Induction Variable Analysis with Delayed Abstractions[1]

SEBASTIAN POP, and GEORGES-ANDRÉ SILBER
CRI, Mines Paris, France
and
ALBERT COHEN
ALCHEMY group, INRIA Futurs, Orsay, France

We present 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 stands a new algorithm that recognizes 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.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers, interpreters, optimization, retargetable compilers*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*partial evaluation, program analysis*

General Terms: Compilers

Additional Key Words and Phrases: Scalar evolutions, static analysis, static single assignment representation, assessing compilers heuristics regressions.

[1]Extension of Conference Paper: part of this work was published in [Pop et al. 2005]. The next points present the new contributions added to this article:

- —Section 1.2 provides more detailed introductory examples,
- —Section 2 makes a clear distinction between MCR and TREC,
- —Section 2.6 is a new discussion on exponential MCR,
- —Section 3 provides a high level view of the analyzer,
- —Section 4.1 gives a more detailed presentation of the components of the analyzer,
- —Section 4.2 gives the termination and complexity of the analyzer,
- —Section 4.4 presents an interprocedural extension of the analyzer,
- —Section 5.3 presents an application algorithm for assessing compilers heuristics regressions.

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 [Wolfe 1996; Allen and Kennedy 2002]. When dealing with common retargetable compilers for general-purpose and/or embedded architectures, the context is different: 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 [Allen and Kennedy 2002]. 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 [KAP]. 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

Several works demonstrated the interest of enriched low-level representation [Amme et al. 2001; Lattner and Adve 2004]. They build on the normalization and simplicity of three address code, adding data types, *Static Single-Assignment* form (SSA) [Cytron et al. 1991; Muchnick 1997] to ease data-flow analysis and scalar optimizations, and control and data annotations (loop nesting, heap structure, etc.). Starting from version 4.0, GCC [GCC 2005] uses such a representation called GIMPLE [Novillo 2003; Merill 2003], a three-address code derived from SIMPLE, the representation of the McCAT compiler [Hendren et al. 1993]. 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 [Wolfe 1992; Gerlek et al. 1995; Pottenger and Eigenmann 1995] 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 [Gerlek et al. 1995; van Engelen 2001; van Engelen et al. 2004], 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 [Cytron et al. 1991; Muchnick 1997] 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. We will sometimes note loop$_1$-ϕ for a loop-ϕ node defined in loop 1. Throughout this paper, we will use a "generic" typed three-address code in SSA form [Cytron et al. 1991]. Its syntax is simplified from the GIMPLE representation [Merill 2003]. 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 [Aho et al. 1986]: *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. The closed-form expression for `f` in Figure 1 is not affine (a second-degree polynomial). In Figure 2, apart from `d`, all variables are *univariate*: they only depend on one

```
a = 3
b = 1
loop (ℓ1)
  c = loop1-φ (a, f)
  d = loop1-φ (b, g)
  if (d ≥ 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
end
```

Fig. 1. First example: polynomial functions. At each step of the loop, an integer value following the sequence $1, 6, 11, \ldots, 126$ is assigned to d, that is the affine function $5\ell_1 + 1$; a value in the sequence $3, 11, 24, \ldots, 1703$ is assigned to f, that is a polynomial of degree 2: $\frac{5}{2}{\ell_1}^2 + \frac{11}{2}\ell_1 + 3$.

```
a = 3
loop (ℓ1)
  c = loop1-φ (a, x)
  loop (ℓ2)
    d = loop2-φ (c, e)
    e = d + 1
    t = d - c
    if (t ≥ 9) goto end2
  end2
  x = e + 3
  if (x ≥ 123) goto end1
end1
```

Fig. 2. Second example: univariate and multivariate functions. The successive values assigned to c are $3, 17, 31, \ldots, 115$, that is the affine univariate function $14\ell_1 + 3$. The evolution of x in the loop follows the values of the sequence $17, 31, \ldots, 129$ that is the affine univariate function $14\ell_1 + 17$. Finally, the evolution of variable d, $3, 4, 5, \ldots, 13, 17, 18, 19, \ldots, 129$ depends on the iteration number of both loops: that is the multivariate affine function $14\ell_1 + \ell_2 + 3$.

```
loop (ℓ1)
  a = loop1-φ (1, b)
  if (a ≥ 100) goto end1
  b = a + 4
  loop (ℓ2)
    c = loop2-φ (a, e)
    e = loop2-φ (b, f)
    if (e ≥ 100) goto end2
    f = e + 6
  end2
end1
```

Fig. 3. Third example: wrap-around. The sequence of values taken by a is $1, 5, 9, \ldots, 101$ that can be written in a condensed form as $4\ell_1 + 1$. The values taken by variable e are $5, 11, 17, \ldots, 95, 101, 9, 15, 21, \ldots, 95, 101$, and generated by the multivariate function $6\ell_2 + 4\ell_1 + 5$. These two variables are used to define the variable c, that will contain the successive values $1, 5, 11, \ldots, 89, 95, 5, 9, 15, \ldots, 89, 95$: the first value of c in the loop ℓ_2 is the value coming from a, while the subsequent values are those of variable e, that is the characteristic of wrap-around variables.

```
loop (ℓ1)
  a = loop1-φ (0, d)
  b = loop1-φ (0, c)
  if (a ≥ 100) goto end
  c = a + 1
  d = b + 1
end:
```

Fig. 4. Fourth example: periodic affine functions. Both a and b have affine evolution functions, and they are taking the values $0, 1, 2, \ldots, 100$ during the execution of the loop ℓ_1, because they both have the same initial value. However, if their initial value is different, their evolution can only be described by a periodic affine function.

```
loop (ℓ1)
  (unsigned char) a = loop1-φ (0, c)
  (int) b = loop1-φ (0, d)
  (unsigned char) c = a + 1
  (int) d = b + 1
  if (d ≥ 1000) goto end
  T[b] = U[a]
end
```

Fig. 5. Fifth example: effects of types on the evolution of scalar variables. The C programming language defines modulo arithmetics for unsigned typed variables. In this example, the successive values of variable a are periodic: $0, 1, 2, \ldots, 255, 0, 1, \ldots$, or in a condensed notation ℓ_1 *mod* 256.

```
loop (ℓ1)
  (char) a = loop1-φ (0, c)
  (int) b = loop1-φ (0, d)
  (char) c = a + 1
  (int) d = b + 1
  if (d > N) goto end
end
```

Fig. 6. Sixth example: inferring properties from undefined behavior. The C programming language does not define the wrapping of signed types: the result of a signed addition is not defined if it is over the types bounds. In this example, the behavior of the program is well defined for all the values of a in the sequence $0, 1, 2, \ldots, 126$, and consequently, the valid values of d are $1, 2, 3, \ldots, 127$, and finally the loop can be proved to run less than 127 iterations.

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, Figure 3, variables `a` and `e` have simple affine evolutions, but `c` is a typical case of *wrap-around* variable [Wolfe 1992], 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. [Gerlek et al. 1995] 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 such *multivariate* variables are more complex than the affine, polynomial or exponential cases considered in [Wolfe 1992; Gerlek et al. 1995]. 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 [Wolfe 1992], or may not even have a known closed form.[2] Figure 4 presents an 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. Figure 5 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 values of scalar variables, or to infer safe bounds for loops, as illustrated in Figure 6.

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,

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

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 formally presents the translation from the SSA to the closed form expressions at the semantic level. Section 4 presents an efficient analysis algorithm to extract closed form expressions for scalar evolutions. Section 5 integrates our method in a data dependence analysis and loop transformation framework. Section 6 compares our method to other existing approaches. Finally, Section 7 concludes and sketches future work.

2. CHAINS AND TREES OF RECURRENCES

In this section, we recall the syntax and semantics of *Multivariate* (a.k.a. multidimensional) *Chains of Recurrences* (MCR) [Bachmann et al. 1994; Kislenkov et al. 1998; Zima 2001; van Engelen 2001], a closed-form that captures the evolution of scalar variables as a function of iteration indices and allows an efficient computation of values at given iteration points. Then, we introduce *Trees of Recurrences* (TREC) that extend the expressive power of MCR by symbolic references. TREC correspond to a compressed part of the SSA graph uniquely dealing with scalar constants and symbols. MCR are obtained after an instantiation pass of all the symbols, defined as an abstraction operator: symbolic evolutions are translated with loss of information into less precise representations, or directly mapped to the "don't know" symbol $\top$. We present abstract envelopes and periodic TREC representations that we proposed as target representations for the instantiation pass. Finally, we propose the peeled TREC, and typed TREC, and we end this section with a discussion about exponential evolutions.

2.1 Chains of Recurrences

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, that can be statically processed by further analyzes and evaluated efficiently at compile-time for a given $\vec{\ell}$. Informally, the MCR corresponding to F is a binary tree of bounded depth, where each node is a constant or a MCR. The syntax of a MCR is inductively defined as: $\Theta = \{\Theta_a, +, \Theta_b\}_k$ or $\Theta = c$, where Θ_a and Θ_b are MCR 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. A MCR Θ is said *univariate* when only one loop affects the evolution and *multivariate* otherwise.

2.1.0.1 *Evaluation of MCR.* The value $\Theta(\ell_1, \ell_2, \ldots, \ell_m)$ of a MCR 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

$$\Theta(\vec{\ell}) = \Theta_a(\vec{\ell}) + \sum_{x=0}^{\ell_k - 1} \Theta_b(\ell_1, \ldots, \ell_{k-1}, x, \ell_{k+1}, \ldots, \ell_m).$$

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

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

Fig. 7. Some arithmetic operations on MCR.

value of the variable and Θ_b the increment for this variable. This definition leads to an exponential algorithm to evaluate a MCR at a given iteration point, but we may compute $\Theta(\vec{\ell})$ with a linear time and space complexity [Bachmann et al. 1994].

2.1.0.2 *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, ..., 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 MCR $\{3, +, 8, +, 5\}_1$. The coefficients of a MCR derive from a finite differentiation table: for example, the coefficients for the MCR 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 [Haghighat and Polychronopoulos 1996]:

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 [van Engelen 2001]. We present our algorithm for extracting MCR coefficients from a classic SSA representation in Section 4.

2.1.0.3 *Arithmetic operations.* Arithmetic operations on MCR can be defined as rewriting rules as illustrated in Figure 7. For a complete table of rewriting rules on MCR we refer to [van Engelen 2001].

2.1.0.4 *Evaluation example.* Considering the second introductory example (see Figure 2), the evolution of `d` can be represented by the affine equation $F(\ell_1, \ell_2) = 14\ell_1 + \ell_2 + 3$. An affine MCR 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:

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

2.2 Trees of Recurrences

We extended the MCR by allowing symbolic expressions instead of scalar coefficients that are loop invariant. We called the resulting representation *Trees of Recurrences* (TREC), as coefficients may contain tree expressions as in abstract syntax trees. We keep the same syntax and the same semantics for the TREC:

$$\{a,+,expr\}_k(x) = a + \sum_{i=0}^{x-1} expr(i).$$

TREC captures a larger class of scalar evolutions than MCR, because TREC may contain symbols referring to other scalar evolutions, and potentially self references. As an example of TREC that is not a MCR, consider the Fibonacci sequence that defines the simplest case of the class of mixers: $fib \rightarrow \{0,+,1,+,fib\}_k$. Optimizations such as symbolic propagation could handle such difficult constructs, however they can lead to problems that are difficult to solve in practice (e.g. determining the number of iterations of a loop whose exit edge is guarded by a Fibonacci sequence). Another difficulty linked to the self referring TREC is that the efficient evaluation based on Newton interpolation series cannot be used, since the self referring TREC correspond to differentiation tables of infinite length.

2.2.0.5 *Instantiation of TREC.* Because a large class of optimizers and analyzers are expecting simpler cases, TREC information is filtered using an instantiation pass. Several abstraction functions can be defined, such as mapping parametric evolutions to $\top$, or mapping non affine functions to $\top$.

2.2.0.6 *Abstract envelopes.* In some 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 maximum and minimum functions over MCR as proposed by [van Engelen et al. 2004] because it is then possible to define other kinds of envelopes using classic scalar abstract domains, such as polyhedra, octagons [Miné 2001], or congruences [Granger 1991].

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

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

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 [Wolfe 1992] are peeled TREC. Indeed, back to the third introductory example (see Figure 3), 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 8. Unifying nested peeled TREC can be done incrementally from the innermost outwards.

Finally, we formalize the notion of peeled TREC equivalence class: given integers $v, a_1, \ldots, a_n$, a MCR $c = \{a_1, +, \ldots, +, a_n\}_1$, a peeled TREC $p = (v, c)_1$, and a MCR $r = \{b_1, +, \ldots, +, b_{n-1}, +, a_n\}_1$, 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 $b_1 = v$.

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 an instantiated form of TREC 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 flip-flop operations as illustrated in Figure 4, that are special cases of self referenced peeled TREC. Variables in a flip-flop exchange their initial values over the iterations, for example:

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

Periodic sequences may also be generated by the wrapping semantics of overflowing typed variables, as illustrated in Figure 5.

2.5 Typed Trees of Recurrences

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 [Warren 2003]: these details are often considered too low level, and language dependent. However, as illustrated in the fifth introductory example, in Figure 5, the analysis of data dependences has to correctly handle the effects of overflowing on variables that are indexing the data.

The C and C++ programming languages define wrapping semantics on overflow for the unsigned types, and leave the semantic of overflowing signed types undefined. The Java programming language defines wrapping semantics on overflow for signed and unsigned types. A compiler can propose a flag for extending the wrapping semantics to all the types, as for example the option `-fwrapv` of GCC. It is this wrapping behavior that has to be preserved during a conversion of a variable to another type, and on any arithmetic operation. One of the solutions is to type the TREC and to map the effects of types 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 $\{(uint)100, +, (uint)\texttt{0xfffffff0}\}_1$. `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 first values of the sequence: $100, 84, 68, 52, 36, 20, 4, 244, 228, \ldots$. When it is impossible to prove that an evolution cannot wrap, it is safe to assume that it wraps, and keep the cast: $(uint)(\{(uchar)100, +, (uchar)240\}_1)$. Another possible solution is to use periodic TREC, but this does not seem to be practical because all the values of a period have to be listed. As we have seen in the previous example we would have to store only 15 values. Using periodic TREC for sequences wrapping over narrow types can seem practical, but for wider types this method is not practical: for a type of size 2^n the maximal period is 2^n that corresponds to the sequence generated by the affine evolution $\{0, +, 1\}_1$.

2.6 Exponential Trees of Recurrences

The exponential MCR [Bachmann et al. 1994] used by [van Engelen 2001] and then extended by [van Engelen et al. 2004] to handle sums or products of polynomial and exponential evolutions are useless in compiler technology for characterizing typed integer sequences. This is mainly due to the fact that integer typed arithmetic limits the definition domain, and any operation whose result is not in the defined domain causes an overflowing effect that either have defined modulo wrapping semantics, or the result of overflowing is left undefined by the programming language standard. The longer exponential integer sequence that can exist for an integer type of size 2^n is $n-1$, that corresponds to the left shifting of the first bit $n-2$ times. Storing exponential evolutions as peeled TREC seems to be efficient, because in general $n \leq 64$, and because exponential evolutions are not very common in codes. We have not yet used this translation in a real experiment.

We acknowledge that exponential MCR can have applications in compiler technology for floating point evolutions, but we have intentionally excluded floating point evolutions from our analysis because floating point arithmetic has even more subtleties than typed integer arithmetic.

The next section will present the semantics of some scalar SSA constructs that are defined in loop structures. We establish the link between the TREC and a subset of the SSA at the semantic level, before exposing an efficient algorithm that translates a part of the SSA representation to TREC in Section 4.

3. SEMANTIC LINK BETWEEN TREC AND A SUBSET OF THE SSA

In the following, we note:

—$\mathcal{S}[\![e]\!]$ for the semantics of an expression e,

—$loop_x - \phi(b, c)$ for an SSA ϕ node defined in loop x, where b is defined outside the loop x, and c is defined in loop x,

—$[n \leftarrow v]$ when the value v is assigned to the variable name n in the environment,

—ℓ_x for the integer-valued indices of loop x numbered from 0,

—$a(\ell_x)$ for the value of variable a at iteration ℓ_x.

We give the denotational semantics for a subset of the SSA expressions contained in an innermost loop x:

$$\begin{aligned}
\mathcal{S}[\![a = loop_x - \phi(b,c)]\!] &= \left[a(\ell_x) \leftarrow \begin{cases} b, & \text{if } \ell_x = 0; \\ c(\ell_x - 1), & otherwise. \end{cases} \right] \\
\mathcal{S}[\![d = e]\!] &= [d(\ell_x) \leftarrow e(\ell_x)] \\
\mathcal{S}[\![f = g + h]\!] &= [f(\ell_x) \leftarrow g(\ell_x) + h(\ell_x)] \\
\mathcal{S}[\![i = j * k]\!] &= [i(\ell_x) \leftarrow j(\ell_x) * k(\ell_x)]
\end{aligned}$$

Knowing the semantics of only these SSA expressions is enough for targeting all the TREC constructs. Following the definition of the updating expression $c(\ell_x - 1)$ in the semantics of loop-ϕ nodes, it is possible to distinguish the following cases:

—c is defined by an expression that is independent of a,

—c is defined by a sum expression that contains a single reference to a,

—c is defined by a sum expression that contains several references to a, or a appears in a product,

—c is transitively dependent on a in general.

For each of these cases, Figure 9 presents the equivalent SSA and TREC notations when they exist.

SSA syntax	conditions	TREC syntax
a = loop_x-ϕ (b, c)	c independent on a	$a = (b, c)_x$ (peeled TREC)
a = loop_x-ϕ (b, a + e)	e independent on a	$a = \{b, +, e\}_x$ (polynomial MCR)
a = loop_x-ϕ (b, k*a + e)	e independent on a, $k > 1$	$a = \{b, +, (k-1) * a + e\}_x$ (or see the exponential MCR syntax)
a = loop_x-ϕ (b, c)	c transitively dependent on a	mixers not represented in general

Fig. 9. Link between some SSA and TREC constructs.

The next section will present an algorithm that extracts a subset of the SSA to the TREC. The crux of the algorithm stands in the gathering of the information that is spread across the whole program, followed by a filtering of difficult constructs to an appropriate abstraction level that can be handled by optimization passes.

4. ANALYSIS OF SCALAR EVOLUTIONS

We now present an algorithm to compute closed-form expressions for inductive variables.[3] 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 queries the database that either returns the cached previously computed evolution function, 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 [Aho et al. 1986]. For all the variables defined in one of the basic blocks of an irreducible region, the answer of our analyzer will be the value $\top$ 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, an interface provides views of different levels of abstractions. This can practically be implemented by several procedures that instantiate TREC.

[3]Interested readers can find an implementation in GCC: `tree-scalar-evolution.c`.

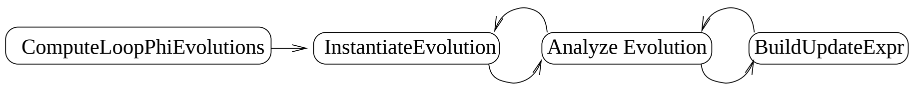

Fig. 10. Bird's eye view of the analyzer

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 nests
    For each loop-ϕ node n in loop l
      InstantiateEvolution(AnalyzeEvolution(l,n), l)

Fig. 11. Driver application.

The structure of our algorithm is quite complex because it is based on a double recursion as sketched in Figure 10. It presents similarities with the algorithm for linear unification [Paterson and Wegman 1976], where the double recursion is hidden behind a single recursion with a stack structure.

4.1 Algorithm

Figure 11 presents a driver application ComputeLoopPhiEvolutions, that computes a TREC for every variable whose value is alive across loop iterations. In practice, the computation of closed form expressions are triggered by applications like the dependence analysis or the evaluation of loop-trip count. As illustrated in this driver, the applications call the analyzer AnalyzeEvolution, presented in Figure 12, for a given loop and a variable name. The results are then filtered through an abstraction function InstantiateEvolution presented in Figure 14.

The first step of AnalyzeEvolution is a query to the database for the evolution function of the analyzed variable. The database is only visible to the AnalyzeEvolution function and is accessed using the construct, Evolution[n], for an SSA name definition n. The value contained initially in the database for a non analyzed variable name is $\perp$. The database ensures that the analysis is performed only once for a given variable name. The main part of the analyzer consists in a pattern matching of five common expressions occurring in a three-address SSA representation, with the corresponding associated action. The first pattern, "`v = constant`", is the simplest one: the resulting evolution is constant. The second pattern, "`v = a`", propagates the evolution function by copy. If the analyzer is restricted to these two patterns, the analyzer has the same role and expressive power as a constant propagation pass. To these basic patterns is added an interpreter, the third pattern "v = a $\odot$ b", that maps the arithmetic operations of the source language onto the arithmetic operations of the target language. Note that cast operations can be implemented as part of this interpreter, but are not presented in Figure 12 for simplifying the presentation. With this extension, the analyzer is slightly more expressive than the classic constant propagation because it is also able to fold some of the arithmetic expressions into constants.

The cornerstone of the analyzer is in the fourth pattern, "v = loop-ϕ(a, b)",

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)
```

Fig. 12. Main analyzer.

that analyses ϕ nodes whose arguments are defined at different loop levels. The recursively defined expression is searched and reconstructed from the low level three-address SSA representation using BuildUpdateExpr. This algorithm is presented in Figure 13 and it 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

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, InstantiateEvolution(AnalyzeEvolution(ln, n), ln),
             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, ⊥)
```

Fig. 13. SSA walker: reconstructs symbolic update expressions from a three-address SSA code.

cycle of the SSA graph: a peeled TREC is returned.

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 computes the number of iterations of the loop by solving a constraint system. As a practical implementation, one can choose the Omega solver [Pugh 1992], but it is also possible to use a solver restricted to univariate affine constraint systems for avoiding any exponential behavior. Then, Apply is used to evaluate the overall effect of the inner loop. Apply implements the efficient evaluation scheme for MCR based on Newton interpolation series (see Section 2.1). Once the overall effect of an inner loop on a scalar variable has been computed, the information is propagated after the loop, extending the limits of a classic constant propagation engine after loop structures.

Finally, in the last pattern of AnalyzeEvolution and BuildUpdateExpr, "`v = condition-φ(a, b)`", it is possible to plug the TREC envelope extension.

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
    If `v` has not been instantiated
      Mark `v` as instantiated
      Return AnalyzeEvolution(l, `v`)
    Else `v` is in a mixer structure: Return $\top$
  Else If *trec* is of the form $\{e_1, +, e_2\}_x$ Then
    $i_1 \leftarrow$ InstantiateEvolution(e_1, l)
    $i_2 \leftarrow$ InstantiateEvolution(e_2, l)
    Return $\{i_1, +, i_2\}_x$
  Else If *trec* is of the form $(e_1, e_2)_x$ Then
    $i_1 \leftarrow$ InstantiateEvolution(e_1, l)
    $i_2 \leftarrow$ InstantiateEvolution(e_2, l)
    Return UnifyPeeled($(i_1, i_2)_x$)
  Else Return $\top$

Fig. 14. A possible filter function.

Although we implemented this extension, there are no optimization or analysis application that uses this extension yet, and thus we will not present this extension of the algorithm in this paper. However, we present in Figure 12 a simple case where both branches have the same evolution functions.

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 termination and complexity of this algorithm are presented in the next subsection, then we give two illustration examples in Section 4.3. Section 4.4 proposes an extension to the interprocedural case, and finally Section 4.5 presents experimental results.

4.2 Termination and Complexity of the Algorithm

When analyzing the code of the algorithm, we can briefly sketch its call graph, as shown in Figure 10. The algorithm is initiated by a call to AnalyzeEvolution, then finishes with the analysis of all the symbols left in the representation, by calling InstantiateEvolution. An overview of the ideas that lead to the proof of the termination consists in remarking that:

—AnalyzeEvolution does not analyze twice the same variable, because after each complete analysis, the evolution is stored in a database that is checked on entry of AnalyzeEvolution,

—InstantiateEvolution does not instantiate twice the same variable, otherwise a mixer is detected, and the recursion is stopped either by returning $\top$ as in Figure 14, or by translating the mixer in an appropriate abstraction, like a periodic function,

—BuildUpdateExpr terminates once it has reached its halting loop-ϕ node, or once it has walked over all the SSA edges connected to the starting loop-ϕ node, in the limits of the analyzed loop.

Consequently, in the worst case, the analyzer stops after having analyzed once all the variables of the program. For giving an idea of the worst case complexity of the analyzer, we will describe with more details the termination process. We will consecutively consider the worst case complexity of each of the building blocks of the algorithm. We deduce from this the overall complexity of the algorithm in the worst case, then the termination of the algorithm in terms of number of basic operations. Figure 15 sketches the computational patterns behind each of the components of the algorithm.

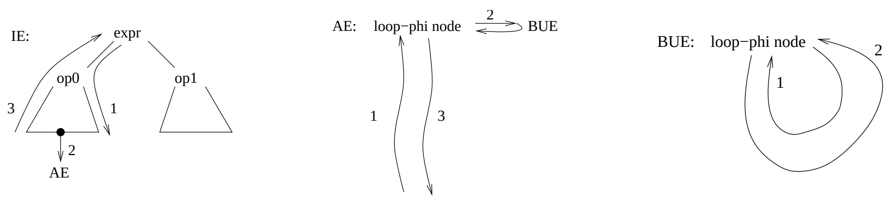

Fig. 15. Computational patterns of InstantiateEvolution (IE), AnalyzeEvolution (AE), and BuildUpdateExpr (BUE).

4.2.1 *Complexity of* InstantiateEvolution. The worst case for InstantiateEvolution corresponds to an expression with n operands, and among them appear all the SSA names defined in the program. Let m stand for the number of SSA names in the program. In the worst case $n > m$. The total cost of instantiating such an expression involves a recursive visit of each leaf: that produces n steps, then for each leaf, a call to AnalyzeEvolution. The total number of operations is equal to n decompositions of the given expression, n calls to AnalyzeEvolution, followed by $n-1$ folds of the obtained subexpressions: the total amounts to $3n-1$ basic operations.

4.2.2 *Complexity of* AnalyzeEvolution. The cost of AnalyzeEvolution for a constant is equal to 1. For a SSA name, the cost is equal to the look-up in the database plus, when the scalar variable was not yet analyzed, the cost of its analysis. Because the first part of this algorithm consists in walking up the definitions to known values or to loop-phi nodes, the algorithm may end as soon as all the needed scalar definitions have values already computed.

In the worst case, the total number of steps is equal to $5m$: at each analysis of a SSA name there are at most 3 reads in the database: for $a = b\ op\ c$, a read for a, then supposing that the variable has not yet been analyzed, a read for each of the operands, then a fold operation on the TREC of the operands, and finally a write of the result in the database.

Because in the worst case $n > m$, the difference $n-m$ corresponds to the number of queries from InstantiateEvolution that hit the cached value in the database.

Thus the total number of basic operations for AnalyzeEvolution is equal to $4m + n$.

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

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

In the particular case where the SSA edges enter an inner loop, the analysis of the definition in the inner loop is triggered. But because we are computing the total number of steps in the worst case, we have already counted these definitions in the input expression to InstantiateEvolution. Thus we can consider all these parts already analyzed, and their cost to the BuildUpdateExpr function is equal to a read in the database, if we count the cost of computing the number of iterations for all the loops separately.

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

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

$$4n - 1 + 4m + \sum_{i \in loop-\phi} e_i + l$$

—n is the number of basic components in TREC to be instantiated,

—m is the number of SSA names in the program,

—e_i is the number of SSA edges reachable from the analyzed loop-ϕ, and not exiting the loop,

—l is the number of steps required to solve the constraint systems for determining the number of iterations for all the loops. l is linear if the solver is restricted to uniquely deal with univariate affine evolutions. In the case of the Omega solver [Pugh 1992], l is exponential in the worst case.

We have proved that the algorithm is terminating on any input SSA representation, and we have analyzed the worst case complexity of the analysis algorithm. The overall cost of the analyzer highlights its structure: it is composed of two preprocessing passes followed by the analysis of the loop-ϕ nodes. The complexity of the algorithm depends on the quality of the expected answer: using an exact solver for constraint systems might not be practical for production compilers. However, we plan to use exact solvers for improving the overall quality of the compiler by assessing regressions with respect to an optimal behavior.

```
a = 3
b = 1
loop (ℓ1)
  c = loop1-φ(a, f)
  d = loop1-φ(b, g)
  if (d ≥ 123)
    goto end
  e = d + 7
  f = e + c
  g = d + 5
end
```

(1) Initial value edge

```
a = 3
b = 1
loop (ℓ1)
  c = loop1-φ(a, f)
  d = loop1-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 = loop1-φ(a, f)
  d = loop1-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 = loop1-φ(a, f)
  d = loop1-φ(b, g)
  if (d ≥ 123)
    goto end
  e = d + 7
  f = e + c
  g = d + 5
end
```

(4) On the "return path"

Fig. 16. Application to the first example

4.3 Application of the Analyzer to the Introductory Examples

We illustrate the analysis of scalar evolutions algorithm on the first two introductory examples in Figures 1 and 2. In addition to clarifying the depth-first search and instantiation phases of the algorithm, this will exercise the recognition of polynomial and multivariate evolutions.

4.3.0.1 *First example.* The depth-first search is best understood with the analysis of `c = loop`$_1$`-`ϕ`(a, f)` in the first example. The SSA edge of the initial value exits the loop, as represented in Figure 16.(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 16.(2). The update edge `c`$\rightarrow$`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`$\rightarrow$`e`, reaching the assignment `e = d + 7`, and finally follows the edge `e`$\rightarrow$`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`$\rightarrow$`c` is followed to the starting loop-ϕ node, as illustrated in Figure 16.(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 16.(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: `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`$\rightarrow$`d` is taken to the definition `d = loop`$_1$`-`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 $\{3, +, 8, +, 5\}_1$.

4.3.0.2 *Second example.* We will now compute the evolution of `x` in the second example, Figure 2, 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 value `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 = loop`$_1$-ϕ`(a, x)`. Since this is a loop-ϕ node, the analyzer must determine its evolution in loop 1. The edge to the initial value 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

```
void F1 () {              void F2 (int x) {
a = 2                     d = 0
loop (ℓ1)                 loop (ℓ2)
  b = loop1-φ(a, c)         e = loop2-φ(d, f)
  F2 (b)                    A[x + e] = ...
  c = b + 3                 f = e + 1
  if (b > 10) goto end1     if (e > 100) goto end2
end1                      end2
}                         }
```

Fig. 17. Interprocedural example

difficult to obtain, as for example the results of an interprocedural extension of the analyzer.

4.4 An Interprocedural Extension

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

We illustrate the algorithm on a simple example given in Figure 17. In this example, the loops are uniquely labeled; the analysis of array `A` triggers the analysis of variable e: $e \rightarrow \{0, +, 1\}_2$; then it triggers the analysis of x, mapped to argument b in the call to function `F2`. The parameter is then analyzed in the calling context, and since there is a unique point from where `F2` is called, the parameter x is identified as the value of b, i.e. $b \rightarrow \{2, +, 3\}_1$. Finally after the instantiation step, the access function of array `A` is recognized to be $\{\{2, +, 3\}_1, +, 1\}_2$. When the function `F2` is called in several contexts, the value of parameter x is obtained by merging the information contained in all these contexts.

4.5 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 [Spec 2000] and JavaGrande [JavaGrande 2000] benchmarks.

Figure 18 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 18 show the precision of the detector of the number of iterations. For the moment, only the single-exit loops are exactly analyzed,

CINT2000	Total	AU	AM	OP	CE	⊤	Loops	Trip	≤	⊤
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**	**AU**	**AM**	**OP**	**CE**	⊤	**Loops**	**Trip**	≤	⊤
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	**Total**	**AU**	**AM**	**OP**	**CE**	⊤	**Loops**	**Trip**	≤	⊤
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

Fig. 18. Scalar induction variables and loop trip count in SPEC CPU2000 and JavaGrande benchmarks. Break-down of scalar evolutions into: "AU" affine univariate, "AM" affine multivariate, "OP" other kinds of polynomials, "CE" other compound expressions containing determined components, such as casts that cannot be further reduced, and "⊤" undetermined evolutions. Last columns describe: "Loops" the number of *natural* loops, "Trip" the number of *single-exit* loops whose trip count is successfully analyzed, "≤" the number of loops for which an upper bound approximation of the trip count is available.

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 as is the case of the 171.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

[Pugh 1992], for precisely computing or approximating the number of iterations in multiple-exit loops.

5. APPLICATIONS

5.1 Data dependence analyzer

Our scalar evolution algorithm is integrated in a dependence analysis pass of GCC applied to scalar loop optimizations, vectorization [Eichenberger et al. 2004; Naishlos 2004], and a generic framework for high level loop transformations [Li and Pingali 1994; Berlin et al. 2004]. These dependence-based transformations use uniform dependence vectors [Banerjee 1992], 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 [Banerjee 1992], and for validating the results of this first data dependence analyzer, we adapted the integer linear programming solver Omega written by William Pugh [Pugh 1992] to solve affine dependence systems. 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 19 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 [Allen and Kennedy 2002]. 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 MIV column 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,

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

Fig. 19. 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".

and the worst case of computation time for SPEC CPU2000 is 15 seconds and 70 percent of the compilation time.

5.2 Evaluation of existing optimizations

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 [Aho et al. 1986] enabled by default with the option `-O2`. In order to use the vector units, the "simdization" pass [Eichenberger et al. 2004] 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 [Naishlos 2004] from IBM Haifa. A linear loop transformation framework has been contributed by

Daniel Berlin from IBM Research and Sebastian Pop, but for the moment only the loop interchange transformation is supported [Berlin et al. 2004], and a pass of value range propagation [Novillo 2005] enabled by default at `-O2` has been contributed by Diego Novillo from Red Hat.

We have experimented with these transformations on SPEC CPU2000 benchmarks and MiBench benchmarks [Guthaus et al. 2001], using for the base compiler the options: `-O2 -msse2 -fno-tree-loop-optimize`, such that all the optimizations based on the scalar evolutions analyzer are disabled, and we compared these results to the peak compiler where we have used the following options: `-O2 -msse2 -ftree-vectorize -ftree-loop-linear`. The only significant speedup for the moment is for the 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.

The main problems for evaluating the potential of this new infrastructure are linked to the fact that all the optimization passes are not tuned enough: the heuristic functions are either too coarse, or not implemented yet. For example, the vectorizer has code that performs loop versioning without having any cost model based on which it could evaluate the profitability of the transformation. The vectorization pass could generate larger codes that slow the execution. In this context we can see the need of an automatic technique to assess regressions of heuristics. We present such a technique in the next subsection.

5.3 Assessing Heuristics Regressions

A heuristic is an analyzer that produces a more abstract representation than an exact solver: a part of the information that would have been extracted by an exact solver is not reached by the heuristic solver. In practice, we speak about heuristics as being exact solvers with cutoff mechanisms based on some features of the solver, such as the time taken to execute the solver, used space, or any other feature that can quantify the behavior of the solver. Also note that the representation of the exact answer might be too costly to be handled in practice, and thus an abstract view might be more practical.

The questions that we want to answer are related to the static evaluation of regressions caused by the use of a heuristic instead of an exact solver. The following questions might be seen as the heart of our problem: Is it possible to define a lattice structure for the set of heuristics? What is the most abstract heuristic that is optimal for a given training set? This question can be reformulated in terms of condensing abstract domains: how do we build a heuristic that is condensing [Giacobazzi et al. 2005] with respect to a given training set? Answering by an algorithm to these questions has practical results in compiler technology, because searching in the heuristics lattice for the most abstract heuristic that is optimal with respect to the training set amounts to search for the most efficient solver that will provide the same quality of answers as an exact solver for a given training set.

Intuitively, the precision of a heuristic can be measured by comparing its output to the answer of an exact solver. Then, based on the precision of the results of heuristic functions, it is possible to define an order on heuristic functions, and finally a structure of lattice in which the bottom element corresponds to a heuristic whose answer is invariably "don't know", while the top element of the heuristics lattice corresponds to the exact solver. Using these remarks, we give the following definitions:

Definition 5.1 *Regression of a heuristic.* Let h be a heuristic function, and ω its associated exact solver from $A \to B$. The regression of h with respect to ω is the set of answers for which h differs from ω. We note: $h \setminus \omega = \{y \in B \mid \forall x \in A, y = h(x), h(x) \neq \omega(x)\}$ for the regression of h with respect to ω.

Definition 5.2 *Amplitude of a regression.* The cardinal of a regression set is also called the amplitude of the regression.

Definition 5.3 *Order on heuristics.* Let h_1 and h_2 be two heuristic functions, and ω the associated exact solver. The order $h_1 \leq_{\mathcal{H}} h_2$ is defined by the relative amplitudes of the regressions with respect to ω as: $h_1 \leq_{\mathcal{H}} h_2$ if $card(h_1 \setminus \omega) \leq card(h_2 \setminus \omega)$.

Definition 5.4 *Lattice of heuristics.* Let

—$\mathcal{H}$ be the set of heuristic functions ordered by $\leq_{\mathcal{H}}$,
—$\wedge$ the meet operator defined by: given two heuristics h_1 and h_2,

$$h_1 \wedge h_2 = \begin{cases} h_1 & \text{if } h_1 \leq_{\mathcal{H}} h_2, \\ h_2 & \text{otherwise,} \end{cases}$$

—$\vee$ the join operator defined by: given two heuristics h_1 and h_2,

$$h_1 \vee h_2 = \begin{cases} h_2 & \text{if } h_1 \leq_{\mathcal{H}} h_2, \\ h_1 & \text{otherwise,} \end{cases}$$

—$\perp$ the heuristic that invariably answers "don't know",
—$\top$ the exact solver.

$(\mathcal{H}, \leq_{\mathcal{H}}, \vee, \wedge, \perp, \top)$ is the lattice of heuristics.

This leads to the following algorithm for assessing the regressions of a heuristic h_1 with respect to the results of a heuristic h_2: evaluate the set of all the possible inputs through h_1, then through h_2. If the output of heuristic h_1 is less precise than the output of h_2, then h_2 is preferable. This algorithm is not realizable in practice, because the set of possible inputs may be infinite. For making the algorithm work in practice for compiler technologies, we have to restrict the input:

Definition 5.5 *Training set.* Let h be a heuristic from $A \to B$. A subset of the input set A is called a training set.

Definition 5.6 *Optimal heuristic for a training set.* A heuristic is said optimal for a training set if it produces the same results as an exact solver.

The engineering of a compiler production version of a heuristic would have to minimize or to maximize several parameters in accordance. It is possible to envision the need of another level of static analysis for extracting information about the behavior of the heuristic, but in the end, a dynamic evaluation on some training set is expected at some abstraction level.

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

Fig. 20. Two sequential loops

As we have described in this subsection, it is possible to use static analysis techniques for an automatic design of heuristics, and for assessing the effectiveness of heuristic functions with respect to optimal exact solvers. In a future work, we will use these techniques for automatically building the missing heuristic functions for which exact solvers are known. We will use these techniques for improving the heuristics of GCC, and in order to guarantee that no future regression will be introduced, we will build a test harness for some of the heuristics used in GCC.

6. 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 [Bachmann et al. 1994], then used by van Engelen [van Engelen 2001; van Engelen et al. 2004]. The representation analyzed by our algorithm is closer to the one used in the Open64 compiler [Gerlek et al. 1995; Liu et al. 1996], but our algorithm avoids the syntactic case distinctions made in [Liu et al. 1996] 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 [Liu et al. 1996].

Path-sensitive approaches have been proposed [van Engelen et al. 2004; Rus et al. 2004] 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 [GNUGO 2005]. However, we consider to adapt this kind of techniques in experimental branches of development, such that we can assess the benefits of having a more precise solver.

Our work is based on the previous research results presented by Robert van Engelen in the early paper [van Engelen 2001]. 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 20 are not correctly handled. The analysis of the first loop yields $i \rightarrow \{0, +, 1\}_1$. Then, analyzing the second loop, the initial value 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. 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.

7. CONCLUSION AND PERSPECTIVES

We introduced *trees of recurrences*, a formalism based on *multivariate chains of recurrences* [Bachmann et al. 1994; Kislenkov et al. 1998], 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

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

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

Amme, W., Dalton, N., von Ronne, J., and Franz, M. 2001. 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).*

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

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

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

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

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

GCC 2005. The gnu compiler collection. http://gcc.gnu.org.

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

Giacobazzi, R., Ranzato, F., and Scozzari, F. 2005. Making abstract domains condensing. *ACM Transactions on Computational Logic (ACM-TOCL) 6,* 1, 33–60.

GNUGO 2005. Gnu go. http://www.gnu.org/software/gnugo/gnugo.html.

Granger, P. 1991. 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.* Springer-Verlag New York, Inc., New York, NY, USA, 169–192.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization.* Austin, TX.

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

Hendren, L., Donawa, C., Emami, M., Gao, G. R., Justiani, and Sridharan, B. 1993. 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. Springer-Verlag, 406–420.

JavaGrande 2000. Java grande forum. http://www.javagrande.org.

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

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

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

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

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

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

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

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

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

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

Novillo, D. 2005. A propagation engine for gcc. In *Proceedings of the 2005 GCC Developers Summit*. 175–185. `http://www.gccsummit.org/2005`.

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

Pop, S., Cohen, A., and Silber, G.-A. 2005. Induction variable analysis with delayed abstractions. In *2005 International Conference on High Performance Embedded Architectures and Compilers*. Barcelona, Spain.

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

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

Rus, S., Zhang, D., and Rauchwerger, L. 2004. 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.

Spec 2000. Standard performance evaluation corporation. `http://www.spec.org`.

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

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

Warren, H. 2003. *Hacker's Delight*. Addison-Wesley.

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

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

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