions

This part of the section introduces a formal definition of ranges, the associated terminology, and some applicable operators.

Let x be a numerical variable defined in a program. The *range* of x is defined as an interval over the real numbers

$$R(x) \subseteq \mathbb{R},$$

such that

$$R(x) = \{\, v \in \mathbb{R} \mid V_{\min} \leq v \leq V_{\max} \,\}.$$

In the context of static analysis, the definition of range is commonly extended to allow unbounded values. In this case, the range of a variable x is defined as

$$R(x) = [\, V_{\min},\, V_{\max} \,],$$

where

$$V_{\min} \in \mathbb{R} \cup \{-\infty\} \quad \text{and} \quad V_{\max} \in \mathbb{R} \cup \{+\infty\}.$$

This extended definition makes it possible to represent variables whose bounds cannot be precisely determined at compile time.

At this point, it is possible to define two special range states: *Bottom*, denoted by $\bot$, and *Top*, denoted by $\top$. The Bottom element represents an inconsistent or impossible range, typically used as an initial state and which, if preserved without modifications until the end of the analysis, indicates an error condition. Formally, Bottom can be represented by any interval $[V_{\min}, V_{\max}]$ such that $V_{\min} > V_{\max}$. A common choice is

$$\bot = [+\infty, -\infty].$$

Conversely, a range is said to be Top when no sufficient information has been identified

to constrain its bounds, and therefore all possible values must be taken into account. It is represented as

$$\top = [-\infty, +\infty].$$

An intuitive ordering over ranges can also be defined, based on their *width*. The width of a range is computed as

$$W = V_{\max} - V_{\min},$$

and it expresses a notion of cardinality for ranges. Given two ranges r_1 and r_2, r_1 is wider than r_2 if

$$W_{r_1} > W_{r_2}.$$

According to the previously introduced terminology, it is possible to state that

$$\bot \ < \ r \ < \ \top$$

for any range r.

Two or more ranges can be merged through two fundamental operations, namely *join* and *meet*, which are formally defined in the following.

Given two ranges

$$r_1 = [\, V_{\min}^{(1)},\, V_{\max}^{(1)} \,] \quad \text{and} \quad r_2 = [\, V_{\min}^{(2)},\, V_{\max}^{(2)} \,],$$

their *join* operator, denoted by $\sqcup$, computes the least upper bound of the two ranges and is defined as

$$r_1 \sqcup r_2 = \Big[\min\Big(V_{\min}^{(1)}, V_{\min}^{(2)}\Big)\,,\ \max\big(V_{\max}^{(1)}, V_{\max}^{(2)}\big)\Big].$$

The join operation represents the smallest range that safely contains both r_1 and r_2.

Given the same ranges r_1 and r_2, their *meet* operator, denoted by $\sqcap$, computes the greatest lower bound of the two ranges and is defined as

$$r_1 \sqcap r_2 = \Big[\max\Big(V_{\min}^{(1)}, V_{\min}^{(2)}\Big)\,,\ \min\big(V_{\max}^{(1)}, V_{\max}^{(2)}\big)\Big].$$

If the resulting interval satisfies $V_{\min} > V_{\max}$, the result of the meet operation is the Bottom element $\bot$.

In addition to these special operations, it is also possible to apply the classical arithmetic operations to ranges, such as addition, subtraction, multiplication, and division.

The concepts introduced so far, including the definition of ranges and the associated

operations, are grounded in the abstract domain in the classical interval formalized within the abstract interpretation framework by Cousot and Cousot [7].

A.2. Common range operations

The following presents a non-exhaustive list of operations on ranges, selected among the most commonly encountered during the development of this thesis.

Let:

- R denote the resulting range,
- $p = [p_{\min}, p_{\max}]$ denote the range of the first operand,
- $q = [q_{\min}, q_{\max}]$ denote the range of the second operand, when applicable.

Addition

The resulting range of an addition operation is obtained by summing the lower bounds of the two operands to determine the lower bound, and the upper bounds to determine the upper bound:

$$R = p + q = [p_{\min}, p_{\max}] + [q_{\min}, q_{\max}] = [p_{\min} + q_{\min},\ p_{\max} + q_{\max}].$$

Subtraction

In the case of subtraction, the lower bound is given by the difference between the minimum of the minuend and the maximum of the subtrahend, while the upper bound is given by the difference between the maximum of the minuend and the minimum of the subtrahend:

$$R = p - q = [p_{\min}, p_{\max}] - [q_{\min}, q_{\max}] = [p_{\min} - q_{\max},\ p_{\max} - q_{\min}].$$

Multiplication

In the case of multiplication, the resulting range is obtained by considering all possible combinations of the bounds of the operands and selecting the minimum and maximum values among the resulting products. Specifically:

$$R = p \cdot q = [p_{\min}, p_{\max}] \cdot [q_{\min}, q_{\max}] = [\min(S),\ \max(S)],$$

where

$$S = \{\, p_{\min} \cdot q_{\min},\ p_{\min} \cdot q_{\max},\ p_{\max} \cdot q_{\min},\ p_{\max} \cdot q_{\max} \,\}.$$

This definition correctly handles all possible cases, including those in which one or both operands may assume negative values or span across zero, thus guaranteeing the soundness of the resulting interval.

Division

The division operation between two ranges presents additional critical aspects compared to the other arithmetic operations, particularly when the range of the divisor includes the value zero. In such cases, the division is not defined for all possible values, and the result must be handled with special care.

Let:

$$p = [p_{\min}, p_{\max}], \qquad q = [q_{\min}, q_{\max}].$$

If the range of the divisor does not include zero, that is:

$$0 \notin [q_{\min}, q_{\max}],$$

the resulting range of the division is defined as:

$$R = p \div q = [p_{\min}, p_{\max}] \div [q_{\min}, q_{\max}] = [\min(S),\ \max(S)],$$

where

$$S = \{\, p_{\min}/q_{\min},\ p_{\min}/q_{\max},\ p_{\max}/q_{\min},\ p_{\max}/q_{\max} \,\}.$$

This formulation is analogous to the one adopted for multiplication and allows a sound interval to be computed when the divisor is either strictly positive or strictly negative.

Conversely, when the range of the divisor includes the value zero, namely:

$$0 \in [q_{\min}, q_{\max}],$$

the division operation is not continuously defined over the entire interval. Under this condition, it is not possible to determine a finite and sound bound for the result of the division using interval arithmetic alone.

In such cases, a conservative choice consists in returning the range $\top = [-\infty, +\infty]$,

indicating that the result may assume any real value. This approximation preserves the soundness of the analysis at the cost of a loss in precision.

Remainder (Modulo)

Similarly to the division operation, the modulo operation presents a critical case when the range of the divisor crosses zero, since under this condition the operation is not defined over the entire interval and must be handled conservatively to preserve soundness.

Let:

$$p = [p_{\min}, p_{\max}], \qquad q = [q_{\min}, q_{\max}].$$

If the range of the divisor does not include zero, that is:

$$0 \notin [q_{\min}, q_{\max}],$$

a sound over-approximation of the result of the remainder operation can be defined as:

$$R = p \bmod q = [r_{\min}, r_{\max}],$$

where

$$r_{\min} = -(|q_{\max}| - 1), \qquad r_{\max} = |q_{\max}| - 1.$$

This bound follows from the fact that, for any integer divisor $d \neq 0$, the remainder of a division by d is bounded by the magnitude of the divisor itself. The resulting interval is therefore independent of the dividend range and depends solely on the maximum absolute value of the divisor.

When the divisor range includes zero, namely:

$$0 \in [q_{\min}, q_{\max}],$$

the remainder operation is undefined for part of the interval. As a consequence, it is not possible to derive a finite and sound bound for the result using interval arithmetic alone.

In this case, a conservative approximation consists in returning the Top element:

$$\top = [-\infty, +\infty],$$

thus preserving the soundness of the analysis at the cost of reduced precision.

Negation

The negation operation (unary minus) applied to a range consists in inverting the sign of all values belonging to the interval. The resulting range is obtained by swapping and negating the bounds of the operand.

Let:

$$p = [p_{\min}, p_{\max}].$$

The resulting range of the negation is defined as:

$$R = -p = [-p_{\max},\ -p_{\min}].$$

This operation preserves the soundness of the analysis and is particularly useful in handling expressions that involve sign changes, in addition to representing a basic case of a monotonic operation on ranges.

Absolute value

The absolute value operation applied to a range transforms all values in the interval into their corresponding non-negative values. The resulting range depends on the position of the interval with respect to zero and therefore requires a case-based distinction.

Let:

$$p = [p_{\min}, p_{\max}].$$

If the range is entirely non-negative, namely:

$$p_{\min} \geq 0,$$

the absolute value operation does not modify the interval:

$$R = |p| = [p_{\min},\ p_{\max}].$$

If the range is entirely non-positive, namely:

$$p_{\max} \leq 0,$$

the result is obtained by applying negation to the range:

$$R = |p| = [-p_{\max},\ -p_{\min}].$$

Finally, if the range crosses zero, that is:

$$p_{\min} < 0 < p_{\max},$$

the lower bound of the result is equal to zero, while the upper bound is given by the maximum of the absolute values of the two extremes:

$$R = |p| = [0,\ \max(|p_{\min}|,\ |p_{\max}|)]\,.$$

This definition yields a sound interval for the absolute value in all possible cases, preserving the correctness of the analysis even in the presence of ranges that include negative values.

Minimum and maximum

The *minimum* and *maximum* operations between two ranges are commonly encountered in numerical code, for instance in clamping expressions, bound enforcement, and conditional updates. In the context of interval analysis, these operations can be soundly approximated by reasoning on the bounds of the operands.

Let:

$$p = [p_{\min}, p_{\max}], \qquad q = [q_{\min}, q_{\max}].$$

Minimum A sound over-approximation of the minimum between two ranges is obtained by selecting the minimum among the lower bounds and the minimum among the upper bounds:

$$R = \min(p, q) = [\min(p_{\min}, q_{\min})\,,\ \min(p_{\max}, q_{\max})]\,.$$

This interval safely contains all possible values that the minimum operation may assume during program execution.

Maximum Similarly, a sound over-approximation of the maximum between two ranges is obtained by selecting the maximum among the lower bounds and the maximum among the upper bounds:

$$R = \max(p, q) = [\max(p_{\min}, q_{\min})\,,\ \max(p_{\max}, q_{\max})]\,.$$

This definition guarantees that the resulting interval encloses all possible outcomes of the maximum operation.

Both operations preserve the soundness of the analysis, although they may introduce a loss of precision in cases where the ranges of the operands significantly overlap. More precise results could be obtained by exploiting additional relational information between the operands, which is however outside the scope of interval-based abstractions.

Widening and narrowing

In the context of static analysis, iterative propagation of ranges along loops may lead to non-terminating ascending chains. To ensure convergence of the analysis, two complementary operations are commonly employed: *widening* and *narrowing*.

Let:

$$p = [p_{\min}, p_{\max}], \qquad q = [q_{\min}, q_{\max}],$$

where p represents the previously computed range and q the newly inferred one.

Widening The widening operator, denoted by ∇, is used to accelerate convergence by extrapolating the evolution of range bounds. A standard widening for interval domains is defined as:

$$p \nabla q = \left[\begin{cases} p_{\min} & \text{if } q_{\min} \geq p_{\min} \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} p_{\max} & \text{if } q_{\max} \leq p_{\max} \\ +\infty & \text{otherwise} \end{cases} \right].$$

This operation guarantees termination of the fixpoint computation, possibly at the cost of a loss in precision.

Narrowing Once convergence has been reached through widening, a narrowing phase can be applied to refine the over-approximation and recover part of the lost precision. A simple narrowing operator for ranges is defined as:

$$p \triangle q = \left[\begin{cases} q_{\min} & \text{if } p_{\min} = -\infty \\ p_{\min} & \text{otherwise} \end{cases}, \begin{cases} q_{\max} & \text{if } p_{\max} = +\infty \\ p_{\max} & \text{otherwise} \end{cases} \right].$$

The narrowing operation preserves soundness while allowing the analysis to converge to a more precise fixpoint.

Together, widening and narrowing provide a practical mechanism to balance termination and precision in interval-based static analyses, and are widely adopted in abstract interpretation frameworks.

B | Appendix B

For completeness, this appendix reports the full set of numerical results underlying the figures presented in the evaluation chapter 6.

B.1. Targeted Recurrence Patterns

The tables B.1 and B.2 report the results obtained by evaluating all the specific recurrences implemented. All affine recurrences were analyzed starting from an input range between -0.5 and 1, and executing the recurrence for 500 iterations. Geometric recurrences were evaluated using the same input widening strategy, but restricted to strictly positive values in order to focus the analysis on non-oscillatory behaviors. For this reason, an input bound of [1, 2.5] was selected. Due to the exponential nature of geometric recurrences, the number of iterations was limited to 20. The linear recurrence test, characterized by a hybrid nature, was evaluated by setting the coefficient range between 1 and 2 and performing 50 iterations.

Benchmark	Speedup	C. Time(s)	Mean Rel. Err.(%)	Mean Abs. Err.
affine_basic	11.51x	0.584	0.70	6.09e-01
affine_crossing	1.54x	0.580	0.00	5.90e-05
affine_delta	10.85x	0.592	0.03	1.04e+01
affine_flattened_S	8.67x	0.584	0.00	1.09e-04
affine_flattened_T	0.52x	0.658	0.04	1.83e-02
fake_with_affine	10.72x	0.586	0.00	1.93e-04
geo_basic	1.00x	0.606	0.00	1.28e-04
geo_crossing	0.80x	0.594	0.00	1.27e-03
geo_delta	2.50x	0.560	0.00	0.00e+00
geo_flattened_S	0.81x	0.558	0.00	8.00e-07
geo_flattened_T	0.62x	0.432	0.00	2.00e-08
linear_rec	0.80x	0.380	0.00	6.16e+02

Table B.1: Results of the recurrence focused test by using Old VRA

Benchmark	Speedup	C. Time(s)	Mean Rel. Err.(%)	Mean Abs. Err.
affine_basic	10.88x	0.558	0.00	1.59e-06
affine_crossing	1.53x	0.608	0.00	6.03e-05
affine_delta	10.90x	0.598	0.03	1.05e+01
affine_flattened_S	10.77x	0.574	0.00	7.92e-05
affine_flattened_T	0.52x	0.654	0.00	7.21e-05
fake_with_affine	9.73x	0.596	0.00	2.53e-04
geo_basic	0.75x	0.584	2.76	5.58e-01
geo_crossing	0.61x	0.606	0.00	2.00e-06
geo_delta	0.65x	0.572	100.00	9.08e+12
geo_flattened_S	0.67x	0.538	0.00	8.00e-07
geo_flattened_T	0.39x	0.446	0.00	2.00e-08
linear_rec	1.07x	0.404	6.67	2.95e+01

Table B.2: Results of the recurrence focused test by using RA-VRA

B.2. Applicable PolyBench algorithm test

The tables B.3 and B.4 report the results obtained by evaluating all the PolyBench [21] benchmarks compatible with the implemented recurrence patterns. In the Old VRA configuration, the existing annotation-based mechanism was preserved, meaning that range annotations were exploited to constrain value intervals during the analysis. In contrast, with the new RA-VRA implementation, all manual annotations were removed, and range inference was entirely delegated to the proposed algorithm. Both sets of results are computed as the average over 20 independent executions.

Tables B.5, B.6 reports the real ranges and the automatically estimated ones for the most significant input and output variables of each benchmark considered.

Finally B.7, tables reports the recurrences present and correctly recognized within the PolyBench benchmarks considered in this evaluation phase. Cases that are not yet supported have been intentionally excluded from the analysis.

Benchmark	Speedup	C. Time(s)	Mean Rel. Err.(%)	Mean Abs. Err.
covariance	1.07x	1.176	5.99	8.40e+01
gemm	0.43x	1.150	0.00	8.81e-04
gemver	1.00x	1.440	0.00	2.98e-02
gesummv	0.52x	1.000	0.00	2.44e-04
symm	0.75x	1.184	0.05	8.76e-03
syr2k	1.09x	1.116	0.00	2.00e-05
syrk	0.42x	1.034	1.05	3.80e-01
trmm	0.85x	1.032	0.02	5.67e-03
2mm	1.17x	1.452	0.00	3.08e-01
3mm	1.59x	1.644	0.16	1.17e+00
atax	1.45x	1.014	0.00	5.61e-02
bicg	0.54x	1.026	0.00	7.02e-05
doitgen	0.61x	1.186	0.00	2.09e-05
mvt	2.36x	1.058	0.00	6.96e-05
floyd-warshall	1.05x	0.846	0.00	0.00e+00
fdtd-2d	0.52x	1.180	1.28	3.84e-01
heat-3d	1.02x	1.254	0.00	0.00e+00
jacobi-1d	0.86x	0.708	0.00	1.49e-07
jacobi-2d	0.73x	0.860	0.00	6.74e-05
seidel-2d	1.67x	0.782	0.00	7.16e-05

Table B.3: PolyBench results obtained using Old VRA

Benchmark	Speedup	C. Time(s)	Mean Rel. Err.(%)	Mean Abs. Err.
covariance	0.98x	1.144	0.00	1.50e-08
gemm	0.98x	1.160	0.00	1.35e-05
gemver	1.98x	1.418	0.00	8.25e-03
gesummv	0.46x	1.040	0.00	8.07e-05
symm	0.46x	1.224	0.09	1.74e-02
syr2k	1.04x	1.142	0.00	1.22e-05
syrk	1.13x	1.082	0.00	6.24e-06
trmm	0.73x	1.020	0.00	2.20e-05
2mm	2.46x	1.434	0.00	2.60e-03
3mm	2.21x	1.664	0.00	6.49e-04
atax	2.17x	1.006	0.00	1.27e-03
bicg	0.50x	1.044	0.00	2.36e-05
doitgen	1.46x	1.184	0.00	6.18e-07
mvt	2.50x	1.082	0.00	2.34e-05
floyd-warshall	1.06x	0.840	0.00	0.00e+00
fdtd-2d	0.46x	1.214	2.13	7.68e-01
heat-3d	0.97x	1.242	0.00	0.00e+00
jacobi-1d	0.79x	0.714	0.00	2.25e-07
jacobi-2d	0.75x	0.870	0.00	5.22e-05
seidel-2d	1.84x	0.802	0.00	5.10e-05

Table B.4: PolyBench results obtained using New RA-VRA

Benchmark	Real Ranges	Computed Ranges
covariance	mean: [−1.29e2, 2.60e2] data: [0.00, 1.29e2] cov: [0.00, 2.70e4]	mean: [0.00, 6.76e4] data: [−6.72e4, 2.60e2] cov: [−4.57e9, 2.70e5]
gemm	A: [0.00, 9.96e−1] B: [0.00, 9.96e−1] C: [0.00, 1.14e2]	A: [0.00, 9.96e−1] B: [0.00, 9.96e−1] C: [0.00, 3.58e2]
gemver	A: [5.21e−7, 1.23] x: [2.00e−2, 2.15e1] W: [1.21, 7.33e3]	A: [0.00, 1.33] x: [0.00, 8.02e1] W: [0.00, 6.41e4]
gemsum	A: [0.00, 9.96e−1] B: [0.00, 9.96e−1] x: [0.00, 9.96e−1] tmp: [4.98e−1, 8.20e1] y: [1.94, 2.22e2]	A: [0.00, 9.96e−1] B: [0.00, 9.96e−1] x: [0.00, 9.96e−1] tmp: [0.00, 2.48e2] y: [0.00, 6.70e2]
symm	tmp: [0.00, 1.62e1] C: [1.56e1, 2.49e1] A: [−9.99e2, 4.25e−1] B: [0.00, 4.95e−1]	tmp: [−9.89e4, 4.90e1] C: [−1.51e5, 7.52e1] A: [−9.99e2, 4.25e−1] B: [0.00, 4.95e−1]
syr2k	A: [0.00, 9.96e−1] B: [0.00, 9.95e−1] C: [4.30e−2, 1.78e2]	A: [0.00, 9.96e−1] B: [0.00, 9.95e−1] C: [0.00, 7.14e2]
syrk	A: [0.00, 9.96e−1] C: [0.00, 1.19e2]	A: [0.00, 9.96e−1] C: [−2.87e2, 6.45e2]
trmm	A: [0.00, 1.00] B: [0.00, 1.08e2]	A: [0.00, 1.00] B: [0.00, 2.99e2]
2mm	A: [0.00, 9.94e−1] B: [0.00, 9.94e−1] C: [0.00, 9.94e−1] D: [7.32e−1, 9.02e3] tmp: [0.00, 1.15e2]	A: [0.00, 9.94e−1] B: [0.00, 9.94e−1] C: [0.00, 9.94e−1] D: [0.00, 5.89e4] tmp: [0.00, 3.12e2]
3mm	E: [0.00, 7.91] F: [0.00, 8.71] G: [0.00, 1.31e4]	E: [0.00, 7.91] F: [0.00, 8.71] G: [0.00, 1.31e4]

Table B.5: Comparison between real and computed ranges (PolyBench, part I).

Benchmark	Real Ranges	Computed Ranges
atax	X: [1.00, 2.00] A: [0.00, 2.10e−1] tmp: [6.09e1, 7.16e1] y: [2.48e3, 2.74e3]	X: [1.00, 2.00] A: [0.00, 2.10e−1] tmp: [0.00, 1.72e2] y: [0.00, 1.48e4]
bicg	A: [0.00, 9.96e−1] p: [0.00, 9.97e−1] r: [0.00, 9.97e−1] s: [5.13e1, 1.36e2] q: [0.00, 6.43e2]	A: [0.00, 9.96e−1] p: [0.00, 9.97e−1] r: [0.00, 9.97e−1] s: [0.00, 3.88e2] q: [0.00, 3.88e2]
floyd-warshall	path: [0.00, 2.00e3]	path: [1.00, 5.00e5]
fdtd-2d	ex: [−2.47e2, 2.58e3] ey: [−2.99e2, 9.99e2] hz: [−1.22e2, 1.20e3]	ex: [−2.90e4, 2.93e4] ey: [−2.92e4, 2.94e4] hz: [−1.95e7, 1.95e7]
heat-3d	A: [2.50e−1, 2.95e1] B: [2.50e−1, 2.95e1]	A: [−7.88e1, 1.09e2] B: [−2.25e1, 5.25e1]
jacobi-1d	A: [1.00e−3, 1.01] B: [1.00e−3, 1.01]	A: [0.00, 1.01] B: [0.00, 1.01]
jacobi-2d	A: [0.00, 2.50e2] B: [0.00, 2.51e2]	A: [0.00, 2.53e2] B: [0.00, 2.53e2]
seidel-2d	A: [5.00e−3, 4.00e2]	A: [5.00e−3, 4.02e2]

Table B.6: Comparison between real and computed ranges (PolyBench, part II).

Benchmark	% Det. Rec.	Affine	Affine Flat. (S)	Affine Flat. (T)	Fake
covariance	100%	11	1	1	3
gemm	100%	12	0	1	1
gemver	100%	10	2	0	2
gesummv	100%	5	2	0	1
symm	100%	10	1	1	1
syr2k	100%	10	0	1	1
syrk	100%	10	0	1	1
trmm	100%	8	0	1	1
2mm	100%	16	0	2	1
3mm	100%	19	0	3	0
atax	100%	8	2	0	0
bicg	100%	8	2	0	0
doitgen	100%	13	1	0	0
mvt	100%	8	2	0	0
floyd-warshall	100%	7	0	1	0
fdtd-2d	100%	17	0	3	0
heat-3d	100%	4	0	0	0
jacobi-1d	100%	5	0	0	0
jacobi-2d	100%	9	0	0	0
seidel-2d	100%	7	0	0	0

Table B.7: Detected recurrences in PolyBench benchmarks

List of Figures

List of Tables

Ringraziamenti

Il mio più grande grazie va senza dubbio a mia moglie Annagiulia, che in questi anni di studio ha dovuto sopportare tutti i miei momenti di ansia e di stress che mi hanno accompagnato. Ma è lei che mi ha dato la spinta per iniziare ed è principalmente grazie al suo sostegno che non ho mollato e sono riuscito a raggiungere questo traguardo, che all'inizio sembrava solo una montagna impervia da scalare. Ringrazio anche mio papà, per quello che mi ha dato e per le passioni che mi ha trasmesso. Ringrazio mia mamma che purtroppo non potrà essere presente fisicamente alla proclamazione, ma che sono sicuro che da lassù, sarà felicissima di quello che ho conquistato. I miei ringraziamenti vanno anche al prof. Agosta, a Niccolò e Gabriele, che sono stati sempre disponibili e pazienti in questi mesi di lavoro, dove non sono mancati i problemi ed i dubbi. Il ringraziamento finale va a tutti i miei amici con cui sto condividendo questi anni di vita, con loro è ora di fare finalmente festa!