The New Framework for Loop Nest Optimization in GCC: from Prototyping to Evaluation

Sebastian Pop[1], Albert Cohen[2], Pierre Jouvelot[1], and Georges-André Silber[1]

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

Abstract. This paper presents a practical prototyping tool for SSA transformations based on PROLOG, and a case study of its applicability using the New Framework for Loop Nest Optimization of the GNU Compiler Collection (GCC). Using this approach, we implemented an induction variable analysis in GCC and performed several experiments for assessing different issues: performance of the generated code, effectiveness of the analyzers and optimizers, compilation overhead, together with possible improvements. This evaluation, based on the SPEC CPU2000, MiBench and JavaGrande benchmarks on different architectures, suggests that our framework could be of valuable use in production compilers developments such as GCC.

1 Introduction

The GNU Compiler Collection (GCC) is the leading compiler suite for the free and open source software. The large set of target architectures and standard compliant front ends makes GCC the de facto compiler for portable developments and system design. However, until recently, GCC was not an option for high performance computing: it had infrastructure neither for data access restructuring nor for automatic parallelization. Furthermore, until recently, the compilation research community had no robust, universal, free, and industry-supported platform where new ideas in compiler technology can be implemented and tested on large scale programs written in standard programming languages. For more than a decade, the field of advanced compilation has been plagued by redundant development efforts on short-lived projects, with little impact on production environments. In this paper, we propose an overview and a preliminary evaluation of the new infrastructure of GCC for analysis and loop nest optimizations. Based on this evaluation we intend to show the potential of GCC to address the needs of a free compiler in the compiler research and high performance computing communities.

Modern compilers implement some of the sophisticated optimizations introduced for supercomputing applications [23, 3]. They provide performance models and transformations to improve fine-grain parallelism and to take into account

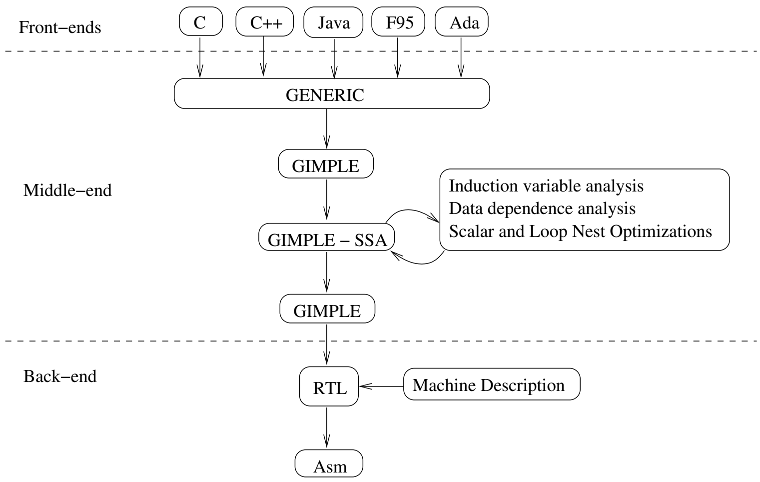

Fig. 1. The infrastructure of GCC 4.

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 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 [12]. 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 for SIMD instructions 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.

Several projects demonstrated the interest of type enriched low-level representation [14]. They build on the normalization and simplicity of three-address code, adding data types, *Static Single-Assignment* (SSA) form [8, 17] to ease data-flow analysis and scalar optimizations, and control and data annotations (loop nesting, heap structure, etc.). Recently, GCC 4 adopted such a representation called GIMPLE [19, 16], a three-address code derived from SIMPLE, the representation of the McCAT compiler [11]. The GIMPLE representation was proposed by Sebastian Pop and Diego Novillo, from RedHat, for minimizing the effort in the development and the maintenance of new analyzes and transformations. GIMPLE is used for building the SSA representation that is then used by the analyzers and the scalar and loop nest optimizers. After these optimizations, the intermediate representation is translated into RTL, where machine-specific informations are added to the representation. Figure 1 presents the overall structure of GCC. The front-ends C, C++, Java, Fortran95 and Ada are translating their intermediate representation to a common representation called GENERIC, then to GIMPLE that is eventually translated to RTL. Before presenting the loop optimizations that we will evaluate, we give an overview of the GIMPLE intermediate representation on which the loop optimizers are acting.

A three-address representation like GIMPLE could seem not suitable to implement program transformations for high performance: control-flow is represented by only two primitives, a conditional expression `if`, and a goto expression `goto`. Loops have to be discovered from the control-flow graph [2]. Although the number of iterations is not captured by the GIMPLE representation, it is often discovered by analyzing the scalar variables involved in the loop exit conditions; this allows to compute 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, or loops with irregular control constructs such as `break` or exceptions. In GIMPLE, subscript expressions, loop bounds and strides are spread across a number of elementary instructions and basic blocks, possibly far away from the original location in the source code. This is due to the translation to the low-level intermediate representation and to optimization phases such as dead-code elimination, partial redundancy elimination, optimization of the control flow, invariant code motion, etc. However, the SSA form provides fast practical ways to extract information concerning values of scalar variables.

We recall some SSA terminology [8, 17] to facilitate further discussions: the *SSA graph* is the graph of def-use chains in the SSA form; each assignment targets a unique variable; ϕ nodes occur at merge points of the control flow and restore the flow of values from the renamed variables. Assuming the natural loops [2] information has been recovered, ϕ nodes whose arguments are defined at different loop depths are called *loop-*ϕ nodes and they have a specific semantics related to a potentially recursive definition. With the following notations:

- $\mathcal{S}[\![e]\!]$ 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 in an enclosing loop y and c is defined in loop x, as illustrated in Figure 3,
- ℓ_x for the integer-valued indices of loop x, $0 \leq \ell_x < N_x$, with N_x the number of iterations in loop x,
- $a(\ell_x)$ 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:

$$\mathcal{S}[\![a = loop_x - \phi(b,c)]\!]\ell_y = \forall \ell_x \left[0 \leq \ell_x < N_x \wedge a(\ell_x) = \begin{cases} b(\ell_y), & \text{if } \ell_x = 0; \\ c(\ell_x - 1), & otherwise. \end{cases} \right]$$

$$\mathcal{S}[\![d = e]\!]\ell_x = [d(\ell_x) = e(\ell_x)]$$

$$\mathcal{S}[\![f = g + h]\!]\ell_x = [f(\ell_x) = g(\ell_x) + h(\ell_x)]$$

We illustrate the semantics of the SSA representation with two examples: Figure 2 illustrates the role of a ϕ node used for merging values assigned in different branches of a condition expression. Figure 3 presents the syntax and the semantics of a loop-ϕ node, variable a is self defined in the argument coming from the loop body.

```
v = 5;
if (E) {
  v = 0;
}
... = v;
```

$\xrightarrow{\text{SSA}}$

```
a = 5
if (E)
  then b = 0
endif
c = φ(a, b)
... = c
```

$\xrightarrow{Sem}$

$$a = 5$$
$$b = 0$$
$$c = \begin{cases} \text{b} & \text{if } E \text{ is true,} \\ \text{a} & \text{otherwise.} \end{cases}$$

Fig. 2. Syntax and semantics of a condition-ϕ node.

```
i = 0;
s = 7;
do {
  i = i + s;
} while (E);
```

$\xrightarrow{\text{SSA}}$

```
b = 0
f = 7
begin1:
  a = loop1-φ(b, c)
  c = a + f
  if (E)
    then goto begin1
    else goto end1
  endif
end1:
```

$\xrightarrow{Sem}$

$$\begin{aligned} a(0) &= b \\ a(x) &= c(x-1) \text{ for } x > 0 \\ a(x) &= a(x-1) + f(x-1) \text{ for } x > 0 \\ &\dots \\ a(x) &= b + \sum_{j=0}^{x-1} f(j) \text{ for } x \geq 0 \\ a(x) &= b + x \cdot f \text{ for } x \geq 0 \\ a(x) &= 7x \text{ for } x \geq 0 \end{aligned}$$

Fig. 3. Syntax and semantics of a loop-ϕ node.

2 Contributions

We developed an analyzer [21] that matches common induction patterns on the SSA graph, based on an algorithm close to linear unification [20]. We develop this idea in Section 3 and give a practical framework for prototyping SSA analyzes and transformations in PROLOG [1], thus paving the way for more analysis and transformations in GCC.

The analyzer that we presented in [21] builds a description of the scalar variables using a practical closed form representation: the chains of recurrences [4, 13, 22]. We presented in [21] some extensions for the chains of recurrences for solving practical problems that we encountered during the integration of our algorithm in GCC. Among these extensions we can cite the symbolic form of chains of recurrences that we called trees of recurrences TREC, enabling the expression of self-defined chains of recurrences, as for example the Fibbonacci sequence, or more generally, exponential evolutions expressed in the program only by additions. We have also proposed the peeled chains of recurrences, that list the first values of a sequence. Periodic sequences can naturally be represented using a combination of these two extensions, as a self-defined TREC listing the elements of the period. We also presented a possible way to handle typed scalar variables and overflow effects. We remarked that the main goal of our analyzer is to reduce the complexity of the representation of the program, by recognizing and removing difficult constructs, so that the optimizers can efficiently process the resulting representation. Finally, we concluded that a new representation (the chains of recurrences) is not needed for constructs that are already captured by the SSA representation. We propose in Section 4 the abstract SSA representation, that simplifies the SSA representation in the sense that a part of the information has been compressed, filtered out, or replaced by abstract descriptions of scalar values, and show that such a simple representation is enough to reach our goals.

Based on the information provided by our analyzer, several optimization passes have been developed: Zdeněk Dvořák from SuSE has contributed strength reduction, induction variable canonicalization and elimination, loop invariant code motion and other classic scalar loop optimizations [2]. All these passes aim to replace the old optimizers that acted at the RTL level, mainly because the type information available at this higher level allows more optimization opportunities. In order to use the vector units of the target architecture, such as the Altivec, SSE or MMX, the "simdization" pass [9] recognizes instruction patterns in loop bodies that can be rewritten using SIMD instructions. This pass has been contributed by Dorit Naishlos [18] from IBM Haifa. A linear loop transformation framework has been integrated in GCC. Daniel Berlin from IBM Research and Sebastian Pop have contributed the loop interchange transformation [5]. In Section 5, we propose an evaluation of the induction variable analyzer by stressing the main version of GCC containing all these optimization passes on standard benchmark programs, thus showing the potentiality of our general framework.

3 Unification Techniques on SSA Problem Formulations

One of the most interesting aspects of the SSA representation comes from the observation that data-flow problems can be described atomically, independently of unrelated problems that would have to be solved in a classic data-flow formulation. For example, in the computation of the number of iterations in a loop, we remark that the SSA problem formulation can be described only in terms of the variables used in the exit conditions, independently of other constructs defined in the program. With respect to the definition of a scalar problem, the SSA representation provides an abstract view of the program, in which only remain those constructs relevant to the resolution of the problem. The SSA representation can thus be viewed as an ideal representation for goal-directed solving techniques, namely the unification algorithms.

3.1 SSA Problem Formulations in PROLOG

Let us represent an *SSA graph* as a set of terms built from boolean and integer variables, well-formed boolean and integer arithmetic expressions over operators $+$, $-$, $<$, $\leq$, $\wedge$, $\vee$, and $\neg$, unary predicate `cond`, binary predicates `assign`, `loop`, and `ephi`, and the ternary predicate `cphi` and `lphi`. Such terms can be represented natively in a language such as GNU PROLOG [1]; we use its syntax in the following, as well as its unification and Horn clause resolution semantics.

Predicates `cphi`, `lphi` and `ephi` capture the conditional flow of values associated with incoming control paths as follows:

condition-ϕ: Term `cphi(p, b, c)` represents the control and data flows of a conditional guarded by a predicate condition `p`, and the respective values of the true and false branches associated, respectively, with terms `b` and `c`;

loop-ϕ: Term `lphi(x, b, c)` represents the flow of a well-structured loop with (virtual) counter `x`, initial value `b` and iteration value `c`.

closed-loop-ϕ: Term `ephi(x, b)` represents the value of `b` after loop `x`.

In the following, we give the denotational semantics for the PROLOG predicates that represent a subset of the SSA expressions, and we note as before:

- $\mathcal{S}[\![e]\!]$ for the semantics of an expression e,
- ℓ_x for the integer-valued indices of loop x, $0 \leq \ell_x < N$, with N the number of iterations in loop x,
- $a(\ell_x)$ for the value of variable a at iteration ℓ_x,
- $last(\ell_x)$ for the last value of loop counter ℓ_x: $N-1$.

$$\begin{aligned}
\mathcal{S}[\![assign(a,b).]\!]\ell_x &= [a(\ell_x) = b(\ell_x)] \\
\mathcal{S}[\![assign(a,b+c).]\!]\ell_x &= [a(\ell_x) = b(\ell_x) + c(\ell_x)] \\
\mathcal{S}[\![assign(a,cphi(p,b,c)).]\!]\ell_x &= \left[a(\ell_x) = \begin{cases} b(\ell_x), \text{ if } p(\ell_x) \text{ is evaluated to true;} \\ c(\ell_x), \, otherwise. \end{cases}\right] \\
\mathcal{S}[\![assign(a,lphi(\ell_y,b,c)).]\!]\ell_x &= \left[a(\ell_y) = \begin{cases} b(\ell_x), & \text{if } \ell_y = 0; \\ c(\ell_y - 1), & otherwise. \end{cases}\right] \\
\mathcal{S}[\![assign(a,ephi(\ell_y,b)).]\!]\ell_x &= [a(\ell_x) = b(last(\ell_y))]
\end{aligned}$$

Unstructured control flow may lead to some back edges being associated with plain conditional terms and others recognized as loops. In general, the program control flow can be recovered from `phi` predicates. Notice the above definition is not aimed at representing all imperative programs in SSA form: it does not provide terms for function calls, array subscripts, etc.

```
a = 2
begin1:
  b = loop1-φ(a, c)
  if (b>=17) goto end1
  c = b + 1
  begin2:
    d = loop2-φ(b, e)
    if (d>=42) goto end2
    e = d + b;
  end2:
end1:
f = closed-φ(c);
```

Fig. 4. Running example in SSA.

$$
\begin{aligned}
\mathtt{a} &= 2\\
\mathtt{b} &= 2,3,4,\ldots,16 \;=\; \ell_1+2\\
\mathtt{c} &= 3,4,5,\ldots,16 \;=\; \ell_1+3\\
\mathtt{d} &= 2,4,6,\ldots,42,3,6,9,\ldots,42,\ldots,16,32\\
&= (\ell_1+2)(\ell_2+1)\\
\mathtt{e} &= 4,6,8,\ldots,42,6,9,12,\ldots,42,\ldots,32\\
&= (\ell_1+2)(\ell_2+2)\\
\mathtt{f} &= 17\\
\text{with } & \ell_1 \in \{0,\ldots,14\}\\
& \ell_2 \in \{0,\ldots,\lfloor 42/(\ell_1+2)\rfloor - 1\}
\end{aligned}
$$

Fig. 6. Sequences and closed forms.

```
assign(a, 2).
assign(b, lphi(l1, a, c)).
assign(c, b + 1).
assign(d, lphi(l2, b, e)).
assign(e, d + b).
assign(f, ephi(l1, c)).
```

Fig. 5. SSA graph.

```
assign(a, 2).
assign(b, lphi(l1, 2, b + 1)).
assign(c, b + 1).
assign(d, lphi(l2, b, d + b)).
assign(e, d + b).
assign(f, 17).
```

Fig. 7. Normalized SSA graph.

Consider the example in Figure 4. The SSA graph of this running example is shown in Figure 5. Each one of its six scalar variables is associated with a function from loop iteration vectors (ℓ_1, ℓ_2) to integers; as shown in Figure 6, these functions can be represented as closed polynomial forms. Variables `b` and `c` are *univariate*: they only depend on one loop counter (ℓ_1); whereas `d` and `e` are *multivariate*. The purpose of induction variable recognition [10] is to statically compute such closed form representations. To compute the evolution of `f`, one must know the *trip count* of the outer loop, i.e., the exact exit value of ℓ_1. To statically evaluate this value, one must already understand the evolutions of `b` and `c`. As we have seen in this example, the SSA graph has partitioned the problem such that the resolution only depends on the variables used to define the exit condition, independently of the existence of the inner loop, or any other constructs.

3.2 Normalizing Semantic Equivalent Constructs

For reducing the size of the code and for preparing the representation for pattern matching, it is possible to use a normalization predicate `trNorm` defined in

Figure 8. The result of this normalization on the running example is illustrated in Figure 7. As shown in Figure 8, `trNorm` unifies simple definition circuits with self-definition loops. It succeeds on all circuits, but fails on more general strongly-connected components. Predicate `trNorm` uses `hasAself` for determining whether the `BackEdge` has a self reference to `X`. In this case, `BackEdge` is replaced by an equivalent expression `X + Step` where `X` appears in front of the expression, such that it can be simpler matched later. The expression contained in `Step` is the result of the `fold` functor that reduces the size of arithmetic expressions by performing basic arithmetic operations on scalar constants.

Note that `Step` can contain self definitions to `X` as the third rule of `hasAself` does not check for self definitions in `B` and directly builds the result `StepA + B`. Thus the expression obtained in `Step` can be quite difficult to handle in general, and this is why some optimizers can decide to ask for a more abstract information.

```
trNorm(assign(X, lphi(LoopId, Init, BackEdge)),
       assign(X, lphi(LoopId, Init, X + Step))) :-
  hasAself(X, BackEdge, Step), !.
trNorm(Default, Default).

hasAself(X, X, 0).
hasAself(X, Name, Step) :-
  assign(Name, Expr), hasAself(X, Expr, Step).
hasAself(X, A + B, Step) :-
  (hasAself(X, A, StepA), fold(StepA + B, Step), !);
  (hasAself(X, B, StepB), fold(A + StepB, Step), !).

fold(_ + unknown, unknown).
fold(unknown + _, unknown).
fold(L + R, Res) :-
  integer(L), fold(R, ResR), integer(ResR), Res is L + ResR, !;
  integer(R), fold(L, ResL), integer(ResL), Res is ResL + R, !.
fold(L + R, ResL + ResR) :- fold(L, ResL), fold(R, ResR), !.
fold(Default, Default).
```

Fig. 8. A normalization equivalence.

4 Abstract SSA

The proper abstraction level depends on the requirements of the intended computation: the value range propagation is able to infer useful information from integer intervals, whereas such information would probably be useless in the computation of the exact loop trip count, that would require a more precise information under the form of a symbolic expression. For this reason, it is possible to either provide the previous normalized, compressed form, or to further process this information, removing difficult constructs, replacing some symbols with abstract elements, leading to an SSA graph with abstract elements: the abstract SSA.

4.1 Abstractions as Herbrand Universes

The theory of abstraction has independently evolved in the abstract interpretation and artificial intelligence domains as practical methods for dealing with either large amounts of data, or for representing uncertainty. In practice, abstractions are defined by a mapping between two sets that preserves some properties and that reduces the complexity. A logic framework for abstractions can be described in terms of Herbrand universes: the starting set can be defined as the set of all ground terms constructed from functors and constants; the abstraction mapping can be defined as a functor transforming every ground term from the starting set to other ground terms composed of constants and functors in the abstract set.

Definition 1 (Abstraction). *An abstraction is a triplet $(\Sigma_1, \Sigma_2, \alpha)$, with Σ_1 and Σ_2 two sets, potentially $\Sigma_2 \subseteq \Sigma_1$, and a total function $\alpha : \Sigma_1 \rightarrow \Sigma_2$ called the abstraction function, that maps the elements of Σ_1 onto that of Σ_2.*

The usefulness of abstractions arises from the fact that it is sometimes simpler to work on a set that contains fewer elements, or on which some properties are decidable, and then to infer the result on the starting set using a concretization function.

Definition 2 (Concretization function). *Let $(\Sigma_1, \Sigma_2, \alpha)$ be an abstraction. A function $\gamma : \Sigma_2 \rightarrow \mathcal{P}(\Sigma_1)$ is called a concretization function. It maps elements from Σ_2 to sets of elements of Σ_1.*

The main idea behind the automatic definition of static analyzers in abstract interpretation [6, 7] is that the safety of computations on an approximation is guaranteed by some order relation that has to be preserved during the translations between the concrete and abstract sets. However, it is sometimes the case that the operations on the abstract domain are exact, i.e. computing on the abstract set produces exactly the same result, without loss of precision, as in the concrete set. In this case the order is trivially preserved, as the abstraction and concretization functions are bijections on some subsets of the abstract and concrete sets.

4.2 Masking Abstractions

Masks or filters are the most intuitive forms of abstractions, as they are defined by abstraction functions that preserve some constructs of the starting set, while the remaining elements are mapped to an "unknown" element in the target representation. Using this technique, it is possible to define several mask functions for obtaining the right level of detail that abstracts away all constructs that are irrelevant for a computation.

We illustrate the definition of a mask in Figure 9: a map of exponential evolutions (e.g., $x = x + x$) matched by the first rule to an "unknown" element. The last rule identically translates all the remaining constructs not matched by the first rule.

```
removeMixers(assign(X, lphi(_, _, X + Step)), assign(X, unknown)) :-
  hasAself(X, Step, _).
removeMixers(Ibidem, Ibidem).
```

Fig. 9. Definition of a mask that maps some exponentials to an "unknown" element.

Another example of masking abstraction is given by the translation of a subset of the SSA graphs to polynomial expressions whose representation can be given either in terms of multivariate chains of recurrences represented by a predicate `mcr`, as illustrated in Figure 10, or more generally, (Figure 11) to lambda expressions represented by predicate `lambda` in which we use binomial expressions represented by the predicate `binom`, and the predicate `sumNFirst` that computes the symbolic sum of the first iterations of a loop by using rewriting rules such as $\sum_{j=0}^{x-1} \binom{j}{k} = \binom{x}{k+1}$. It is then possible to define some rewriting rules on MCR as described in [22] for obtaining a MCR normal form, or define the arithmetic operations on polynomials or lambda expressions for normalizing this other equivalent representation. As illustrated by the definition of `fromSSAtoMCR`, the chains of recurrences are obtained by an exact rewrite of an SSA graph. That justifies our earlier remark: *there is no need to use the chains of recurrences representation*, as the chains of recurrences are nothing else than a subset of the SSA graphs.

```
fromSSAtoMCR(assign(X, lphi(_, _, X + Step)),
             assign(X, unknown)) :-
  hasAself(X, Step, _).
fromSSAtoMCR(assign(X, lphi(LoopId, Init, X + Step)),
             assign(X, mcr(LoopId, Init, Step))).
fromSSAtoMCR(Ibidem, Ibidem).

% Some possible Prolog queries and their answers ``Q = some chain of recurrence'':
% | ?- fromSSAtoMCR(assign(a, lphi(l1, 2, a + 2)), Q).
% Q = assign(a,mcr(l1,2,2))
% | ?- fromSSAtoMCR(assign(a, lphi(l1, 3, a + mcr(l1, 4, 5))), Q).
% Q = assign(a,mcr(l1,3,mcr(l1,4,5)))
```

Fig. 10. From a subset of the SSA graphs to chains of recurrences.

4.3 Replacing Symbols with Abstract Elements

A commonly used technique consists in replacing symbols with a safe description of the values that they represent during the execution of the program. In this case, the safety of the abstract representation and of the abstract operations should be given by some order relation. We illustrate this with the translation

```
fromSSAtoLambda(assign(X, lphi(_, _, X + Step)),
                assign(X, unknown)) :-
  hasAself(X, Step, _).
fromSSAtoLambda(assign(X, lphi(LoopId, Init, X + Step)),
                Init + Result) :-
  sumNFirst(LoopId, LoopId, Step, Result).
fromSSAtoLambda(Ibidem, Ibidem).

% compute the symbolic sum of the first N iterations of LoopId: from 0 to N - 1
sumNFirst(_, N, Constant, Constant * lambda(N, binom(N, 1))) :-
  integer(Constant).
sumNFirst(LoopId, N, Constant * lambda(LoopId, binom(LoopId, K)),
          Constant * lambda(N, binom(N, ResK))) :-
  integer(Constant), fold(K + 1, ResK).
sumNFirst(LoopId, N, A + B, ResA + ResB) :-
  sumNFirst(LoopId, N, A, ResA), sumNFirst(LoopId, N, B, ResB).
sumNFirst(LoopId, N, lambda(LoopId, A + B), lambda(LoopId, ResA + ResB)) :-
  sumNFirst(LoopId, N, A, ResA), sumNFirst(LoopId, N, B, ResB).

% Some possible Prolog queries and their answers ``Q = some lambda expression'':
% | ?- fromSSAtoLambda(assign(a, lphi(l1, 2, a + 2)), Q).
% Q = 2+2*lambda(l1,binom(l1,1))
% | ?- fromSSAtoLambda(assign(a, lphi(l1, 3, a + (4 + 5*lambda(l1,binom(l1,1))))), Q).
% Q = 3+(4*lambda(l1,binom(l1,1))+5*lambda(l1,binom(l1,2)))
```

Fig. 11. From a subset of the SSA graphs to lambda expressions.

of the representation of the meet over all paths in terms of an abstract value in Figure 12. We see that the predicate of the condition-ϕ disappears in the target representation and that the condition-ϕ is replaced by a predicate `meet`. The meet operation can be defined using classic abstractions, such as for example integer intervals, or the set of values. The meet operation is usually defined based on the needs of the computation that uses the abstraction.

```
meetOverAllPaths(assign(X, cphi(_, A, B)), assign (X, meet(A, B))).
meetOverAllPaths(Ibidem, Ibidem).
```

Fig. 12. Definition of a meet over all paths abstraction.

4.4 PROLOG as Prototyping Language

As we have seen in this section, the tools provided by PROLOG suit the needs of prototyping SSA graph transformations. PROLOG's unification engine provides a natural way to describe complex algorithms that transform SSA graphs. The complex description of the algorithms that we have described in [21] is mainly due to the description of the unification algorithm itself [20].

After this very high level description of our induction variable detection algorithm, we have to warn the reader that the implementation of this same algorithm, which has been integrated in GCC, is written in the C language. This has its drawbacks, as illustrated by the size of the resulting implementation, some 3000 lines of C code, but it also is more efficient in terms of execution time than an equivalent program written in PROLOG. Yet, the most important point is that the C language is more portable, a strategic point for GCC. In the next section we describe some experiments that show the central role of our analyzer on optimizations in GCC, and that show the effectiveness of our implementation.

5 Evaluation of Existing Optimizations

For evaluating the importance of our induction variable analysis on existing optimizations in GCC, we used two compilers based on *gcc version 4.1.0 20051104 (experimental)* with the following options "-O3 -msse2 -ftree-vectorize -ftree-loop-linear": the peak compiler is the compiler without any modifications; in the base compiler we disabled the analysis of induction variables. Figure 13 presents the percent improvement for execution time for the SPEC CPU2000 and JavaGrande benchmarks for an AMD Athlon(tm) 64 Processor 3700+ with 128 Kb of L1 cache, 1024 Kb of L2, and 1 GB of RAM, on SuSE with a Linux kernel 2.6.13. Finally, Figure 13 presents the percent improvement of execution time for a part of the MiBench on an ARM XScale-IXP42x processor at 133 MHz with 32 Mb of RAM on Debian with Linux kernel 2.6.12. It is possible to remark that knowing more information about the compiled program can produce worse code, as some of the transformations are applied based on a machine model that is not enough accurate. It is also possible to remark that overall, the use of this information is crucial for aggressive optimizations.

Looking at compilation time issues, a possible technique for stressing our induction variable analyzer is to look at the compilation time of an optimizer that uses our analyzer: the vectorizer is based on a pattern matching of the instructions contained in loops, and on the data dependence information. The design of the data dependence analysis is based on a compromise between precision and compilation-time effectiveness. The results of the data dependence tests are obtained on demand for a loop nest; thus the analysis can be restricted to the loop to be optimized. Furthermore, the dependence tests are ordered such that the fastest tests are executed first, then more expensive analyzes are triggered if the previous ones fail. In order to show the effectiveness of the analysis framework, we have measured the compilation time of the vectorization pass. For the SPEC CPU2000 benchmarks, the vectorization pass does not exceed 1 second per compiled file, or 5 percent of the compilation time per file, showing that the dependence analyzer is fast in practice. This experiment was performed on a Pentium4 2.40 GHz.

These experiments show that our analysis framework is robust for a large set of benchmarks written in several programming languages (C, C++, Java, Fortran) and has a low overhead compilation time.

6 Conclusion and Perspectives

In this paper, we have used PROLOG as a practical tool for prototyping SSA transformations. We have proposed the abstract SSA representation, and we have shown by construction that the chains of recurrences are nothing else than a subset of the SSA graph. We gave a more high level description of the induction variables algorithm that we have integrated to GCC. We evaluated the implementation of our algorithm in the current sources of GCC, and shown that the compiler is robust for a large set of benchmarks written in several programming languages (C, C++, Java, Fortran) and has a low overhead compilation time.

The infrastructure of GCC for loop nest optimizations is quite new, and more work is needed to enhance these preliminary results. However, it constitutes a starting basis for a complete infrastructure for advanced loop restructuring. We are still working on refining the data dependence tests, as optimizations need more precision for enabling more transformations. More precisely, we are working on better data dependence analysis, handling of symbolic dependence tests, and the embedding of polyhedral techniques [15, 24] to refine dependence information.

GCC is much appreciated for its excellent production quality, standards compliance and speed. We believe we have shown that GCC has all the merits for academic research and for target-specific research and development in industry. We hope that the high performance and compilation communities are going to use GCC as their main research and implementation platform in the future.

References

1. Gnu prolog. `http://pauillac.inria.fr/~diaz/gnu-prolog/`.
2. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
3. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.
4. O. Bachmann, P. S. Wang, and E. V. Zima. Chains of recurrences a method to expedite the evaluation of closed-form functions. In *Proceedings of the international symposium on Symbolic and algebraic computation*, pages 242–249. ACM Press, 1994.
5. D. Berlin, D. Edelsohn, and S. Pop. High-level loop optimizations for GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 37–54, 2004. `http://www.gccsummit.org/2004`.
6. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In 4^{th} *ACM Symp. on Principles of Programming Languages*, pages 238–252, Los Angeles, California, Jan. 1977.
7. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

8. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.
9. A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 82–93. ACM Press, 2004.
10. M. P. Gerlek, E. Stoltz, and M. J. Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. on Programming Languages and Systems*, 17(1):85–122, Jan. 1995.
11. L. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in LNCS, pages 406–420. Springer-Verlag, 1993.
12. KAP C/OpenMP for Tru64 UNIX and KAP DEC Fortran for Digital UNIX. `http://www.hp.com/techsevers/software/kap.html`.
13. V. Kislenkov, V. Mitrofanov, and E. Zima. Multidimensional chains of recurrences. In *Proceedings of the 1998 international symposium on symbolic and algebraic computation*, pages 199–206. ACM Press, 1998.
14. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *ACM Symp. on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.
15. V. Loechner and D. Wilde. Parameterized polyhedra and their vertices. *Int. J. of Parallel Programming*, 25(6), Dec. 1997. `http://icps.u-strasbg.fr/PolyLib`.
16. J. Merill. GENERIC and GIMPLE: a new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers Summit*, pages 171–180, 2003. `http://www.gccsummit.org/2003`.
17. S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.
18. D. Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004. `http://www.gccsummit.org/2004`.
19. D. Novillo. Tree SSA - a new optimization infrastructure for GCC. In *Proceedings of the 2003 GCC Developers Summit*, pages 181–193, 2003. `http://www.gccsummit.org/2003`.
20. M. S. Paterson and M. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.
21. S. Pop, A. Cohen, and G.-A. Silber. Induction variable analysis with delayed abstractions. In *2005 International Conference on High Performance Embedded Architectures and Compilers*, 2005. Barcelona, Spain.
22. R. A. van Engelen. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'01)*, pages 118–132, 2001.
23. M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
24. D. Wonnacott and W. Pugh. Nonlinear array dependence analysis. In *Proc. Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, 1995. Troy, New York.

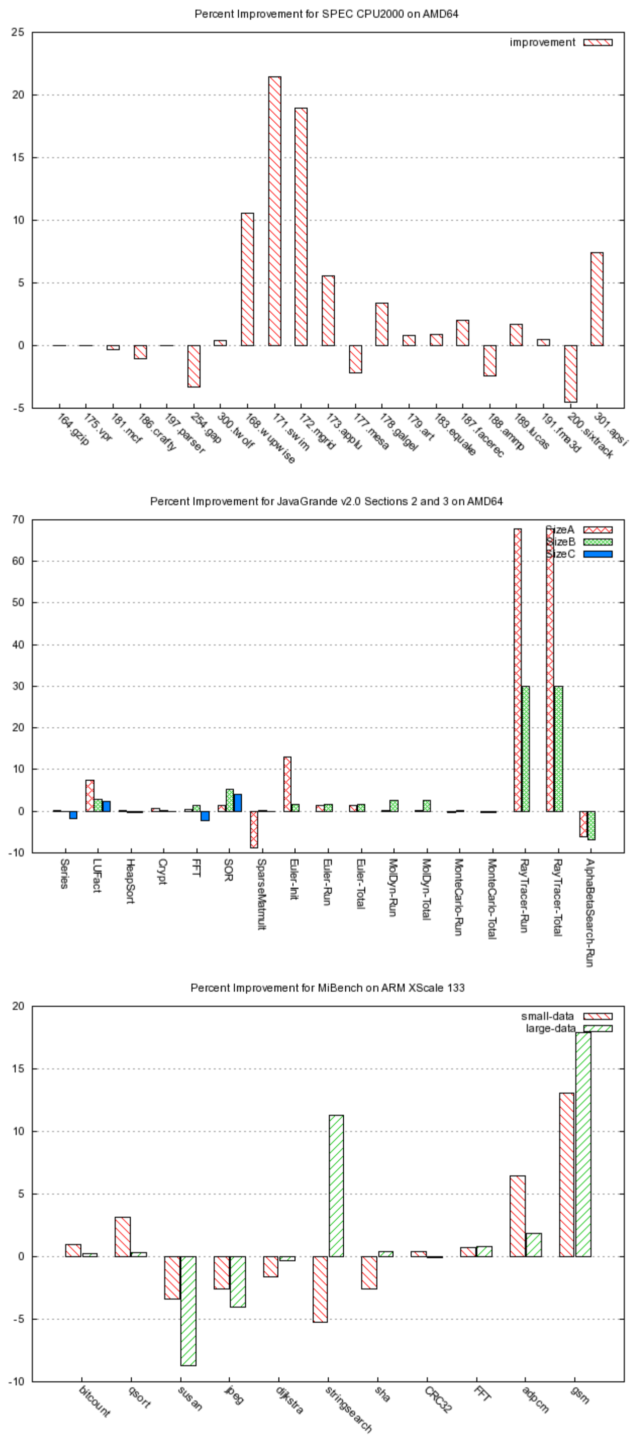

Fig. 13. Percent improvement at run time for SPEC CPU2000, JavaGrandeon AMD64, and for MiBench on ARM.