Fast Recognition of Scalar Evolutions on Three-Address SSA Code

Sebastian Pop [1], Philippe Clauss [3], Albert Cohen [2], Vincent Loechner [3], and 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

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

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 [29, 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 [13]. 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, 15]. They build on the normalization and simplicity of three address code, adding data types, *Static Single-Assignment* form (SSA) [6, 19] 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 [20, 18], a three-address code derived from SIMPLE, the representation of the McCAT compiler [10]. 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 [28, 7, 22] 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.*

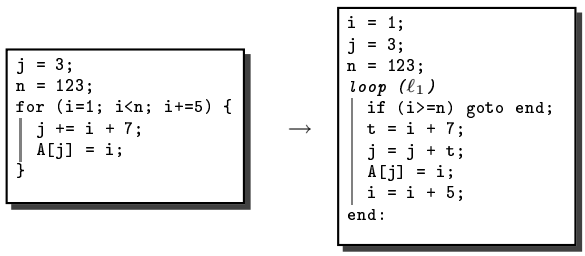


Fig. 1. Lowering to three-address code

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.

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

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 [28, 7, 9, 30, 26, 25].

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 [27] 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. [28, 7, 16] (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 [9, 26, 25, 23] would succeed as well, but their symbolic differencing or substitution strategies have a higher overhead. Finally, the first technique by Wu et al. [31] would fail due to the multiple intermediate assignments in the evolution path associated with the inductive definition of variable `j`; yet the second one [30] 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 [7, 26, 25], 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 [6, 19] 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:
```

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

Fig. 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:
```

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

Fig. 3. Second example

Throughout this paper, we will use a "generic" three-address code in SSA form [6]. Its syntax is simplified from the GIMPLE representation [18]. 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

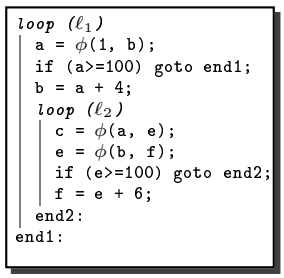


Fig. 4. Third example

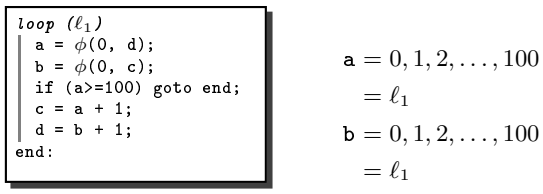


Fig. 5. Fourth example

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 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 describes 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 [28], 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 [28, 7]. 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 [28], or may not even have a known closed form.[1] Figure 5 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 intrication of the SSA graph.

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.

1.3 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 algorithm is a key component of a wider effort to bring advanced loop optimizations to the GNU Compiler Collection (GCC). Its effectiveness is assessed by the application to dependence testing: 83.2% of the dependences were successfully resolved on the SpecFP 2000 benchmarks [24], 62.5% on SpecINT 2000 [24] and 99.2% on JavaGrande [12]. Our method is also scalable since the total computation time for scalar evolution and dependence analysis never exceeds a few percents 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) [4, 14, 32, 26], with equivalent expressive power but more appropriate to implement algebraic operations on scalar evolutions.

Let $F(\ell_1, \ell_2, \ldots, \ell_m)$ — or $F(\boldsymbol{\ell})$ 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 $\boldsymbol{\ell}$. 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 $\boldsymbol{\ell}$ will be defined in Section 2.1. The syntax of a TREC is derived from MCR and inductively defined as:

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

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 [14], 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.

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

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 $\Theta(\boldsymbol{\ell}) = c$, otherwise, Θ is of the form $\{\Theta_a, +, \Theta_b\}_k$ and

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

Intuitively, the evaluation of $\{\Theta_a, +, \Theta_b\}_k$ for a given vector $\boldsymbol{\ell}$ 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 $\Theta(\boldsymbol{\ell})$ with a linear time and space complexity — in terms of arithmetic operations. The proof and computation process is adapted from the MCR framework [4].

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,

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

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 [9] 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:

$$\{\{\Theta_a, +, \Theta_b\}_k, +, \Theta_c\}_k \to \{\Theta_a, +, \Theta_b + \Theta_c\}_k,$$
$$\{\{\Theta_a, +, \Theta_b\}_k, +, \Theta_c\}_{k'} \to \{\{\Theta_a, +, \Theta_c\}_{k'}, +, \Theta_b\}_k.$$

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 [4], 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.

Most closed forms for wrap-around variables [28] 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			

Fig. 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 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 [28] 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 [26]. Wolfe also introduced the class of *monotonic* induction variables, his method and definition being extended by Wu et al. [30] 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 [25, 23]. 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.

Several constraints lead to the design of our analyzer. 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 [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.

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 [28]. 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).

To prove this algorithm, it is sufficient to check for termination (similar to depth-first search) and that the semantics of the SSA representation is captured by the computed trees of recurrences. The proof 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. In addition, peeled TREC are built directly from the syntax of the loop-ϕ nodes. The rest of 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 [21] for details.

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] $\neq \bot$ 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` $\odot$ `b`" (with $\odot \in \{+, -, *\}$) Then
    res ← AnalyzeEvolution(l, `a`) $\odot$ 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 ← $(\texttt{a}, \texttt{b})_l$
    Else If $update$ is $\top$ Then
      res ← $\top$
    Else
      res ← $\{\texttt{a}, +, update\}_l$
  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 ← $\top$
  Else
    res ← $\top$
  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, $\bot$),
  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, $\bot$)
    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, $\top$)
    ($exist$, $update$) ← BuildUpdateExpr(h, `b`)
    If $exist$ Then Return (true, $\top$)
  Return (false, $\bot$)

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 $\{e_1, +, e_2\}_x$ Then
    i_1 ← InstantiateEvolution(e_1, l)
    i_2 ← InstantiateEvolution(e_2, l)
    Return $\{i_1, +, i_2\}_x$
  Else If $trec$ is of the form $(e_1, e_2)_x$ Then
    i_1 ← InstantiateEvolution(e_1, l)
    i_2 ← InstantiateEvolution(e_2, l)
    Return UnifyPeeled($(i_1, i_2)_x$)
  Else Return $\top$

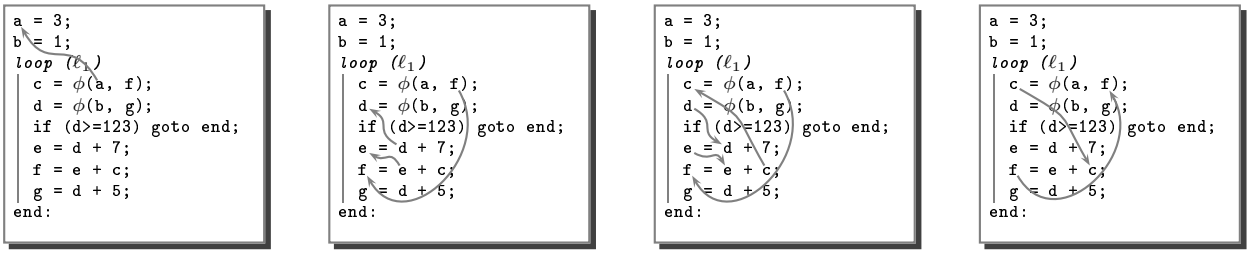
Fig. 7. Algorithm to compute scalar evolutions

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 `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).



(1) Initial condition edge (2) Search "c" (not found) (3) Found the halting phi (4) On the "return path"

Fig. 8. Application to the first example

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 $\{\{a,+,14\}_1,+,1\}_2$. Then back to `e = d + 1` we get a symbolic TREC for `e`: $\{\{a+1,+,14\}_1,+,1\}_2$; and finally back to `x = e + 3`, we get a symbolic TREC for `x`: $\{a+14,+,14\}_1$. A final instantiation of `a` yields the closed form of `x` and all other variables.

3.3 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 [24] and JavaGrande [12] 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.

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

Fig. 9. Types of TREC representing scalar evolutions in Spec CPU2000 and JavaGrande benchmarks

Computation of scalar evolutions. Figure 9 summarizes our experiments. The first column is the benchmark name. 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 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 [7] 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, hence further motivates our extension of the chains of recurrence formalism.

Loop-trip count. The last three columns in Figure 9 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

Our scalar evolution algorithm is integrated in a dependence analysis pass of GCC applied to vectorization and modulo scheduling [17]. These dependence-based transformations use uniform dependence vectors [5], but our method identifying conflicting accesses between trees of recurrences is applicable to the computation of more general dependence abstractions as well.

Other applications, not addressed here, include dead-code elimination [21], interprocedural region and dependence analysis [11, 21], closed-form substitution [7, 22] — for scalar dependence removal — and strength-reduction, based on the efficient run-time evaluation of MCR [14].

Retrieving dependence information from affine TREC of individual array references is no different from the classical dependence analysis of affine subscripts. Existing dependence analysis frameworks are directly applicable: we implemented an extended Banerjee test unifying all cases of multivariate affine evolutions [5]. Our implementation handles multidimensional arrays and peeled TREC. Tests for periodic, polynomial, exponential or envelope chains of recurrences [25] are also applicable to our framework, but we did not include them in our current experiments for lack of a robust and scalable implementation.

Our implementation performs the analysis 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 10 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%.

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

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%

Fig. 10. Dependence analysis results on Spec CPU2000 and JavaGrande benchmarks

role in loop optimizations, as described in the introduction of this paper. 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 [26, 25].

Our representation is closer to the one used in the Open64 compiler [7, 16], but our algorithm avoids the syntactic case distinctions made in [16] 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 leads 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 [16].

The algorithm described by van Engelen [26] is much simpler than ours but has an additional caveat due to the lack of control-flow information: it may lead to wrong results when considering a sequence of loops, as illustrated in figure 11. The analysis of the first loop yields $i \rightarrow \{0, +, 1\}_1$. Then, analyzing the second loop, the initial condition is $i \rightarrow \{0, +, 1\}_1$, leading to a MCR: $i \rightarrow \{\{0, +, 1\}_1, +, 2\}_2$. However producing this chain of recurrence for two sequential — *not* nested — loops is violating the semantics of the chains of recurrences, 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.

Path-sensitive approaches have been proposed [25, 23] 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: even quadratic space complexity can be unacceptable on benchmarks like GNU Go [8].

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

Fig. 11. Two sequential loops

6 Conclusion and Perspectives

We introduced *trees of recurrences*, a formalism based on *multivariate chains of recurrences* [4, 14], with syntactic 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 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. 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 our algorithm.

This algorithm is well suited for the recognition of *piecewise affine and polynomial 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, including vectorization, 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. 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. 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.
7. 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.
8. Gnu go. `http://www.gnu.org/software/gnugo/gnugo.html`.
9. M. Haghighat and C. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Trans. on Programming Languages and Systems*, 18(4):477–518, July 1996.
10. 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.

11. F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *ACM Int. Conf. on Supercomputing (ICS'2)*, Cologne, Germany, June 1991.
12. Java grande forum. `http://www.javagrande.org`.
13. KAP C/OpenMP for Tru64 UNIX and KAP DEC Fortran for Digital UNIX. `http://www.hp.com/techsevers/software/kap.html`.
14. 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.
15. 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.
16. 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.
17. J. Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *Parallel Architectures and Compilation Techniques (PACT'96)*, 1996.
18. 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`.
19. S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.
20. 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`.
21. 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`.
22. 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.
23. 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.
24. Standard performance evaluation corporation. `http://www.spec.org`.
25. 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.
26. 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.
27. M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 13(2):181–210, 1991.
28. 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.
29. M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
30. 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.
31. 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.
32. 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.