Induction Variable Analysis with Delayed Abstractions

Sebastian Pop [1] Albert Cohen [2]
Georges-André Silber [1]

[1] Centre de recherche en informatique, École des mines de Paris, Fontainebleau, France
[2] ALCHEMY group, INRIA Futurs, Orsay, France

Abstract

This paper presents the design of an induction variable analyzer suitable for the analysis of typed, low-level, three address representations in SSA form. At the heart of our analyzer is a new algorithm recognizing scalar evolutions. We define a representation called trees of recurrences that is able to capture different levels of abstractions: from the finer level that is a subset of the SSA representation restricted to arithmetic operations on scalar variables, to the coarser levels such as the evolution envelopes that abstract sets of possible evolutions in loops. Unlike previous work, our algorithm tracks induction variables without prior classification of a few evolution patterns: different levels of abstraction can be obtained on demand. The low complexity of the algorithm fits the constraints of a production compiler as illustrated by the evaluation of our implementation on standard benchmark programs.

1 Introduction and Motivation

Supercomputing research has produced a wealth of techniques to optimize a program and tune code generation for a target architecture, both for uniprocessor and multiprocessor performance [34, 2]. But the context is different when dealing with common retargetable compilers for general-purpose and/or embedded architectures: the automatic exploitation of parallelism (fine-grain or thread-level) and the tuning for dynamic hardware components become far more challenging.

Modern compilers implement some of the sophisticated optimizations introduced for supercomputing applications [2]. 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 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 [15]. 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, these compilers require a huge implementation effort that cannot be matched by small development teams for general-purpose and/or free software compilers.

1.1 Loop Transformations on a Low-Level Representation

Conversely, several works demonstrated the interest of enriched low-level representation [3, 17]. They build on the normalization and simplicity of three address code, adding data types, *Static Single-Assignment* form (SSA) [7, 22] to ease data-flow analysis and scalar optimizations, and control and data annotations (loop nesting, heap structure, etc.). Starting from version 4.0, GCC uses such a representation called GIMPLE [24, 20], a three-address code derived from SIMPLE, the representation of the McCAT compiler [13]. 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 [33, 9, 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.*

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 [9, 32, 31], our method captures affine and polynomial closed forms without restrictions on the number and the intricateness of ϕ nodes. 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 [7, 22] 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. Throughout this paper, we will use a "generic" typed three-address code in SSA form [7]. Its syntax is simplified from the GIMPLE representation [20]. We consider only two control-flow primitives: a conditional expression `if`, and a goto expression `goto`. In addition, we include a loop annotation discovered from the control-flow graph [1]: *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 computed from the scalar evolution of variables involved in the loop exit conditions; this 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.

To illustrate the main issues and concepts, we consider several examples in Figure 1. The closed-form expression for `f` in the first example is not affine (a second-degree polynomial). In the second example, apart from `d`, all variables are *univariate*: they only depend on one loop counter; `d` is called *multivariate*. To compute the evolution of `c`, `x` and `d` in the second example, one must know the *trip count* of the inner loop, here 10 iterations. Yet, to statically evaluate the trip count of ℓ_2 one must already understand the evolutions of `c` and `d`. In the third example, variables `a` and `e` have simple affine evolutions, but `c` is a typical case of *wrap-around* variable [33], a variable that takes a special value at the first iteration and becomes an inductive variable with a regular evolution in 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. [9] does not handle wrap-around variables in nested loops where the initial value is varying in the outer loops. Indeed, the closed form representation for

```
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:
```

First example

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

```
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:
```

Second example

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

```
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:
```

Third example

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

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

Fourth example

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

```
loop (ℓ1)
  (unsigned char) a = φ(0, c);
  (int) b = φ(0, d);
  (unsigned char) c = a + 1
  (int) d = b + 1
  if (d >= 1000) goto end;
  T[b] = U[a];
end:
```

Fifth example

$$\begin{aligned} \mathtt{a} &= 0, 1, 2, \ldots, 255, 0, 1, \ldots = \ell_1 \ mod\ 256 \\ \mathtt{b} &= 0, 1, 2, \ldots, 999 = \ell_1 \end{aligned}$$

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

Sixth example

$$\begin{aligned} \mathtt{a} &= 0, 1, 2, \ldots, 126 = \ell_1 \\ \mathtt{d} &= 1, 2, 3, \ldots, 127 = \ell_1 + 1 \\ \mathtt{N} &\in [0, 126] \end{aligned}$$

Figure 1: Examples

such *multivariate* variables are more complex than the affine, polynomial or exponential cases considered in [33, 9]. Here the value of `c` is reinitialized to a different value at each iteration of the outer loop: in the first iteration of ℓ_1, its values in the inner loop are $1, 5, 11, 17$, then in the second iteration of ℓ_1, its values are $5, 9, 15, 21, \ldots$, etc. We show the closed form of `c` in Section 2.3, after the introduction of a representation that captures such evolutions. 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 [33], or may not even have a known closed form.[1] Figure 1 presents a fourth example where `a` and `b` have a linear closed form but derive from an intricate inductive definition scheme. Unlike our algorithm, previous works could not compute this closed form — although it is a linear one — due to the intricateness of the SSA graph. The fifth example illustrates an unusual data dependence problem. Variable `a` is incremented at each iteration of ℓ_1, however the unsigned char type constraints its evolution to the range $[0, 255]$. When language standards define modulo arithmetics for a type, the compiler has to handle the effects of wrapping overflows on induction variables. When the effect of overflowing operations is not defined by the language as wrapping, then based on the defined part of the domain, the compiler is allowed to deduce constraints on the loop bounds as illustrated in the sixth example.

[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.

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 data dependence analysis and loop transformation framework. Section 5 compares our method to other existing approaches. Finally, section 6 concludes and sketches 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 modified and extended version of *Multivariate* (a.k.a. multidimensional) *Chains of Recurrences* (MCR) [4, 16, 36, 32]. The expressive power is extended by symbolic references. MCR are obtained after an instantiation pass of all the varying symbols, defined as an abstraction operator: some evolutions are mapped to the "don't know" symbol $\top$. TREC correspond to a compressed part of the SSA graph uniquely dealing with scalar constants and symbols.

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: $\Theta = \{\Theta_a, +, \Theta_b\}_k$ or $\Theta = c$, where Θ_a and Θ_b are trees of recurrences and c is a constant. Subscript k indexes the varying dimension. 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.

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 [4].

Newton series. Efficient computation of MCR is based on Newton interpolation series. Considering a univariate chain of recurrences with c_0, c_1, c_2, $\ldots$, c_n, constant parameters (i.e., either scalar constants, or symbolic names defined outside loop k)

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

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 the first example of Figure 1 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 $\{3, +, 8, +, 5\}_1$. The 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 either by differencing the successive values taken by the scalar variable in successive loop iterations, and construct the differentiation table like Haghighat and Polychronopoulos [12]:

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			

or, avoid the construction of this differentiation table, by directly extracting the coefficients from the code [32]. We present our algorithm for extracting TREC coefficients from a classic SSA representation in Section 3.

Arithmetic operations. Arithmetic operations on TREC can be defined as on MCR using rewriting rules. For a complete table of rewriting rules on MCR we refer to [32].

Evaluation example. Considering the second introductory example (see Figure 1), 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 $\Theta(\ell_1, \ell_2) = \{\{3, +, 14\}_1, +, 1\}_2$, that can be evaluated for $\ell_1 = 10$ and $\ell_2 = 15$ as follows:

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

2.2 Instantiation of TREC and Abstract Envelopes

The analyzers use an instantiation pass for limiting the expressiveness of the TREC to cases that are simpler to handle. TREC capture a larger class of scalar evolutions than MCR because TREC may contain symbols referring to other scalar evolutions. The symbolic MCR restrict the use of symbols to variables that are defined outside the loop where the variable is varying: the symbols stand for constants with respect to the varying loop. As an example of TREC that is not a MCR, consider the Fibonacci sequence that define the simplest case of the class of mixers: $fib \to \{0, +, 1, +, fib\}_k$. Optimizers such as symbolic propagation could handle such difficult constructs, however they lead to problems that are difficult to solve (e.g. determining the number of iterations of a loop whose exit edge is guarded by a Fibonacci sequence). Because a large class of optimizers and analyzers are expecting simpler cases, TREC information is filtered using an instantiation pass. Several abstractions can be defined, such as mapping every non polynomial scalar evolution to $\top$, or even more practically, mapping non affine functions to $\top$. In appropriate cases, it is natural to map uncertain values to an abstract value. We have experimented instantiations of TREC with intervals, in which case we obtain a set of possible evolutions that we call an envelope. Allowing the coefficients of TREC to contain abstract scalar values is a more natural extension than the use of max and min functions over MCR as proposed by van Engelen in [31] because it is then possible to define other kinds of envelopes using classic scalar abstract domains, such as polyhedra, octagons [21], or congruences [11].

2.3 Peeled Trees of Recurrences

A common case, quite frequently used by programmers consists in variables that contain a value used during the first iteration of a loop, that is replaced by the values of another induction variable for the rest of iterations. 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. Most closed forms for wrap-around variables [33] are peeled TREC. Indeed, back to the third introductory example (see Figure 1), the closed form for `c` can be represented by a peeled multivariate affine TREC: $(\{1, +, 4\}_1, \{\{5, +, 4\}_1, +, 6\}_2)_2$. A peeled TREC describes 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 tries to add a new column to the differentiation table without changing the last element in that column. Since this first column contains the coefficients of the TREC, the transformation is possible if it does not modify the last coefficient of the column. This is illustrated in Figure 2. 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 2: 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: $b_{n-1} = a_{n-1} - a_n$, $b_{n-2} = a_{n-2} - b_{n-1}$, $\ldots$, $b_1 = a_1 - b_2$, we say that r is equivalent to p if and only if $b_1 = v$.

The importance of this method is illustrated by the number of cases that occur in benchmarks: in the SPEC CPU2000 we have found 29 wrap around loop-ϕ that can be unified, on the GCC code itself we have found 337 unification opportunities, and on the JavaGrande benchmarks we have found 5 occurrences. In all these cases, the subsequent passes that are using TREC information are ineffective if the unification is not performed: they conservatively reject difficult constructs such as the wrap around evolutions.

2.4 Periodic Trees of Recurrences

Periodic sequences may be generated by the wrapping semantics of overflowing, as illustrated by the fifth example of Figure 1, and as we will see in Section 3.3. But they can also be produced by flip-flop operations, that are special cases of self referenced peeled TREC. Variables in a flip-flop exchange their initial values over the iterations (see [26]), for example:

$$flip \rightarrow (3, 5, flip)_k(x) = [3, 5]_k(x) = \begin{cases} 3 & \text{if } x = 0 \text{ mod } 2, \\ 5 & \text{if } x = 1 \text{ mod } 2. \end{cases}$$

2.5 Exponentials and Other Extensions

We have conducted some experiments using the extensions of trees of recurrences to exponential functions. However these extensions are not as critical to a generic production compiler such as GCC, as a fast generic algorithm for detecting affine functions. Loops iterating over the powers of 2 are frequently used in bit operations, but they are handled by a different technique than the exponential MCR used by van Engelen [32]. We do not think that evolutions that are mixing polynomial and exponential evolutions [31] are important in compiler technology because these cases are not frequent enough. The absence of mixed polynomial and exponential evolutions in real life code is probably linked to the fact that overflowing semantics are hard to control even for the simpler cases of affine functions.

Wolfe [33] showed a classification method to recognize some cases of *exponential* evolutions and 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].

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. Several constraints have led the design of our analyzer.

First, our algorithm does not assume a particular control-flow structure and makes no restriction on the recursive intricate variable definitions. It however fails to detect any meaningful induction variable on irreducible control flow graphs that cannot be analyzed into natural loop structures [1]. 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. Furthermore, the algorithm is able to delay a part of the analysis until more information is known, by leaving symbolic names in the representation. The representation that is obtained from AnalyzeEvolution function is the most instantiated with respect to the instantiation context, that is, no early approximations have been performed. Based on this representation, symbolic solvers, as for example the computation of the number of iterations in a loop, may produce safe and precise informations that improve the information available in the instantiation context. The last constraint that is important for the inclusion of an implementation of the algorithm in a production compiler is that the analyzer should be linear in time and space. In order to satisfy this constraint and to allow further possible refinements we provide an interface that provides different levels of abstractions by several instantiation functions. The structure of our algorithm is quite complex because it is based on a double recursion as sketched in Figure3. It presents similarities with the algorithm for linear unification by Paterson and Wegman [25], where the double recursion is hidden behind a single recursion with a stack structure. We will further investigate these similarities in a later paper.

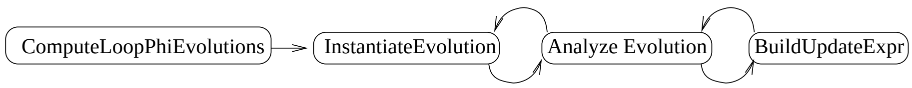

Figure 3: Bird's eye view of the analyzer

3.1 Algorithm

Figure 4 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 or the evaluation of loop-trip count. All these applications call directly the analyzer: AnalyzeEvolution for a given loop number and a variable name. The evolution functions are stored in a database that is visible only to 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.

```
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 loop nest
    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 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 4: Algorithm to compute scalar evolutions.

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, $\perp$), 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, Apply is used 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 proof of this algorithm 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 proof exploits 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 (for update expressions) and loop-trip counts. See the associated technical report [26] for details.

3.2 Application of the Analyzer to the Introductory Examples

We illustrate the analysis of scalar evolutions algorithm on the first two introductory examples in Figure 1. In addition to clarifying the depth-first search and instantiation phases of the algorithm, this will exercise the recognition of polynomial and multivariate evolutions.

```
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) Searching for “c”

```
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 “return path”

Figure 5: Application to the first example

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 5.(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 5.(2). The update edge `c`→`f` is followed 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 `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 5.(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 5.(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 $\{\mathtt{a}, +, \mathtt{e}\}_1$ starts with the substitution of the first operand: $\mathtt{a} = 3$, 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: $\{8, +, 5\}_1$, and replacing `e` with its evolution finishes the instantiation of the TREC of `c` that yields $\{3, +, 8, +, 5\}_1$.

Second example. We will now compute the evolution of `x` in the second example of Figure 1, 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, the edge `e`→`d` is followed to its definition in loop 2, ending on the definition of a loop-ϕ node. At this point, `d` is known to be 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`: the edge `e`→`d` is followed to the loop-ϕ node from which the search has started. 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. The edge to the initial condition is ignored, and the update edge is taken, 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`, ending on a statement of the loop 2. Again, edge `e`→`d` is followed, ending on the definition of `d` that has already been 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. Finally, the starting loop-ϕ node `c` is reached. 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. The symbolic TREC of `c` has been computed: $\{\mathtt{a}, +, 14\}_1$. The last step consists in propagating this information from the loop-ϕ node of `c` to the node where the computation has started: `x`. Back from `c` to `d`, the TREC for `d` can partially be instantiated: $d \rightarrow \{\{\mathtt{a}, +, 14\}_1, +, 1\}_2$. Then back to `e = d + 1`, $e \rightarrow \{\{\mathtt{a} + 1, +, 14\}_1, +, 1\}_2$; and finally back to `x = e + 3`, $x \rightarrow \{\mathtt{a} + 14, +, 14\}_1$. A final instantiation yields $x \rightarrow \{17, +, 14\}_1$

As can be seen in these examples, the scalar evolution information gathered on demand is reused several times, and for this reason it is stored in a cache for avoiding the computation in later queries. This is an important practical design because the scalar evolution information is used by all the loop nest optimizers, from the scalar variable optimizations to the data dependence analyzers whose results are later used in auto-vectorization or loop transforms. A drawback of using a database is that a part of the information might be invalidated by the loop optimizers: when one of the variables is removed or renamed in the SSA graph, all the scalar evolutions that contain this name have to be invalidated and analyzed again. In practice, it is faster to erase all the results and to start the analysis again. This is not a final result, and we probably will use incremental updates for saving results more difficult to obtain, as for example the results of an inter-procedural extension of the analyzer (see the associated technical report [26] for details).

3.3 Typed induction variables

Induction variable analysis in the context of typed operations is not new: all the compilers that have loop optimizers for typed intermediate representations have solved this problem. However there is little literature that describes the problems and solutions: these details are often considered too low level, and language dependent. As illustrated in the fifth introductory example in Figure 1, the analysis of data dependences has to correctly handle the effects of overflowing on variables that are indexing the data. The C language defines wrapping semantics on overflow for the unsigned types, and leaves the semantic of overflowing signed types undefined. A compiler can propose a flag for extending the wrapping semantics to all the types, as for example the option `-fwrapv` of GCC. This wrapping behavior has to be preserved during a conversion to another type, and basically the following cases have to be considered separately: conversions to narrower or to wider types, and conversions between signed and unsigned types. One of the solutions is to type the TREC and to map the effects of type conversions from the SSA representation to the TREC representation. For example, the conversion from unsigned char to unsigned int of TREC `{(uchar) 100, +, (uchar) 240}_1` is `{(uint) 100, +, (uint) 0xfffffff0}_1`. `0xfffffff0` is the only possible value for the step in the range of unsigned int that keeps the original sequence unchanged $(100, 84, 68, \ldots)$. This conversion has to be performed carefully with respect to the number of iterations of loop 1 in order to ensure that the original TREC does not wrap. If the number of iterations in loop 1 is greater than 6, the converted TREC should also contain a wrap modulo 256, as illustrated by the sequence of the original TREC $(100, 84, 68, 52, 36, 20, 4, 244, 228, \ldots)$. A solution is to keep the cast and avoid the conversion: `(uint) ({(uchar) 100, +, (uchar) 240}_1)`. Another possible solution is to use periodic TREC (see Section 2.4), but this does not seem to be practical because all the values of a period have to be listed, and as we have seen in the previous example we would have to store 15 values.

3.4 Empirical Study

To show the robustness and language-independence of our implementation, and to evaluate the accuracy of our algorithm, we determine a compact representation of *all* variables defined by loop-ϕ nodes in the SPEC CPU2000 [30] and JavaGrande [14] benchmarks.

Figure 6 summarizes our experiments: affine univariate variables are very frequent because well structured loops are most of the time using simple constructs, affine multivariate are also quite frequent because they are used for iterating over multi dimensional arrays. As one could expect, difficult to understand (or even to read) constructs such as polynomials of degree greater or equal to two occur very rarely: we have detected only three occurrences in SPEC CPU2000, and none in JavaGrande. Even if their detection does not involve more computation, it makes the compiler more complex with no profit, leading to code maintaining problems. We plan to restrict the analyzer included in the production compiler to only deal with affine evolutions. The remaining cases have to be refined following the needs of optimizations or analyzers.

The last four columns in Figure 6 show the precision of the detector of the number of iterations. For the moment, only the single-exit loops are exactly analyzed, excluding a big number of loops that contain irregular control flow (probably containing exception exits) such as in the case of java programs. The effectiveness of the loop transforms is reduced because a large number of loops are not correctly analyzed. In some cases an approximation of the loop count can enable aggressive loop transformations

CINT2000	Total	A.U.	A.M.	O.P.	Other	$\top$	Loops	Trip	Approx.	$\top$
164.gzip	1138	285	0	119	144	590	223	90	4	55
175.vpr	4084	1384	0	321	575	1804	505	109	0	207
176.gcc	18403	2835	4	1646	1636	12282	3506	663	25	982
181.mcf	80	21	0	7	38	14	72	2	0	32
186.crafty	2503	513	6	101	386	1497	464	119	13	192
197.parser	2220	564	0	120	450	1086	786	67	22	308
252.eon	2115	601	0	164	146	1204	541	186	6	49
253.perlbmk	8048	919	1	791	380	5957	993	155	0	297
254.gap	26913	3782	9	3051	1115	18956	2143	350	10	773
255.vortex	3106	479	0	308	278	2041	332	19	2	81
256.bzip2	42	26	0	4	2	10	14	2	0	4
300.twolf	10536	1577	0	895	849	7215	1014	47	0	699
CFP2000	**Total**	**A.U.**	**A.M.**	**O.P.**	**Other**	$\top$	**Loops**	**Trip**	**Approx.**	$\top$
168.wupwise	235	94	0	12	123	6	84	70	0	1
171.swim	191	84	0	26	22	59	32	25	1	0
172.mgrid	962	165	0	420	157	220	71	58	0	0
173.applu	1808	645	5	630	199	329	184	150	0	12
177.mesa	10308	4100	1	1304	1123	3780	1158	689	34	124
179.art	496	213	2	72	64	145	100	38	0	33
183.equake	1212	438	2	104	102	566	105	52	1	24
188.ammp	1632	649	6	102	286	589	546	169	14	34
189.lucas	1901	250	0	425	121	1105	109	68	1	31
191.fma3d	6759	3691	14	593	1112	1349	2835	1971	17	493
200.sixtrack	8549	1723	17	2298	781	3730	1109	563	0	274
301.apsi	3986	1087	5	1190	785	919	387	284	0	15
JavaGrande v2.0	**Total**	**A.U.**	**A.M.**	**O.P.**	**Other**	$\top$	**Loops**	**Trip**	**Approx.**	$\top$
section1	777	93	0	0	396	288	201	79	0	77
section2	311	121	0	20	19	151	107	2	0	3
section3	567	120	0	2	18	427	173	3	0	6

Figure 6: Scalar induction variables and loop trip count in SPEC CPU2000 and JavaGrande benchmarks. Break-down of scalar evolutions into: "A.U." affine univariate, "A.M." affine multivariate, "O.P." other kinds of polynomials, "Other" other compound expressions containing determined components, and "$\top$" undetermined evolutions. Last columns describe: "Loops" the number of *natural* loops, "Trip" the number of *single-exit* loops whose trip count is successfully analyzed, "Approx." the number of loops for which an upper bound approximation of the trip count is available.

as is the case of the swim test in SPEC CPU2000: the size of data accessed in the loop is used to provide an upper bound estimation of the number of iterations, allowing the dependence analyzer to correctly refine the dependence relations, and the loop interchange to be performed. Further refinements of this analyzer will either provide more hints for approximating the number of iterations, or use an integer programming solver, such as Pugh's Omega solver [28], for precisely computing or approximating the number of iterations in multiple-exit loops.

4 Applications

Our scalar evolution algorithm is integrated in a dependence analysis pass of GCC applied to scalar loop optimizations, vectorization [8, 23], and a generic framework for high level loop transformations [18, 6]. These dependence-based transformations use uniform dependence vectors [5], but our method for identifying conflicting accesses between TREC is applicable to the computation of more general dependence abstractions as well. We implemented an extended Banerjee test [5]. Tests for periodic, polynomial, exponential or envelope TREC are also applicable to our framework, but we did not include them in our current experiments for lack of a robust and scalable implementation.

In order to show the effectiveness of the data dependence analyzer as used in an optimizer, we have measured the compilation time of the vectorization pass that make use of the data dependence information. For SPEC CPU2000 benchmarks, the vectorization pass does not exceed 1 second per compiled file, nor 5 percent of the compilation time per file, showing that the dependence analyzer is fast in practice. The experiments were performed on a Pentium4 2.40 GHz with 512 Kb of cache, 2 GB of RAM, on a Debian Sarge with a Linux kernel 2.6.8. Our implementation performs the analysis on demand: a reduced part of the information is computed 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 7 shows an evaluation of the precision of the information provided by the analyzer while computing the data dependence relations for all the array accesses in a function. Trivial dependence tests, such as when independence is deduced from accesses to different arrays, are not included in this table. First column is the name of the benchmark, the second column gives the total number of dependence tests. Following columns give: the number of tests that have been classified as dependent, independent, undetermined (either because the analyzer is too weak, or because the test is not decidable at compile time). Following columns split the dependence tests into zero induction variable "ZIV" tests (i.e. both array are referenced by constant functions with respect to the analyzed loop nest), single induction variable "SIV" tests and multiple induction variables "MIV" tests [2]. We have to stress that this last evaluation is quite artificial because an optimizer, such as the vectorizer, would focus the data dependence analysis only on a few loop nests. The number of dependence tests and the MIVcolumn witness the stress on the dependence analyzer: an important number of tests involve arrays accessed in different loops, that could be successive loops separated by an important number of statements. Even with these extreme test conditions, our data dependence analyzer catches an important number of dependence relations, and the worst case of computation time for SPEC CPU2000 is 15 seconds and 70 percent of the compilation time.

Based on our induction variable analysis, several scalar optimizations have been contributed by Zdeněk Dvořák from SuSE: strength reduction, induction variable canonicalization and elimination, loop invariant code motion [1] enabled by default with the option `-O2`. In order to use the vector units, the "simdization" pass [8] recognizes loop patterns that can be rewritten using SIMD instructions for Altivec, SSE or MMX. This pass is enabled with the option `-ftree-vectorize` and has been contributed by Dorit Naishlos [23] from IBM Haifa. A linear loop transformation framework has been contributed

CINT2000	# tests	d	i	u	ZIV	SIV	MIV
164.gzip	2130	228	301	1601	380	53	43
175.vpr	1465	362	452	651	542	147	69
176.gcc	91493	29022	30040	32431	52955	2041	2783
181.mcf	147	6	1	140	2	2	1
186.crafty	38387	5967	8544	23876	12354	1210	705
197.parser	1609	296	40	1273	103	45	44
252.eon	97386	34474	62080	832	96369	1078	19
253.perlbmk	12345	733	534	11078	993	32	693
254.gap	35084	1273	1798	32013	3065	495	725
255.vortex	15303	129	70	15104	138	10	7
256.bzip2	13	1	0	12	0	0	0
300.twolf	7873	689	1404	5780	2041	188	45
CFP2000	**# tests**	**d**	**i**	**u**	**ZIV**	**SIV**	**MIV**
168.wupwise	818	6	0	812	0	0	6
171.swim	677	212	57	408	9	29	197
172.mgrid	2014	39	20	1955	23	17	0
173.applu	32786	3312	9109	20365	450	1074	13081
177.mesa	302103	14414	19600	268089	26379	5705	38411
179.art	168	51	19	98	49	6	8
183.equake	845	417	216	212	308	176	210
187.facerec	9185	284	102	8799	206	76	25
188.ammp	19578	3443	8846	7289	10899	1170	692
189.lucas	93946	221	31	93694	74	86	26
191.fma3d	54013	14841	18448	20724	19874	6867	4029
200.sixtrack	130773	9657	41939	79177	46935	2260	3518
301.apsi	8149	1006	295	6848	223	434	340
JavaGrande v2.0	**# tests**	**d**	**i**	**u**	**ZIV**	**SIV**	**MIV**
section1	73028	9247	63767	14	72999	0	14
section2	1926	386	430	1110	212	411	83
section3	12185	3724	3169	5292	3043	2230	819

Figure 7: Classification of data dependence tests in SPEC CPU2000 and JavaGrande. Columns "d", "i" and "u" correspond to the number of tests that have been classified as dependent, independent, and undetermined. Last columns split the dependence tests into zero induction variable "ZIV", single induction variable "SIV" and multiple induction variable "MIV".

by Daniel Berlin from IBM Research and Sebastian Pop, but for the moment only the loop interchange transformation is supported [6]. We have experimented with these transformations on SPEC CPU2000 171.swim benchmark, for which a critical loop is interchanged: on peak it obtains 1320 points compiled with `-O2 -ftree-loop-linear` compared to the base compiled with `-O2` at 796 points represents a 65.83% benefit.

5 Comparison with the Most Closely Related Works

Induction variable detection has been studied extensively in the past because of its central role in loop optimizations. Our target closed form expressions is an extension of the chains of recurrences algebra described first by Bachman, Wang and Zima [4], then used by van Engelen [32, 31]. The representation analyzed by our algorithm is closer to the one used in the Open64 compiler [9, 19], but our algorithm avoids the syntactic case distinctions made in [19] that have severe consequences in terms of generality (when analyzing intricate SSA graphs) and maintainability. Syntactic information is altered by the translation to low-level representations, or by earlier optimizations that may insert new variables or eliminate redundancies in array subscript expressions. Pattern matching at a low level may lead to an explosion of the number of cases to be recognized; e.g., if a simple recurrence is split across two variables, its evolution would be detected as a wrap around if not handled correctly in a special case; in practice the analysis would have to approximate the result by not handling these special cases [19]. Path-sensitive approaches have been proposed [31, 29] to increase precision in the context of conditional variable updates. These techniques may lead to an exponential number of paths, and although interesting, seem not yet suitable for a production compiler, where even quadratic space complexity is unacceptable on benchmarks like GNU Go [10].

Our work is based on the previous research results presented by Robert van Engelen in the early paper [32]. We have experimented with similar algorithms and dealt with several restrictions and difficulties that remained unsolved in his later papers. For example, loop sequences as illustrated in Figure 8 are not correctly handled. The analysis of the first loop yields $i \to \{0, +, 1\}_1$. Then, analyzing the second loop,

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

Figure 8: Two sequential loops

the initial condition is $i \to \{0, +, 1\}_1$, leading to a MCR: $i \to \{\{0, +, 1\}_1, +, 2\}_2$. However producing this chain of recurrence for two sequential — *not* nested — loops is violating the semantics of the chains of recurrences. This case cannot be correctly handled without inserting at the end of each loop an assignment for each scalar variable that is modified in the loop and then used after the loop, as is the case in the loop closed SSA form. Because van Engelen is using a representation that is not in SSA form, he has to deal with all the difficulties of building an "SSA-like" form. With some minor changes, as the one described above, their algorithm can be seen as a translation from an unstructured list of instructions to a weak SSA form, that is a restriction of the original program to operations on scalars. Constructing a weak SSA form could be of great interest for representations that cannot efficiently be translated to a classic SSA form (for example the RTL representation of GCC). Another interesting result for their algorithm would be a proof that constructing a weak SSA representation is faster than building the classic SSA representation. However, they have not presented experimental results on real codes or standard benchmarks for showing the effectiveness of their approach.

In contrast, our algorithm is analyzing a classic SSA representation, and instead of worrying about the expressiveness power of the intermediate representation, we are more concerned about a completely opposite question: how to limit the expressiveness of the SSA representation in order to provide the optimizers and the analyzers the right level of abstraction that they can process. It might well be argued that a new representation is not necessary for concepts that can already be expressed in the SSA representation. This point is well taken. We acknowledge that we could have presented the current algorithm as a transformer from SSA to an extended version of the SSA containing abstract elements. However, we deliberately have chosen to present the analyzer producing trees of recurrences for highlighting the sources of our inspiration and for presenting the extensions that we proposed to the chains of recurrences. Finally, we wanted the algorithm presented in this paper to reflect the underlying implementation in GCC.

6 Conclusion and Perspectives

We introduced *trees of recurrences*, a formalism based on *multivariate chains of recurrences* [4, 16], with symbolic and algebraic extensions, such as the peeled chains of recurrences. These extensions increase the expressiveness of standard chains of recurrences and alleviate the need to resort to intractable exponential expressions to handle wrap-around and mixer induction variables. We extended this representation with the evolution envelopes that handle abstract elements as approximations of runtime values.

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. The algorithm is proven 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 an implementation of our algorithm.

This algorithm is well suited for the recognition of *piecewise affine and polynomial evolutions*. Our framework is implemented in a production compiler the GNU Compiler Collection (4.0): it is the basis for several optimizations being developed, including vectorization, loop transformations 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. Independently of 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] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

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

[3] 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.

[4] 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.

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

[6] D. Berlin, D. Edelsohn, and S. Pop. High-level loop optimizations for GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 37–54, 2004. `http://www.gccsummit.org/2004`.

[7] 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.

[8] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 82–93. ACM Press, 2004.

[9] 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.

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

[11] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT '91: Proceedings of the international joint conference on theory and practice of software development on Colloquium on trees in algebra and programming (CAAP '91): vol 1*, pages 169–192, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

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

[13] 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.

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

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

[16] 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.

[17] 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.

[18] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *Intl. J. of Parallel Programming*, 22(2):183–205, April 1994.

[19] 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.

[20] 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`.

[21] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

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

[23] D. Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004. `http://www.gccsummit.org/2004`.

[24] 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`.

[25] M. S. Paterson and M. N. Wegman. Linear unification. In *STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186, New York, NY, USA, 1976. ACM Press.

[26] S. Pop, P. Clauss, A. Cohen, V. Loechner, and G.-A. Silber. Fast recognition of scalar evolutions on three-address ssa code. Technical Report A/354/CRI, Centre de Recherche en Informatique (CRI), École des mines de Paris, 2004. `http://www.cri.ensmp.fr/classement/doc/A-354.ps`.

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

[34] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[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] 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.