Dealing with arithmetic overflows in the polyhedral model

Bruno Cuervo Parrino
INRIA Camus
Universidad de Buenos Aires
bcuervo@dc.uba.ar

Julien Narboux
INRIA Camus
Université de Strasbourg
CNRS UMR 7005
Julien.Narboux@inria.fr

Eric Violard
INRIA Camus
Université de Strasbourg
CNRS UMR 7005
Eric.Violard@inria.fr

Nicolas Magaud
INRIA Camus
Université de Strasbourg
CNRS UMR 7005
Nicolas.Magaud@inria.fr

ABSTRACT

The polyhedral model provides techniques to optimize Static Control Programs (SCoP) using some complex transformations which improve data-locality and which can exhibit parallelism. These advanced transformations are now available in both GCC and LLVM. In this paper, we focus on the correctness of these transformations and in particular on the problem of integer overflows. Indeed, the strength of the polyhedral model is to produce an abstract mathematical representation of a loop nest which allows high-level transformations. But this abstract representation is valid only when we ignore the fact that our integers are only machine integers. In this paper, we present a method to deal with this problem of mismatch between the mathematical and concrete representations of loop nests. We assume the existence of polyhedral optimization transformations which are proved to be correct in a world without overflows and we provide a self-verifying compilation function. Rather than verifying the correctness of this function, we use an approach based on a validator, which is a tool that is run by the compiler after the transformation itself and which confirms that the code produced is equivalent to the original code. As we aim at the formal proof of the validator we implement this validator using the Coq proof assistant as a programming language [4].

Keywords

polyhedral model, arithmetic overflows, validator

1. INTRODUCTION

The polyhedral model provides techniques to optimize programs using some complex transformations which improve data-locality and which can exhibit parallelism. Thanks to the Graphite [14] and Polly [5] projects, these advanced transformations are now available in the GCC and LLVM mainstreams compilers. These program transformations are very complex and hence it is hard to guarantee their correctness.

Our goal in collaboration with Alexandre Pilkiwiecz, currently at INRIA Rocquencourt, is to prove formally the correctness of a compiler based on the polyhedral model and to integrate it in the Compcert compiler [7, 8]. Compcert is a compiler for a large subset of the C language which has been proved to be correct using the Coq proof assistant [4].

As the polyhedral transformations apply to affine loop nests in a mathematical framework where each loop variable is considered to be a mathematical integer, and not a machine integer, we must therefore warrant that no arithmetic overflow occurs when the considered loop nests are executed.

There are in fact two problems concerning arithmetic overflows and the polyhedral model:

1. The representation as a polyhedral is incorrect if the original program is performing some overflows.
2. The representation of a polyhedral by a loop using machine integers is incorrect if these integers are not large enough.

In this paper, we focus on these two problems. We assume the existence of an optimization pass in the polyhedral model, which is proved correct in a world *without arithmetic overflows*. First, we propose a solution to produce a compiler which does not ignore the problem of overflows. Second, we propose a validation algorithm for our transformation about overflows. The validator is a function which after every run of the compiler checks *a posteriori* that the transformation is correct. As we aim at a formal proof using the Coq proof assistant, we implemented our validation algorithm using Coq, although the correctness of the validator is not yet proven.

The remainder of this paper is organized as follows: first we describe the related work, then we propose a solution to deal with the overflows in the polyhedral model, and before describing our solution more precisely we provide a small domain specific language. Then we describe our translation

IMPACT 2012
Second International Workshop on Polyhedral Compilation Techniques
Jan 23, 2012, Paris, France
In conjunction with HiPEAC 2012.

http://impact.gforge.inria.fr/impact2012

validation algorithm. Finally we compare our approach to another approach based on an extension of the polyhedral model.

2. RELATED WORK

Arithmetic overflows in Polly. In [5], Tobias Grosser presents the solution he adopted in the implementation of Polly. To solve the first problem, they only deal with loops using signed integers for the array subscripts. This is correct because the C standard states that the behavior of the program is undefined if an overflow occurs during a computation using *signed* integers. But this fact about the C standard does not correspond to the concrete usage of the language as shown by the existence of the options `-fwrapv` and `-no-strict-overflow` of GCC. In particular, having a defined semantics can be useful when writing code to detect overflows. For this reason, Xavier Leroy has chosen to give a precise semantics to signed integer overflows in Compcert (two's complement wrapping). Hence we cannot use this assumption. To solve the second problem *"Polly uses always 64 bit induction variables and signed calculations. This is correct as long as [...], these variables have at most 64 bit size and the schedules of the statements do not produce any larger value"*. Even if we believe that *"However, in practical programs we have not yet seen such an overflow as most loops do not get close to the maximal value possible in a 64 bit counter"*, we cannot use this argument in the context of Compcert because this would require to change the statement of the correctness theorem to restrict it to some class of "practical programs" and then the user of the compiler would have to wonder if his program is in this class or not. Moreover overflows are a source of bugs in critical software, as shown for example by the well-known bug of Ariane flight 501 [9]. Tobias Grosser states that *"we can conveniently derive the minimal type needed to ensure that no unforeseen integer overflows occur"*. We believe that such type does not always exists as the original program may be already using the largest integer type, and hence we need to find another solution.

Translation validation. The idea of translation validation was introduced by Pnueli *et al.* [11]. The use of a validator to check the correctness of the output of a compiler has been applied to GCC [10] and SGI Pro-64 [6]. Jean-Baptiste Tristan has proven some validation algorithms using Coq to certify the output of some optimizations within the Compcert compiler [15, 16, 17, 18]. A validator for the polyhedral model has been proposed by Zuck *et al.* [21], but this validator ignores the problem of arithmetic overflows.

3. A SOLUTION

We cannot use static analysis to detect the presence or absence of overflows because the SCoPs usually contains parameters whose values are unknown at compile time. Hence, we propose to use a dynamic approach. For efficiency reason we can neither perform computations using arbitrary precision nor check the presence of overflows at every computation step. Our solution consists in generating a formula which captures the presence of overflows in the SCoP, then asking to an external tool [19] for a condition about the parameters of SCoP which implies the absence of overflows. Finally we check this condition dynamically. If the condition holds we can use the optimized version of the SCoP. If it does not, in order to preserve the semantics of the SCoP we keep the original version.

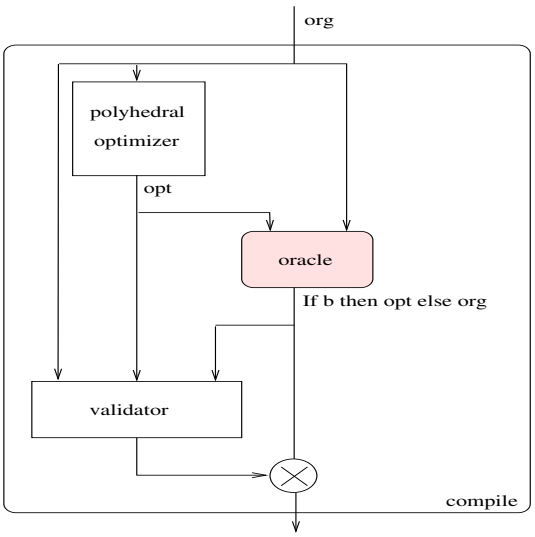

Figure 1: Overview of the approach

Figure 1 illustrates our solution for overcoming the problem of arithmetic overflows and for ensuring the correctness of polyhedral transformations. In addition to the polyhedral optimizer, our compiler uses an oracle and a validator. On the figure, we denote using a grayed-out box the Ocaml code. The correctness of the approach does not depend on the correctness of this piece of code.

The oracle returns a boolean expression (b) which denotes a sufficient condition to ensure that both the original program (org) and the optimized program (opt) do not produce any overflow.

Our transformation then builds a program that we shall call the resulting program in the rest of the paper. This resulting program is of the shape "If b then opt else org". It dynamically evaluates the boolean expression b and executes org, *i.e.* the original program, if the condition is not fulfilled or opt, *i.e.* the optimized program, if the condition holds. The resulting program is then transmitted to the validator.

The validator is a function which takes the original program (org), the optimized program (opt) and the resulting program, and returns a boolean: if it returns true, then the resulting program is equivalent to the original one and our compiler therefore produces the resulting program.

The compile function executes the polyhedral optimizer. If the polyhedral optimizer does not fail ($\neq$ `None`) it executes the oracle. Finally it runs the validator on the output and returns the resulting program only if the validator confirms the equivalence otherwise it returns `None`.

```
Definition compile org :=
```

```
 match (polyhedral_optim org) with
  None => None
| Some opt =>
   let new := If (oracle org opt) opt org in
    if validator org opt new then
      Some new else None
 end.
```

In this work we are interested in ensuring that no arithmetic overflow occurs while transforming loops. Because of that, we assume the existence of a polyhedral optimizer proven correct using integers of arbitrary size, which is the focus of Alexandre Pilkiewicz PhD. This polyhedral optimizer also uses an approach based on a posteriori validation using off-the-shelf polyhedral tools such as Pluto [3], CLooG [1], ISL [19], PPL [13], PolyLib [12].

4. A DOMAIN SPECIFIC LANGUAGE

Here we introduce the language that we will consider. Programs in this language are affine loop nests to which the polytope model is applicable. It is a "toy" language whose main purpose is to reason about arithmetic overflows in polyhedral transformations. For the sake of simplicity we omit some functions (modulo, min, max) which would be required in a more realistic polyhedral language, but these functions do not impact the problem of overflows. Moreover, we assume that the lower bound of a loop is 0 and the loop counter increment is always 1.

4.1 Syntax

Our base language has the following syntax:

$$\begin{array}{lcl} l & ::= & n \mid x \mid n * l \mid l + l \mid l - l \\ e & ::= & n \mid x \mid T[l] \mid e + e \mid e - e \mid e * e \mid e/e \\ i & ::= & \text{skip} \mid T[l] := e \mid i; i \mid \text{Loop } x \text{ from 0 to } l \text{ do } i \text{ done} \end{array}$$

In this language, l represents linear index expressions, e expressions with integer values, and i the instructions of the language we consider. In this grammar x stands for variables and n for integers.

We also add the possibility to enclose our SCoPs inside a conditional instruction:

$$\begin{array}{lcl} b & ::= & \text{true} \mid \text{false} \mid l \leq l \mid b \text{ and } b \mid b \text{ or } b \\ j & ::= & \text{If } b \text{ then } i \text{ else } i \end{array}$$

The construct "If b then i else i" allows to describe programs which branch depending on a condition. This construction only appears in the resulting program where the condition is the one ensuring computations do not trigger overflows.

4.2 Semantics

In this section we present a structural operational semantics for our language allowing formal analysis of the behaviour of programs. We aim at proving that the original program and the resulting one behave the same way. This operational semantics has been formalized using Coq.

In fact, we give two semantics for our language: a "concrete" one where index variables are machine integers, *i.e.*, 32-bit signed integers represented using two's complement, and an "abstract" one where index variables are mathematical integers. For the sake of concision, our semantics is parametrized by ι which can take two values o or no to refer to the concrete (with overflows) or the abstract semantics (without overflows) respectively.

Linear expressions. The semantics of linear expressions is summarized in Fig. 3, where $+_o$ denotes the addition modulo and $+_{no}$ denotes the standard addition over mathematical integers. To evaluate a linear expression we must know what values the variables stand for. Therefore the semantics is defined using an environment which is a function denoted by $\mathcal{L}$ from the set of variables to the set of values. The judgment $\mathcal{L} \vdash_\iota e \longrightarrow n$ means that the linear expression e evaluates to the integer n in environment $\mathcal{L}$.

Expressions. The semantics for arithmetic expressions is described in Fig. 4. We assume that we have only the four usual arithmetic expressions, in a more realistic language we could add calls to some pure, side-effect free functions. The semantics is defined using an environment E which is a pair $\mathcal{L}$,$\mathcal{T}$, where $\mathcal{T}$ is a function which maps array names to tuples of values. Note that as the polyhedral optimization does not change the actual computations performed by the code but only their order, the **non linear** arithmetic expressions could contain any side-effect free function. The absence or presence of overflow in the **non linear** arithmetic expressions is not important for our study, hence in our toy language, we just assume the standard arithmetic expressions with overflows (+ - * /).

$$\frac{}{\mathcal{L} \vdash_\iota n \longrightarrow n} \qquad \frac{\mathcal{L}(x) = n}{\mathcal{L} \vdash_\iota x \longrightarrow n}$$

$$\frac{\mathcal{L} \vdash_\iota l_1 \longrightarrow n_1 \qquad \mathcal{L} \vdash_\iota l_2 \longrightarrow n_2}{\mathcal{L} \vdash_\iota l_1 + l_2 \longrightarrow n_1 \ +_\iota \ n_2}$$

$$\frac{\mathcal{L} \vdash_\iota l_1 \longrightarrow n_1 \qquad \mathcal{L} \vdash_\iota l_2 \longrightarrow n_2}{\mathcal{L} \vdash_\iota l_1 - l_2 \longrightarrow n_1 \ -_\iota \ n_2}$$

$$\frac{\mathcal{L} \vdash_\iota l \longrightarrow m}{\mathcal{L} \vdash_\iota n * l \longrightarrow n \ \times_\iota \ m}$$

Figure 3: Semantics for Index (linear) Expressions

Instructions. The semantics for instructions is defined as usual (see Fig. 2). Note that we use two constructs for loops. The constructor `Loop` defines loops in our language. In order to express its semantics, we have an alternate instruction `Loop'`, which corresponds to the loop after its initialization. We assume that loops are always incremented by one. To formalize the semantics we introduced some notation for modifying environments. For instance the notation $\mathcal{L}[x \mapsto v]$ stands for the environment mapping x to v and all other variables y to $\mathcal{L}(y)$.

Additional constructs for dealing with overflows. Dealing with overflows requires to add a conditional instruction to our language. Indeed, depending on some preconditions

$$\frac{E = \mathcal{L},\mathcal{T} \quad \mathcal{L} \vdash_\iota l \longrightarrow i \quad 0 \le i < n \quad \mathcal{T}(T) = v_0 v_1 \cdots v_{n-1} \quad \mathcal{L},\mathcal{T} \vdash_\iota e \longrightarrow v}{E \vdash_\iota T[l] := e \rightarrow (\text{skip} \ , \mathcal{L}, \mathcal{T}[T \mapsto v_0 v_1 \cdots v_{i-1} v v_{i+1} \cdots v_{n-1}])}$$

$$\frac{}{E \vdash_\iota \text{skip}; i \rightarrow (i \ , E)} \qquad \frac{E \vdash_\iota i_1 \rightarrow (i_1' \ , E')}{E \vdash_\iota i_1; i_2 \rightarrow (i_1'; i_2 \ , E')}$$

$$\frac{E = \mathcal{L},\mathcal{T} \qquad \mathcal{L}(x) = m \qquad m \ge n}{E \vdash_\iota \text{Loop' } x \text{ from 0 to } n \text{ do } i \text{ done} \rightarrow (\text{skip} \ , E)}$$

$$\frac{E = \mathcal{L},\mathcal{T} \qquad \mathcal{L}(x) = m \qquad m < n}{E \vdash_\iota \text{Loop' } x \text{ from 0 to } n \text{ do } i \text{ done} \rightarrow (i; \text{Loop' } x \text{ from 0 to } n \text{ do } i \text{ done} \ , \mathcal{L}[x \mapsto m+1], \mathcal{T})}$$

$$\frac{E = \mathcal{L},\mathcal{T} \qquad \mathcal{L} \vdash_\iota l \longrightarrow n}{E \vdash_\iota \text{Loop } x \text{ from 0 to } l \text{ do } i \text{ done} \rightarrow (\text{Loop' } x \text{ from 0 to } n \text{ do } i \text{ done} \ , \mathcal{L}[x \mapsto -1], \mathcal{T})}$$

Figure 2: Semantics for Instructions

$$\frac{}{E \vdash_\iota n \longrightarrow n} \qquad \frac{E = \mathcal{L},\mathcal{T} \quad \mathcal{L}(x) = n}{E \vdash_\iota x \longrightarrow n}$$

$$\frac{E \vdash_\iota e_1 \longrightarrow n_1 \qquad E \vdash_\iota e_2 \longrightarrow n_2}{E \vdash_\iota e_1 + e_2 \longrightarrow n_1 +_o n_2}$$

$$\frac{E \vdash_\iota e_1 \longrightarrow n_1 \qquad E \vdash_\iota e_2 \longrightarrow n_2}{E \vdash_\iota e_1 - e_2 \longrightarrow n_1 -_o n_2}$$

$$\frac{E \vdash_\iota e_1 \longrightarrow n_1 \qquad E \vdash_\iota e_2 \longrightarrow n_2}{E \vdash_\iota e_1 * e_2 \longrightarrow n_1 \times_o n_2}$$

$$\frac{E \vdash_\iota e_1 \longrightarrow n_1 \qquad E \vdash_\iota e_2 \longrightarrow n_2 \qquad n_2 \neq 0}{E \vdash_\iota e_1 / e_2 \longrightarrow n_1 /_o n_2}$$

$$\frac{E = \mathcal{L},\mathcal{T} \quad \mathcal{T}(T) = v_0 \cdots v_{n-1} \quad E \vdash_\iota l \longrightarrow i \quad 0 \le i < n}{E \vdash_\iota T[l] \longrightarrow v_i}$$

Figure 4: Semantics for Arithmetic Expressions

$$\frac{}{\mathcal{L} \vdash \text{true} \longrightarrow true} \qquad \frac{}{\mathcal{L} \vdash \text{false} \longrightarrow false}$$

$$\frac{\mathcal{L} \vdash_o l_1 \longrightarrow n_1 \qquad \mathcal{L} \vdash_o l_2 \longrightarrow n_2 \qquad n_1 \le n_2}{\mathcal{L} \vdash l_1 \le l_2 \longrightarrow true}$$

$$\frac{\mathcal{L} \vdash_o l_1 \longrightarrow n_1 \qquad \mathcal{L} \vdash_o l_2 \longrightarrow n_2 \qquad n_1 > n_2}{\mathcal{L} \vdash l_1 \le l_2 \longrightarrow false}$$

$$\frac{\mathcal{L} \vdash b \longrightarrow v \qquad \mathcal{L} \vdash b' \longrightarrow v'}{\mathcal{L} \vdash b \text{ and } b' \longrightarrow v \ \wedge \ v'}$$

$$\frac{\mathcal{L} \vdash b \longrightarrow v \qquad \mathcal{L} \vdash b' \longrightarrow v'}{\mathcal{L} \vdash b \text{ or } b' \longrightarrow v \ \vee \ v'}$$

Figure 5: Semantics for Boolean Expressions

$$\frac{E = \mathcal{L},\mathcal{T} \qquad \mathcal{L} \vdash b \longrightarrow true}{E \vdash_o \text{If } b \text{ then } i \text{ else } i' \rightarrow (i \ , \mathcal{L}, \mathcal{T})}$$

$$\frac{E = \mathcal{L},\mathcal{T} \qquad \mathcal{L} \vdash b \longrightarrow false}{E \vdash_o \text{If } b \text{ then } i \text{ else } i' \rightarrow (i' \ , \mathcal{L}, \mathcal{T})}$$

Figure 6: Semantics for Conditional Instructions

which can be statically determined but only dynamically checked, we should use the optimized program or stick to the original one. For the evaluation of these preconditions, we use the concrete semantics which may produce overflows (see Fig. 5). We will see in section 6 that for the correctness of our approach we will need to refine this definition. The semantics for conditional instructions is given in Fig. 6.

5. THE ORACLE

We also have an oracle that given the original and optimized programs, returns a sufficient condition to ensure that no overflow occurs. To build this oracle we first define a function which captures the presence of overflows in the SCoP. We call this function `cond_overflow` and define it by induction on the structure of programs:

$$\begin{aligned}
&\texttt{cond_overflow}(\text{skip}) = \text{true}\\
&\texttt{cond_overflow}(T(l) := e) =\\
&\quad \textstyle\bigwedge_{expr \in subexpr(l) \cup lsubexpr(e)} (\texttt{MININT} \le expr \le \texttt{MAXINT})\\
&\texttt{cond_overflow}(i_1; i_2) =\\
&\quad \texttt{cond_overflow}(i_1) \wedge \texttt{cond_overflow}(i_2)\\
&\texttt{cond_overflow}(\text{Loop } x \text{ from 0 to } l \text{ do } i \text{ done}) =\\
&\quad 0 \le x \le l \Rightarrow \texttt{cond_overflow}(i) \wedge\\
&\quad \textstyle\bigwedge_{expr \in subexpr(l)} (\texttt{MININT} \le expr \le \texttt{MAXINT})
\end{aligned}$$

`subexpr(l)` denotes the set of all the sub-expressions of the linear expression l. `lsubexpr(e)` denotes the set of all the sub-expressions of the linear expression which appear in the

array accesses of e.

Note that in the context of certified compilers, we can analyse the overflows in the source language. If the compiler performs optimizations which do not preserve the structure of subexpressions and the absence/presence of overflows, then the analysis should be carried out on the machine code.

The oracle is implemented using Ocaml and it makes use of the `iscc` calculator that offers an interface to the barvinok library [20]. This library allows amongst other things to simplify a set of linear inequalities and also to count the number of elements in parametric affine sets.

Example. Let us consider the following program written in our domain specific language:

```
p ::    Loop i from 0 to n do
          Loop j from 0 to m do
            C[i+j] = C[i+j] + A[i]*B[j]
          done
        done
```

We look for a boolean expression involving only the parameters (*i.e.* n and m) and denoting a sufficient condition to ensure that no overflow occurs during the execution of program p. To construct this expression, first, we compute the expression `cond_overflow`(p), which denotes the condition of absence of overflow for this SCoP:

`cond_overflow`$(p) = (0 \leq i \leq n \implies (0 \leq j \leq m \implies -2^{31} \leq i+j \leq 2^{31}-1))$.

From this expression, we then build a query for the `iscc` tool. The query expresses the set of values (n, m) for which no overflow occurs:

$$\{(n,m) \mid \forall i.\forall j.(0 \leq i \leq n \implies (0 \leq j \leq m \implies -2^{31} \leq i+j \leq 2^{31}-1))\}$$

This formula has to be rewritten in order to use only operations that are allowed by `iscc`. It is rewritten as:

$$\{(n,m) \mid \neg(\exists i.\exists j.(0 \leq i \leq n \wedge (0 \leq j \leq m \wedge (-2^{31} > i+j \vee i+j > 2^{31}-1))))\} =$$

$$\{(n,m) \mid \text{true}\} - \{(n,m) \mid \exists i.\exists j. (0 \leq i \leq n \wedge (0 \leq j \leq m \wedge (-2^{31} > i+j \vee i+j > 2^{31}-1)))\}$$

Here is the query that we send to `iscc`:

```
 { [n,m] } - { [n,m] :  exists i :  exists j :
0 <= i <= n and 0 <= j <= m and
(-2147483648 > i+j or i+j > 2147483647) };
```

And here is the result of the query:

```
 { [n, m] :  m <= -1 or (m >= 0 and n <= -1) or
(m >= 0 and n >= 0 and m <= 2147483647 - n) }
```

Finally, we obtain the boolean expression:

$$m \leq -1 \text{ or } (0 \leq m \text{ and } n \leq -1) \text{ or } (0 \leq m \text{ and } 0 \leq n \text{ and } m \leq 2147483647 - n)$$

Note that this formula expresses the fact that there is no overflow if we don't even enter into the inner or outer loop ($m \leq -1$ or $n \leq -1$). Note also that the formulas we obtain from the `iscc` tool are linear expressions w.r.t the parameters.

6. THE VALIDATOR

As we aim at a formal proof using the Coq proof assistant, we implemented our validation algorithm using Coq and use the Coq extraction mechanism to generate an Ocaml program.

Our validator takes three programs as input: the original program (*org*), the optimized program (*opt*) which is obtained from *org* by applying the polyhedral optimizer (assuming no overflow occurs in *org*) and the resulting program (*new*). If the validator returns true, then it means that the resulting program is equivalent to the original one. If the validator returns false, then it means that we do not know. Here is the definition of our validator in Coq:

```
Definition validator
   (org: instr) (opt: instr) (new: instrgen) :=
  match new with
     If b p q =>
       (instr_eq p opt)  &&
       (instr_eq q org)  &&
       disjoint_lists (loopvarlist p) (bvars b) &&
       disjoint_lists (loopvarlist q) (bvars b) &&
       overflow_checker p b &&
       overflow_checker q b
   | Instr st => false
  end.
```

Our validation algorithm is the following one: we first look at the shape of the resulting program. The resulting program must have the shape "If b then p else q" where p is equal to *opt* (`instr_eq p opt`) and q is equal to *org* (`instr_eq q org`). Moreover, we verify that the boolean expression b involves only parameters, *i.e.*, it does not contain any loop variable. Last, we check that b is a sufficient condition to ensure that no overflow occurs during the execution of program p and program q.

Remark Here we assume every variable is of type 32-bit signed integer. Note that in a more realistic language, the function `cond_overflow` should be parametrized by the real type of the involved variables. The original program may use signed and unsigned variables. The type of index variables of the original program may be different from the ones of the optimized version. Depending on the polyhedral transformation the type of the index variables may (should) be 64-bit signed integers, in order to reduce the chance of overflows. Note also that the overflow conditions may reduce to true if it can be determined statically that no overflow can occur.

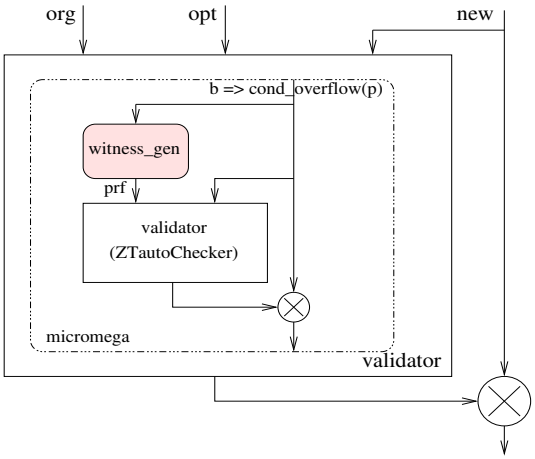

Figure 7: Two nested validators

The function `overflow_checker`(p, b) calls the internal Coq function of the tactic `micromega` [2] to check whether the condition $b \Rightarrow$ `cond_overflow`(p) holds. Note that this Coq tactic is also based on an approach using a validator. Thus we have two nested validators as shown in Fig.7 which focuses on the validator box of Fig.1. As in Fig.1, we denote by a grayed-out box the Ocaml code which does not belong to the trusted code base. The proof of the formula $b \Rightarrow$ `cond_overflow`(p) is generated by an external tool (`witness_gen`) and the Coq function `ZTautoChecker` checks that the external proof is correct.

```
Definition overflow_checker p b :=
  let f := Tauto.I (bformula_of_bexpr b)
                   (condOverflow_to_bformula p) in
  let w := witness_gen f in
   match w with
     None => false
   | Some prf => ZTautoChecker f prf end.
```

In this code written in Coq, the variable `f` represents the formula $b \Rightarrow$ `cond_overflow`(p) in the internal datatype of `micromega`.

Dealing with overflow in the evaluation of condition for absence of overflow. The micromega checker also assumes that expressions are considered in a mathematical world. But during the evaluation of b an overflow may also occur. In this case, we are not sure that b holds and then we cannot conclude from $b \Rightarrow$ `cond_overflow`(p) that `cond_overflow`(p). To deal with this problem, we could assume that b is evaluated in a world without arithmetic overflows using an arbitrary precision library. This assumption could be implemented only using a library such as GNU MP for example. In the context of a compiler, especially Compcert this would require to prove formally an arbitrary precision library within the compiler. We choose a simpler solution which consists in checking if the evaluation of b produces an overflow and executing the original program in this case. To state this we need to change the operational semantics associated with boolean expressions of our language. The new semantics is presented in Figure 9. We introduce a judgment $\mathcal{L} \vdash_{oc} l \longrightarrow (n, e)$ for the evaluation of linear expression with overflow checking (see Figure 8). In this judgment, e is a boolean which is *true* if the evaluation of the linear expression l does not produce any overflow and $false$ otherwise. To implement this judgment we would need either inline assembly to obtain the processor flag if there is one, or using small built-in functions to get the same effect. We change the evaluation of the boolean expression: it returns false if an overflow occurred during the evaluation and returns the boolean value otherwise.

$$\frac{}{\mathcal{L} \vdash_{oc} n \longrightarrow (n, true)} \qquad \frac{\mathcal{L}(x) = n}{\mathcal{L} \vdash_{oc} x \longrightarrow (n, true)}$$

$$\frac{\mathcal{L} \vdash_{oc} l_1 \longrightarrow (n_1, e_1) \qquad \mathcal{L} \vdash_{oc} l_2 \longrightarrow (n_2, e_2)}{\mathcal{L} \vdash_{oc} l_1 + l_2 \longrightarrow (n_1 +_o n_2, (\mathrm{OK}(n_1 +_{no} n_2) \wedge e_1 \wedge e_2))}$$

$$\frac{\mathcal{L} \vdash_{oc} l_1 \longrightarrow (n_1, e_1) \qquad \mathcal{L} \vdash_{oc} l_2 \longrightarrow (n_2, e_2)}{\mathcal{L} \vdash_{oc} l_1 - l_2 \longrightarrow (n_1 -_o n_2, (\mathrm{OK}(n_1 -_{no} n_2) \wedge e_1 \wedge e_2))}$$

$$\frac{\mathcal{L} \vdash_{oc} l \longrightarrow (m, e)}{\mathcal{L} \vdash_{oc} n * l \longrightarrow (n \times_o m, (\mathrm{OK}(n \times_{no} m) \wedge e))}$$

where $\mathrm{OK}(p) = -2^{31} \leq p \leq 2^{31} - 1$

Figure 8: Semantics for Index (linear) Expressions with Overflow Checking

Soundness of the validation algorithm. To state the soundness of the validation algorithm we need to define program equivalence. As our SCoP denotes only programs which terminate we can use the following definition using the reflexive-transitive closure of the smallstep semantics:

Definition 6.1 (Program equivalence). $p \equiv_\iota p'$ *iff for all environments* E *and* E',

$$E \vdash_\iota p \rightarrow^* (skip\ ,\ E') \Leftrightarrow E \vdash_\iota p' \rightarrow^* (skip\ ,\ E').$$

Assuming that for any program p, $p \equiv_{no}$ polyhedral_optim(p) we make the following conjecture which states the soundness property of our validator:

Conjecture 6.2 (Validator soundness). *if* $p \equiv_{no} p'$ *and* `validate(p, p', new) = true`*, then* $p \equiv_o$ *new*

7. DISCUSSION

As pointed by Tobias Grosser and Sven Verdoolaege, there are also some polyhedral libraries such as ISL which implement an extension of the polyhedral model which can deal

$$\frac{}{\mathcal{L} \vdash \text{true} \longrightarrow true} \qquad \frac{}{\mathcal{L} \vdash \text{false} \longrightarrow false}$$

$$\frac{\mathcal{L} \vdash_{oc} l_1 \longrightarrow (n_1, true) \qquad \mathcal{L} \vdash_{oc} l_2 \longrightarrow (n_2, true) \qquad n_1 \leq n_2}{\mathcal{L} \vdash l_1 \leq l_2 \longrightarrow true}$$

$$\frac{\mathcal{L} \vdash_{oc} l_1 \longrightarrow (n_1, e_1) \qquad \mathcal{L} \vdash_{oc} l_2 \longrightarrow (n_2, e_2) \qquad \neg e_1 \vee \neg e_2 \vee (n_1 > n_2)}{\mathcal{L} \vdash l_1 \leq l_2 \longrightarrow false}$$

$$\frac{\mathcal{L} \vdash b \longrightarrow v \qquad \mathcal{L} \vdash b' \longrightarrow v'}{\mathcal{L} \vdash b \text{ and } b' \longrightarrow v \ \wedge \ v'} \qquad \frac{\mathcal{L} \vdash b \longrightarrow v \qquad \mathcal{L} \vdash b' \longrightarrow v'}{\mathcal{L} \vdash b \text{ or } b' \longrightarrow v \ \vee \ v'}$$

Figure 9: Semantics for Boolean Expressions with Overflow Checking

with piecewise affine functions. These extensions are sufficient to model the overflow behavior of signed and unsigned integer computations. Using such a library it is possible to generate optimized versions of the original program which are correct even in the presence of overflows. It means that the semantics of the loop is defined even for the values of the parameters which produce overflows. For the values of the parameters which do not produce overflow we think this approach would generate a code similar to our approach, but it would also generate code for the value of parameters which do produce overflows. If the optimizer tries to generate the same code for the two cases this may block some optimizations.

8. CONCLUSION AND FUTURE WORK

We proposed a way to preserve the polyhedral representation of loop nests from correctness bugs involving arithmetic overflows. We believe that our approach does not reduce the efficiency of the polyhedral optimizations because it adds a simple boolean test per SCoP. We also proposed a validator to ensure after every run of the compilation pass that the transformation is correct.

Our next goal is to prove formally using the Coq proof assistant the correctness of the validator. In the future we should extend our approach to dynamically check the absence of pointer alias.

Acknowledgments. We would like to thank our six reviewers for their numerous remarks which helped improving this paper. We also would like to thank Albert Cohen, Tobias Grosser and Sven Vooderlaege for their very precise comments about this work.

9. REFERENCES

[1] C. Bastoul. Generating loops for scanning polyhedra. Technical Report 2002/23, PRiSM, Versailles University, 2002. Related to the CLooG tool.

[2] F. Besson. Fast reflexive arithmetic tactics: the linear case and beyond. In *In Types for Proofs and Programs (Types'06)*, volume 4502 of *LNCS*, pages 48–62. Springer, 2007.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. P.: Pluto: A practical and fully automatic polyhedral program optimization system. In *In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08)*, pages 101–113, New York, NY, USA, 2008. ACM.

[4] Coq development team, The. *The Coq Proof Assistant Reference Manual, Version 8.3*. LogiCal Project, 2011.

[5] T. Grosser. Enabling polyhedral optimizations in llvm. Master's thesis, University of Passau, April 2011.

[6] A. Kanade, A. Sanyal, and U. P. Khedker. A PVS Based Framework for Validating Compiler Optimizations. In *SEFM*, pages 108–117. IEEE Computer Society, 2006.

[7] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, July 2009.

[8] X. Leroy. The Compcert verified compiler, software and commented proof. Available at `http://compcert.inria.fr/`, Jan. 2010.

[9] J. L. Lions. Ariane 5 - flight 501 failure. Technical report, 1996. `http://en.wikisource.org/wiki/Ariane_501_Inquiry_Board_report`.

[10] G. C. Necula. Translation validation for an optimizing compiler. *SIGPLAN Not.*, 35:83–94, May 2000.

[11] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166, London, UK, 1998. Springer-Verlag.

[12] Polylib. `http://icps.u-strasbg.fr/PolyLib`.

[13] PPL: The parma polyhedra library. `http://www.cs.unipr.it/ppl/`.

[14] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, and R. Upadrasta. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *GCC Research Opportunities Workshop (GROW'10)*, Pisa, Italy, Jan. 2010.

[15] J.-B. Tristan. *Formal verification of translation validators*. These, Université Paris-Diderot - Paris VII, Nov. 2009.

[16] J.-B. Tristan and X. Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *35th ACM Symposium on Principles of Programming Languages (POPL 2008)*, pages 17–27, San Francisco, United States, 2008. ACM, ACM Press.

[17] J.-B. Tristan and X. Leroy. Verified Validation of Lazy Code Motion. In *ACM SIGPLAN conference on Programming Language Design and Implementation*

(PLDI), pages 316–326, Dublin, Ireland, June 2009. ACM.

[18] J.-B. Tristan and X. Leroy. A simple, verified validator for software pipelining. In ACM, editor, *37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, Madrid, Spain, Jan. 2010.

[19] S. Verdoolaege. *barvinok: User Guide*, Sept. 2011. http://www.cs.kuleuven.be/cgi-bin/dtai/barvinok.cgi.

[20] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, June 2007.

[21] L. Zuck, A. Pnueli, Y. Fang, B. Goldberg, and Y. Hu. Translation and run-time validation of optimized code. In *In 2nd Workshop on Runtime Verification*, pages 180–201, 2002.