Fast Recognition of Scalar Evolutions on Three-Address SSA Code

Sebastian Pop [1] Philippe Clauss [3] Albert Cohen [2] Vincent Loechner [3]
Georges-André Silber [1]

[1] Centre de recherche en informatique, École des mines de Paris, Fontainebleau, France
[2] ALCHEMY group, INRIA Futurs, Orsay, France
[3] ICPS - LSIIT, Université Louis Pasteur, Strasbourg, France

CRI/ENSMP Research Report
A/354/CRI
April 1, 2004

Abstract

This paper presents the design and implementation of a scalar evolutions analyzer for a low level representation of imperative programs. This representation is a three address code in static single assignment form, enriched with type information. The heart of our analysis is a new algorithm recognizing scalar evolutions, represented by closed form expressions called trees of recurrences. These symbolic expressions generalize linear and polynomial evolutions. Unlike previous work, our algorithm tracks induction variables without prior classification of a few evolution patterns. The computation of all scalar evolutions in a procedure is achieved through a bidirectional (non-iterative) traversal of the control-flow graph. The low complexity and abilitity to work on a low-level representation with an intricate SSA graph fit the constraints of a production compiler, even in a dynamic or iterative optimization framework. This analysis is implemented in a development branch (4.0) of GCC. We evaluate this implementation on standard benchmark programs.

1 Introduction and Motivation

Most optimizing compilers implement architecture-independent optimizations to produce efficient code for high-level languages. They also provide architecture-aware mechanisms and models to select machine instructions and tune code generation parameters for a particular target. In addition, compilers for high-performance or embedded architectures must exploit several kinds of on-chip parallelism, including vector/SIMD, instruction level and thread level. They also face increasingly complex hardware components, from the memory subsystem (cache hierarchies, load/store queue speculation, write buffers, miss address files, hardware prefetching), to the dynamic structures of deeply pipelined microarchitectures (predictors, trace cache, replay mechanisms).

Supercomputing research has produced a wealth of techniques to optimize a program and generate efficient code, both for uniprocessor and multiprocessor performance. But the context is different when dealing with common retargetable compilers for general-purpose and/or embedded architectures: the automatic exploitation of fine-grain parallelism and the tuning for dynamic hardware components become far more challenging. We recall three important goals that require a stronger attention when designing a general-purpose optimizing compiler.

Flexibility. The optimizer must cope with a wide spectrum of programming languages and application codes, with general control and memory addressing structures, including unstructured control-flow.

Robustness. Time and memory constraints are stringent, with programs larger than scientific codes in general. In addition, the software industry does not easily trade extra performance for extra compilation time, and will not use an optimization flag that incurs more bugs than the rest of the compiler.

Autonomy. Although expert users may collaborate with the compiler by tuning optimization switches, following programming guidelines, or inserting pragmas, such level of effort is not expected from the vast majority of programmers. This reduces the applicability of simple, pattern-based transformations, and advocate for general methods to discover the legality and profitability of optimizations.

Modern compilers implement some of the sophisticated optimizations introduced for supercomputing applications [4]. They provide performance models and transformations to improve fine-grain parallelism and exploit the memory hierarchy. Most of these optimizations are loop-oriented and assume a high-level code representation with rich control and data structures: `do` loops with regular control, constant bounds and strides, typed arrays with linear subscripts. Yet these compilers are architecture-specific and designed by processor vendors, e.g., IBM, SGI, HP, Compaq and Intel. In addition, the most advanced optimizations are limited to Fortran and C, and performance is very dependent on the recognition of specific patterns in the source code. Some source-to-source compilers implement advanced loop transformations driven by architecture models and profiling [19]. However, good optimizations require manual efforts in the syntactic presentation of loops and array subscripts (avoiding, e.g., `while` loops, imperfect nests, linearized subscripts, pointers). In addition, the source-to-source approach is not suitable for low-level optimizations, including vectorization and software pipelining. It is also associated with pattern-based strategies that do not easily adapt to syntactic variations in the programs. Finally, they are not language-independent, and they require a huge implementation effort that cannot be matched by small development teams for general-purpose and/or free software compilers.

This paper presents a symbolic representation and an efficient algorithm for the recognition of scalar evolutions. This framework is applied to dependence analysis on low-level three-address code. Our algorithm is more effective in retrieving precise closed form expressions than state-of-the-art induction-variable recognition methods. It is a key component of a wider effort to bring advanced loop optimizations to the GNU Compiler Collection (GCC), in an attempt to build a modern optimizing compiler that fulfills the three above goals.

1.1 Loop Transformations on a Low-Level Representation

Implementing complex compiler passes on a high-level representation seems natural, because the source code exposes a richer semantics; it also (arguably) reflects more information about the design and algorithms chosen by the programmer. However, source-level analyzes and transformations require a lot of efforts to achieve comparable robustness, speed and maintainability as the rest of the compiler. The added complexity is mostly due to the lack of normalization of the control and data structures.

Driving program transformations on a high-level representation is also less appropriate when precise architecture models come into play: the source program is very far from the machine code for which such models are designed. For example, branch predictors or instruction caches may not be accurately modeled at the source level where the number and the layout of assembly instructions is unknown. Lower level code has fewer constructs and simpler semantics, control and memory access structures are normalized to a few basic primitives, and language independence is easier to achieve. Also, analyzes and transformations on low level code prove to be very reliable in practice. However, a very low-level representation may not suit the requirements of advanced analyzes and transformations. For example, the Register Transfer Language (RTL) in the GNU Compiler Collection (GCC) uses concrete register names and sub-word operations that dramatically reduce the accuracy of data-flow analyzes.

Independently, several works demonstrated the interest of enriched low-level representation [5, 21]. They build on the normalization and simplicity of three address code, adding data types, *Static Single-Assignment* form (SSA) [9, 25] to ease data-flow analysis and scalar optimizations, and control and data annotations (loop nesting, heap structure, etc.). Recently, the tree-ssa branch of GCC was built on such a representation, GIMPLE [26, 24], a three-address code derived from SIMPLE, the representation of the McCAT compiler [16]. Although these representations were not designed for loop transformations, the availability of abstract type and control-flow information triggered our interest.

In a three-address representation like GIMPLE, subscript expressions, loop bounds and strides are spread across a number of instructions and basic blocks, possibly far away from the original location in the source code. This is due to the lowering itself and to intermediate optimization phases, such as dead-code elimination, partial redundancy elimination, optimization of the control flow, invariant code motion, etc. The most popular techniques to retrieve scalar evolutions [34, 13, 27] are not well suited to work on such loosely structured loops because they rely on classification schemes into a set of predefined forms, often based on pattern-matching rules. Such rules are sometimes sufficient at the source level (for numerical codes), but *too restrictive to cover the wide variability of inductive schemes induced by scalar and control-flow optimizations on a low-level representation.*

```
j = 3;
n = 123;
for (i=1; i<n; i+=5) {
  j += i + 7;
  A[j] = i;
}
```

$\longrightarrow$

```
i = 1;
j = 3;
n = 123;
loop (ℓ1)
  if (i>=n) goto end;
  t = i + 7;
  j = j + t;
  A[j] = i;
  i = i + 5;
end:
```

Figure 1: Lowering to three-address code

Figure 1 illustrates this increased variability and complexity. Looking at the original loop in the left-hand side, variable `j` is inductively updated with a non-negative value, meaning it is a strictly increasing function of the loop counter `i`. Indeed, `i` takes the values $1, 6, 11, \ldots, 126$ and `j` is successively evaluated to $3, 11, 24, \ldots, 1703$. This observation is sufficient to prove there are no dependences on array `A`. The evolution of `i` and `j` may also be characterized as closed form expressions, parameterized by the number of iterations ℓ_1 of the enclosing loop.

$$\begin{aligned} \mathtt{i} &= 5\ell_1 + 1, \\ \mathtt{j} &= \frac{5}{2}{\ell_1}^2 + \frac{11}{2}\ell_1 + 3. \end{aligned}$$

Although the evolution of `j` is not an affine (or linear) expression, several analyzes would retrieve enough information (at the source-code level) to detect the absence of dependence [34, 13, 15, 35, 32, 31].

Lowering the loop to three-address code makes the inductive definitions more intricate, as shown on the right-hand side of Figure 1. Pattern-matching rules for induction variable classification [33] would fail on this example where the inductive scheme involves 3 scalar variables and 5 definitions (`n` is not read in any inductive definition). The method by Wolfe et al. [34, 13, 22] (implemented in the SGI compiler and Open64/ORC) is not pattern-based but belongs to the classification category as well; it would succeed in recognizing a polynomial expression for variable `j` but require a matrix inversion with symbolic coefficients, an expensive formal computation that does not always yield a result. Some sophisticated analyzes with unified closed form algebra [15, 32, 31, 29] would succeed as well, but their symbolic differencing or substitution strategies have a higher overhead. Finally, the first technique by Wu et al. [36] would fail due to the multiple intermediate assignments in the evolution path associated with the inductive definition of variable `j`; yet the second one [35] would prove that `t` is non-negative and succeed in proving the absence of dependences on `A`, although it would not yield a closed form expression for `j`.

To address the challenges of induction variable recognition on a low-level representation, we designed a general and flexible algorithm to build closed form expressions for scalar evolutions. *This algorithm can retrieve the array subscripts, loop bounds and strides lost in the lowering process to three-address code and intermediate optimization phases, as well as many other scalar evolution properties that did not explicitly appear in the source code.* We claim that enriched three-address code representations like GIMPLE have a high potential for the design and implementation of analyzes of abstract program properties. We demonstrate that induction-variable recognition and dependence analysis can be effectively implemented at such a low level. We also show that our method is more flexible and robust than comparable solutions on high-level code, since *it retrieves precise dependence information without restrictions on the complexity of the flow of control and recursive scalar definitions*. In particular, unlike [13, 32, 31], our method captures affine and polynomial closed forms without restrictions on the number and intrication of ϕ nodes. We may also recognize closed forms in the presence of *irreducible control-flow* within loop nests. Finally, speed, robustness of the implementation and language-independence are natural benefits of using a low-level static single assignment representation.

1.2 Introductory Examples

We recall some SSA terminology to facilitate further discussions, see [9, 25] for details: the *SSA graph* is the graph of def-use chains in the SSA representation; ϕ nodes (or functions) occur at merge points and restore the flow of values from the renamed variables; the ϕ arguments are variables listed in the order of the associated control-flow edges; ϕ nodes are split into the *loop-ϕ* — nodes whose second argument correspond to a back-edge in the control-flow graph — and *condition-ϕ* categories.

```
a = 3;
b = 1;
loop (ℓ1)
  c = φ(a, f);
  d = φ(b, g);
  if (d>=123) goto end;
  e = d + 7;
  f = e + c;
  g = d + 5;
end:
```

$$\begin{aligned} \texttt{d} &= 1, 6, 11, \ldots, 126 \\ &= 5\ell_1 + 1 \\ \texttt{f} &= 3, 11, 24, \ldots, 1703 \\ &= \frac{5}{2}{\ell_1}^2 + \frac{11}{2}\ell_1 + 3 \end{aligned}$$

Figure 2: First example

```
a = 3;
loop (ℓ1)
  c = φ(a, x);
  loop (ℓ2)
    d = φ(c, e);
    e = d + 1;
    t = d - c;
    if (t>=9) goto end2;
  end2:
  x = e + 3;
  if (x>=123) goto end1;
end1:
```

$$\begin{aligned} \texttt{c} &= 3, 17, 31, \ldots, 115 \\ &= 14\ell_1 + 3 \\ \texttt{x} &= 17, 31, \ldots, 129 \\ &= 14\ell_1 + 17 \\ \texttt{d} &= 3, 4, 5, \ldots, 13, 17, 18, 19, \ldots, 129 \\ &= 14\ell_1 + \ell_2 + 3 \end{aligned}$$

Figure 3: Second example

```
loop (ℓ1)
  a = φ(1, b);
  if (a>=100) goto end1;
  b = a + 4;
  loop (ℓ2)
    c = φ(a, e);
    e = φ(b, f);
    if (e>=100) goto end2;
    f = e + 6;
  end2:
end1:
```

$$\begin{aligned} \texttt{a} &= 1, 5, 9, \ldots, 101 \\ &= 4\ell_1 + 1 \\ \texttt{e} &= 5, 11, 17, \ldots, 95, 101, 9, 15, 21, \ldots, 95, 101 \\ &= 6\ell_2 + 4\ell_1 + 5 \\ \texttt{c} &= 1, 5, 11, \ldots, 89, 95, 5, 9, 15, \ldots, 89, 95 \end{aligned}$$

Figure 4: Third example

Throughout this paper, we will use a "generic" three-address code in SSA form [9]. Its syntax is simplified from the GIMPLE representation [24]. We consider only two control-flow primitives: a conditional expression

```
loop (ℓ1)
  a = φ(0, d);
  b = φ(0, c);
  if (a>=100) goto end;
  c = a + 1;
  d = b + 1;
end:
```

$$\begin{aligned} \texttt{a} &= 0, 1, 2, \ldots, 100 \\ &= \ell_1 \\ \texttt{b} &= 0, 1, 2, \ldots, 100 \\ &= \ell_1 \end{aligned}$$

Figure 5: Fourth example

`if`, and a goto expression `goto`. In addition, we include a loop annotation discovered from the control-flow graph [2]: *loop* (ℓ_k) is the annotation for loop number k, and ℓ_k denotes the implicit counter associated with this loop (loop numbers are unique). Loop annotations do not hold any information on the bounds, and ℓ_k counters do not correspond to any concrete variable in the program. Thanks to a loop restructuring pass, we assume that every loop exits on a single `goto` controlled by a conditional expression. Although the number of iterations is not captured by the representation, it is often discovered from the scalar evolution of variables involved in the loop exit conditions; this is allows to recompute precise information lost in the translation to a low-level representation, but it may also be useful when the source level does not expose enough syntactic information, e.g., in `while` loops. The rest of the representation is roughly equivalent to C, but we only recognize the `+` operators in scalar updates (other operations yield an unknown evolution represented by $\top$).

To illustrate the main issues and concepts, we consider the two examples in Figures 2 and 3. The first program corresponds to the loop introduced in Figure 1 (omitting the array reference), and the second one illustrates scalar evolutions across nested loops.

The right-hand side of Figures 2 and 3 describe the evolution of the main inductive variables in each program. The closed-form expression for `f` in Figure 2 is not affine (a second-degree polynomial). In Figure 3, apart from `d`, all variables described on the right-hand side are *univariate*: they only depend on one loop counter; `d` is called *multivariate*. To compute the evolution of `c`, `x` and `d` in Figure 3, one must know the *trip count* of the inner loop, here 10 iterations. Yet, to statically evaluate the trip count of loop 2, one must already understand the evolutions of `c` and `d`. Up to our knowledge, our method is the first to succeed in recovering the loop-trip count in this case, hence to compute the closed forms for variables `c`, `x` and `e`.

Figure 4 presents a third example. Variables `a` and `e` have simple affine evolutions, but `c` is a typical case of *wrap-around* variable [34], a variable that takes a special value at the first iteration and becomes an inductive variable with a regular evolution in the further iterations. Such variables are defined by loop-ϕ nodes appearing in strongly-connected components of the SSA graph that hold *other* loop-ϕ nodes. The classification by Wolfe et al. does not handle wrap-around variables in nested loops where the special initial value is not invariant by the outer loops. Indeed, the closed form representation for such *multivariate* variables are more complex than the affine, polynomial or exponential cases considered in [34, 13]. Here the value of `c` is reinitialized to a different special value at each iteration of the outer loop: for $\ell_1 = 0$ its values in the inner loop are $1, 5, 11, 17$, then for $\ell_1 = 1$ its values are $5, 9, 15, 21, \ldots$, etc. We will show the closed form of `c` in Section 2, after the presentation of a representation that captures such evolutions. Once again on this third example, our algorithm is the first to compute affine closed forms for all scalar variables. Interestingly, we will also show that the complexity of the computation is no higher than for the second example.

Some complex SSA graphs with multiple loop-ϕ nodes in a cycle do not match the wrap-around class. They define more general induction variables that include the *periodic* or *exponential* classes [34], or may not even have a known closed form.[1] Figure 5 presents a fourth example where `a` and `b` have linear closed forms but derive from an intricate inductive definition scheme. We will not address the exponential and periodic cases of induction variables, because we do not have any application for these expressions at the moment, and because few programs exhibit such evolutions.

[1] A complete algebra does not exist for the Turing-complete computation model of scalar evolutions.

1.3 Overview of the Paper

In the following, we expose a set of techniques to extract and to represent evolutions of scalar variables in the presence of complex control flow and intricate inductive definitions. We focus on designing low-complexity algorithms that do not sacrifice on the effectiveness of retrieving precise scalar evolutions, using a typed, low-level, SSA-form representation of the program.

Our framework is very effective in computing closed forms for induction variables: 83.2% of the dependences were successfully resolved on the SpecFP 2000 benchmarks [30], 62.5% on SpecINT 2000 [30] and 99.2% on JavaGrande [18]. It is also scalable since the total computation time for scalar evolution and dependence analysis never exceeds 10% percent of the total compilation time.

Section 2 introduces the algebraic structure that we use to capture a wide spectrum of scalar evolution functions. Section 3 presents the analysis algorithm to extract closed form expressions for scalar evolutions. Section 4 integrates our method in a dependence analysis framework. Section 5 compares our method to other existing approaches. Finally, section 6 concludes and sketch future work.

2 Trees of Recurrences

In this section, we introduce the notion of *Tree of Recurrences* (TREC), a closed-form that captures the evolution of induction variables as a function of iteration indices and allows an efficient computation of values at given iteration points. This formalism is a slightly modified and extended version of *Multivariate* (a.k.a. multidimensional) *Chains of Recurrences* (MCR) [6, 20, 37, 32], with equivalent expressive power but more appropriate to implement algebraic operations on scalar evolutions.

Let $F(\ell_1, \ell_2, \ldots, \ell_m)$ — or $F(\vec{\ell})$ for short — represent the evolution of a variable inside a loop of depth m as a function of $\ell_1, \ell_2, \ldots, \ell_m$, the integer-valued loop indices (numbered from 0).

We wish to convert F into a closed form Θ, called TREC, that can be statically processed by further analyzes and evaluated efficiently at compile-time for a given $\vec{\ell}$. Informally, the TREC corresponding to F is a binary tree of unbounded depth, where each node is a constant or a TREC. The evaluation of a TREC for a given $\vec{\ell}$ will be defined in Section 2.1. The syntax of a TREC is derived from MCR and inductively defined as:

$$\Theta = \{\Theta_a,\, +,\, \Theta_b\}_k \quad \text{or} \quad \Theta = c,$$

where Θ_a and Θ_b are trees of recurrences and c is a integer constant. Subscript k is called the *dimension* of the TREC. When recognizing scalar evolutions in nested loops, k denotes a loop number. Unlike MCR [20], recurrences can be nested without constraints on their dimension k, but this makes no difference on expressiveness.

As a form of syntactic sugar, $\{\Theta_a, +, \{\Theta_b, +, \Theta_c\}_k\}_k$ may be flattened into $\{\Theta_a, +, \Theta_b, +, \Theta_c\}_k$, and likewise for longer *right-extended branches* subscripted with the same loop k.

We say that a TREC Θ is *univariate* when only one loop affects the evolution (i.e. k is identical for each recurrence of the tree) and *multivariate* otherwise. We also provide an additional TREC, denoted by $\top$, to capture any evolution that we fail to recognize.

2.1 Evaluation of Trees of Recurrences

The value $\Theta(\ell_1, \ell_2, \ldots, \ell_m)$ of a TREC Θ is defined as follows: if Θ is a constant c then $\Theta(\vec{\ell}) = c$, otherwise, Θ is of the form $\{\Theta_a, +, \Theta_b\}_k$ and

$$\Theta(\vec{\ell}) = \Theta_a(\vec{\ell}) + \sum_{l=1}^{\ell_k} \Theta_b(\ell_1, \ldots, \ell_{k-1}, l, \ell_{k+1}, \ldots, \ell_m).$$

Intuitively, the evaluation of $\{\Theta_a, +, \Theta_b\}_k$ for a given vector $\vec{\ell}$ matches the inductive updates of a scalar variable across ℓ_k iterations of loop k, with Θ_a the initial value of the variable and Θ_b the increment for this variable. This definition leads to an exponential algorithm to evaluate a TREC at a given iteration point, but we may compute $\Theta(\vec{\ell})$ with a linear time and space complexity — in terms of arithmetic operations. The proof and computation process is adapted from the MCR framework [6].

Newton series. Efficient computation of MCR is based on Newton interpolation series. Considering a univariate chain of recurrences with constant parameters c_0, c_1, c_2, …, c_n,

$$\{c_0, +, c_1, +, c_2, +, \ldots, +, c_n\}_k(\vec{\ell}) = \sum_{p=0}^{n} c_p \binom{\ell_k}{p}. \tag{1}$$

This result comes from the following observation: a sum of multiples of binomial coefficients — called *Newton series* — can represent any polynomial. The closed form for `f` in Figure 2 is the second order polynomial $F(\ell_1) = \frac{5}{2}{\ell_1}^2 + \frac{11}{2}\ell_1 + 3$ which can be represented by the sum of multiples of binomial coefficients $c_0\binom{\ell_1}{0} + c_1\binom{\ell_1}{1} + c_2\binom{\ell_1}{2}$, with $c_0 = 3$, $c_1 = 8$ and $c_2 = 5$. This corresponds to the TREC $\{3, +, 8, +, 5\}_1$. Notice that the length of the TREC gives the order of the polynomial it represents.

Interestingly, coefficients of a TREC derive from a finite differentiation table. For example, the coefficients for the TREC associated with $\frac{5}{2}{\ell_1}^2 + \frac{11}{2}\ell_1 + 3$ can be computed as follows:

ℓ_1	0	1	2	3	4
c_0	**3**	11	24	42	65
c_1	**8**	13	18	23	
c_2	**5**	5	5		
c_3	**0**	0			

However, unlike Haghighat and Polychronopoulos [15] we do not actually construct this differentiation table to find coefficients of a TREC, we will explain in a further section how to extract those coefficients directly from the SSA representation of the code.

Normalization. The evaluation process begins with the recursive application of the following rules — where the left-hand and right-hand sides describe the same evolution:

$$\begin{aligned}
\{\{\Theta_a, +, \Theta_b\}_k, +, \Theta_c\}_k &\rightarrow \{\Theta_a, +, \Theta_b + \Theta_c\}_k, \\
\{\{\Theta_a, +, \Theta_b\}_k, +, \Theta_c\}_{k'} &\rightarrow \{\{\Theta_a, +, \Theta_c\}_{k'}, +, \Theta_b\}_k.
\end{aligned}$$

This allows to simplify and normalize the TREC, such that sub-trees of a constant k are linearized into right-extended branches, and such that nodes of dimension k may only enclose nodes of dimension less than or equal to k. This matches the definition of a *pure sum* MCR. Following [6], we recursively substitute every flattened TREC of constant dimension k with its Newton interpolation. If the original TREC has N "+" operators, the result is a sum of at most $N + 1$ terms, hence the linear complexity.

Evaluation example. Considering the second introductory example (see Figure 3), the evolution of `d` can be represented by the affine equation $F(\ell_1, \ell_2) = 14\ell_1 + \ell_2 + 3$. A multivariate affine TREC for `d` is $\Theta = \{\{3, +, 1\}_2, +, 14\}_1$, another equivalent one is $\Theta = \{\{3, +, 14\}_1, +, 1\}_2$ according to the second normalization rule given above.

The evaluation of $\Theta(\ell_1, \ell_2)$ for $\ell_1 = 10$ and $\ell_2 = 15$ proceeds as follows:

$$\begin{aligned}
\Theta(10, 15) &= \{\{3, +, 14\}_1, +, 1\}_2(10, 15) \\
&= \{3 + 14\binom{10}{1}, +, 1\}_2(15) = \{143, +, 1\}_2(15) \\
&= 143 + \binom{15}{1} = 158
\end{aligned}$$

2.2 Algebraic Properties

Given Θ_a, Θ_b, Θ_c and Θ_d some trees of recurrences and x a constant, the following algebraic properties prove that trees of recurrences are closed for addition and multiplication:

$$\begin{aligned}
x + \{\Theta_a, +, \Theta_b\}_k &= \{x + \Theta_a, +, \Theta_b\}_k \\
x \times \{\Theta_a, +, \Theta_b\}_k &= \{x \times \Theta_a, +, x \times \Theta_b\}_k \\
\{\Theta_a, +, \Theta_b\}_k + \{\Theta_c, +, \Theta_d\}_k &= \{\Theta_a + \Theta_c, +, \Theta_b + \Theta_d\}_k \\
\{\Theta_a, +, \Theta_b\}_k \times \{\Theta_c, +, \Theta_d\}_k &= \{\Theta_a \times \Theta_c, +, \{\Theta_a, +, \Theta_b\}_k \times \Theta_d \\
&\quad + \{\Theta_c, +, \Theta_d\}_k \times \Theta_b + \Theta_b \times \Theta_d\}_k
\end{aligned}$$

2.3 Peeled Trees of Recurrences

A common case used quite frequently by programmers consists in variables that contain a value used during the first iteration of a loop, that is replaced by the value of another variable. We have chosen to represent these variables by explicitly listing the first value that they contain, and then the evolution function that they follow. The *peeled* TREC are described by the syntax $(a, b)_k$ whose semantics is given by:

$$(a, b)_k(x) = \begin{cases} \text{a} & \text{if } x = 0, \\ \text{b (x - 1)} & \text{for } x \geq 1, \end{cases}$$

where a is a TREC with no evolution in loop k, b is a TREC that can have an evolution in loop k, and x is indexing the iterations in loop k. As a form of syntactic sugar, we write $(a, b, c)_k$ for the nested $(a, (b, c)_k)_k$.

Most closed forms for wrap-around variables [34] are peeled TREC. Indeed, back to the third introductory example (see Figure 4), the closed form for `c` can be represented by a peeled multivariate affine TREC:

$$(\{1, +, 4\}_1, \{\{5, +, 4\}_1, +, 6\}_2)_2.$$

2.4 Unification of Values Described by a Peeled TREC

A peeled TREC describe the first values of a closed form chain of recurrence. In some cases it is interesting to replace it by a simpler MCR, and vice versa, to peel some iterations out of a MCR. For example, the peeled TREC $(0, \{1, +, 1\}_1)_1$ describes the same function as $\{0, +, 1\}_1$. This last form is a unique representative of a class of TREC that can be generated by peeling one or more elements from the beginning.

Simplifying a peeled TREC amounts to the unification of its first element with the function represented in the right-hand side of the peeled TREC. A simple unification algorithm looks whether a TREC can be built by adding a column to the differentiation table without changing the last element in that column. Since this first column contains the coefficients of the chain of recurrence, the transformation is possible if it does not modify the last coefficient of the column. This is illustrated in figure 6. Unifying nested peeled TREC can be done incrementally from the innermost outwards.

ℓ_1	0	1	2	3	4
c_0	3	11	24	42	65
c_1	8	13	18	23	
c_2	5	5	5		
c_3	0	0			

ℓ_1	0	1	2	3	4	5
c_0	**0**	3	11	24	42	65
c_1	**3**	8	13	18	23	
c_2	**5**	5	5	5		
c_3	**0**	0	0			

Figure 6: Adding a new column to the differentiation table of the chain of recurrence $\{3, +, 8, +, 5\}_1$ leads to the chain of recurrence $\{0, +, 3, +, 5\}_1$.

Finally, we formalize the notion of peeled TREC equivalence class: given integers $v, a_1, \ldots, a_n$, a TREC $c = \{a_1, +, \ldots, +, a_n\}_1$, a peeled TREC $p = (v, c)_1$, and a TREC $r = \{b_1, +, \ldots, +, b_{n-1}, +, a_n\}_1$, with the integer coefficients $b_1, \ldots, b_{n-1}$ computed as follows: $b_{n-1} = a_{n-1} - a_n$, $b_{n-2} = a_{n-2} - b_{n-1}$, ..., $b_1 = a_1 - b_2$, we say that r is equivalent to p if and only if $b_1 = v$.

2.5 Periodic Evolutions

One of the consequences of adopting the symbolic TREC is the ability to handle complex evolutions that do not have closed forms. It is then possible to further analyze these complex evolutions or just decide that the game is too risky and stop the analysis in time. However we will play the game in this paper and see how it is possible to detect periodic behaviors of induction variables. First we have to define a syntax that will serve as a support for our extraction algorithm. We call the interdependent peeled TREC mixers in the general case because the result of the computation is the mix of the initial values as entered in the loop. In the special case where the initial values are not modified during the iterations of a loop, we call the interdependent peeled TREC exchangers, because the variables exchange their initial values without modification in the loop. In that case, it is possible to detect a minimal number of iterations, i.e. the period of the exchanger, that makes the values of each variable to be reassigned to the original variable. Based on this intuitive description, we will formalise the notion of periodic evolution using the the syntax $[a_0, \ldots, a_{n-1}]_k$ for describing a periodic evolution in loop k of period n with the following semantics:

$$[a_0, \ldots, a_{n-1}]_k(x) = \begin{cases} a_0 & \text{if } x = 0 \ mod\ n, \\ \ldots & \\ a_{n-1} & \text{if } x = n-1 \ mod\ n. \end{cases}$$

2.6 Symbolic, Exponential and Envelopes TREC

To use TREC in a symbolic computation process, we extend them with symbolic parameters. In the intermediate steps of the computation, a symbolic TREC may not always correspond to a closed form: some symbolic parameters may inductively depend on the variable whose TREC is being evaluated.

Our scalar evolution algorithm may also return a symbolic TREC, as the remaining parameters correspond to statically unknown program variables *invariant in the loops of interest*, such as function parameters or variables defined outside these loops.

Several extensions of the TREC framework would be interesting, to capture a wider range of scalar evolutions. For example, Wolfe [34] showed a classification method to recognize some cases of *exponential* evolutions, and Engelen used a MCR framework to retrieve both polynomial and exponential evolutions with the same algorithm [32]. Wolfe also introduced the class of *monotonic* induction variables, his method and definition being extended by Wu et al. [35] to build a fast dependence test. Experiments are on the way, along with further extensions including evolution envelopes (that improve on the analysis of monotonic induction variables) and more expressive symbolic expressions [31, 29]. However, it is still unclear whether such extensions will be effective for dependence testing and static analysis on a low level code representation.

3 Analysis of Scalar Evolutions

We will now present an algorithm to compute closed-form expressions for inductive variables. The interface to our analyzer is designed as an interface to a database that contains, for a given variable definition, its evolution function under the form of a TREC. For example, when the data dependence analyzer needs the evolution function of a variable that indexes an array, it simply launches a query that either returns the cached evolution function because it was previously computed, or otherwise triggers the analysis of the asked variable, triggering the analysis of all the variables, loop counts, etc. needed to determine the evolution function. We have observed that during the analysis of the evolution of all the loop-ϕ nodes in all the loops of a program the average use of the cache is of about two queries per stored scalar evolution, and consequently not having the database at all can also be considered as a potentially good solution, but this entirely depends on the number of applications that are using the information of evolution functions.

There are several constraints that lead to the design of our analyzer, and that we would like to describe before entering in the details of the algorithm.

First, our algorithm does not assume a particular control-flow structure and makes no restriction on the recursive intrication of variable definitions. It however fails to detect any meaningful induction variable on irreducible control flow graphs that cannot be analyzed into natural loop structures [2]. For all the variables defined in one of the basic blocks of an irreducible region, the answer of our analyzer will be $\top$ that stands for an uncomputable evolution.

Another characteristic of this algorithm is that it does not use the syntactic information of the analyzed SSA representation. In other words, it makes no distinction between the names of variables defined in the source code and those that are introduced by the lowering to three-address code, or by other optimizers. We have experimented with a version of the algorithm that considers the syntactic information of the variables, however the number of special cases exploses, and often leads to a large number of wrap around variables that are only the consequence of the code lowering.

Now that we have set the landscape, we will carefully describe the components that construct our algorithm.

3.1 Algorithm

Figure 7 presents our algorithm to compute the scalar evolutions of all variables defined by loop-ϕ nodes: ComputeLoopPhiEvolutions returns a TREC for every variable whose value is alive across loop iterations. It is important to note that this function is just a driver that we have used for illustrating the query system of our algorithm. In general the computation of closed form expressions are triggered by applications like the dependence analysis, the evaluation of loop-trip count, strength-reduction, or the removal of loop-carried scalar dependences by closed-form substitution [34]. All these applications call directly the real analyzer: AnalyzeEvolution for a given loop number and a variable name. The evolution functions are stored in a database that is used only in the AnalyzeEvolution function and that is accessed using the following construct: Evolution$[n]$, for an SSA name definition n. The value contained initially in the database for a non analyzed name definition is $\perp$ that stands for a not yet analyzed variable.

The cornerstone of the algorithm is the search and reconstruction of the symbolic update expression on a path of the SSA graph. Let us start with the description of the BuildUpdateExpr algorithm. This corresponds to a depth-first search algorithm in the SSA graph with each step composed of a look-up of an SSA definition, and then followed by a recursive call of the search algorithm on the symbolic operands. The search halts when the starting loop-ϕ node is reached. When analyzing an assignment whose right-hand side is a sum, the search algorithm examines the first operand, and if the starting loop-ϕ node is not reachable through this path, it examines the second operand. When one of the operands contains a path to the starting loop-ϕ node, the other operand of the sum is added to the update expression, and the result is propagated to the lower search steps together with the reconstructed update expression. If the starting loop-ϕ node cannot be found by depth-first search, i.e., when BuildUpdateExpr returns (false, $\perp$), we know that the definition does not belong to a cycle of the SSA graph: a peeled TREC is returned.

InstantiateEvolution substitutes symbolic parameters in a TREC. It computes their statically known value, i.e., a constant, a periodic function, or an approximation with intervals, possibly triggering other computations of TREC in the process. The call to InstantiateEvolution is postponed until the end of the depth-first search, ensuring termination of the recursive nesting of depth-first searches, and avoiding early approximations in the computation of update expressions. Combined with the introduction of symbolic parameters in the TREC, postponing the instantiation alleviates the need for a specific ordering of the computation steps.

The overall effect of an inner loop may only be computed when the exit value of the variable is a function of the entry value. In such a case, the whole loop is behaving as a macro-increment operation. When the exit condition depends on affine TREC only, function NumberOfIterations deduces the number of iterations of the loop. Then we call Apply to evaluate the overall effect of the inner loop. Apply implements the efficient evaluation scheme for TREC based on Newton interpolation series (see Section 2.1). As a side-effect, the algorithm does indeed compute the loop-trip count for many natural loops in the control-flow graph. Our method recovers information that was lost during the lowering process or syntactically hidden in the source program (e.g., when using `while` loops instead of Fortran `do` loops).

The algorithm is proved by structural induction on the semantics of the three-address representation

```
Algorithm: ComputeLoopPhiEvolutions
Input: SSA representation of the procedure
Output: a TREC for every variable defined by loop-φ nodes
  For each loop l in a depth-first traversal of the tree of nested loops
    For each loop-φ node n in loop l
      InstantiateEvolution(AnalyzeEvolution(l, n), l)

Algorithm: AnalyzeEvolution(l, n)
Input: l the current loop, n the definition of an SSA name
Output: TREC for the variable defined by n within l
  v ← variable defined by n
  ln ← loop of n
  If Evolution[n] ≠ ⊥ Then
    res ← Evolution[n]
  Else If n matches "v = constant" Then
    res ← constant
  Else If n matches "v = a" Then
    res ← AnalyzeEvolution(l, a)
  Else If n matches "v = a ⊙ b" (with ⊙ ∈ {+, −, ∗}) Then
    res ← AnalyzeEvolution(l, a) ⊙ AnalyzeEvolution(l, b)
  Else If n matches "v = loop-φ(a, b)" Then
    (notice a is defined outside loop ln and b is defined in ln)
    Search in depth-first order a path from b to v:
    (exist, update) ← BuildUpdateExpr(n, definition of b)
    If (not exist) (i.e., if such a path does not exist) Then
      res ← (a, b)_l
    Else If update is ⊤ Then
      res ← ⊤
    Else
      res ← {a, +, update}_l
  Else If n matches "v = condition-φ(a, b)" Then
    eva ← InstantiateEvolution(AnalyzeEvolution(l, a), ln)
    evb ← InstantiateEvolution(AnalyzeEvolution(l, b), ln)
    If eva = evb Then
      res ← eva
    Else
      res ← ⊤
  Else
    res ← ⊤
  Evolution[n] ← res
  Return Eval (res, l)

Algorithm: BuildUpdateExpr(h, n)
Input: h the halting loop-φ, n the definition of an SSA name
Output: (exist, update), exist is true if h has been reached,
update is the reconstructed expression for the overall effect in the loop of h
  If (n is h) Then
    Return (true, 0)
  Else If n is a statement in an outer loop Then
    Return (false, ⊥),
  Else If n matches "v = a" Then
    Return BuildUpdateExpr(h, definition of a)
  Else If n matches "v = a + b" Then
    (exist, update) ← BuildUpdateExpr(h, a)
    If exist Then Return (true, update + b),
    (exist, update) ← BuildUpdateExpr(h, b)
    If exist Then Return (true, update + a)
  Else If n matches "v = loop-φ(a, b)" Then
    ln ←loop of n
    (notice a is defined outside ln and b is defined in ln)
    If a is defined outside the loop of h Then
      Return (false, ⊥)
    s ← Apply(ln, AnalyzeEvolution(ln, n), NumberOfIterations(ln))
    If s matches "a + t" Then
      (exist, update) ← BuildUpdateExpr(h, a)
      If exist Then
        Return (exist, update + t)
  Else If n matches "v = condition-φ(a, b)" Then
    (exist, update) ← BuildUpdateExpr(h, a)
    If exist Then Return (true, ⊤)
    (exist, update) ← BuildUpdateExpr(h, b)
    If exist Then Return (true, ⊤)
  Return (false, ⊥)

Algorithm: InstantiateEvolution(trec, l)
Input: trec a symbolic TREC, l the instantiation loop
Output: an instantiation of trec
  If trec is a constant c Then Return c
  Else If trec is a variable already instantiated Then
    Return ResolveMixer(l, v)
  Else If trec is a variable v Then
    Return AnalyzeEvolution(l, v)
  Else If trec is of the form {e1, +, e2}_x Then
    i1 ← InstantiateEvolution(e1, l)
    i2 ← InstantiateEvolution(e2, l)
    Return {i1, +, i2}_x
  Else If trec is of the form (e1, e2)_x Then
    i1 ← InstantiateEvolution(e1, l)
    i2 ← InstantiateEvolution(e2, l)
    Return UnifyPeeled((i1, i2)_x)
  Else Return ⊤
```

Figure 7: Algorithm to compute scalar evolutions

in section 3.3: global correctness is derived from the local properties of operations on symbolic TREC, exploiting the associativity of assignments along paths of the SSA graph, the ability to substitute any symbolic parameter with another TREC, and the simultaneous evaluation of overall effects and loop-trip counts.

3.2 Application to the Introductory Examples

Let us now illustrate the algorithm on the first two introductory examples in Figures 2 and 3. In addition to clarifying the depth-first search and instantiation phases of the algorithm, this will exercise the recognition of polynomial and multivariate evolutions.

First example. The depth-first search is best understood with the analysis of `c = φ(a, f)` in the first example. The SSA edge of the initial value exits the loop, as represented in Figure 8.(1). Here, the initial value is left in a symbolic form, but GCC would replace it by 3 through constant propagation.

To compute the parametric evolution function of `c`, the analyzer starts a depth-first search algorithm, as illustrated in Figure 8.(2). We follow the update edge `c`→`f` to the definition of `f` in the loop body: assignment `f = e + c`. The depth-first algorithm follows the first operand, `f`→`e`, reaching the assignment

```
a = 3;
b = 1;
loop (ℓ1)
| c = φ(a, f);
| d = φ(b, g);
| if (d>=123) goto end;
| e = d + 7;
| f = e + c;
| g = d + 5;
end:
```

(1) Initial condition edge

```
a = 3;
b = 1;
loop (ℓ1)
| c = φ(a, f);
| d = φ(b, g);
| if (d>=123) goto end;
| e = d + 7;
| f = e + c;
| g = d + 5;
end:
```

(2) Search "c" (not found)

```
a = 3;
b = 1;
loop (ℓ1)
| c = φ(a, f);
| d = φ(b, g);
| if (d>=123) goto end;
| e = d + 7;
| f = e + c;
| g = d + 5;
end:
```

(3) Found the halting phi

```
a = 3;
b = 1;
loop (ℓ1)
| c = φ(a, f);
| d = φ(b, g);
| if (d>=123) goto end;
| e = d + 7;
| f = e + c;
| g = d + 5;
end:
```

(4) On the "returning path"

Figure 8: Application to the first example

`e = d + 7`, and finally follows the edge `e`→`d` that leads to a loop-ϕ node of the same loop. Since this is not the loop-ϕ node from which the analyzer has started the depth-first search, the search continues on the other operands that were not yet analyzed: back on `e = d + 7`, operand `7` is a scalar and there is nothing more to do, then back on `f = e + c`, the edge `f`→`c` is followed to the starting loop-ϕ node, as illustrated in Figure 8.(3).

At this point, the analyzer has found the strongly connected component that corresponds to the path of iterative updates. Following this path in execution order, as illustrated in Figure 8.(4), the analyzer builds the update expression as an aggregation of the operands that are not on the updating path: in this example, the update expression is just `e`. As a result, the analyzer assigns to the definition of `c` the parametric evolution function $\{\mathtt{a}, +, \mathtt{e}\}_1$.

The instantiation of the parametric expression $\{\mathtt{a}, +, \mathtt{e}\}_1$ starts with the substitution of the first operand of the TREC: $\mathtt{a} = 3$, then the analysis of `e` is triggered. First the assignment `e = d + 7` is analyzed, and since the evolution of `d` is not yet known, the edge `e`→`d` is taken to the definition `d = φ(b, g)`. Since this is a loop-ϕ node, the depth-first search algorithm is used as before and yields the evolution function of `d`, $\{b, +, 5\}_1$, and after instantiation, $\{1, +, 5\}_1$. Finally the evolution of `e = d + 7` is computed: $\{8, +, 5\}_1$. A final instantiation yields the polynomial TREC of `c`: $\{3, +, 8, +, 5\}_1$.

Second example. We will now compute the evolution of `x` in the nested loop example of Figure 3, to illustrate the recognition of multivariate induction variables and the computation of the trip count of a loop. The first step consists in following the SSA-edge to the definition of `x`. Consider the right-hand side of the definition: since the evolution of `e` along loop 1 is not yet analyzed, we follow the edge `e`→`d` to its definition in loop 2, ending on the definition of a loop-ϕ node.

At this point we know that `d` is updated in loop 2. The initial condition `c` is kept under a symbolic form, and the iteration edge `d`→`e` is followed in the body of loop 2. The depth-first search algorithm starts from right-hand side of the assignment `e = d + 1`: following the edge `e`→`d` we end on the loop-ϕ node from which we have started the search, meaning that the search succeeded. Back on the path `d`→`e`→`d`, the analyzer gathers the evolution of `d` along the whole loop, an increment of 1, and ends on the following symbolic TREC: $\{\mathtt{c}, +, 1\}_2$.

From the evolution of `d` in the inner loop, the analyzer determines the overall effect of loop 2 on `d`, that is the evaluation of function $f(i) = \mathtt{c} + i$ for the number of iterations of loop 2. Fortunately, the exit condition is the simple expression `t>=9`, and the TREC for `t` (or `d - c`) is $\{0, +, 1\}_2$, an affine (non-symbolic) expression. It comes that 10 iterations of loop 2 will be executed for each iterations of loop 1. Calling Apply(2, $\{\mathtt{c}, +, 1\}_2$, 10) yields the overall effect `d = c + 10`.

The analyzer does not yet know the evolution function of `c`, and consequently it follows the SSA edge to its definition: `c = φ(a, x)`. Since this is a loop-ϕ node, the analyzer must determine its evolution in loop 1. We ignore the edge to the initial condition, and walk back the update edge, searching for a path from `c` to itself.

First, edge `c`→`x` leads to the statement `x = e + 3`, then following the SSA edge `x`→`e`, we end on a statement of the loop 2. Again, edge `e`→`d` is followed, ending on the definition of `d` that we have already analyzed: $\{\mathtt{c}, +, 1\}_2$. The depth-first search selects the edge `d`→`c`, associated with the overall effect statement `d = c + 10` that summarizes the evolution of the variable in the inner loop. We finally reached the starting loop-ϕ node `c`. From this point, the path is walked back gathering the stride of the loop: 10 from the assignment `d = c + 10`, then 1 from the assignment `e = d + 1`, and 3 from the last assignment on the return path. We have computed the symbolic TREC of `c`: $\{\mathtt{a}, +, 14\}_1$.

The last step consists in the propagation of this evolution function from the loop-ϕ node of `c` to the original node of the computation: the definition of `x`. Back from `c` to `d`, we can partially instantiate its evolution: a symbolic TREC for `d` is $\{\{\mathtt{a}, +, 14\}_1, +, 1\}_2$. Then back to `e = d + 1` we get a symbolic TREC for `e`: $\{\{\mathtt{a}{+}1, +, 14\}_1, +, 1\}_2$; and finally back to `x = e + 3`, we get a symbolic TREC for `x`: $\{\mathtt{a}{+}14, +, 14\}_1$. A final instantiation of `a` yields the closed form of `x` and all other variables.

```
loop (ℓ1)
| a = φ(0, b);
| b = φ(1, a);
| T[a] = ...
end:
```

Figure 9: An exchanger

Third example. We consider the analysis of an exchanger as presented in figure 9. Suppose that the evolution of a has been triggered during the dependence analysis of array access $T[a]$. The first step leads to the definition of a, then on the edge entering the loop on the definition of b that is different than the halting phi node, and consequently the evolution is $a \to (0, b)_1$. The instantiation of this form triggers the analysis of b that results in $b \to (1, a)$, and finally to $a \to (0, (1, a)_1)_1$. The last step unifies this recursive definition of a into the periodic evolution: $a \to [0, 1]_1$. For the clarity of the algorithm and space constraints, the UnifyPeeled function has not been included in figure 7, but the algorithm is a simple application of the ideas exposed in sections 2.4 and 2.5.

3.3 Proof of the algorithm

To prove that the result of our algorithm is correct, we will prove that the semantic of the SSA representation is preserved during the translation to the TREC. The proof of this translation is established by structural induction: a first induction is based on the degree of the chain of recurrence, the second induction is based on the dimension of the chain of recurrence. The next steps consist in identifying the syntax of peeled TREC directly from the syntax of the loop-ϕ nodes, then in proving by induction on the period of a periodic TREC the link between the semantics of peeled and periodic TREC.

The translation of programs from the SSA representation into the TREC is based on a pattern matching algorithm. This is why we will consider in our proof several cases as shown in figure 10. For systematizing the proof of our algorithm, we have chosen to represent the first argument of a loop-ϕ node as the initial condition of the scalar variable, and the second argument of the loop-ϕ nodes as the reconstructed overall effect of a loop on a scalar variable.

	SSA syntax	conditions	semantics of SSA
1	a = loop-ϕ (init, a + expr)	*expr* independent on a	$a(x) = init + \sum_{j=0}^{x-1} expr(j)$
2	a = loop-ϕ (init, k*a + expr)	*expr* independent on a, $k > 1$	exponential case
3	a = loop-ϕ (init, expr)	*expr* independent on a	$a(x) = \begin{cases} init, & \text{when x = 0;} \\ expr(x-1), & otherwise. \end{cases}$
4	a = loop-ϕ (init, expr)	*expr* transitively dependent on a	general case: mixers

Figure 10: SSA representation of loop-ϕ nodes after tree reconstruction.

3.3.1 Construction of pure sum univariate TREC

When *expr* **is a constant** with respect to the analyzed loop l, or in other words when *expr* is a polynomial of degree 0, the evolution of the variable a in the loop is an affine function with an initial condition *init*, and a step *expr*. In order to show this, we start from the semantics of the SSA representation:

$$a(x) = init + \sum_{j=0}^{x-1} expr(j)$$

and since $expr(j)$ is a constant with respect to j,

$$a(x) = init + x * expr$$

Finally, based on the semantics of the pure sum chains of recurrences, by identification we get the corresponding syntax:

$$a(x) = init * \binom{x}{0} + expr * \binom{x}{1}$$
$$a(x) \rightarrow \{init, +, expr\}_l(x)$$

When *expr* **is a polynomial of degree** $n-1$ with respect to the analyzed loop l for which we have already determined that it is written under the syntax:

$$expr(x) \rightarrow \{b_0, +, \ldots, +, b_{n-1}\}_l(x)$$

We start from the semantics of the SSA program:

$$\begin{array}{rcl}
a(x) & = & init + \sum_{j=0}^{x-1} expr(j) \\
a(x) & = & init + \sum_{j=0}^{x-1} (b_0 * \binom{j}{0} + \ldots + b_{n-1} * \binom{j}{n-1}) \\
a(x) & = & init + \sum_{j=0}^{x-1} \sum_{k=0}^{n-1} (b_k * \binom{j}{k}) \\
a(x) & = & init + \sum_{k=0}^{n-1} (b_k * \sum_{j=0}^{x-1} \binom{j}{k}) \\
a(x) & = & init + \sum_{k=0}^{n-1} (b_k * \binom{x}{k+1}) \\
a(x) & = & init + b_0 * \binom{x}{1} + \ldots + b_{n-1} * \binom{x}{n} \\
a(x) & = & init * \binom{x}{0} + b_0 * \binom{x}{1} + \ldots + b_{n-1} * \binom{x}{n} \\
a(x) & \rightarrow & \{init, +, b_0, +, \ldots, +, b_{n-1}\}_l(x)
\end{array}$$

that is a polynomial of degree n in loop l.

3.3.2 Analysis of multivariate TREC

When a variable is assigned outside of any loop, i.e. at depth zero, its value is either a constant or a parameter that is passed in by functions arguments. We present the interprocedural case in section 3.6, and for simplifying the presentation, we don't consider parameters until that section. In an intraprocedural analysis the parameters can be considered as constants, i.e. entities that are not varying in any loop, and computations can be performed symbolically.

The case of a single surrounding loop has been analyzed in the previous section. The value of a variable on exit of a loop can be expressed as either a constant or a parameter expression in function of the number of iterations of the crossed loop. Loops are behaving as compound statements that assign complex values to the variable, independently on their iteration counters.

We shall see by induction on the SSA structure how the multivariate TREC are constructed. For this we analyze the different cases that can occur during the analysis of an assignment.

When right hand side is a constant. The dimension of the evolution function is equal to zero. The defined variable does not vary in any loop.

When right hand side is a loop-ϕ node. the first step is to consider an initial condition that is a constant, and observe as presented in section 3.3.1 that the analysis leads to a univariate evolution when the loop modifies the initial value of the variable. If the value is not modified by the loop, the dimension is zero. If the initial condition is a multivariate function of dimension n, and the considered loop modifies the initial condition, the dimension of the evolution function is equal to $n + 1$. If the current loop does not modify the value of the variable, the dimension is equal to n.

When right hand side is an expression. there are two cases following the number of variables in the expression. This simplification is a direct effect of considering GIMPLE expressions.

Lemma: When the right hand side is an expression containing a single variable name whose evolution function is of dimension n, then the assigned variable has also an evolution function of dimension n.

Lemma: When the right hand side is an expression containing two variable names whose evolution function is respectively of dimension n and m, then the assigned variable has an evolution function of dimension at most equal to $n + m$, and at least equal to $max(m, n)$.

Proof: Trivial by considering the arithmetic operations defined on TREC in section 2.

3.3.3 Peeled TREC

When the SSA chain don't end on the same loop-ϕ node, the reconstructed expression does not contain a self reference, however it is possible that the reconstructed expression transitively depends on itself. These loop-ϕ nodes are represented by the cases 3 and 4 in figure 10.

In the case 3. We identify the semantics of the SSA representation with the semantics of the peeled TREC defined in section 2.3. The initial value is used during the first iteration, then this value is discarded and replaced by the value of another induction variable. This is the definition of a wrap around variable.

The case 4. represents in general the mixers whose initial values are modified during the loop. An example of mixers is the Fibonacci function. It is still possible to identify the semantics of this SSA representation with the semantics of peeled TREC by allowing a peeled TREC to be self referenced. The case of the periodic TREC is a subcase of this one, in which the initial values of the variables are not modified in the loop, but exchanged.

3.3.4 Periodic TREC.

Lets concentrate on how the periodic TREC are constructed. Intuitively, the periodic TREC are the result of a chain of loop-ϕ nodes that don't contain arithmetic operations, such that they don't modify the initial values of variables. The same initial values are exchanged until they are assigned back to their initial container. Following the number of loop-ϕ nodes in the exchanging information chain it is possible to detect the period of the exchanger.

Two loop-ϕ nodes Suppose that the analyzed SSA representation contains two interdependent loop-ϕ nodes:

$$a = phi(c, b)$$

$$b = phi(d, a)$$

Because the starting loop-ϕ node is not reachable in a single tour of the loop, the analyzer builds a symbolic peeled TREC, then by instantiation, it ends on a symbolic peeled TREC with a self reference: $a \to (c, d, a)_l$. The detection of the self reference triggers the translation of the peeled TREC into a periodic TREC. Indeed, starting from the semantic of peeled TREC in section 2.5:

$$a(x) \to (c, d, a)_l(x) =_{def} \begin{cases} \text{c} & \text{if } x = 0, \\ \text{d} & \text{if } x = 1, \\ \text{a (x - 2)} & \text{for } x \geq 2, \end{cases}$$

following the first two cases, $a(x - 2k) = c$ for $x = 2k$ and $a(x - 2k) = d$ for $x = 2k + 1$. Following the third case $a(x) = a(x - 2)$ for $x \geq 2$, that initiates the recursive definition, $a(x) = a(x - 2k)$ for $x \geq 2k$, and more precisely by combining these two results, $a(x) = a(x - 2k) = c$ for $x = 2k$, and $a(x) = a(x - 2k) = d$ for $x = 2k + 1$. More synthetically:

$$a(x) \to (c, d, a)_l(x) = \begin{cases} \text{c} & \text{for } x = 2k, \\ \text{d} & \text{for } x = 2k + 1. \end{cases} = \begin{cases} \text{c} & \text{if } x = 0 \ mod\ 2, \\ \text{d} & \text{if } x = 1 \ mod\ 2. \end{cases} =_{def} [c, d]_l(x)$$

Initial values are exchanged by n loop-ϕ nodes For constants $c_0, \ldots, c_{n-1}$, the same arguments lead to the general result:

$$a \to (c_0, \ldots, c_{n-1}, a)_l(x) = \begin{cases} a \to c_0 & \text{if } x = 0 \ mod\ n, \\ a \to c_1 & \text{if } x = 1 \ mod\ n, \\ \ldots & \\ a \to c_{n-1} & \text{if } x = n - 1 \ mod\ n, \end{cases} =_{def} [c_0, \ldots, c_{n-1}]_l(x)$$

3.3.5 Termination of the algorithm

When analyzing the code of the algorithm, we can briefly sketch its call graph, as shown in Figure 11. The algorithm is initiated by a call to ANALYZEEVOLUTION, then finishes with the analysis of all the symbols left in the representation, by calling INSTANTIATEEVOLUTION.

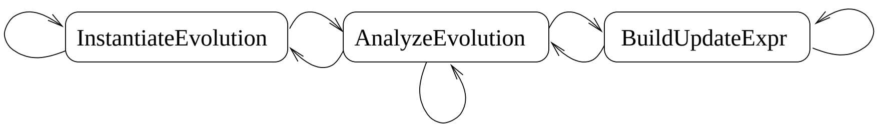

Figure 11: Call graph of the scalar analysis algorithm.

A very quick overview of the ideas that lead to the proof of the termination consists in remarking that:

- ANALYZEEVOLUTION does not process twice the same variable, because of the use of the database,
- INSTANTIATEEVOLUTION does not instantiate twice the same variable, because when a variable is seen twice, a mixer is detected,

- BuildUpdateExpr terminates once it has reached its halting loop-ϕ node, or once it has walked over all the SSA edges in the analyzed loop.

Consequently, after having analyzed all the variables of the program, the analyzer stops. Now for giving an idea of the worst case of the analyzer, we will describe with more details the termination process.

In order to prove the termination, we will consecutively consider the worst case complexity of each of the building blocks of the algorithm. We deduce from this the overall complexity of the algorithm (in the worst case), then the termination of the algorithm in terms of number of basic operations. Figure 12 summarizes the ideas behind each of the components of the algorithm.

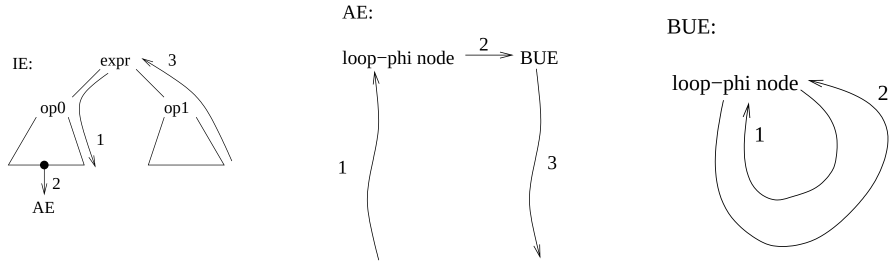

Figure 12: General idea of the InstantiateEvolution, AnalyzeEvolution, and BuildUpdateExpr algorithms.

Cost of InstantiateEvolution In the worst case for InstantiateEvolution, we're given an expression with n elements, and among them, there are all the SSA names, that we call m, defined in the program. The total cost is equal to recursively visit each leaf of the given expression: that produces n steps, then for each leaf calling AnalyzeEvolution. The total number of operations is equal to n decompositions of the given expression, n calls to AnalyzeEvolution, followed by n folds of the obtained subexpressions.

Cost of AnalyzeEvolution The cost of AnalyzeEvolution for a constant is equal to 1.

For a SSA name, the cost is equal to the look-up in the database plus, when the scalar variable was not yet analyzed, the cost of its analysis. Because the first part of this algorithm consists in walking up the definitions to known values or to loop-phi nodes, the algorithm may end as soon as all the needed scalar definitions have values already computed. Furthermore because in this study we're interested in the worst case, we're analyzing all the SSA name definitions in the program. Because the analysis of a single SSA name of a SSA chain triggers the analysis of the entire chain, we can consider that the cost for analyzing the current name is equal to 1, the cost of a database read.

In the worst case, the analysis of all the SSA chains is triggered succesively, starting close to the loop-ϕ nodes. In that case, the total number of steps is equal to $5 \cdot m$, because at each analysis of a SSA name there are at most 3 reads in the database: for $a = b\ op\ c$, a read for the left hand side of the assignment, then a read for each of the operands, then a fold operation on the TREC of the operands, and finally a write of the result in the database.

When AnalyzeEvolution ends on a loop-ϕ node whose evolution is not yet analyzed, AnalyzeEvolution calls BuildUpdateExpr.

Cost of BuildUpdateExpr BuildUpdateExpr is called only from AnalyzeEvolution, when analyzing a loop-ϕ node. The loop-ϕ edge exiting the loop is left under a symbolic form, while the edge pointing in the loop is the one followed by BuildUpdateExpr in a depth first search order until the starting loop-ϕ node is reached. The number of operations triggered by one call to BuildUpdateExpr is equal to the number of edges explored during this depth first search. Once all the paths reachable by following SSA edges in the analyzed loop are explored and the halting loop-ϕ node is still not found, BuildUpdateExpr ends by returning a peeled TREC.

In the particular case where the SSA edges enter an inner loop, the analysis of the definition in the inner loop is triggered. But because we're computing the total number of steps in the worst case, we have already counted these definitions in the input expression to INSTANTIATEEVOLUTION. Thus we can consider all these parts already analyzed, and their cost to the BUILDUPDATEEXPR function is equal to a read in the database.

BUILDUPDATEEXPR is called only on the loop-ϕ nodes not yet analyzed, in other words, once per loop-ϕ node in the worst case. Thus the overall complexity of BUILDUPDATEEXPR is equal to $\sum_{i \in loop-\phi} e_i$, where e_i is the number of SSA edges reachable from the analyzed loop-ϕ , not exiting the loop, and ending on the first loop-ϕ node of the same loop.

Cost of the whole algorithm Putting all together, we obtain the following worst case complexity:

$$3 \cdot n + 5 \cdot m + \sum_{i \in loop-\phi} e_i$$

- n is the number of basic components in TREC to be instantiated,
- m is the number of SSA names in the program,
- e_i is the number of SSA edges reachable from the analyzed loop-ϕ , not exiting the loop, and ending on the first loop-ϕ node of the same loop.

3.3.6 Concluding remarks

We successively have proved that the analyzer produces the TREC representation whose semantics matches the semantics of the SSA representation, then we have proved that the algorithm is terminating on any input SSA representation, and finally we have analyzed the worst case complexity of the analysis algorithm. The overall cost of the analyzer highlights its structure: it is composed of two preprocessing passes followed by the analysis of the loop-ϕ nodes.

3.4 Fixed-Width Integers

The formal definition of a TREC and the previous algorithm assume arithmetic operations over unbounded integers. This is also the case for all induction variable analyzes we are aware of. To be applicable in a general-purpose compiler like GCC, we cannot simply assume that no overflow will occur due to programming errors, to repeated increments on short-width scalars, or to the very common practice of subtracting interval lower bounds from unsigned integers to implement two comparisons in one.

Our current solution is partial: it is simple and efficient but is not a guarantee of correctness. Since GIMPLE is a typed representation, we tag each node in a tree of recurrences with the result type of the ϕ or arithmetic operation it was derived from.

- When a byte-sized variable occurs on the path in the SSA graph, its TREC is conservatively set to $\top$.
- When the algorithm computes the number of iterations of a loop, it calls APPLY on the trip count. When the result does not fit the type of the variable, its TREC is set to $\top$.

Recently, scalable and accurate methods were designed and applied to the verification of reactive control codes for critical embedded systems [12, 8]. We advocate for the integration of such techniques as an optional verification engine to guarantee the correctness of every TREC.

3.5 Application to Redundant Check Elimination

Like most high-level languages with automatic garbage collection, the semantics of Java requires systematic array bound checks. Unlike run-time pointer or type checks, array bound checks can often be statically resolved from the analysis of scalar evolutions. Even in C or Fortran, the programmer may introduce run-time checks for debugging purposes (assertions) or to detect special boundary cases. These checks are assumed to be satisfied during the normal execution of the program, and after procedure inlining, specialization and cloning by the compiler, some of them become redundant and can be statically eliminated.

Practically, for each comparison (equality or inequality, strict or not) between two scalar variables `a` and `b`, we compute the TREC of `a-b`. With the same method as to compute loop-trip counts, we may deduce the number of iterations for which the condition holds, and respectively for which it does not. When the condition holds for all or no iterations, the check may be removed or the branch may be eliminated, respectively.

This method is currently limited to affine TREC and does not take the context of the enclosing conditionals into account. It is mostly intended as a validation tool to debug and enhance our algorithm.

3.6 Interprocedural extensions

To enable interprocedural dependence analysis, we follow the scheme proposed by the PIPS parallelizing compiler project [17]. To avoid name conflicts, loops have to be uniquely identified and procedure bodies are considered as loops that iterate only once. To handle some recursion cases, procedures are also assigned a unique identifier.

Algorithm The interprocedural algorithm is based on an extension of the instantiation mechanism of our analyzer. Instead of stopping the instantiation when analyzing a function parameter, the instantiation continues to follow up the link into the caller bodies. If the analyzer detects several callers, the instantiation values are merged into a single TREC. This TREC is then registered as the initial condition of the parameter in the callee.

Example We illustrate the algorithm on a simple example given in figure 13.

```
void F1 () {
a = 2
loop (ℓ1)
| b = φ(a, c)
| F2 (b);
| c = b + 3;
| if (b > 10) goto end;
end:
}
```

```
void F2 (int x) {
d = 0
loop (ℓ2)
| e = φ(d, f)
| A[x + e] = ...
| f = e + 1;
| if (e > 100) goto end;
end:
}
```

Figure 13: Interprocedural example

In this example, the loops are uniquely labeled; the analysis of array `A` triggers the analysis of variable e: $e \rightarrow \{0, +, 1\}_2$; then it triggers the analysis of x, mapped to argument b in the call to function `F2`. The parameter is then analyzed in the calling context, and since there is a unique point from where `F2` is called, the parameter x is identified as the value of b, i.e. $b \rightarrow \{2, +, 3\}_1$. Finally after the instantiation step, the access function of array `A` is recognized to be $\{\{2, +, 3\}_1, +, 1\}_2$.

When the function `F2` is called in several contexts, the value of parameter x is obtained by merging the information contained in all these contexts.

3.7 Empirical Study

To show the robustness and language-independence of our implementation, and to evaluate the accuracy of our algorithm, we compute scalar evolutions for *all* variables defined by loop-ϕ nodes in the Spec CPU2000 [30] and JavaGrande [18] benchmarks. Six SpecFP and three SpecINT benchmarks were omitted due to compilation errors in the ISO C and Fortran 95 front-ends of GCC's branch.

Computation of scalar evolutions. Figure 14 summarizes our experiments. The first column gives the name of the benchmark. The "Total" column counts the number of scalar evolutions we are trying to recognize. The next 5 columns describe the break-down of these scalar evolutions into the following

INT2000	Total	Affine Uni.	Affine Multi.	Peeled	Other	$\top$	Loops	Trip	No Trip
164.gzip	1183	150	6	404	1	622	224	32	117
175.vpr	2404	308	0	1285	2	809	503	24	291
181.mcf	140	20	4	78	0	38	73	1	34
186.crafty	4857	215	11	4310	0	321	461	104	220
197.parser	1847	219	2	1132	1	493	786	5	392
254.gap	4421	1258	66	2006	92	999	2148	39	1097
255.vortex	748	110	0	448	1	189	333	1	101
256.bzip2	951	118	1	641	5	186	193	32	107
300.twolf	3652	477	8	2167	3	997	1038	18	741
Total	20203	2875	98	12471	105	4654	5759	256	3100
FP2000	**Total**	**Affine Uni.**	**Affine Multi.**	**Peeled**	**Other**	$\top$	**Loops**	**Trip**	**No Trip**
168.wupwise	649	157	0	271	2	219	84	2	69
171.swim	89	50	0	32	0	7	32	0	25
172.mgrid	364	111	0	192	2	59	73	8	52
173.applu	759	304	12	264	0	179	184	66	96
177.mesa	3554	1304	6	1285	18	941	1158	54	794
179.art	439	80	2	282	0	75	100	4	67
183.equake	407	84	0	269	0	54	105	45	32
188.ammp	872	235	9	517	2	109	546	13	203
Total	7133	2325	29	3112	24	1643	2282	192	1338
JavaGrande	**Total**	**Affine Uni.**	**Affine Multi.**	**Peeled**	**Other**	$\top$	**Loops**	**Trip**	**No Trip**
section1	1995	78	0	1685	74	158	201	2	174
section2	251	77	1	154	3	16	107	5	59
section3	718	71	1	461	0	185	173	15	85
Total	2964	226	2	2300	77	359	481	22	318

Figure 14: Types of TREC representing scalar evolutions for loop-ϕ nodes in Spec CPU2000 and JavaGrande benchmarks

classes: affine univariate TREC, affine multivariate TREC, peeled TREC, other kinds of non-$\top$ TREC (e.g., polynomial), and unknown ($\top$) evolutions. Several peeled TREC correspond to wrap-around variables [13] but most of them are associated with loop-ϕ nodes that do not belong to a cycle of the SSA graph, i.e., to non-induction variables whose value is produced in the previous iteration of a loop; these variables incur scalar loop-carried dependences that may be removed by closed form substitution. The amount of detected peeled TREC is a direct consequence of the analysis of low level code. This motivates our extension of the chains of recurrence formalism.

Loop-trip count. The last three columns in Figure 14 describe, respectively: the number of *natural* loops in the benchmark (i.e., the number of control-flow points targeted by back-edges, possibly in irreducible graphs or with multiple exits), the number of *single-exit* loops whose trip count is successfully analyzed, and the number of *single-exit* loops where we fail to compute the trip count. We do not compute trip counts on multiple-exit loops yet.

Unfortunately, a minority of loops have a statically known number of iterations that can be recovered by our method. This is due to a current limitation of our method, which does not compute symbolic trip counts. Indeed, the trip-count information is mostly used in the Apply function of the scalar evolution algorithm, which does not currently handle symbolic evaluations on parametric iteration counts (yielding higher degree TREC in general). This will be improved in future versions of the algorithm and implementation.

4 Dependence Analysis

As we have seen until now, the chains of recurrences are a common representation for functions evaluated at integer points. We have explained how we used this representation for describing the relation between the values taken by variables modified in loops and an idealized loop counter. In this section we show how to represent affine dependence relations using this same formalism of chains of recurrences.

4.1 Data dependence problems

There are several interesting problems around the notion of data dependence testing: one of the most important is the proof of independence between data accesses, that allows the parallelization of the loop. When independence cannot be proven, it is still interesting to have a way to characterize the dependence relation, because it is possible to prove the legality of a loop transformation using the degrees of liberty extracted from the dependence description. In compilers litterature, different abstraction levels have been proposed for the data dependence description, going from the most abstract one to more precise ones, we can list:

- the sign lattice, also known as dependece directions,
- the integer domain, known as dependence distance,
- the polyhedral domain.

We propose in the following to relate the conflicting elements of the referenced data using integer functions, then we extend this representation to what we call the "trecohedral" domain. From these representations we show how to extract the classic dependence representations, and we give a running example for illustrating the extraction algorithm. Finally we give some experimental results and expose the status of the implementation and integration with GCC's optimizers.

4.2 Description of conflicting iterations

The main question that we will answer in this paragraph is what is needed for representing the relations between memory accesses? For referencing memory elements an integer function is used: it is also called access function. When two access functions f_1 and f_2 reference the same array, they potentially access some elements twice. Furthermore, if the access functions f_1 and f_2 are affine functions, it is possible to describe the conflict relation with two affine characteristic functions g_1 and g_2, such that:

$$f_1(g_1(x)) = f_2(g_2(x))$$

for some integer values $x \in [0, k]$, with k the last conflicting iteration, that can either be a determined integer value or not analyzable at compile time, in which case it can be a symbolic parameter. We call functions g_1 and g_2 the conflicting elements functions, because for successive values of x, these functions enumerate the iterations for which the two access functions are equal.

The computation of these conflicting iteration functions is performed as described by Utpal Banerjee in [7], where these functions are computed, and are used in the computation of dependence relations, but they are lost once the computation is over. We will show that these functions can be useful where the classic abstractions are not enough precise.

4.3 Running example

Let us study the simple loop in Figure 15, in a SSA form three-address representation. Consider the three references to array `A`: from left to right, the access functions are $\{2, +, 2\}_1$, $\{0, +, 4\}_1$ and $\{0, +, 2\}_1$.

To analyze dependences between a pair of references, we first compute the sequence of loop iterations which support accesses from both references. Here, `A[2*i+2]` conflicts with `A[2*i']` when `i` $= 0, 1, 2, \ldots, 49$ — $\{0, +, 1\}_1$ — and `i'` $= 1, 2, 3, \ldots, 50$ — $\{1, +, 1\}_1$ — respectively. For `A[2*i+2]` and `A[4*i']`, the respective overlapping iterations are `i` $= 1, 3, 5, \ldots, 99$ — $\{1, +, 2\}_1$ — and `i'` $= 1, 2, 3, \ldots, 50$ — $\{1, +, 1\}_1$.

This suffices to prove that references to `A` induce loop-carried dependences. To get more precise information, one may observe that first element of the sequence of a given reference conflicts with the first element of the other one. Here, the dependence from `A[2*i+2]` to `A[4*i]` is not uniform: indeed, the update coefficients in the TREC of conflicting sequences are different (1 and 2). Repeating the manual computation on references `A[2*i+2]` and `A[2*i]` yields a uniform dependence with a distance of 1 iteration.

```
for (i=0; i<100; i++)
| A[2*i+2] = A[4*i] + A[2*i];
```

↓ lowering

```
i = 0;
loop (ℓ1)
| j = φ(i, k);
| if (j>=100) goto end;
| t = 2 * j;
| u = 4 * j;
| v = t + 2;
| A[v] = A[u] + A[t];
| k = j + 1;
end:
```

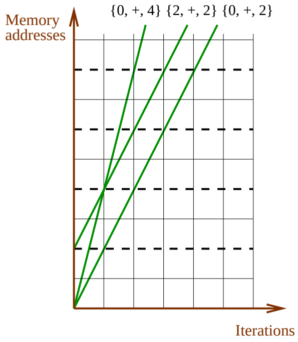

Figure 15: Dependence analysis example

The previous example shows that retrieving dependence information from affine TREC of individual array references is no different from the classical dependence analysis case. Existing dependence analysis frameworks are directly applicable.[2]

INT2000	DD Tests	No DD	% of tests	Distance DD	Other DD	% of tests	Unknown	% of tests
164.gzip	3167	578	18.3%	264	24	9.1%	2301	72.7%
175.vpr	1459	972	66.6%	417	26	30.4%	44	3.0%
181.mcf	163	2	1.2%	6	2	4.9%	153	93.9%
186.crafty	41236	17138	41.6%	10332	314	25.8%	13452	32.6%
197.parser	1319	84	6.4%	349	56	30.7%	830	62.9%
254.gap	9790	3698	37.8%	2613	14	26.8%	3465	35.4%
255.vortex	1112	90	8.1%	90	0	8.1%	932	83.8%
256.bzip2	2536	112	4.4%	275	100	14.8%	2049	80.8%
300.twolf	3363	1894	56.3%	549	62	18.2%	858	25.5%
Total	64145	24568	38.3%	14895	598	24.2%	24084	37.5%
FP2000	**DD Tests**	**No DD**	**% of tests**	**Distance DD**	**Other DD**	**% of tests**	**Unknown**	**% of tests**
168.wupwise	556	0	0%	6	0	1.1%	550	98.9%
171.swim	520	104	20%	205	0	39.4%	211	40.6%
172.mgrid	3623	40	1.1%	61	0	1.7%	3522	97.2%
173.applu	11139	7684	69.0%	1747	20	15.9%	1688	15.2%
177.mesa	12272	3020	24.6%	3066	4748	63.7%	1438	11.7%
179.art	89	38	42.7%	42	4	51.7%	5	5.6%
183.equake	1005	336	33.4%	327	312	63.6%	30	3.0%
188.ammp	25641	17362	67.7%	5803	710	25.4%	1766	6.9%
Total	54845	28584	52.1%	11257	5794	31.1%	9210	16.8%
JavaGrande	**DD Tests**	**No DD**	**% of tests**	**Distance DD**	**Other DD**	**% of tests**	**Unknown**	**% of tests**
section1	99013	92878	93.8%	6135	0	6.2%	0	0%
section2	670	94	14.0%	229	153	57.0%	194	29.0%
section3	2380	322	13.5%	775	615	58.4%	668	28.1%
Total	102063	93294	91.4%	7139	768	7.7%	862	0.8%

Figure 16: Dependence analysis results on Spec CPU2000 and JavaGrande benchmarks

The TREC describing sequences of conflicting accesses can be very complex. Even though most expressions found in benchmarks are affine, computing dependence information from general multivariate affine expressions is equivalent to solving a Diophantine equation [7]. The general solution requires a polyhedral abstraction [11, 28] when taking loop bounds into account.

Since we aim for speed, we look for uniform dependences only and use the Banerjee test [7]. Approximate information (directions) may still be obtained when dependences are not uniform, i.e., when the update parameters in the TREC of both references differ. We also implemented the extension of the Banerjee to

[2] To improve accuracy, symbolic TREC from both array references are subtracted before instantiating the result.

multidimensional arrays [4].

We are currently implementing simple extensions of the Banerjee test to handle peeled TREC, and to refine the accuracy of the analysis with loop-trip count. Since our algorithm can retrieve polynomial expressions and can be extended to recognize flip-flop, periodic, exponential, and monotonic evolutions, we will study graceful degradations of the dependence analysis to avoid complete failure on these cases.

4.4 Experimental results

Our scalar evolution algorithm is integrated in a dependence analysis pass of GCC that is used by vectorization and linear loop transforms. These dependence-based transformations use uniform dependence vectors [7], but our method identifying conflicting accesses between TREC is applicable to the computation of more general dependence abstractions as well.

Other applications, not addressed here, include closed-form substitution [13, 27] — for scalar dependence removal — and strength-reduction, based on the efficient run-time evaluation of MCR [20].

In our implementation, the analysis is performed on demand, when loop transformations or code motion requires dependence information. Yet to illustrate the scalability and accuracy of the analysis, we computed all dependences between pairs of references — both of them addressing the same array — in every function.

Figure 16 exercises our dependence analysis on the Spec CPU2000 and JavaGrande benchmarks. The first column gives the name of the benchmark. The "DD Tests" column gives the total number of data dependence tests performed. The "No DD" column gives the number of tests proving that there is *no* dependence between two memory accesses.

The two following columns count the number of dependences that were detected by our method: column "Distance DD" gives the number of *distance data dependence*, and column "Other DD" gives the number of data dependence relations that could *not* be represented by a distance dependence vector (more general affine form). Finally, the "Unknown" column gives the number of dependence tests that our analysis is unable to perform (e.g., with a $\top$ or polynomial TREC), and the associated percentage of all dependence tests.

Overall, 83.2% of the dependences were successfully resolved on the SpecFP benchmarks, 62.5% on SpecINT and 99.2% on JavaGrande. These results are also very accurate since 38.3% of the tests prove the absence of any dependence on SpecINT, 52.1% on SpecFP and 91.4% on JavaGrande. We also note that a majority of the remaining dependences are successfully represented as distance dependence vectors.

The cost of this dependence test is very low, although we perform thousands of tests per benchmark. In our experiments, the total computation time for scalar evolution and dependence analysis never exceeded 10% of the compilation time, with a mean overhead of less than 1%.

4.5 Vectorization and Modulo-Scheduling

Our dependence analyzer is used in the vectorization phase, currently in development in GCC, targeting the SIMD extensions of modern microprocessors. Selection of SIMD instructions require more efforts than tagging parallel loops. It requires accurate loop-trip count information and well behaved strided array access patterns. Our algorithm can provide some of the required information, in addition to the detection of loop-carried dependences. We are also beginning the implementation of the fission-based Kennedy and Allen algorithm [3, 10] to extract more vectorizable loops from reduced dependence graphs.

Other optimizations are on the way, relying on our method and implementation to compute dependence vectors, closed forms for array subscripts and loop-trip count. These works include the implementation of the Swing modulo-scheduling [23].

These developments already deliver performance on loop kernels, but are not robust enough to bring significant performance gains on full benchmarks.

5 Comparison with the most closely related works

Induction variable recognition consists in computing a closed form expression from a source intermediate representation. Induction variable detection has been studied extensively in the past because of its central role in loop optimizations, as described in the introduction of this paper.

Target intermediate representation Our target closed form expressions is an extension of the chains of recurrences algebra described first by Bachman, Wang and Zima [6], then used by van Engelen [32, 31]. This section emphasizes the technical motivations for revisiting the algorithm proposed by van Engelen, the main one being the different nature of the intermediate representation on which the analysis is performed.

Starting intermediate representation Unlike the analyzer described by van Engelen, the algorithm described in our paper analyzes a classic SSA representation [9, 33]. As we have seen, the analyzed representation does not contain informations about the syntactical loop constructs. The loop structures are directly detected together with the number of iterations from a low level representation of the program [1] close to the one promoted by Chris Lattner in the Low Level Virtual Machine [21]. One of the benefits of working on this low level SSA representation is that our analyzer contains much less cases and consequently is simpler to maintain than an analyzer that works on syntactic information such as the one described by van Engelen.

An interesting approach in analyzing a classic SSA representation has been proposed in [22] and implemented in the Open64 compiler. However this analyzer considers the syntactic information encapsulated in the SSA names of the variables. Having implemented an initial analyzer that considers this syntactic information, we have observed a huge amount of special cases that were detected as wrap around variables, but that were generated by the introduction of temporary variables during the lowering of the representation into the three address code. We would like to stress the fact that by not using the syntactic information at all, in other words considering versions of a same syntactic variable as different variables, is enough for decreasing the number of special cases and consequently the analyzer becomes simpler to write, to extend and to maintain. Our paper corrects this previous approach, and extends the technique to the detection of polynomials of larger degree, and to the detection and representation of more complex induction schemes.

Our starting representation is closer to the one used in the Open64 compiler [13, 22], but our algorithm avoids the syntactic case distinctions made in [22] that have severe consequences in terms of generality (when analyzing intricate SSA graphs) and maintainability.

Syntactic information In most of cases the syntactic information is altered by the translation of high level constructs into the low level representation, or by earlier optimizations that can insert variable names that are completely unrelated to the ones used in the original program. Considering syntactic information at such a low level leads to an explosion of the number of cases to be recognized (as an example, we can take the split of a simple recurrence into two unrelated names that would be detected as a wrap around if not handled correctly in a special case), and often the analysis algorithms have to approximate the result by not handling these special cases [22].

In the case of the algorithm described by van Engelen [32], the syntactic information can lead to wrong results when considering a sequence of loops as illustrated in figure 17. The analysis of the first loop leads to the result $i \rightarrow \{0, +, 1\}_1$. Then, analyzing the second loop, the initial condition is $i \rightarrow \{0, +, 1\}_1$, leading to a multivariate chain of recurrence: $i \rightarrow \{\{0, +, 1\}_1, +, 2\}_2$. However producing this chain of recurrence for two sequential (i.e. non nested) loops is violating the semantics of the chains of recurrences, and could lead to incorrect data dependence information. One of the solutions for correcting this problem is to insert an assignment at the end of each loop. Note that this is what the classic SSA representation does with ϕ nodes placed at the end of a loop.

```
for (i = 0; i < N; i+=1) { ... };
for (; i < M; i+=2) { ... };
```

Figure 17: Two sequential loops

Misuse of the SSA representation We would like to stress the fact that the analysis algorithm described by Van Engelen is not based on an SSA form. It is not even an "SSA-like" form as claimed in his papers. The first difference between the SSA and the representation considered in the algorithm described by van Engelen is that variable definitions are unique in the SSA form. We illustrate this anomaly by taking the example in Figure 18, where the program is not in SSA form because the variables j and k have two different definitions: one in the loop and another outside the loop, whereas in a SSA representation a loop-ϕ node would have been expected.

```
DO i=0,n
  ...
  j = j + h
  k = k + 2 * i
ENDDO
```

Figure 18: Example from [32] page 5.

Another problem with the algorithm described by van Engelen is that it considers a list of assignments sorted in a topological way, but all the information about the control flow is lost in this list. It is thus impossible to retrieve the context with respect to the CFG starting only from the assignments that are stored in that list. This bounds the capabilities of his algorithm and misleads the design of his extensions: as an example, we can take the extension of his analyzer for evolution envelopes where he has to consider in parallel the structure of the CFG and the list of statements. Moreover this design leads to an analyzer that is exponential in the number of control flow junctions in a loop.

Complexity issues Path-sensitive approaches have been proposed [31], to increase precision in the context of unpredictable control-flow. These techniques lead to an exponential number of paths, and although interesting in theory, seem unsuitable for a production compiler: even quadratic space complexity can be unacceptable on benchmarks like GNU Go [14].

Demand driven vs. whole program analysis. Because our analyzer has been designed around an on demand interface, it does not need a preprocessing analysis of the scalar dependence graph, and consequently it is faster than the algorithm described by van Engelen. Another benefit of not considering the scalar dependence graph separately, but discovering it during the analysis, is that the analyzer is able to handle more difficult cases such as the mixers and the exchangers. We have proposed appropriate extensions of the target representation for handling the results of the analyzer. Always compared to the algorithm of van Engelen, our analyzer is faster because it does not need to analyze all the scalar variables occuring in a loop, but only a subset that has been asked for by an optimizer. In fact, because these two algorithms work on different starting intermediate representations, and use different strategies for gathering the information, the only common part is the target representation, the chains of recurrences, that we have extended for representing more difficult cases that are left under the "fail" case by van Engelen.

Experimental comparison, robustness and maintainability of the code. We would like to express the difficulty of experimentally comparing our algorithm to the one described by van Engelen because of the missing stable implementation of their algorithm. We have started by implementing it, then due to deficiencies in their algorithm we have proposed several versions of the analyzer that are completely different than van Engelen's approach. We can comment on the early versions of the analyzer that contain some ideas from the analyzer described by van Engelen: in these analyzers we have experimented with separate scalar dependence graph topological orderings and detection of inductions based on the syntactic information. Some of these first versions that we have implemented are freely available in the CVS of the GCC project in the "lno-branch". It is interesting to note that the size of the analyzer was of about 4500 lines, and didn't dealt correctly with all the wrap-around cases. Once we have corrected the analyzer not to deal with the syntactic information, the size of the analyzer dropped down to less than 3000 lines of code, and from that we have reduced it to about to 2500 lines for inclusion in the mainline branch: GCC-4.0. The main benefit of not dealing with the syntactic information has been a big reduction on the number of cases to be handled, and the fact that the scalar dependence graph is irrelevant to be analyzed on an SSA representation.

We consider that all the induction variable analyzers that use the syntactic information and that work on a SSA representation, are misusing the properties of the SSA. The best example for illustrating our early versions of the analyzer that misuse the SSA representation is [22]. This paper depicts Open64's implementation of the induction variable detector. Unlike van Engelen's analyzer, Open64's analyzer was implemented on a true SSA form, and it was aimed to entirely exploit the benefits of this representation. However, one of the drawbacks of Open64's analyzer is the use of syntactic information for detecting scalar inductions.

6 Conclusion and Perspectives

We introduced *trees of recurrences*, a formalism based on *multivariate chains of recurrences* [6, 20] with several syntactic and algebraic extensions, such as the peeled and periodic chains of recurrences. These extensions increase the expressiveness of standard chains of recurrences and alleviate the need to resort to untractable exponential expressions to handle wrap-around and mixer induction variables.

We also presented a novel algorithm for the analysis of scalar evolutions. This algorithm is capable of traversing an arbitrary program in Static Single-Assignment (SSA) form, without prior classification of the induction variables. A proof of the algorithm is provided, by induction on the structure of the SSA graph. Unlike prior works, our method does not attempt to retrieve more complex closed form expressions, but focuses on generality: starting from a low-level three-address code representation that has been seriously scrambled by complex phases of data- and control-flow optimizations, the goal is to recognize simple and tractable induction variables whose algebraic properties allow precise static analysis, including accurate dependence testing. We have implemented and integrated our algorithm in a production compiler, showing the scalability and robustness of our algorithm.

This algorithm is well suited for the recognition of *piecewise affine, polynomial, and periodic evolutions*. Our framework is implemented in a development branch (4.0) of the GNU Compiler Collection (GCC): it is the basis for several optimizations being developed in this future version of GCC, including vectorization (for SIMD extensions), loop interchange and modulo-scheduling. We presented experimental results on the Spec CPU2000 and JavaGrande benchmarks, with an application to dependence analysis. Our results show no degradations in compilation time. In parallel with the algorithmic and formal contributions to induction variable recognition, this work is part of an effort to bring competitive loop transformations to the free production compiler GCC.

References

[1] Gcc internal manual. `http://gcc.gnu.org/`.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] J. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[5] W. Amme, N. Dalton, J. von Ronne, and M. Franz. SafeTSA: a type safe and referentially secure mobile-code representation based on static single assignment form. In *ACM Symp. on Programming Language Design and Implementation (PLDI'01)*, 2001.

[6] O. Bachmann, P. S. Wang, and E. V. Zima. Chains of recurrences a method to expedite the evaluation of closed-form functions. In *Proceedings of the international symposium on Symbolic and algebraic computation*, pages 242–249. ACM Press, 1994.

[7] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Boston, 1992.

[8] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Minet, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM Symp. on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, June 2003.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[10] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhaser, Boston, 2000.

[11] P. Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, pages 429–441, St. Malo, France, July 1988.

[12] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[13] M. P. Gerlek, E. Stoltz, and M. J. Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. on Programming Languages and Systems*, 17(1):85–122, Jan. 1995.

[14] Gnu go. `http://www.gnu.org/software/gnugo/gnugo.html`.

[15] M. Haghighat and C. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Trans. on Programming Languages and Systems*, 18(4):477–518, July 1996.

[16] L. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in LNCS, pages 406–420. Springer-Verlag, 1993.

[17] F. Irigoin, P. Jouvelot, and R. Triolet. Overview of the PIPS project. In P. Feautrier and F. Irigoin, editors, *2^{nd} Intl. Workshop on Compilers for Parallel Computers*, pages 199–212, Paris, Dec. 1990.

[18] Java grande forum. `http://www.javagrande.org`.

[19] KAP C/OpenMP for Tru64 UNIX and KAP DEC Fortran for Digital UNIX. `http://www.hp.com/techsevers/software/kap.html`.

[20] V. Kislenkov, V. Mitrofanov, and E. Zima. Multidimensional chains of recurrences. In *Proceedings of the 1998 international symposium on symbolic and algebraic computation*, pages 199–206. ACM Press, 1998.

[21] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *ACM Symp. on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.

[22] S.-M. Liu, R. Lo, and F. Chow. Loop induction variable canonicalization in parallelizing compilers. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, page 228. IEEE Computer Society, 1996.

[23] J. Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *Parallel Architectures and Compilation Techniques (PACT'96)*, 1996.

[24] J. Merill. GENERIC and GIMPLE: a new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers Summit*, pages 171–180, 2003. `http://www.gccsummit.org/2003`.

[25] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[26] D. Novillo. Tree SSA - a new optimization infrastructure for GCC. In *Proceedings of the 2003 GCC Developers Summit*, pages 181–193, 2003. `http://www.gccsummit.org/2003`.

[27] B. Pottenger and R. Eigenmann. Parallelization in the presence of generalized induction and reduction variables. In *ACM Int. Conf. on Supercomputing (ICS'95)*, June 1995.

[28] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, Aug. 1992.

[29] S. Rus, D. Zhang, and L. Rauchwerger. The value evolution graph and its use in memory reference analysis. In *Proceedings of the 2004 Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004.

[30] Standard performance evaluation corporation. `http://www.spec.org`.

[31] R. van Engelen, J. Birch, Y. Shou, B. Walsh, and K. Gallivan. A unified framework for nonlinear dependence testing and symbolic analysis. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 106–115, 2004.

[32] R. A. van Engelen. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'01)*, pages 118–132, 2001.

[33] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 13(2):181–210, 1991.

[34] M. J. Wolfe. Beyond induction variables. In *ACM Symp. on Programming Language Design and Implementation (PLDI'92)*, pages 162–174, San Francisco, California, June 1992.

[35] P. Wu, A. Cohen, and D. Padua. Induction variable analysis without idiom recognition: Beyond monotonicity. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, LNCS, Cumberland Falls, Kentucky, Aug. 2001.

[36] P. Wu, A. Cohen, D. Padua, and J. Hoeflinger. Monotonic evolution: an alternative to induction variable substitution for dependence analysis. In *ACM Int. Conf. on Supercomputing (ICS'01)*, Sorrento, Italy, June 2001.

[37] E. V. Zima. On computational properties of chains of recurrences. In *Proceedings of the 2001 international symposium on symbolic and algebraic computation*, pages 345–352. ACM Press, 2001.