POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Recurrence-Aware Value Range Analysis for Precision Tuning

Laurea Magistrale in Computer Science Engineering - Ingegneria Informatica

Author: Luca Achini

Advisor: Prof. Giovanni Agosta

Co-advisors: Niccolò Nicolosi, Gabriele Magnani

Academic year: 2025-2026

---

1. Introduction

In the context of compilation, *precision tuning* encompasses techniques aimed at improving the performance of numerical computations by reducing precision in a controlled manner. TAFFO [2], an LLVM-based framework, implements this approach by converting high-precision floating-point values, such as `float` and `double`, into fixed-point representations. The trade-off between precision loss and performance gain represents the fundamental evaluation metric.
Since the total number of available bits is limited, it is necessary to determine how many bits must be reserved for the integer part in order to prevent overflow while maximizing fractional precision. In this context, Value Range Analysis (VRA) inspects the program to infer the minimum and maximum values that must be safely represented. Accurate range information therefore enables an efficient fixed-point conversion, although the analysis must be carefully designed to avoid excessive complexity or loss of precision.

1.1. Objectives and motivation

The current VRA module integrated in TAFFO adopts a lattice-based approach, in which the abstract domain is represented by intervals associated with program instructions [4]. The initial lattice state is determined through the analysis of constants, variable initializations, or programmer-provided annotations, and ranges are propagated using interval arithmetic.
The main limitation emerges in the presence of loops: the current implementation requires analyzing all expected iterations, leading to excessive compilation times or even non-termination when the trip count cannot be statically determined. To prevent such behavior, the analysis is interrupted and explicit annotations are used to specify final ranges. However, this approach forces the programmer to manually reconstruct the ranges of variables involved in loops, transferring part of the analytical burden to the developer and reducing the overall level of automation.
This limitation motivates the need for a method capable of inferring the final ranges of loops without executing all iterations and without relying on manual configuration.
The proposed method must therefore guarantee:

1. **Soundness**: the computed ranges must conservatively include all possible values.
2. **Termination**: the algorithm must reach a fixed point in the lattice domain.
3. Acceptable compilation times.

4. Scalability and adaptability to heterogeneous usage scenarios.

These objectives are pursued and satisfied through the introduction of a *Recurrence-Aware Value Range Analysis* (RA-VRA), an extension of the traditional VRA that explicitly accounts for iterative constructs.
The relevant state of the art is discussed in Section 2. Section 3 presents the methodological framework and describes the RA-VRA algorithm. The experimental evaluation is reported in Section 4. Finally, Section 5 concludes the work and outlines possible future developments.

2. State of the art

From the analysis of the state of the art [3], range analysis techniques can be classified into two main approaches: static and dynamic. The former infers bound information entirely at compile time, without requiring program execution. The latter, instead, relies on observing the program behavior during one or more executions, using the collected data to estimate or progressively refine the computed ranges. Given the general-purpose nature of TAFFO and the need to guarantee portability, determinism, and independence from input datasets, the static approach was selected. This choice ensures soundness and reproducibility of the analysis.
The lattice-based approach already adopted in the current version was considered fundamental to guarantee both termination and soundness of the analysis.
The abstract domain D consists of the set of intervals associated with each program instruction. Each interval is defined as:

$$R = [r_{\min}, r_{\max}] \subseteq \mathbb{R}$$

A complete configuration of all computed ranges represents a lattice state S.
Given two intervals $R_1 = [a_1, b_1]$ and $R_2 = [a_2, b_2]$, the fundamental domain operations are:

$$R_1 \sqcup R_2 = [\min(a_1, a_2), \max(b_1, b_2)]$$

$$R_1 \sqcap R_2 = [\max(a_1, a_2), \min(b_1, b_2)]$$

The partial order of the lattice is defined by interval inclusion:

$$R_1 \sqsubseteq R_2 \iff a_2 \leq a_1 \wedge b_1 \leq b_2$$

An interval is therefore more precise when it is contained within another.
The domain admits a minimum element $\bot$ (empty interval) and a maximum element $\top$ (unbounded interval).
Since the interval domain admits infinite ascending chains (e.g., $[0,1] \sqsubseteq [0,2] \sqsubseteq [0,3] \sqsubseteq \ldots$), a purely monotonic fix-point iteration does not guarantee termination. To ensure convergence, a widening operator ∇ is introduced.
The application of widening allows the fix-point to be reached without explicitly iterating all loop executions. This is particularly relevant when the number of iterations is unknown at compile time or when it is very large.
In practice, only a limited number of loop iterations are analyzed: if a monotonic growth of the bounds is detected, widening forces convergence by setting the lower bound to $-\infty$ or the upper bound to $+\infty$. This guarantees termination of the analysis while preserving soundness, since all concrete values remain included in the computed interval.
However, applying widening indiscriminately often leads to excessive over-approximation, especially in iterative constructs. To mitigate this effect, principles derived from Scalar Evolution [1] were considered, which analyze the behavior of variables updated recurrently inside loops.
By explicitly modeling structured evolution patterns, the application of widening can be restricted to cases where no precise characterization of the variable growth is available. In this sense, the analysis adopts a form of controlled widening: acceleration toward $\top$ is not triggered at the first sign of monotonic growth, but only when a more precise description of the variable evolution cannot be derived.
This approach preserves both termination and soundness, while significantly improving the precision of the inferred ranges.

3. Methodology

3.1. Recurrences

The adopted method aims at identifying and extracting predictable behaviors of variables influenced by the presence of loops, in order to determine the range assumed at the end of the final iteration without fully simulating the entire iterative construct.

Such behaviors are modeled as *recurrences*, namely relations describing the evolution of a variable as a function of the iteration index.
Among the analyzed typologies, the two primary classes identified are:

- **Affine recurrences**, characterized by linear growth with respect to the number of iterations;
- **Geometric recurrences**, characterized by exponential growth when the ratio is strictly positive, or by oscillatory behavior when the ratio includes negative values.

Each recurrence is defined by the following fundamental elements:

- an initial range r_0, representing the interval assumed by the variable before entering the loop;
- a set of parameters governing the evolution of the range;
- an algebraic function $at(k)$, applied regularly at each iteration k, which describes the variable update and is deterministically computable as a function of the iteration index.

The example shown in listing 1 illustrates a loop with N iterations containing (a) an affine recurrence and (b) a geometric recurrence.

```
float a = 0.0f, b = 1.0f;
while (i < N) {
    a = a + 3;     // (a) affine rec.
    b = b * 1.5f; // (b) geometric rec.
    i++;
}
```

Listing 1: Example of recurrences

In the presence of monotonic behavior with respect to the iteration index k, and assuming the trip count (TC), i.e., the maximum number of loop iterations, is known, the final range can be determined by evaluating the function $at(k)$ at the endpoints of the iteration interval. In particular:

$$R = [\min(at(0), at(TC)), \max(at(0), at(TC))]$$

In the case of non-monotonic behaviors, as may occur for certain geometric recurrences, it is instead necessary to explicitly determine the minimum and maximum values assumed by the function $at(k)$ over the discrete interval $k \in [0, TC]$.

In such situations, the final range is computed as:

$$R = \left[\min_{0 \le k \le TC} at(k), \max_{0 \le k \le TC} at(k)\right]$$

The bounds are therefore determined according to the analytical nature of the recurrence, while still avoiding the explicit simulation of all loop iterations.
Finally, in addition to the knowledge of the trip count, possible dependencies among recurrences were taken into account. In Listing 2, it can be observed that the affine recurrence in *Loop*2 cannot be computed before determining the final value of x at the end of *Loop*1. This implies that recurrence resolution must respect their interdependencies, requiring a consistent ordering during the analysis process.

```
int x = 0;
for (int i = 0; i < N; i++){ // Loop 1
  x = x + 3; //Affine: start=0, incr=+3
}
for (int j = 0; j < M; j++) { // Loop 2
  x = x + 2; //Affine: start=?, incr=+2
}
```

Listing 2: Two recurrences dependency example

3.2. RA-VRA algorithmic workflow

The RA-VRA procedure is formalized in Algorithm 1.

Algorithm 1 RA-VRA High-Level Workflow

Require: Program P in SSA form with loops in canonical form
Ensure: Final range mapping $\mathcal{R}$

```
PRESEED(P)
INSPECT(P)
while true do
  solvedRR ← 0
  solvedTC ← 0
  ASSEMBLERRS(P, by ref solvedRR)
  COMPUTETCS(P, by ref solvedTC)
  PROPAGATE(P)
  if solvedRR + solvedTC = 0 then
    FALLBACKREMAININGS(P)
    PROPAGATE(P)
    break
  end if
end while
COLLECTRESULTS(P)
```

A necessary precondition for the application of RA-VRA is that the program is represented in SSA form and that loops are expressed in canonical form. The SSA representation simplifies the analysis of variable dependencies, while canonical loop form enables the unambiguous identification of induction variables and termination conditions, thereby facilitating trip count computation.
At convergence, RA-VRA produces a sound mapping of value ranges for each relevant program instruction. These ranges are subsequently collected and provided as input to the following phases of the TAFFO precision tuning process.
The implementation was carried out within the LLVM framework, although the method is not inherently tied to this infrastructure. LLVM provides dedicated passes to enforce the preconditioned structural properties, which motivates its adoption for the implementation.

RA-VRA structures the analysis into five main phases: *preseed*, *inspect*, *assemble*, *trip count computation*, and *propagate*. These phases are complemented by a *fallback* mechanism, activated for unresolved cases, and by a final stage dedicated to collecting the computed ranges.

Preseed The *Preseed* phase performs two main tasks: it initializes the lattice state S_0 using constant values and programmer annotations, and it determines a consistent analysis order. A correct instruction order is required both to ensure sound range propagation through interval operations and to construct recurrences according to execution semantics.
In the LLVM Intermediate Representation (LLVM-IR), instructions are organized into *BasicBlocks*, which preserve intra-block program order and reflect control-flow structure. Blocks are scheduled according to dominance relations, ensuring that each block is processed only after its dominators. The resulting schedule is stored and reused during propagation to avoid redundant structural analysis.

Inspection The *Inspection* phase identifies potential recurrence roots by constructing a dependency graph derived from def-use relations among instructions and detecting cycles within it. Instructions belonging to such cycles are stored for subsequent assembly and resolution phases.
Within the LLVM-IR context, the recurrence roots observed in practice fall into two primary categories:

- *PHI* nodes located at loop headers, modeling iterative scalar updates;
- *Store* instructions, when the stored value depends on a *load* operation from the same memory base.

Assembly The *Assembly* phase iteratively classifies and constructs detected recurrences once their dependencies are resolved.
Each recurrence is modeled as a tree rooted at the recurring variable. Classification is performed by analyzing the main path, defined as the shortest path from the root to its terminal node: for *PHI* roots, this corresponds to the back-edge value; for *Store* roots, to the *load* from the same memory base. The operators along this path determine the recurrence class (e.g., affine or geometric).
Secondary paths are inspected to detect unresolved dependencies and to extract the ranges required to compute the recurrence parameters. Recurrences are finalized only when all dependencies are satisfied; otherwise, they are either discarded or reconsidered in subsequent iterations.

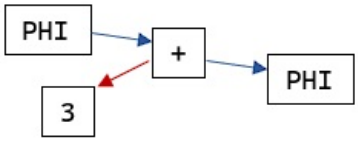

Figure 1: Tree representation of the affine recurrence in Loop 1 of Listing 2. The main path is highlighted in blue, while secondary paths are shown in red.

Trip count computation The *Trip count computation* phase determines the number of loop iterations for loops in canonical form. The analysis starts from the branch instruction controlling loop exit, traces back to the corresponding comparison, and identifies the induction variable together with the loop-invariant operand. Whenever possible, the trip count is derived directly from the characteristic function

of the recurrence associated with the induction variable.
The computation may be postponed if one of the operands depends on recurrences that have not yet been resolved. In cases where the trip count cannot be determined statically, the previous VRA-based approach supported by explicit annotations is adopted.

Propagation The *Propagation* phase reanalyzes instruction blocks according to the order defined in *Preseed*, applying interval arithmetic and evaluating resolved recurrences through their characteristic function at the computed trip count. Loop blocks are analyzed only once.
If at least one new recurrence is resolved during an iteration, the algorithm returns to the *Assembly* phase. Termination occurs either when all recurrences are resolved or when no further recurrences can be resolved.
In the latter case, the *fallback* mechanism is activated: remaining recurrences are discarded or conservatively handled (e.g., by relying on existing annotations or applying aggressive widening), followed by a final propagation step to preserve soundness.
At completion, the resulting ranges constitute the final lattice state S_f, which is then provided to the subsequent precision tuning phase for numerical type conversion.

3.3. Complexity Analysis

To ensure a bounded compilation-time overhead, the computational complexity of RA-VRA is analyzed.
Let I denote the number of program instructions and R the number of detected recurrences, with $R \leq I$. The main phases of the algorithm require a traversal of the instruction set ($O(I)$), while recurrence assembly scales with the number of detected recurrences ($O(R)$).
The overall complexity is therefore:

$$O(I + R)$$

and, in the worst case, linear in the number of instructions:

$$O(I)$$

RA-VRA thus preserves compilation times proportional to the size of the analyzed program.

4. Experimental evaluation

The experimental evaluation is structured into three main stages.
The first stage concerns the verification of the soundness of the proposed recurrence model. For each recurrence type, the characteristic function is validated by comparing the range obtained through explicit loop execution with the one derived from the corresponding closed-form expression. In no case does the dynamic range exceed the analytically computed one, confirming the correctness and soundness of the model.
The second stage analyzes the algorithms ability to recognize, construct, and resolve recurrences, comparing its performance with the previous VRA implementation. The considered metrics include compilation time, performance speed-up, mean relative conversion error, and median absolute error, the latter adopted to mitigate the influence of extreme cases. All recurrences are correctly classified. Compilation time and achieved speed-up remain comparable to the previous version, while the average absolute error is reduced by approximately one order of magnitude. Stress tests, obtained by progressively widening the input bounds, highlight the stability of affine recurrences even under large initial ranges, whereas geometric recurrences exhibit a rapid degradation in numerical quality due to exponential growth and interval saturation.

Table 1: Average results on affine recurrences

Metric	Old VRA	RA-VRA
Speed-up	7.30x	**7.39x**
Comp. Time (s)	**0.597**	0.599
Mean Rel. Err. (%)	0.128	**0.005**
Median Abs. Err.	9.20e-04	**6.90e-05**

The third stage extends the analysis to the PolyBench suite, comparing the annotation-based approach with the fully automatic RA-VRA version. In addition to the previously adopted metrics, the *Normalized Bound Error (NBE)* is introduced to quantify the normalized deviation between computed bounds $C = [c_{\min}, c_{\max}]$ and real bounds $R = [r_{\min}, r_{\max}]$:

$$\text{NBE} = \frac{|r_{\min} - c_{\min}| + |r_{\max} - c_{\max}|}{r_{\max} - r_{\min}}.$$

NBE is evaluated exclusively for the RA-VRA version. The previous implementation relies on manually annotated ranges, which do not result from an inference procedure and therefore do not provide a meaningful basis for assessing bound accuracy. Since annotated bounds may be arbitrarily tight or conservative, a comparison through NBE would primarily reflect the quality of manual specifications rather than the effectiveness of the analysis itself. Results into

Table 2: PolyBench aggregated results

Metric	Old VRA	RA-VRA
Speed-up	0.99x	**1.25x**
Comp. Time (s)	**1.107**	1.116
Mean Rel. Err. (%)	0.428	**0.111**
Median Abs. Err.	5.06e-04	**2.35e-05**

Table 2 show comparable speed-up and compilation time between the two VRA versions, with a reduction in absolute error in the automatic approach.

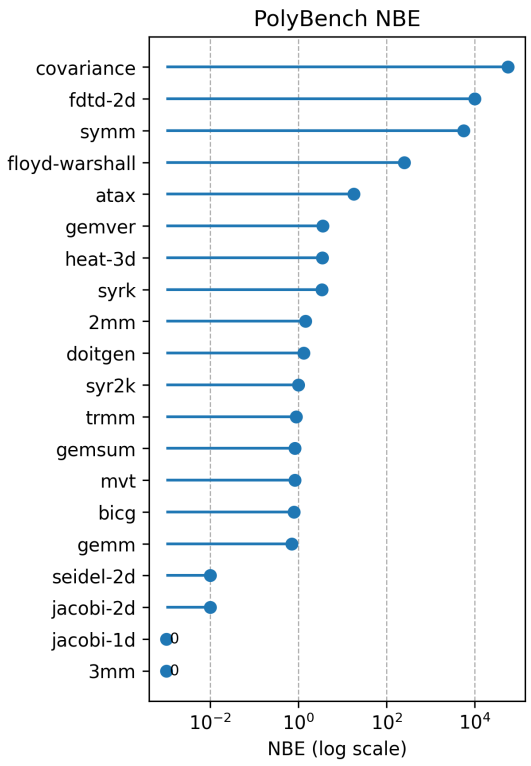

Figure 2: NBE for PolyBench benchmarks using RA-VRA (log scale).

Higher NBE (Figure 2) values reflect conservative over-approximation consistent with the soundness requirement, which assumes the accumulation of extremal values even when such scenarios are unlikely in concrete executions. A structural limitation emerges in the presence of divisions whose divisor range includes zero, which generate excessively wide intervals under classical interval arithmetic.

5. Conclusions

The proposed method enables the extraction of range information by analyzing recurrences without fully iterating loops, significantly reducing the need for manual annotations.
It is a scalable, modular, and extensible approach that preserves soundness as its guiding principle, even at the cost of more conservative bounds. Its structure allows the analysis to evolve without altering its theoretical foundations.
It therefore provides a solid basis for future developments, either toward improving precision through additional semantic information or toward integrating probabilistic techniques that accept a controlled relaxation of soundness in exchange for tighter estimates and explicitly quantified risk.

References

[1] Javed Absar. Scalar evolution demystified. EuroLLVM Developers Meeting, 2018. Presentation and video.

[2] Daniele Cattaneo, Michele Chiari, Giovanni Agosta, and Stefano Cherubin. TAFFO: The compiler-based precision tuner. *SoftwareX*, 20:101238, 2022.

[3] Stefano Cherubin and Giovanni Agosta. Tools for reduced precision computation: A survey. *ACM Computing Surveys*, 53(2), April 2020.

[4] Patrick Cousot and Radhia 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.