Symbolic Expression Evaluation to Support Parallelizing Compilers

Thomas Fahringer
Institute for Software Technology and Parallel Systems
University of Vienna
Liechtensteinstrasse 22, A-1090 Vienna, Austria
e-mail: tf@par.univie.ac.at

Published in Proc. of 5th IEEE Euromicro Workshop on Parallel and Distributed Processing, pp. 173-180, London, UK, Jan. 1997

Abstract

Symbolic analysis is of paramount importance to further advance the state-of-the-art of parallelizing compilers. The quality of various compiler analyses and optimizing code transformations depend on the ability to evaluate symbolic expressions for equality and inequality ($=, <, >$) relationships. This paper describes a powerful algorithm that computes lower and/or upper bounds of wide classes of linear and non-linear symbolic expressions given a set of constraints on loop variables and loop invariants. The algorithm is used to compare symbolic expressions, examine non-linear array index functions for data dependences, and simplify systems of constraints. Among others the algorithm supports dependence analysis, detecting zero-trip-loops, dead code elimination, and performance prediction. We have implemented the algorithm and use it as part of a parallelizing compiler and a static performance estimator.

1 Introduction

Many parallelizing compilers fail to effectively parallelize programs caused by the deficit of the compiler to analyze complex – in particular non-linear – loop bounds, conditionals, and array index functions. Non-linear symbolic expressions are commonly caused by induction variable substitution, linearizing arrays, parameterizing data parallel programs with symbolic number of processors and/or problem size, and so forth. Numerous researchers [7, 2, 9, 8] have reported on the occurrence of non-linear symbolic expressions in practical codes and the need of effective techniques to analyze such programs. Non-linear symbolic expressions seriously hamper crucial compiler analysis including testing for data dependences, optimizing communication, simplifying code, dead code elimination, performance prediction, detecting zero-trip-loops, etc.

Consider the loop nest in Example 1.1 which contains linearized and indirect array referencing and is very similar to a code excerpt as found in the TRFD code of the Perfect Benchmarks [3]. In loop $L1$ array IA is initialized with a non-linear array index function $(I*(I-1))/2$, which is then used in $S4$ to read $X(IJ, KL)$ and write $X(KL, IJ)$ with $IJ = (I*(I-1))/2+J$ and $KL = (K*(K-1))/2+L$, respectively. In order to examine whether loop $L5$ can be executed in parallel, we must determine whether statement $S4$ implies a true dependence (array elements are written before they are read) with respect to $L5$. Conventional dependence tests cannot examine non-linear array index functions and therefore assume a data dependence that sequentializes loop $L5$. This paper describes an algorithm that can compare non-linear array index functions such as KL and IJ, and consequently determines the absence of a true dependence, which enables the underlying compiler to parallelize loop $L5$.

Example 1.1

```
L1:  DO I=1,N
S1:      IA(I)=(I*(I-1))/2
     ENDDO
     ...
L2:  DO I=1,N
L3:      DO J=1,I
S2:          IJ=IA(I)+J
             ...
L4:          DO K=1,I
                 ...
L5:              DO L=1,J-1
S3:                  KL=IA(K)+L
                     ...
S4:                  X(KL,IJ)=X(IJ,KL)+VAL
                 ENDDO
             ENDDO
         ENDDO
     ENDDO
```

For the loop nest in Example 1.2 a compiler may determine whether the condition of statement S1 ever evaluates to TRUE. If not then we can simply eliminate the entire conditional statement (dead code elimination). Detecting zero-trip-loops [8] is a similar problem which tries to determine whether the loop body of a given loop nest is ever executed. Other related problems require loop iteration or statement execution counts which are key figures to estimate a program's performance [7, 4]. All of these problems can be formulated as a set of linear and non-linear constraints $\mathcal{I}$ defined over loop *variables* and *parameters* (loop invariants), which are commonly derived from loop bounds and conditional statements. For instance, $\mathcal{I}$ is given by $\{1 \leq I1 \leq N, N/(2*I1) \leq I2 \leq N, I1*I2 \leq N\}$ based on Example 1.2. Most analyses based on systems of constraints are significantly alleviated if $\mathcal{I}$ is simplified by eliminating redundant constraints and detecting contradictions.

Example 1.2

```
     DO I1=1,N
       DO I2=N/(2*I1),N
S1:      IF (I1*I2 ≤ N) THEN
           A(I1,I2) = ...
         ENDIF
         ...
       ENDDO
     ENDDO
```

This paper describes an algorithm that computes lower and/or upper bounds of wide classes of linear and non-linear symbolic expressions given a set of constraints defined over loop variables and loop invariants. The algorithm is based on substituting variables and parameters by their associated lower and upper bound expressions. Rewrite rules for lower and upper bound functions and simplification techniques are applied until the symbolic expression consists only of constants, $-\infty$ or ∞. Among others this algorithm can be used to compare symbolic expressions for equality and inequality ($=, <, >$) relationships, examine non-linear array index functions for data dependences, and detect redundant inequalities in a set of constraints.

We have implemented the algorithm and use it as part of P^3T [5, 4], a performance estimator for parallel programs, and *VFCS* [1], a parallelizing compiler for data parallel programs on distributed memory parallel architectures. Experiments will be shown that demonstrate the usefulness of our approach.

This paper is organized as follows: In Section 2 we present our algorithm for computing lower and upper bounds of symbolic expressions and how this algorithm is used to compare symbolic expressions and examine data dependences of non linear array index functions. Section 3 shows how to simplify a set of symbolic constraints. Related work is discussed in Section 4. Section 5 gives a detailed example of how to detect redundant inequalities in a set of symbolic constraints. Conclusions and final remarks are given in Section 6.

2 Evaluating Symbolic Expressions

In this section we present our algorithm which compares symbolic expressions for equality or inequality relationships. The following notations and definitions are used in the remainder of this paper:

- Let $\mathcal{I}$ be a set of non-linear and linear constraints defined over loop *variables* and *parameters* (loop invariants) which are derived from the loop bounds and conditional statements of a program. $\mathcal{V}$ is the set of variables and $\mathcal{P}$ the set of parameters appearing in $\mathcal{I}$.
- $low(E)$ and $up(E)$ are two functions, which define the minimum and maximum value of a symbolic expression E for all values of variables and parameters appearing in E. These two functions (up and low) will be referred to as *δ-functions*. $\delta(E)$ which is either $low(E)$ or $up(E)$ is denoted as *δ-expression*. Note that E may again contain δ-expressions.
- A *symbolic (integer-valued) expression* E may consist of arbitrary division, multiplication, subtraction, addition, exponentiation, maximum, minimum, and δ-functions. Its operands can be variables, parameters, integer constants and infinity symbols ($\infty, -\infty$).

2.1 Algorithm

Figure 1 shows algorithm EXPR_BOUND which tries to compute the lower and/or upper bound of a symbolic expression E defined in $\mathcal{V} \cup \mathcal{P}$. The input to EXPR_BOUND is E, $\mathcal{B}$, and $\mathcal{Q}$. $\mathcal{B}$ is the set of known upper and lower bound expressions (constraints) for each variable and parameter. These constraints are derived from $\mathcal{I}$, the set of constraints of the underlying problem.

Assume that the cardinality of $\mathcal{V}$ and $\mathcal{P}$ respectively is n and m, then $\mathcal{Q}$=$\{(v_1;v_1^l,v_1^u)$,...,$(v_n;v_n^l,v_n^u)$, $(p_1;p_1^l,p_1^u)$,...,$(p_m;v_m^l,v_m^u)\}$. $(v_i; v_i^l, v_i^u)$ – where $1 \leq i \leq n$ – specifies a variable $v_i \in \mathcal{V}$ with its associated maximum lower bound v_i^l and minimum upper bound v_i^u. $(p_j; p_j^l, p_j^u)$ – where $1 \leq j \leq m$ – specifies a parameter $p_j \in \mathcal{P}$ with its associated maximum lower bound p_j^l and minimum upper bound p_j^u. The algorithm maintains the maximum lower and minimum upper bound for each variable and parameter in $\mathcal{Q}$. If a new lower bound for a variable is deduced by the algorithm that is larger than the current maximum lower bound as stored in $\mathcal{Q}$, then $\mathcal{Q}$ is updated accordingly. Similar accounts for the minimum upper bound.

Let us continue Example 1.1 of Section 1. The set of constraints $\mathcal{I}$ for this problem is derived from the corresponding loop nest. $\mathcal{I}$ = {1≤I ≤ N,1 ≤ J ≤ I,1 ≤ K ≤ I,1 ≤ L ≤ J-1 }. This yields the following sets of variables, parameters, lower and upper bounds:

EXPR_BOUND(E,$\mathcal{B}$,$\mathcal{Q}$)

- **INPUT:**
  E: δ-expression defined in $\mathcal{V} \cup \mathcal{P}$
  $\mathcal{B}$: Set of lower and upper bound expressions for variables and parameters defined in $\mathcal{V} \cup \mathcal{P}$.
- **INPUT-OUTPUT:**
  - $\mathcal{Q}$: Set of single lower and upper bound for each variable and parameter updated by EXPR_BOUND.
- **OUTPUT:**
  - Bound for E which is a constant, ∞, or -∞.
- **ALGORITHM:**

```
V = Set of variables appearing in E.
P = Set of parameters appearing in E.
E_orig = E
REPEAT
    E = E_orig
    Q_old = Q
    WHILE variables appear in E DO
        S1: Choose a δ-expression in E for rewriting
        S2: Rewrite δ-expression of S1 in E
        S3: Simplify E
        S4: Update Q
    ENDWHILE
    WHILE parameters appear in E DO
        S5: Choose a δ-expression in E for rewriting
        S6: Rewrite δ-expression of S5 in E
        S7: Simplify E
        S8: Update Q
    ENDWHILE
UNTIL Q ≡ Q_old
S9: Determine Bound for E
```

Figure 1. A Demand driven algorithm to compute the lower and/or upper bound of a symbolic expression defined in $\mathcal{V} \cup \mathcal{P}$.

$$\mathcal{V} = \{I, J, K, L\}, \mathcal{P} = \{N\}, LB_I = \{1, J, K\}, UB_I = \{N\},$$
$$LB_J = \{1, L-1\}, UB_J = \{I\}, LB_K = \{1\}, UB_K = \{I\},$$
$$LB_L = \{1\}, UB_L = \{J-1\}, LB_N = \{I\}, UB_N = \{\infty\},$$

$$\mathcal{B} = \bigcup_{q \in \{I,J,K,L,N\}} LB_q \cup UB_q$$

Initially all lower and upper bounds in $\mathcal{Q}$ are set to $-\infty$ and ∞, respectively.

The algorithm traverses E and replaces all δ-expressions of E using bound expressions in $\mathcal{B}$ and $\mathcal{Q}$, and rewrite-rules of Table 1 until E is a constant, $-\infty$, or ∞. During a single iteration of the REPEAT loop, the bounds in $\mathcal{Q}$ may become tighter as compared to the previous loop iteration. Tighter bounds in $\mathcal{Q}$ may induce tighter and consequently more precise bounds for E as well. Therefore, the algorithm iterates as long as $\mathcal{Q}$ changes between any pair of consecutive REPEAT loop iterations. At the end of a specific iteration, E might be a constant or an infinity symbol which cannot be changed by additional iterations of the algorithm as no variables or parameters are available for substitution. For this reason E is set to the original expression at the beginning of each iteration of the REPEAT loop.

2.2 Rewrite δ-Expressions

Rewriting $\delta(x)$ describes the process of applying a rewrite rule (see Table 1) to $\delta(x)$. For instance, a δ-expression $low(x+y)$ is rewritten as $low(x) + low(y)$ according to rule 7: $low(x+y) \rightarrow low(x) + low(y)$. This means that $low(x+y)$ is textually *replaced* (semantically equivalent) by $low(x) + low(y)$. Rewrite rules can be applied to either a single variable, parameter or a symbolic expression. If x in $\delta(x)$ is a variable or a parameter then we frequently use the term *substituting* instead of rewriting.

Continuing Example 1.1 of the previous section, where we try to prove that $IJ \geq KL$ in statement S4: X(KL,IJ)=X(IJ,KL)+VAL for all loop iterations of $L5$ which implies that $L5$ can be executed in parallel. Note that this may still imply an anti dependence (array elements are read before written). If array X is block distributed then statement $S4$ still causes communication, however, all communication can be hoisted outside of $L5$ through communication vectorization and aggregation [1].

As IJ $= \frac{I*(I-1)}{2} + J$ and KL $= \frac{K*(K-1)}{2} + L$ according to statements $S1$, $S2$ and $S3$ of Example 1.1, we have to show that $\frac{I*(I-1)}{2} + J \geq \frac{K*(K-1)}{2} + L \Longleftrightarrow \frac{I*(I-1)}{2} + J - \frac{K*(K-1)}{2} - L \geq 0$. The set of constraints $\mathcal{I}$ for this problem is given in Section 2.1. In order to prove the absence of a true dependence we have to show that low$(\frac{I*(I-1)}{2} + J - \frac{K*(K-1)}{2} - L) \geq 0$:

$$\begin{aligned} & low(\frac{I*(I-1)}{2} + J - \frac{K*(K-1)}{2} - L) \\ \rightarrow \quad & low(\frac{I*(I-1)}{2}) + low(J) - up(\frac{K*(K-1)}{2}) - up(L) \quad (1) \end{aligned}$$

Which is based on rewrite rules (7) and (8) of Table 1. Note that Table 1 displays primarily rewrite rules for low functions. Rewrite rules for up functions are similar. Rewrite rules for expressions containing infinity symbols have not been included, since they are easy to determine.

The aim of rewriting δ-expressions is to reduce all subexpressions, variables, and parameters of E until E is a constant or an infinity symbol, which is then returned as the result of the algorithm. In order to find the correct rewrite rule to be applied to a multiplication, division or exponentiation expression, the algorithm may have to determine the value ranges (signs) for the associated sub-expressions. This is done by recursively calling EXPR_BOUND.

It may actually occur that variables are induced again by replacing parameters, as the bounds of parameters (in $\mathcal{B}$) may contain variables. In this case we recursively call the algorithm for the variable bounds required.

In order to guarantee that EXPR_BOUND terminates we use the following *termination condition (TC)*: If the algorithm is trying to rewrite *low(expr)* in a recursive call of EXPR_BOUND while trying to rewrite *low(expr)* in a previous recursion, then *low(expr)* is replaced by $-\infty$. The same accounts for *up(expr)*, which is replaced by ∞. Furthermore, if there are no lower or upper bounds given for a variable or a parameter, then the associated lower and upper bound is assumed to be $-\infty$ and ∞, respectively.

Whenever EXPR_BOUND deduces a new possible tighter lower or upper bound for a variable, it is stored (statement S4 and S8) in $\mathcal{Q}$, which is implemented as a global data structure across all recursive calls to EXPR_BOUND.

Some of the rewrite rules for δ-functions may contain δ-functions applied to x^- (all negative integers of x) or x^+ (all positive integers of x). If the algorithm is unable to apply rewrite rules to such expressions because there is no information available for the value range (sign) of x^+ and x^-, then the following rules – which guarantee conservative and therefore correct bounds – are used:

$$\begin{aligned} low(x^-) &\rightarrow -\infty \\ low(x^+) &\rightarrow 1 \\ up(x^-) &\rightarrow -1 \\ up(x^+) &\rightarrow \infty \end{aligned}$$

2.3 Rewrite Policies

In statement S1 of Figure 1 it is first tried to find subexpressions of E for rewriting that contain variables. Each variable corresponds to a loop variable with a unique loop nest level. Those δ-expressions $\delta(expr)$ that contain the variable with highest loop nest level (loop variable appearing in innermost loop) have to be rewritten first; the one with the second innermost loop variable second, ..., the one with the outermost loop variable will be rewritten at the very end. If there are several δ expressions which contain a variable with highest loop nest level then choose any of these δ expressions for rewriting.

If there are no more variables in E then rewrite (statements S5 and S6) those δ-expressions in E first that contain parameters which depend (their lower or upper bounds contain other parameters) on other parameters.

Substituting variables – with their lower or upper bounds – before parameters enables us in many cases to find tighter bounds for δ-expressions. Each variable is in fact a loop variable in the underlying problem. Loop variables may directly or indirectly depend on constants or parameters (loop invariants). Overly conservative expression bounds may be computed if we reduce all variables to constants, $-\infty$, and ∞ without ever trying to simplify intermediate expressions. See also Section 2.4.

Continuing rewriting (1) of the previous section, we determine:

$$\begin{aligned} & low(\frac{I*(I-1)}{2}) + low(J) - up(\frac{K*(K-1)}{2}) - up(L) \\ \rightarrow\ & low(\frac{\text{I*(I-1)}}{2}) + low(J) - up(\frac{\text{K*(K-1)}}{2}) - up(\text{J-1}) \quad (2) \\ \rightarrow\ & low(\frac{I*(I-1)}{2}) - up(\frac{K*(K-1)}{2}) + 1 \quad (3) \\ \rightarrow\ & low(\frac{\text{I*(I-1)}}{2}) - up(\frac{I*(I-1)}{2}) + 1 \quad (4) \end{aligned}$$

First we have to rewrite the term that contains the innermost loop variable L which is $up(L)$. By replacing L with its upper bound $(J-1)$ we obtain (2). Simplifying (2) yields (3), which is explained in the next section. Thereafter, we search again for the term with the highest loop nest level. Only I and K appear in (3). K is substituted next as it has a higher loop nest level than I, which results in (4).

2.4 Simplify Expressions

In statement S3 and S7 we try to simplify E by first saving the outermost δ-function of E in δ_c, then replacing all *δ(expr)* in E by *expr* which yields E', and then simplify E' using standard symbolic expression simplification. Finally, E is replaced by $\delta_c(E')$ which is the new simplified E. Note that $\delta_c(E')$ is guaranteed (proof in [6]) to represent either the same or a tighter bound for E. E.g. let $E = \textbf{low}(\frac{up(n^2)+up(n)}{low(n)} - 1)$, then the algorithm first saves the outermost δ-function of E – which is **low** – in δ_c. Replacing all δ(*expr*) in E by *expr* yields $E' = \frac{n^2+n}{n} - 1$, which can be further simplified to $E' = n$. We then re-apply δ_c to E' which yields $low(n)$, the new simplified E. If we would have tried to reduce all variables and parameters to constants and infinity symbols, then we may get very conservative bounds. Consider an example where $E = low(n) - up(n) + 1$, $low(n) = 1$, and $up(n) = \infty$. If we simplify E according to the method described above, then we obtain $n - n + 1$ which is further reduced to 1. On the other hand, we may reduce E as follows: $low(n) - up(n) + 1 \rightarrow 1 - \infty + 1 \rightarrow -\infty$. Clearly, the first result represents a much tighter bound for E than the second result.

Continuing Example 1.1, we can simplify (4) as follows:

$$\begin{aligned} & low(\frac{I*(I-1)}{2}) - up(\frac{I*(I-1)}{2}) + 1 \\ \rightarrow\ & \frac{I*(I-1)}{2} - \frac{I*(I-1)}{2} + 1 \\ \rightarrow\ & 1 \quad (5) \end{aligned}$$

2.5 Determine Result

Finally, in statement S9 we reach a point where E is an expression consisting only of constants and infinity symbols. If E is a constant, $-\infty$ or ∞ then E is returned as the

$$
\begin{aligned}
up(x) &\rightarrow min(up(UB_1(x)),\ldots,up(UB_u(x))) && : \text{ if } x \text{ is a symbolic variable with u upper bounds} && (1)\\
up(c) &\rightarrow c && : \text{ if } c \text{ is an integer constant} && (2)\\
low(x) &\rightarrow max(low(LB_1(x)),\ldots,low(LB_l(x))) && : \text{ if } x \text{ is a symbolic variable with } l \text{ lower bounds} && (3)\\
low(c) &\rightarrow c && : \text{ if } c \text{ is an integer constant} && (4)\\
low(-x) &\rightarrow -up(x) && && (5)\\
up(-x) &\rightarrow -low(x) && && (6)\\
low(x+y) &\rightarrow low(x)+low(y) && && (7)\\
low(x-y) &\rightarrow low(x)-up(y) && && (8)
\end{aligned}
$$

$$
low(x*y) \rightarrow \begin{cases} low(x)*up(y) & : \text{ if } x \le 0 \text{ and } y > 0 \\ up(x)*low(y) & : \text{ if } x \ge 0 \text{ and } y < 0 \\ low(x)*low(y) & : \text{ if } x, y \ge 0 \\ up(x)*up(y) & : \text{ if } x, y < 0 \\ min(low(x)*up(y), up(x)*low(y), low(x)*low(y), up(x)*up(y)) & : \text{ otherwise} \end{cases} \qquad (9)
$$

$$
low(\frac{x}{y}) \rightarrow \begin{cases} \frac{up(x)}{up(y)} & : \text{ if } x \ge 0 \text{ and } y < 0 \\ \frac{up(x)}{low(y)} & : \text{ if } x, y < 0 \\ \frac{low(x)}{up(y)} & : \text{ if } x \ge 0 \text{ and } y > 0 \\ \frac{low(x)}{low(y)} & : \text{ if } x < 0 \text{ and } y > 0 \\ min(\frac{up(x^-)}{low(y)}, \frac{low(x)}{low(y^+)}, \frac{up(x)}{up(y^-)}, \frac{low(x^+)}{up(y)}) & : \text{ otherwise} \end{cases} \qquad (10)
$$

$$
low(x^c) \rightarrow \begin{cases} (low(x))^c & : \text{ if } (x \ge 0 \text{ and } c \ge 0) \text{ or} \\ & : \quad (x \le 0 \text{ and } c \ge 0 \text{ and } c \text{ odd}) \text{ or} \\ & : \quad (x < 0 \text{ and } c < 0 \text{ and } c \text{ even}) \\ (up(x))^c & : \text{ if } (x > 0 \text{ and } c < 0) \text{ or} \\ & : \quad (x \le 0 \text{ and } c \ge 0 \text{ and } c \text{ even}) \text{ or} \\ & : \quad (x < 0 \text{ and } c < 0 \text{ and } c \text{ odd}) \\ min((low(x))^c, (up(x))^c, (low(x_0^+))^c) & : \text{ otherwise} \end{cases} \qquad (11)
$$

Table 1. Rewrite rules for simplifying expressions containing low and up functions. x and y are symbolic integer-valued expressions. x^- and x^+ respectively refer to the set of negative and positive integer values (neither contains 0) of x. x_0^+ is defined by x^+ including also 0. c is an integer constant.

result. In all other cases the algorithm makes a worst-case assumption and respectively returns ∞ or $-\infty$ depending on whether the algorithm has been called for an upper or lower bound.

In continuation of Example 1.1 we determine that (5) has already been simplified to an integer constant, which is the lower bound of $low(\frac{I*(I-1)}{2} + J - \frac{K*(K-1)}{2} - L)$. This enables us to conclude the following: As $low(\frac{I*(I-1)}{2} + J - \frac{K*(K-1)}{2} - L) \ge 1$ and therefore $IJ \ge KL$, we can safely parallelize loop $L5$ of Example 1.1. Conventional dependence analysis techniques assume a true dependence for statement $S4$ in this code and consequently serialize loop $L5$ as they fail to evaluate non-linear array index functions.

3 Simplification of Systems of Constraints

The complexity of many compiler analyses as well as the associated results which are based on a set of constraints $\mathcal{I}$ can be considerably reduced, if tautologies, contradictions, equalities and redundant constraints can be detected and eliminated. Simplifying a set of constraints has been shown [7, 8] to be critical in order to alleviate performance as well as compiler analysis. We have developed and implemented the following simplification techniques:

- Tautology inequalities (e.g. $2 > 0$, $n \le n$, and $n \le n*n$) in $\mathcal{I}$ are detected and removed by using standard symbolic expression simplification and algorithm EXPR_BOUND.
- Search for two inequalities $I_1, I_2 \in \mathcal{I}$ which induce an equality. E.g. let I_1 be $2*i*j-n \le 0$ and I_2 is $2*i*j-$

$n \geq 0$, which implies $I_3 : 2*i*j-n=0$. If so, then we search for a variable v with a coefficient as simple as possible (e.g. constant) in the resulting equality, solve it for v and substitute in $\mathcal{I}$ for all occurrences of v. Once we have successfully substituted an equality I_3 – deduced from two inequalities I_1 and I_2 – we can delete I_1, I_2, and I_3 in $\mathcal{I}$.

- Examine whether two inequalities of $\mathcal{I}$ contradict each other (e.g. $2*i*j+m>0$ and $2*i*j+m+3 \leq 0$), which means that $\mathcal{I}$ has no solution. If contradictions are detected many compiler analyses can be immediately terminated.

- Try to detect and eliminate redundant inequalities in $\mathcal{I}$. This is in particular important for inequalities involving variables, as the number of inequalities in $\mathcal{I}$ is in many cases directly proportional to the complexity of finding a solution to $\mathcal{I}$.

  The following theorem states a condition which allows us to eliminate redundant inequalities in a set of constraints.

  THEOREM: Let E_1, E_2 be two symbolic expressions defined in $\mathcal{V} \cup \mathcal{P}$, and $q \in \mathcal{V} \cup \mathcal{P}$. I_1, I_2 are two inequalities of $\mathcal{I}$ such that I_1 is defined by E_1 REL q and I_2 by E_2 REL q. REL is an inequality relationship in $\{\leq, <\}$. Then I_2 can be eliminated from $\mathcal{I}$, if at least one of the following two conditions is satisfied.

  1. $low(E_1 - E_2) \geq 0$ for all values of variables and parameters appearing in E_1 and E_2
  2. $up(E_2 - E_1) \leq 0$ for all values of variables and parameters appearing in E_1 and E_2.

  This theorem has been proved in [6]. The theorem for inequality relationships $>$ and $\geq$ is similar. Algorithm EXPR_BOUND can therefore also be used to detect and eliminate redundant inequalities.

4 Related Work

Blume [2] developed an algorithm to compare two expressions based on symbolic ranges. A range $[a:b]$ for a variable x defines a single lower and upper bound on x, which can also be written as $a \leq x \leq y$. The main difference between our algorithm and the one of Blume is that Blume's techniques are based on manipulation of symbolic expressions containing variable ranges, whereas our method applies δ-functions to symbolic expressions. Although Blume's algorithm is quite general, we encountered sufficient problems to justify the development of our own method for comparing symbolic expressions:

- Blume assumes for the range $[a:b]$ of a variable v that $a \leq b$. This restriction makes it difficult for some important compiler analyses to apply his algorithm. For instance, in order to compute loop iteration counts [5, 7] or detect zero-trip-loops [8], we must allow loop variables whose lower bounds can be larger than their upper bounds for some loop iterations. Our algorithm EXPR_BOUND has been designed to handle such cases.

- Blume does not discuss how to handle multiple lower and upper bounds for a variable v, which is a critical issue for symbolic sum algorithms [7] or dead code elimination. Again, our approach models such cases.

- Our rewrite rules for simplifying expressions are less conservative than those of Blume, which means that our approach may obtain tighter bounds for expressions. For instance, Blume's rewrite rules [2] for multiplication of a variable range [a:b] with a constant c – if the sign for c is unknown – is $[-\infty{:}\infty]$. Our rewrite rules for multiplication operations (Table 1) do not contain any $-\infty$ or ∞. If we apply Blume's rewrite rule to the expression $x*y$ where $0 \leq x \leq y$ and the sign of y is unknown, then $x*y \rightarrow [0:y]*y \rightarrow [-\infty:\infty]$. Applying rewrite rule (9) of Table 1 to $low(x*y)$ yields the following substitution: $low(x*y) \rightarrow min(low(x)*up(y), up(x)*low(y), low(x)*low(y), up(x)*up(y)) \rightarrow min(low(x)*up(y), up(y)*low(x), low(x)*low(x), up(y)*up(y))$by substituting $up(y)$ for up(x), and $low(x)$ for $low(y)$. In our algorithm according to Section 2.2, we assume $up(y)$ is equal to ∞ as there is no upper bound information available for y. Consequently the previous expression can be rewritten as $min(0*\infty, \infty*0, 0*0, \infty*\infty) \rightarrow min(0,0,0,\infty) \rightarrow 0$. Therefore, our method deduces the tightest lower bound possible based on the constraints given, whereas Blume's approach derives a worst case answer $-\infty \leq x*y$.

- Our approach for obtaining a replacement order for variables in expressions is similar effective as Blume's technique – which requires a dependence graph of variable bounds and topological sorting of strongly-connected components in this graph – but much cheaper in terms of complexity.

Other related work has been done by M. Haghighat and C. Polychronopoulos [8]. They describe techniques to prove that a symbolic expression is strictly increasing or decreasing, which enables them to analyze codes for data dependences. P. Tu and D. Padua [9] present an algorithm to determine the relationship between two symbolic expressions which is based on gated static single assignment form.

Rec	Step	E	$\mathcal{Q}$	Comments
1	1	$low(n - \frac{n}{i_2})$		
	2	$\rightarrow low(n) - up(\frac{n}{i_2})$		rule (8)
2	1	$low(i_2)$		
	2	$\rightarrow low(\frac{n}{2*i_1})$		rule (3)
3	1	$low(i_1)$		
	2	$\rightarrow max(low(\frac{n}{2*i_2}), 1)$	$(i_1; 1, \infty)$	rule (3)
			$(i_2; -\infty, \infty)$	TC for $low(i_2)$
4	1	$up(i_2)$		
	2	$\rightarrow min(up(n), up(\frac{n}{i_1}))$		rule (3)
5	1	$low(n)$		
	2	$\rightarrow max(low(i_1), low(i_2), low(i_1 * i_2))$		rule (3)
	3	$\rightarrow max(1, low(i_2), min(low(i_1) * up(i_2),$	$(n; 1, \infty)$	
		$up(i_1) * low(i_2), low(i_1) * low(i_2), up(i_1) * up(i_2))$		rule (9)
6	1	$up(i_1)$		
	2	$\rightarrow min(up(n), up(\frac{n}{i_2}))$		rule (1)
	3	$\rightarrow min(up(n), \infty)$		TC for $up(\frac{n}{i_2})$
	4	$\rightarrow up(n)$	$(i_1; 1, n)$	
5	4	$\rightarrow max(1, -\infty, min(1 * \infty, up(n) * -\infty, 1 * -\infty, up(n) * \infty))$		TC for $up(n)$
	5	$\rightarrow max(1, -\infty, min(\infty, -\infty, -\infty, \infty))$		simplify
	6	$\rightarrow max(1, -\infty, -\infty)$		simplify
	7	$\rightarrow 1$		simplify
4	3	$\rightarrow min(up(n), \frac{up(n)}{low(i_1)})$		rule (10)
	4	$\rightarrow min(up(n), \frac{up(n)}{1})$		substitute $low(i_1)$
	5	$\rightarrow up(n)$	$(i_2; -\infty, n)$	
3	3	$\rightarrow max(min(\frac{up(n^-)}{low(2*i_2)}, \frac{low(n)}{low(2*i_2^+)}, \frac{up(n)}{up(2*i_2^-)}, \frac{low(n)}{up(2*i_2)}), 1)$		rule (10)
	4	$\rightarrow max(min(\frac{low(n)}{2}, \frac{up(n)}{2*(-1)}, \frac{low(n)}{2*up(n)}), 1)$		substitute
	5	$\rightarrow max(-\frac{up(n)}{2}, 1)$		simplify
	6	$\rightarrow 1$		simplify
2	3	$\rightarrow \frac{low(n)}{2*up(i_1)}$		rule (10)
	4	$\rightarrow \frac{low(n)}{2*up(n)}$		substitute $up(i_1)$
	5	$\rightarrow \frac{1}{2}$		simplify
	6	$\rightarrow 1$	$(i_2; 1, n)$	i_2 integer
1	3	$\rightarrow low(n) - \frac{up(n)}{low(i_2)}$		rule (10)
	4	$\rightarrow low(n) - \frac{up(n)}{1}$		substitute $low(i_2)$
	5	$\rightarrow low(n) - up(n)$		simplify
	6	$\rightarrow 0$		proved $n \geq \frac{n}{i_2}$

Table 2. Using EXPR_BOUND to prove that $n \geq \frac{n}{i_2}$.

5 Experiments

In this section we continue Example 1.2 of Section 1 which demonstrates how algorithm EXPR_BOUND can be used to eliminate a redundant constraint on a variable and as a consequence simplifies other compiler analyses significantly. Consider the following set of inequalities as obtained from the code of Example 1.2.

$$\mathcal{I} = \{1 \leq i_1 \leq n,\ \frac{n}{2*i_1} \leq i_2 \leq n,\ i_1 * i_2 \leq n\}$$

This yields the following sets of variables, parameters, lower and upper bounds:
$\mathcal{V} = \{i_1, i_2\}, \mathcal{P} = \{n\}, LB_{i_1} = \{1, \frac{n}{2*i_2}\}, UB_{i_1} = \{n, \frac{n}{i_2}\}$,
$LB_{i_2} = \{\frac{n}{2*i_1}\}, UB_{i_2} = \{n, \frac{n}{i_1}\}, LB_n = \{i_1, i_2, i_1 * i_2\}$,
$UB_n = \{2*i_1*i_2\}, \mathcal{B} = LB_n \cup UB_n \cup \bigcup_{1 \leq j \leq 2} LB_{i_j} \cup \bigcup_{1 \leq j \leq 2} UB_{i_j}$

where LB_r and UB_r are respectively the sets of lower and upper bounds for a variable or parameter r which are derived from $\mathcal{I}$. An interesting question arises whether one of the two upper bounds for i_1 in $\mathcal{I}$ is redundant. If we can show that $low(n - \frac{n}{i_2}) \geq 0$ for all possible values of variables and parameters in $\mathcal{V} \cup \mathcal{P}$ according to the theorem on page 6, then $i_2 \leq n$ is redundant in $\mathcal{I}$. For this purpose we invoke EXPR_BOUND(E=low(n-$\frac{n}{i_2}$),B,Q={i_1;$-\infty$,∞),(i_2,$-\infty$,∞),(n,$-\infty$,∞)}). Table 2 shows the most interesting steps for each recursion of EXPR_BOUND. Note that there are no rule numbers for up functions as they are not shown in Table 1.

In what follows we will describe the most important recursions of EXPR_BOUND as shown in Table 2 in more detail. *Rec* specifies the current recursion of the algorithm, and *Step* the processing step within the recursion. E is the current symbolic expression as processed by EXPR_BOUND. *TC* refers to the termination condition of EXPR_BOUND. The current contents of $\mathcal{Q}$ at a specific table entry is the union of the most recent table entries for each different variable and parameter. E.g. $\mathcal{Q} = \{(i_1; 1, n), (i_2; 1, n), (n; 1, \infty)\}$ at recursion 2 step 6.

An item $i.j$ of the following itemized list corresponds to recursion i, step j of algorithm EXPR_BOUND. We will specify the corresponding statements of the algorithm according to Figure 1 where appropriate.

1.2 In order to identify the rewrite rule to apply to $up(\frac{n}{i_2})$, the value range for i_2 needs to be known. Therefore, the algorithm is recursively (recursion 2) called for $E = low(i_2)$ to find out whether i_2 can ever be negative.

3.2 Allows to immediately deduce $low(i_1) \geq 1$, which is added into $\mathcal{Q}$. Then the algorithm is evaluating $low(\frac{n}{2*i_2})$ to investigate whether this expression is larger than 1. In order to determine the correct rewrite rule to apply to $low(\frac{n}{2*i_2})$, we need to determine the value range for i_2. As recursion 2 already tried to deduce $low(i_2)$, we add $-\infty \leq i_2$ into $\mathcal{Q}$ based on the termination condition (see Section 2.2) of EXPR_BOUND. Then the algorithm is recursively called for $E = up(i_2)$ to determine whether i_2 can ever be positive.

5.3 As the maximum function for $low(n)$ contains 1, we can immediately add $1 \leq n$ into $\mathcal{Q}$. In order to further substitute the minimum function we need to know the sign for $up(i_1)$.

6.3 As in recursion 1 step 2 the algorithm already tried to evaluate $up(\frac{n}{i_2})$, we substitute $up(\frac{n}{i_2})$ by ∞ according to the termination condition of EXPR_BOUND. After step 4 of this recursion the algorithm returns to recursion 5.

2.5 The expression of recursion 2 step 4 has been simplified according to statement S7, which yields $\frac{1}{2}$. This means that $low(i_2) \geq \frac{1}{2}$, and consequently, $low(i_2) \geq 1$ as $low(i_2)$ must be an integer.

1.5 Simplifies $low(n) - up(n)$ according to statement S7, which yields $n - n$ and finally 0.

1.6 Proves that $n \geq \frac{n}{i_2}$ for all values of i_1, i_2, and n. Therefore, the inequality $i_2 \leq n$ is redundant in $\mathcal{I}$ according to the theorem of page 6.

Other important compiler analyses such as detecting zero-trip-loops, dead code elimination, and performance prediction commonly examine whether $\mathcal{I}$ has a solution at all. If $\mathcal{I}$ contains program unknowns (parameters) then these analyses may yield guarded solutions. In [7] we have described a symbolic sum algorithm that computes the number of solutions of a set of linear and non-linear constraints which are defined over variables and parameters. For instance, by detecting and eliminating the redundant inequality $i_2 \leq n$ in $\mathcal{I}$, the number of guarded solutions to $\mathcal{I}$ as determined by our symbolic sum algorithm is reduced by 50 %.

Please note that Table 2 shows an overly conservative application of our algorithm for the sake of illustrating the most interesting aspects of EXPR_BOUND. The actual implementation of EXPR_BOUND detects at the beginning of the first recursion that $i_1, i_2, n \geq 1$ by simply checking the lower bound expressions of $\mathcal{B}$ and initializes $\mathcal{Q}$ with this information. Based on $\mathcal{Q}$ the following rewrite rules are applied: $low(n - \frac{n}{i_2}) \rightarrow low(n) - up(\frac{n}{i_2}) \rightarrow low(n) - \frac{up(n)}{low(i_2)} \rightarrow low(n) - \frac{up(n)}{1} \rightarrow low(n) - up(n) \rightarrow 0$. Therefore, our implementation of EXPR_BOUND can prove in 5 steps that $n \geq \frac{n}{i_2}$ under the constraints of $\mathcal{I}$.

6 Conclusions

Numerous researchers have shown the importance of symbolic compiler analysis [8, 2, 7, 9] for optimizing and predicting performance of parallel programs. A crucial problem of many symbolic compiler analyses is to determine the relationship between symbolic expressions.

We have described a demand driven algorithm that computes the upper and/or lower bound of linear and non-linear symbolic expressions. This algorithm is used to examine non-linear array index functions for data dependences, compare symbolic expressions, and simplify systems of constraints. Previous methods are based on algorithms that are either restricted to a smaller class of symbolic expressions, are more expensive in terms of complexity, or produce less accurate results (tighter bounds).

We have demonstrated the effectiveness of our techniques by proofing the absence of a data dependence in a loop with non-linear array references that enables the compiler to parallelize this loop. Conventional dependence analysis would have serialized this loop. We also showed how to use our algorithm to detect redundant inequalities in a set of non-linear constraints, which as a result significantly reduces the complexity of other important compiler and performance analysis.

References

[1] S. Benkner, S. Andel, R. Blasko, P. Brezany, A. Celic, B. Chapman, M. Egg, T. Fahringer, J. Hulman, Y. Hou, E. Kelc, E. Mehofer, H. Moritsch, M. Paul, K. Sanjari, V. Sipkova, B. Velkov, B. Wender, and H. Zima. *Vienna Fortran Compilation System - Version 2.0 - User's Guide*, October 1995.

[2] W. Blume. *Symbolic Analysis Techniques for Effective Automatic Parallelization*. PhD thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, June 1995.

[3] M. Berry. et al. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *International*

Journal of Supercomputing Applications, pages 3(3): 5–40, 1989.

[4] T. Fahringer. Estimating and Optimizing Performance for Parallel Programs. *IEEE Computer*, 28(11):47 – 56, November 1995.

[5] T. Fahringer. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Publishers, Boston, USA, ISBN 0-7923-9708-8, March 1996.

[6] T. Fahringer. Symbolic Expression Evaluation. Technical report, Institute for Software Technology and Parallel Systems, Univ. of Vienna, Oct 1996.

[7] T. Fahringer. Toward Symbolic Performance Prediction of Parallel Programs. In *IEEE Proc. of the 1996 International Parallel Processing Symposium*, pages 474–478, Honolulu, Hawaii, April 1996.

[8] M. Haghighat and C. Polychronopoulos. Symbolic Analysis for Parallelizing Compilers. CSRD Report No. 1355, CSRD, University of Illinois at Urbana-Champaign, IL, 1994.

[9] P. Tu and D. Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. In *9th ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.