Efficient Symbolic Analysis for Parallelizing Compilers and Performance Estimators

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

Accepted for publication in the Journal of Supercomputing, Kluwer Academic Publishers

Abstract

Symbolic analysis is of paramount importance for parallelizing compilers and performance estimators to examine symbolic expressions with program unknowns such as number of processors and array dimension sizes and to solve queries based on systems of constraints (equalities and inequalities). This paper describes techniques for counting the number of solutions to a system of constraints, simplifying systems of constraints, computing lower and upper bounds of symbolic expressions, and determining the relationship between symbolic expressions. All techniques target wide classes of linear and non-linear symbolic expressions and systems of constraints. Our techniques have been implemented and are used as part of a parallelizing compiler and a performance estimator for data dependence analysis, detecting zero-trip-loops, dead code elimination, communication optimization, and estimation of work distribution for parallel programs. Various examples and experiments demonstrate the effectiveness of our symbolic analysis techniques.

1 Introduction

Many parallelizing compilers and performance tools fail to effectively parallelize and evaluate programs caused by the deficit to analyze linear and non-linear symbolic expressions (expressions with program unknowns). Numerous researchers [3, 20, 11, 4, 22, 14, 15, 6] have reported on the occurrence of symbolic expressions in practical codes and the need of effective techniques to analyze such programs. Non-linear symbolic expressions are commonly caused by induction variable substitution, linearizing arrays, parameterizing parallel programs with symbolic number of processors and problem sizes, tiling with symbolic block sizes, recurrences, and so forth. Symbolic expressions can seriously hamper crucial compiler and performance analysis including testing for data dependences, optimizing communication, privatizing arrays, simplifying code, dead code elimination, detecting zero-trip-loops, performance prediction, etc. The inability to effectively handle symbolic expressions and constraints commonly results in worst case assumptions for many anlayses, moves analysis from compile time into runtime and consequently causes considerable performance losses.

Consider the loop nest with linearized and indirect array referencing in Example 1.1 which is very similar to a code excerpt as found in the TRFD code of the Perfect Benchmarks [7]. In loop $L1$ array IA is initialized with a non-linear array subscript expression $(J_1 * (J_1 - 1))/2$, which is then used in $S4$ to read $X(IJ, KL)$ and write $X(KL, IJ)$ with $IJ = (J_1 * (J_1 - 1))/2 + J_2$ and $KL = (J_3 * (J_3 - 1))/2 + J_4$, 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 subscript expressions and therefore assume a data dependence that sequentializes loop $L5$. This paper describes an algorithm that can compare non-linear array subscript expressions such as KL and IJ, and consequently determines the absence of a true dependence, which enables the underlying compiler to parallelize loop $L5$.

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).

Example 1.1

```
L1:  DO J1=1,N
S1:      IA(J1)=(J1*(J1-1))/2
     ENDDO
     ...
L2:  DO J1=1,N
L3:      DO J2=1,J1
S2:          IJ=IA(J1)+J2
             ...
L4:          DO J3=1,J1
                 ...
L5:              DO J4=1,J2-1
S3:                  KL=IA(J3)+J4
                     ...
S4:                  X(KL,IJ)=X(IJ,KL)+VAL
                 ENDDO
             ENDDO
         ENDDO
     ENDDO
```

Example 1.2

```
   DO J1=1,N
     DO J2=N/(2*J1),N
S1:      IF (J1*J2 ≤ N) THEN
           A(J1, J2) = ...
         ENDIF
         ...
     ENDDO
   ENDDO
```

Detecting zero-trip-loops [14] is a similar problem which tries to determine whether the loop body of a given loop nest is ever executed. Counting the number of loop iterations has been shown [11, 12, 14, 21] to be crucial for many performance analyses such as modeling work distribution [10], data locality [11] and communication overhead [9]. All of these problems can be formulated as queries based on 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, conditional statements, data and work distribution specifications [16, 2], etc. For instance, $\mathcal{I}$ is given by $\{1 \leq J_1 \leq N,\ N/(2 * J_1) \leq J_2 \leq N,\ J_1 * J_2 \leq N\}$ for Example 1.2. The number of solutions of $\mathcal{I}$ can then be used to answer all problems mentioned above.

Furthermore, many analyses based on systems of constraints $\mathcal{I}$ are significantly alleviated if $\mathcal{I}$ is simplified by eliminating redundant constraints, substituting equations, detecting contradictions, etc.

This paper describes a variety of techniques that enable effective examining of linear and non-linear symbolic expressions and counting the number of solutions to a system of constraints. First, we describe an algorithm for computing lower and upper bounds of wide classes of linear and non-linear symbolic expressions based on a set of constraints. Among others this algorithm supports comparing symbolic expressions for equality and inequality $(=, <, >)$ relationships, simplifying systems of constraints, examining non-linear array subscript expressions for data dependences, and optimizing communication.

Second, we present a symbolic sum algorithm that estimates the number of integer solutions to a system of constraints. This algorithm is used to support detection of zero-trip loops, elimination of dead code, and performance prediction of parallel programs.

We have implemented all described techniques and are currently integrating them with a parallelizing compiler and a performance estimator for data parallel programs on distributed memory parallel architectures. Experiments will be shown that demonstrate the usefullness of our approach.

This paper is organized as follows: Basic definitions and notations are given in Section 2. In Section 3 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 subscript expressions. Our algorithm for counting the number of solutions to a system of linear and non-linear constraints is described in Section 4. Section 5 shows how to simplify a set of symbolic constraints. Related work is discussed in Section 6. In the following section we present experiments that demonstrate the effectiveness of our symbolic analysis. Conclusions and final remarks are given in Section 8.

2 Preliminaries

The following notations and definitions are used in the remainder of this paper:

- Our symbolic analysis has been implemented and is currently being integrated with *VFCS* [2], a High-Performance Fortran-style parallelizing compiler and P^3T [8, 12], and with a performance estimator for

data parallel programs on distributed memory parallel architectures. The *VFCS* parallelization strategy is based on data decomposition in conjunction with the single-program, multiple-data programming model. With this method, each array is partitioned and mapped to the target archtitecture's processors. The specification of the array-element mapping to the processors is called the program's data distribution. A processor is then thought of as owning the data assigned to it; these data elements are stored in its local memory. The work contained in the program is distributed according to a data distribution, and a processor performs computations that define the data elements that the processor owns. Both data and work distribution are commonly specified through directives, such as HPF directives [16]. If a processor requires data that is owned by a remote processor, the non-local data is accessed through interprocessor communication which is automatically optimized and inserted by *VFCS*.

- Let $\mathcal{I}$ be a set of non-linear and linear constraints (equalities and inequalities) defined over loop *variables* and *parameters* (loop invariants) which are commonly derived from loop bounds, array subscript expressions, conditional statements and data and work distribution specifications 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 ($\infty, -\infty$). Symbolic expressions therefore describe both linear as well as non-linear expressions. Note that our algorithm for counting solutions to a system of constraints (see Section 4) is restricted to a sub-class of these expressions.

In order to provide a useful framework for manipulating symbolic expressions and constraints, we implemented standard techniques [19] which include multiplying out products and powers, reduce products of factors, put all terms over a common denominator, separate into terms with simple denominators, cancel common factors between numerators and denominators, etc.

3 Evaluate Symbolic Expressions

In this section we present our techniques to evaluate symbolic expressions for equality and inequaltiy relationships which are based on the following theorem:

Theorem 3.1 Compare Expressions

Let E_1, E_2 be two symbolic expressions defined in $\mathcal{V} \cup \mathcal{P}$ then $E_1 \geq E_2$ 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.*

In order to compare symbolic expressions we developed an algorithm that computes the lower and upper bound of symbolic expressions based on a system of constraints.

Inherently, computing bounds of a symbolic expression is in general undecidable and depends mostly on the set of constraints given. Our objective is therefore to cover a wide class of both linear and non-linear symbolic expressions for which we selectively compute lower and/or 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 $\mathcal{V} \cup \mathcal{P}$.

3.1 Compute Lower and Upper Bounds of Symbolic Expressions

Figure 1 shows algorithm EXPR_BOUND which computes 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 for each variable and parameter. These expressions are derived from $\mathcal{I}$, the set of constraints of the underlying problem.

Assume that the cardinality of $\mathcal{V}$ and $\mathcal{P}$ is respectively n and m, then $\mathcal{Q}$ is given by a set of triples as follows:

$$\mathcal{Q}=\{(v_1,v_1^l,v_1^u),...,(v_n,v_n^l,v_n^u),\ (p_1,p_1^l,p_1^u),...,(p_m,v_m^l,v_m^u)\}.$$

The triple (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. The triple (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 $\mathcal{Q}$. For instance, 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. We similarly determine 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 as follows:

$$\mathcal{I} \quad = \quad \{1 \leq J_1 \leq N, 1 \leq J_2 \leq J_1, 1 \leq J_3 \leq J_1, 1 \leq J_4 \leq J_2 - 1\} \tag{1}$$

This yields the following sets of variables, parameters, lower and upper bounds:

$\mathcal{V} = \{J_1, J_2, J_3, J_4\}, \mathcal{P} = \{N\}, LB_{J_1} = \{1, J_2, J_3\}, UB_{J_1} = \{N\}, LB_{J_2} = \{1, J_4 + 1\}, UB_{J_2} = \{J_1\}, LB_{J_3} = \{1\}, UB_{J_3} = \{J_1\}, LB_{J_4} = \{1\}, UB_{J_4} = \{J_2 - 1\}, LB_N = \{J_1\}, UB_N = \{\infty\}, \mathcal{B} = LB_N \cup UB_N \cup \bigcup_{1 \le i \le 4} (LB_{J_i} \cup UB_{J_i})$

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.

3.2 Rewrite δ-Expressions

Rewriting $\delta(E)$ – where E is a symbolic expression – describes the process of applying a rewrite rule (see Table 1) to $\delta(E)$. For instance, a δ-expression $low(E_1 + E_2)$ is rewritten as $low(E_1) + low(E_2)$ according to rule 7: $low(E_1 + E_2) \rightarrow low(E_1) + low(E_2)$. E_1 and E_2 are symbolic expressions. This means that $low(E_1 + E_2)$ is textually *replaced* (semantically equivalent) by $low(E_1) + low(E_2)$. If E in $\delta(E)$ 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 [2].

As $IJ = \frac{J_1*(J_1-1)}{2} + J_2$ and $KL = \frac{J_3*(J_3-1)}{2} + J_4$ according to statements $S1$, $S2$ and $S3$ of Example 1.1, we have to show that $\frac{J_1*(J_1-1)}{2} + J_2 \geq \frac{J_3*(J_3-1)}{2} + J_4 \Longleftrightarrow \frac{J_1*(J_1-1)}{2} + J_2 - \frac{J_3*(J_3-1)}{2} - J_4 \geq 0$. The set of constraints $\mathcal{I}$ for this problem is given by (1). In order to prove the absence of a true dependence we have to show that $low(\frac{J_1*(J_1-1)}{2} + J_2 - \frac{J_3*(J_3-1)}{2} - J_4) \geq 0$:

$$\begin{aligned} & low(\frac{J_1 * (J_1 - 1)}{2} + J_2 - \frac{J_3 * (J_3 - 1)}{2} - J_4) \\ \rightarrow \quad & low(\frac{J_1 * (J_1 - 1)}{2}) + low(J_2) - up(\frac{J_3 * (J_3 - 1)}{2}) - up(J_4) \end{aligned} \tag{2}$$

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 sub-expressions, 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) of 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 holds 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 Figure 1) 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:

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

3.3 Rewrite Policies

In statement S1 of Figure 1 it is first tried to find sub-expressions of E containing variables for rewriting. Each variable corresponds to a loop variable with a unique loop nest level. Those δ-expressions 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 3.4.

Continuing rewriting (2) of the previous section, we determine:

$$\begin{array}{rcll} & & low(\frac{J_1 * (J_1 - 1)}{2}) + low(J_2) - up(\frac{J_3 * (J_3 - 1)}{2}) - up(J_4) & \\ & \rightarrow & low(\frac{J_1 * (J_1 - 1)}{2}) + low(J_2) - up(\frac{J_3 * (J_3 - 1)}{2}) - up(J_2 - 1) & (3) \\ & \rightarrow & low(\frac{J_1 * (J_1 - 1)}{2}) - up(\frac{J_3 * (J_3 - 1)}{2}) + 1 & (4) \\ & \rightarrow & low(\frac{J_1 * (J_1 - 1)}{2}) - up(\frac{J_1 * (J_1 - 1)}{2}) + 1 & (5) \end{array}$$

Firstly, we have to rewrite the term that contains the innermost loop variable J_4 which is $up(J_4)$. By replacing J_4 with its upper bound $(J_2 - 1)$ we obtain (3). Simplifying (3) yields (4) which is explained in the next section. Thereafter, we search again for the term with the highest loop nest level. Only J_1 and J_3 appear in (4). J_3 is substituted next as it has a higher loop nest level than J_1 which implies (5).

3.4 Simplify Expressions

In statement S3 and S7 of Figure 1 we try to simplify E by first saving the outermost δ-function of E in δ', then replacing all $\delta(y)$ in E by $y - y$ is a symbolic expression – which yields E'. Thereafter, we simplify E' using standard symbolic expression simplification. Finally, E is replaced by $\delta'(E')$ which is the new simplified E. Note that $\delta'(E')$ is guaranteed to represent either the same or a tighter bound for E. For instance, let $E = \mathbf{low}(\frac{up(x^2)+up(x)}{low(x)} - 1)$ where x is a symbolic expression, then the algorithm first saves the outermost δ-function of E – which is **low** – in δ'. Replacing all $\delta(y)$ in E by y yields $E' = \frac{x^2+x}{x} - 1$, which can be further simplified to $E' = x$. We then re-apply δ' to E' which yields $low(x)$, 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 $E = low(x) - up(x) + 1$, $low(x) = 1$, and $up(x) = \infty$. If we simplify E according to the method described above, then we obtain $x - x + 1$ which is further reduced to 1. On the other hand, we may reduce E as follows: $low(x) - up(x) + 1 \rightarrow 1 - \infty + 1 \rightarrow -\infty$. Clearly, the first result represents a much tighter bound for E than the second result.

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

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

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

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

Table 1: Rules for rewriting symbolic 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.

Continuing Example 1.1, we can simplify (5) as follows:

$$
\begin{array}{rcll}
& low(\frac{J_1*(J_1-1)}{2}) - up(\frac{J_1*(J_1-1)}{2}) + 1 & & \\
\rightarrow & \frac{J_1*(J_1-1)}{2} - \frac{J_1*(J_1-1)}{2} + 1 & & \\
\rightarrow & 1 & & (6)
\end{array}
$$

3.5 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 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 (6) has already been simplified to an integer constant, which is the lower bound of the original expression $low(\frac{J_1*(J_1-1)}{2} + J_2 - \frac{J_3*(J_3-1)}{2} - J_4)$. This enables us to conclude the following: As $low(\frac{J_1*(J_1-1)}{2} + J_2 - \frac{J_3*(J_3-1)}{2} - J_4) \geq 1$ and therefore $IJ \geq KL$, we can safely parallelize loop $L5$ of Example 1.1. Most existing 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 subscript expressions.

4 Count Solutions to a System of Constraints

Counting the number of integer solutions to a set of constraints has been shown to be a key issue in performance analysis of parallel programs. Numerous applications [8, 21, 14] include: estimating statement execution counts, branching probabilities, work distribution, number of data transfers and cache misses. Even compiler analysis can be supported, for instance, by detecting and eliminating dead code such as loops that never iterate (zero-trip loops).

In what follows, we present a non-linear loop nest for which we compute the iteration count in order to motivate our research.

```
DO J1=1,N1
    DO J2=1,N2*J1
        if (J1 ≤ N2) statement S
    ENDDO
ENDDO
```

Computing how many times statement S is executed in the loop nest is equivalent to counting the number of integer solutions of $\mathcal{I} = \{1 \leq J_1 \leq N_1,\ 1 \leq J_2 \leq N_2 * J_1,\ J_1 \leq N_2\}$. J_1 and J_2 are (loop) *variables* and N_1, N_2 are *parameters* (loop invariants).

Note that we consider $J_2 \leq N_2 * J_1$ to be non-linear, although N_2 is loop invariant. The statement execution count for S is given by:

$$\sum_{J_1=1}^{min(N_1,N_2)} \sum_{J_2=1}^{N_2*J_1} 1 = \begin{cases} \frac{N_1^2*N_2}{2} + \frac{N_1*N_2}{2} & : \text{ if } 1 \leq N_1 \leq N_2 \\ \frac{N_2^3}{2} + \frac{N_2^2}{2} & : \text{ if } 1 \leq N_2 < N_1 \end{cases}$$

In general, every loop implies at least two constraints on its loop variable, one for its upper and one for its lower bound. Additional constraints on both parameters and variables can be implied, for instance, by conditional statements, min and max functions, etc.

Algorithms [8] that count the number of integer solutions of arbitrary large sets of linear constraints over loop variables have already been implemented and successfully used in the context of performance analysis and parallelizing compilers. The drawback with these algorithms, however, is that program unknowns (*parameters*) must be replaced by constant integers. In order to overcome this disadvantage M. Haghighat and C. Polychronopoulos [14] described an algebra of conditional values and a set of rules for transforming symbolic expressions. However, they did not present an algorithm that decides which rule to apply when. N. Tawbi [21] developed a symbolic sum algorithm which handles loop nests with parameters. Her approach is restricted to sums based on linear inequalities implied by loop header statements. W. Pugh [18] improved Tawbi's algorithm by extending it to techniques that count the number of integer solutions to an arbitrary set of linear constraints.

So far non-linear constraints have not been considered for counting integer solutions. In this section we describe a symbolic sum algorithm that goes beyond previous work by estimating the number of integer solutions to a class of non-linear constraints.

4.1 Symbolic Sum Computation

In this section we describe a symbolic algorithm which computes the number of integer solutions of a set of linear and non-linear constraints $\mathcal{I}$ defined over $\mathcal{V} \cup \mathcal{P}$. $I \in \mathcal{I}$ is restricted to be of the following form:

$$p_1(\vec{P}) * v_1 + \ldots + p_k(\vec{P}) * v_k \quad REL \quad 0 \tag{7}$$

where REL $\in \{\leq, \geq, <, >, =, \neq\}$ represents an equality or inequality relationship. $\vec{P}$ is a vector over parameters of $\mathcal{P}$. $p_i(\vec{P})$ are linear or non-linear expressions over $\mathcal{P}$, whose operations can be addition, subtraction, multiplication, division, floor, ceiling, and exponentiation. Min and max functions are substituted where possible by constraints free of min and max functions which is shown in Section 4.4. Not all variables in $\{v_1, \ldots, v_k\}$ are necessarily pairwise different where $v_i \in \mathcal{V}$ $(1 \leq i \leq k)$. Then

$$\mathcal{W} = \{\vec{V} | \vec{V} \text{ satisfies } \mathcal{I}\} \tag{8}$$

is the set of integer solutions over all variables in $\mathcal{V}$ which satisfy the conjunction of constraints in $\mathcal{I}$. Every $\vec{V}$ is a vector over variables in $\mathcal{V}$, which represents a specific solution to $\mathcal{I}$. Figure 2 shows the algorithm for counting the number of elements in $\mathcal{W}$, given $\mathcal{I}$, $\mathcal{P}$, $\mathcal{V}$, E, and $\mathcal{R}$. E is an intermediate result (symbolic expression) for a specific solution E_i. The result $\mathcal{R}$ is a set of tuples (C_i, E_i) where $1 \leq i \leq k$. Each tuple corresponds to a conditional solution of the sum algorithm. Note that the conditions C (satisfying (7)) among all solution tuples are not necessarily disjoint. The result has to be interpreted as the sum over all E_i under the condition of C_i as follows:

$$\sum_{1 \leq i \leq k} \gamma(C_i) * E_i \tag{9}$$

where γ is defined as

$$\gamma(C) = \begin{cases} 1 & : \text{ if } C = \text{TRUE} \\ 0 & : \text{ otherwise} \end{cases} \tag{10}$$

E and $\mathcal{R}$ must be respectively set to 1 and ϕ (empty set) at the initial call of the algorithm.

4.2 The Algorithm

In each recursion the algorithm is eliminating one variable $v \in \mathcal{V}$. First, all lower and upper bounds of v in $\mathcal{I}$ are determined. Then the maximum lower and minimum upper bound of v is searched by splitting $\mathcal{I}$ into disjoint subsets of constraints. For each such subset I, the algebraic sum (see Section 4.3) of the current E over v is computed. Then the sum algorithm is recursively called for I, the newly computed E, $\mathcal{V} - \{v\}$, $\mathcal{P}$, and $\mathcal{R}$. Eventually at the deepest recursion level, $\mathcal{V}$ is empty, then E and its associated $\mathcal{I}$ represent one solution tuple defined solely over parameters.

In statement S1 we simplify $\mathcal{I}$, which includes detection and elimination of tautologies, equalities and redundant inequalities. Our simplification techniques and algorithms are described in detail in Section 5. If $\mathcal{I}$ contains a contradiction then statement S2 returns without adding a solution tuple to $\mathcal{R}$. In statement S3 the algorithm checks for an empty set of variables. If so, then there are no variables remaining for being eliminated, the algorithm adds the current solution tuple $(\mathcal{I}, E)$ into $\mathcal{R}$ and returns.

Statement S4 describes the core of the algorithm, which addresses several issues:

- In statement S4.1 a variable is chosen which is being eliminated in this recursion. A heuristic is used which selects the variable appearing in as few constraints of $\mathcal{I}$ as possible. Note that an inequality which contains more than one variable is a (lower or upper) bound for all of these variables.

- A variable $v \in \mathcal{V}$ may have several lower and upper bounds in $\mathcal{I}$. In order to compute the sum of all integer solutions for v over all constraints in $\mathcal{I}$, we need to know the maximum lower bound and minimum upper bound of v. In general we may not know them, therefore, we have to split $\mathcal{I}$ into disjoint subsets of constraints. For each such subset $\mathcal{I}'_{i,j}$ it is assumed that l_i and u_j is the associated maximum lower and minimum upper bound for v, respectively. This is shown in statement S4.3. Clearly, in this step we may create sets of constraints $\mathcal{I}'_{i,j}$ that contain contradictions which are tried to be detected in S2 by using our simplification techniques (see Section 5). Choosing v according to the previous heuristic implies a minimum number of different $\mathcal{I}'_{i,j}$ created at this step.

- For all of the previously created subsets $\mathcal{I}'_{i,j}$, we compute the algebraic sum $E_{i,j}$ (see Section 4.3) based on the current E for v bounded by l_i and u_i.

- The algorithm is recursively called with $\mathcal{I}'_{i,j}$, $\mathcal{V} - \{v\}$, $\mathcal{P}$, $E_{i,j}$, and $\mathcal{R}$

We further extended the sum algorithm of Figure 2 to a larger class of constraints that goes beyond $\mathcal{I}$ as described by (7). In the following we assume that $\mathcal{I}$ represents a set of constraints that satisfy the conditions mentioned by (7). Let $\mathcal{I}_1$ be a set of constraints defined over $\mathcal{V} \cup \mathcal{P}$ which is given by

$$p_1(\vec{P}) f_1(\vec{V_1}) + \ldots + p_k(\vec{P}) f_k(\vec{V_k}) \;\; REL \;\; 0 \tag{11}$$

SUM($\mathcal{I}$,$\mathcal{V}$,$\mathcal{P}$,E,$\mathcal{R}$)

- **INPUT:**
  $\mathcal{I}$: set of linear and non-linear constraints defined over $\mathcal{V} \cup \mathcal{P}$
  $\mathcal{V}$: set of variables
  $\mathcal{P}$: set of parameters
  E: symbolic expression defined over $\mathcal{V} \cup \mathcal{P}$
- **INPUT-OUTPUT:**
  $\mathcal{R}$: set of solution tuples $(\mathcal{C}_i, \mathcal{S}_i)$ where $1 \leq i \leq k$. $\mathcal{C}_i$ is a conjunction of linear or non-linear constraints defined over $\mathcal{P}$. S_i is a linear or non-linear symbolic expression defined over $\mathcal{P}$.
- **ALGORITHM:**

S1: Simplify $\mathcal{I}$

S2: **if** $\mathcal{I}$ is inconsistent (no solution) **then**
  return
  endif

S3: **if** $\mathcal{V} = \phi$ **then**
  $\mathcal{R} := \mathcal{R} \cup \{\mathcal{I}, E\}$
  return
  endif

S4: Split $\mathcal{I}$

S4.1: Choose variable $v \in \mathcal{V}$ for being eliminated

S4.2: $\mathcal{I}'' :=$ subset of $\mathcal{I}$ not involving v
  $\mathcal{L} :=$ set of lower bounds of v in $\mathcal{I}$
  $\mathcal{U} :=$ set of upper bounds of v in $\mathcal{I}$
  $a :=$ cardinality of $\mathcal{L}$
  $b :=$ cardinality of $\mathcal{U}$

S4.3: **for each** $(l_i, u_j) \in \mathcal{L} \times \mathcal{U}$ **do**

$$\mathcal{I}'_{i,j} := \mathcal{I}'' \cup \{ l_1 < l_i, .., l_{i-1} < l_i, l_{i+1} \leq l_i, .., l_a \leq l_i \} \cup \{u_1 > u_j, .., u_{j-1} > u_j, u_{j+1} \geq u_j, .., u_b \geq u_j\} \cup \{l_i \leq u_j\}$$

$$E_{i,j} := \sum_{v=l_i}^{u_j} E$$

SUM($\mathcal{I}'_{i,j}, \mathcal{V} - \{v\}, \mathcal{P}, E_{i,j}, \mathcal{R}$)

endfor

S5: **return**

Figure 2: Symbolic Sum Algorithm

$p_i(\vec{P})$ and REL have the same meaning as described by (7). $\vec{V_i}$ describes a vector of variables over V_i (subset of variables in $\mathcal{V}$) with $1 \leq i, j \leq k$. $f_i(\vec{V_i})$ are linear or non-linear expressions defined over $\mathcal{V}$ of the following form (simplified to their simplest form):

$$\frac{v_1 * \ldots * v_j}{w_1 * \ldots * w_l}$$

where all variables in $\{v_1, \ldots, v_j, w_1, \ldots, w_l\} \subseteq \mathcal{V}$ must be pairwise different after simplification. Furthermore, any variable $v \in \mathcal{V}$ may appear in several f_i with the restriction that v must always appear either in the denominator or in the numerator of all f_i in which it appears. For instance, v may appear in the numerator of all f_i. However, v cannot appear in the numerator of f_i and in the denominator of f_j with $i \neq j$.

Clearly, $\mathcal{I} \subseteq \mathcal{I}_1$. Unfortunately, in general the algorithm cannot handle constraints of the form described by (11), as for certain $\mathcal{I}_1$ the algorithm may create a new set of constraints in statement S4.3 that does not honor the constraints of (11). We have not yet found a way to narrow down the constraints of $\mathcal{I}_1$, such that the algorithm never creates a constraint out of it that does not satisfy (11). Nevertheless, we extended the symbolic sum algorithm to handle a larger class of constraints as described in (7) by simply checking at the beginning of the algorithm whether $\mathcal{I}$ satisfies (11). If yes, then the algorithm proceeds, otherwise, the algorithm returns and indicates to the user that the original $\mathcal{I}$ cannot be processed for the reasons mentioned above. For instance, this will enable the algorithm to count the integer solutions to the following set of constraints which is not covered by (7):

$$1 \leq J_1 \leq N_1,\ 1 \leq J_1 * J_2 \leq N_2 * J_1,\ N_1 * N_2 \leq J_1 * J_2$$

4.3 Algebraic Sum

In the following we describe the computation of the algebraic sum $\sum_{v=l}^{u} E$ over a variable $v \in \mathcal{V}$, where l and u have been extracted (S4.3 in Figure 2) from constraints of $\mathcal{I}$. E is a symbolic expression over $\mathcal{V} \cup \mathcal{P}$, which has been simplified down to its simplest form by using our own symbolic manipulation package [19]. Computing the algebraic sum is then reduced to solving:

$$\sum_{v=l}^{u} E_0 + E_1 * v^{\pm 1} + E_2 * v^{\pm 2} + \ldots + E_q * v^{\pm q} \tag{12}$$

E_i is a linear or non-linear expression over $\mathcal{P} \cup \mathcal{V} - \{v\}$. $\pm q$ (q is an integer constant) means that the sign of q can either be positive or negative. The problem, therefore, amounts to computing the following: $\sum_{v=1}^{n} v^i$. In the following we show how to approximate all terms in (12) by an upper or lower bound or an approximate expression (average between upper and lower bound):

For the case where $i \geq 0$ we can use standard formulas for sums of powers of integers. They are described in [13] and reviewed in [21]. For $i < 0$ there are no closed forms known. However, for $i = -1$ and $2 \leq n$ it has been shown [13] that (ln is the natural logarithm):

$$\begin{array}{rcccl} ln(n) & < & \sum_{v=1}^{n} \frac{1}{v} & < & ln(n) + 1 \\ -ln(n) - 1 & < & \sum_{v=-1}^{-n} \frac{1}{v} & < & -ln(n) \end{array} \tag{13}$$

and also [5] that $\sum_{v=1}^{\infty} \frac{1}{v^2} = \frac{\pi^2}{6} \approx 1.65$. Based on this approximation it can be shown that for all $i \geq 2$ the following holds: $\sum_{v=1}^{\infty} \frac{1}{v^2} \geq \sum_{v=1}^{\infty} \frac{1}{v^i}$, and finally

$$\begin{array}{rcccll} 1 & < & \sum_{v=1}^{\infty} \frac{1}{v^i} & \leq & \frac{\pi^2}{6} & \text{if } i \geq 2 \\ 1 & < & \sum_{v=-1}^{-\infty} \frac{1}{v^i} & \leq & \frac{\pi^2}{6} & \text{if } i \geq 2 \text{ and } i \text{ is even} \\ -\frac{\pi^2}{6} & \leq & \sum_{v=-1}^{-\infty} \frac{1}{v^i} & < & -1 & \text{if } i \geq 2 \text{ and } i \text{ is odd} \end{array} \tag{14}$$

4.4 Miscellaneous

In this section we briefly describe how to handle equalities, $\neq$ inequalities, min/max functions, and fractional expressions in our symbolic sum algorithm.

- Equalities are eliminated by our simplification techniques according to Section 5.
- If $\mathcal{I}$ contains a "$\neq$" inequality I such as e.g. $J_1 * J_2 \neq n$, then we split $\mathcal{I}$ into two disjoint subsets $\mathcal{I}_1$ and $\mathcal{I}_2$ such that $\mathcal{I}_1 := \mathcal{I} - I \cup \{J_1 * J_2 > N\}$ and $\mathcal{I}_2 := \mathcal{I} - I \cup \{J_1 * J_2 < N\}$. The sum algorithm is then called for both subsets $\mathcal{I}_1$ and $\mathcal{I}_2$ and the combined result is equivalent to the original problem including the "$\neq$" inequality.
- In many cases we can replace *min* and *max* functions appearing in inequalities. E.g let $I \in \mathcal{I}$ be an inequality $J_1 \leq min(J_2, J_3)$, then we replace I by the conjunction of two inequalities $J_1 \leq J_2$ and $J_1 \leq J_3$ in $\mathcal{I}$. If, however, I is given by $J_1 \leq max(J_2, J_3)$, then we have to split $\mathcal{I}$ into two disjoint subsets $\mathcal{I}_1$ and $\mathcal{I}_2$ such that $\mathcal{I}_1 := \mathcal{I} - I \cup \{J_1 \leq J_2, J_2 \geq J_3\}$ and $\mathcal{I}_2 := \mathcal{I} - I \cup \{J_1 \leq J_3, J_2 < J_3\}$. The algorithm is then called for both subsets $\mathcal{I}_1$ and $\mathcal{I}_2$ and the combined result is equivalent to the original problem including the max function. $\geq$ inequalities are handled similarly.
- There are several options to handle fractional expressions (floor and ceiling functions) in constraints and general symbolic expressions:
  - The easiest way to handle floor and ceiling operations is to assume that all operations are defined over rational numbers. This is clearly an approximation of the integer sum computation which may induce high estimation errors. However, we also observed that in many cases this approach achieves very reasonable estimates.
  - If a fractional contains only parameters and constants, then we can replace it by a newly introduced parameter and proceed as if there were no fractional. In the result we will substitute all such parameters back to their original fractional expression. This alternative may imply quite complex inequalities and result expressions (even though exact) as the dependencies among such fractional parameters can only partially be exploited by our simplification techniques. Of course, simplification within fractionals can still be done.
  - Fractional expressions can be replaced by their corresponding lower or upper bounds based on the following observation:

    $$\frac{E_1}{E_2} - 1 < \lfloor \frac{E_1}{E_2} \rfloor \leq \frac{E_1}{E_2} \leq \lceil \frac{E_1}{E_2} \rceil < \frac{E_1}{E_2} + 1$$

    The difference between the lower and upper bound may be used to specify the estimation accuracy. Also the average between lower and upper bound can be used to replace a fractional expression.

A detailed example of how to apply our symbolic sum algorithm is given in Section 7.2.

5 Simplify 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 [11, 14, 17] to be critical in order to alleviate performance as well as compiler analysis. We have implemented the following simplification techniques:

- Tautology inequalities (e.g. $2 > 0$, $N \leq N$, and $N \leq N * N$) 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. For instance, let I_1 be $2 * J_1 * J_2 - N \leq 0$ and I_2 is $2 * J_1 * J_2 - N \geq 0$, which implies $I_3 : 2 * J_1 * J_2 - N = 0$. 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. Searching for a coefficient with small complexity in a non-linear symbolic expression can be in general quite tricky. We apply a heuristic that proceeds as follows: First search for constant

coefficients. If there are no constant coefficients search for single parameter coefficients, then for coefficients that are linear symbolic expressions. Otherwise take any coefficient. Only if there are no variables in the equality we will try to solve it for a parameter and proceed as for variables. Eliminating variables before parameters has higher priority as many symbolic compiler analyses try to eliminate variables in a set of constraints and provide results as symbolic expressions defined over parameters. Once we have successfully substituted equality I_3 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. $2*J_1*J_2+N>0$ and $2*J_1*J_2+N+3\leq 0$), which means that $\mathcal{I}$ has no solution. If contradictions are detected most 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.

  The theorem for inequality relationships $>$ and $\geq$ is similar. Algorithm EXPR_BOUND can therefore also be used to detect and eliminate redundant inequalities.

6 Related Work

Blume [4] 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. Furthermore, 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 or detect zero-trip-loops [14], loop variables whose lower bounds can be larger than their upper bounds for some loop iterations must be modeled. 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 or dead code elimination.

M. Haghighat and C. Polychronopoulos [14] described 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 [22] present an algorithm to determine the relationship between two symbolic expressions which is based on gated static single assignment (G-SSA) form. V. Maslov and W. Pugh [17] propose a method to simplify polynomial constraints over integers to make dependence analysis more precise. They try to transform a general polynomial constraint into a conjunction of affine constraints by applying factorization and affinization. Although their general techniques seems very powerful, they do not address the issue of detecting and eliminating redundant constraints.

P. Havlak [15] uses G-SSA for comparing symbolic expressions and for data dependence testing as part of interprocedural compiler analysis. His techniques are efficient for linear expressions but are limited in case of non-linear expressions.

E. Su et al. [20] implemented an interface between the Paradigm parallelizing compiler [1] and a commercial symbolic manipulation package. Based on symbolic analysis they developed several compiler analyses and transformations including integer area estimation, computation partitioning and cyclic array distribution.

Our symbolic sum algorithm as shown in Figure 2 is an extension to what has been sketched by W. Pugh as the convex sum for linear inequalities in [18]. Pugh only superficially outlined his techniques and has not implemented his algorithm according to [18]. Whereas we have presented a precise definition for the symbolic sum algorithm, fully implemented it, and are currently integrating our techniques with P^3T to apply them to

performance prediction applications. Pugh described his algorithm only for a set of linear inequalities, whereas we extended the algorithm to cover also non-linear inequalities. His techniques are based on simplifications for linear inequalities only, whereas our simplification techniques (see Section 5) handle also classes of non-linear inequalities and expressions. Pugh's algebraic sum algorithm is restricted to variables with positive exponents only, whereas our method also handles negative exponents.

Other related work about counting solutions to a system of constraints has been reflected throughout Section 4.

7 Experiments

All algorithms described in this paper have been implemented. We are currently developing symbolic dependence analysis, symbolic communication optimization and symbolic performance prediction as part of *VFCS* and P^3T all of which is based on the described algorithms. In what follows, we describe three experiments to measure the potential benefits of our techniques for improving the analysis and performance of parallel programs.

7.1 Eliminate Redundant Constraints

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.

$$\mathcal{I} = \{1 \leq J_1 \leq N,\ \tfrac{N}{2*J_1} \leq J_2 \leq N,\ J_1 * J_2 \leq N\}$$

This yields the following sets of variables, parameters, lower and upper bounds:
$\mathcal{V} = \{J_1, J_2\}, \mathcal{P} = \{N\}, LB_{J_1} = \{1, \frac{N}{2*J_2}\}, UB_{J_1} = \{N, \frac{N}{J_2}\}, LB_{J_2} = \{\frac{N}{2*J_1}\}, UB_{J_2} = \{N, \frac{N}{J_1}\}, LB_N = \{J_1, J_2, J_1 * J_2\}, UB_N = \{2 * J_1 * J_2\}, \mathcal{B} = LB_N \cup UB_N \cup \bigcup_{1 \leq i \leq 2} (LB_{J_i} \cup UB_{J_i})$

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 J_1 in $\mathcal{I}$ is redundant. If we can show that $low(N - \frac{N}{J_2}) \geq 0$ for all possible values of variables and parameters in $\mathcal{V} \cup \mathcal{P}$ according to the theorem on page 13, then $J_2 \leq N$ is redundant in $\mathcal{I}$. For this purpose we invoke EXPR_BOUND(E=low(N-$\frac{N}{J_2}$),B,Q=$\{(J_1,-\infty,\infty),(J_2,-\infty,\infty),(N,-\infty,\infty)\}$). 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. $\mathcal{Q} = \{(J_1, 1, N), (J_2, 1, N), (N, 1, \infty)\}$ 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 $up(\frac{N}{J_2})$, the value range for J_2 needs to be known. Therefore, the algorithm is recursively (recursion 2) called for $E = low(J_2)$ to find out whether J_2 can ever be negative.

3.2 Allows to deduce that $low(J_1) \geq 1$ which is used to update the lower bound of J_1 in $\mathcal{Q}$. Then the algorithm is evaluating $low(\frac{N}{2*J_2})$ to investigate whether this expression is larger than 1. In order to determine the correct rewrite rule to apply to $low(\frac{N}{2*J_2})$, we need to determine the value range for J_2. As recursion 2 already tried to deduce $low(J_2)$, we add $-\infty \leq J_2$ into $\mathcal{Q}$ based on the termination condition (see Section 3.2) of EXPR_BOUND. Then the algorithm is recursively called for $E = up(J_2)$ to determine whether J_2 can ever be positive.

5.3 As the maximum function for $low(N)$ contains 1, we can add $1 \leq N$ into $\mathcal{Q}$. In order to further substitute the minimum function we need to know the sign for $up(J_1)$.

6.3 As in recursion 1 step 2 the algorithm already tried to evaluate $up(\frac{N}{J_2})$, we substitute $up(\frac{N}{J_2})$ by ∞ according to the termination condition of EXPR_BOUND. After step 4 of this recursion the algorithm returns to recursion 5.

Rec	Step	E	$\mathcal{Q}$	Comments
1	1	$low(N - \frac{N}{J_2})$		
	2	$\rightarrow low(N) - up(\frac{N}{J_2})$		rule (8)
2	1	$low(J_2)$		
	2	$\rightarrow low(\frac{N}{2*J_1})$		rule (3)
3	1	$low(J_1)$		
	2	$\rightarrow max(low(\frac{N}{2*J_2}), 1)$	$(J_1, 1, \infty)$	rule (3)
			$(J_2, -\infty, \infty)$	TC for $low(J_2)$
4	1	$up(J_2)$		
	2	$\rightarrow min(up(N), up(\frac{N}{J_1}))$		rule (3)
5	1	$low(N)$		
	2	$\rightarrow max(low(J_1), low(J_2), low(J_1 * J_2))$		rule (3)
	3	$\rightarrow max(1, low(J_2), min(low(J_1) * up(J_2),$	$(N, 1, \infty)$	
		$up(J_1) * low(J_2), low(J_1) * low(J_2), up(J_1) * up(J_2))$		rule (9)
6	1	$up(J_1)$		
	2	$\rightarrow min(up(N), up(\frac{N}{J_2}))$		rule (1)
	3	$\rightarrow min(up(N), \infty)$		TC for $up(\frac{N}{J_2})$
	4	$\rightarrow up(N)$	$(J_1, 1, N)$	
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	$\rightarrow min(up(N), \frac{up(N)}{low(J_1)}))$		rule (10)
	4	$\rightarrow min(up(N), \frac{up(N)}{1})$		substitute $low(J_1)$
	5	$\rightarrow up(N)$	$(J_2, -\infty, N)$	
3	3	$\rightarrow max(min(\frac{up(N^-)}{low(2*J_2)}, \frac{low(N)}{low(2*J_2^+)}, \frac{up(N)}{up(2*J_2^-)}, \frac{low(N)}{up(2*J_2)}), 1)$		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	$\rightarrow \frac{low(N)}{2*up(J_1)}$		rule (10)
	4	$\rightarrow \frac{low(N)}{2*up(N)}$		substitute $up(J_1)$
	5	$\rightarrow \frac{1}{2}$		simplify
	6	$\rightarrow 1$	$(J_2, 1, N)$	J_2 integer
1	3	$\rightarrow low(N) - \frac{up(N)}{low(J_2)}$		rule (10)
	4	$\rightarrow low(N) - \frac{up(N)}{1}$		substitute $low(J_2)$
	5	$\rightarrow low(N) - up(N)$		simplify
	6	$\rightarrow 0$		proved $N \geq \frac{N}{J_2}$

Table 2: Using EXPR_BOUND to prove that $N \geq \frac{N}{J_2}$.

2.5 The expression of recursion 2 step 4 has been simplified according to statement S7, which yields $\frac{1}{2}$. This means that $low(J_2) \geq \frac{1}{2}$, and consequently, $low(J_2) \geq 1$ as $low(J_2)$ must be an integer.

1.5 Simplifies $low(N) - up(N)$ according to statement S7, which yields $N - N = 0$.

1.6 Proves that $N \geq \frac{N}{J_2}$ for all values of J_1, J_2, and N. Therefore, the inequality $J_2 \leq N$ is redundant in $\mathcal{I}$ according to the theorem of page 13.

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 Section 4 we have described a symbolic sum algorithm that computes the number of solutions of a set of constraints which are defined over variables and parameters. For instance, by detecting and eliminating the redundant inequality $J_2 \leq N$ 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 $J_1, J_2, N \geq 1$ 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: $low(N - \frac{N}{J_2}) \rightarrow low(N) - up(\frac{N}{J_2}) \rightarrow low(N) - \frac{up(N)}{low(J_2)} \rightarrow low(N) - \frac{up(N)}{1} \rightarrow low(N) - up(N) \rightarrow 0$. Therefore, our implementation of EXPR_BOUND can prove in 5 steps that $N \geq \frac{N}{J_2}$ under the constraints of $\mathcal{I}$.

7.2 Counting Solutions to a System of Constraints

In what follows we present an experiment that shows how the symbolic sum algorithm can be used to estimate the amount of work to be processed by every processor of a data parallel program. The following code shows a High Performance Fortran - HPF code excerpt with a processor array PR of size P.

```
INTEGER A(N2)
!HPF$ PROCESSORS :: PR(P)
!HPF$ DISTRIBUTE (BLOCK) ONTO PR :: A
DO 10 J1=1,N1
    DO 10 J2=1,J1 * N1
        if ( J2  ≤  N2 ) A(J2) = ...
10  CONTINUE
```

The loop contains a write operation to a one-dimensional array A which is block-distributed onto P processors. Let k $(1 \leq k \leq P)$ denote a specific processor of the processor array. Computations that define the data elements owned by a processor k are performed exclusively by k. For the sake of simplicity we assume that P evenly divides N_2. Therefore, a processor k is executing the assignment to A based on the underlying block distribution if $\frac{N_2*(k-1)}{P} + 1 \leq J_2 \leq \frac{N_2*k}{P}$. The precise work to be processed by a processor k is the number of times k is writing A, which is defined by *work(k)*.

N_1	N_2	$\sum_{1 \leq k \leq P} work(k)$	$\sum_{1 \leq k \leq P} \overline{work(k)}$	error in %
100	100	10000	10025	0.25
200	100	20000	20238	1.19
200	200	40000	40450	1.12
400	200	80000	80474	0.59
400	400	160000	160900	0.56
800	400	320000	320950	0.29
800	800	640000	641800	0.28

Table 3: Measured versus estimated values for the amount of work to be done by all processors for $\mathbf{P = 4}$.

The problem to estimate the amount of work to be done by processor k can now be formulated as counting the number of integer solutions to $\mathcal{I}$ which is given by:

$$\begin{array}{rcccl} & 1 & \leq & J_1 & \leq & N_1 \\ & 1 & \leq & J_2 & \leq & J_1 * N_1 \\ & & & J_2 & \leq & N_2 \\ \frac{N_2*(k-1)}{P} + 1 & & \leq & J_2 & \leq & \frac{N_2*k}{P} \end{array}$$

In the following we substitute $\frac{N_2*(k-1)}{P} + 1$ by LB and $\frac{N_2*k}{P}$ by UB. By using our simplification techniques of Section 5 we can determine that $1 \leq J_2$ is made redundant by $LB \leq J_2$ and $J_2 \leq N_2$ by $J_2 \leq UB$. Therefore, the simplified $\mathcal{I}$ with all redundant inequalities removed is given by

$$\begin{array}{rcccl} 1 & \leq & J_1 & \leq & N_1 \\ LB & \leq & J_2 & \leq & UB \\ & & J_2 & \leq & J_1 * N_1 \end{array}$$

In the first recursion we can either choose variable J_1 or J_2 for being eliminated according to the heuristic of S4.1 of our sum algorithm (see Figure 2). Both variables appear in 3 inequalities of the simplified $\mathcal{I}$. By selecting J_2 we obtain two upper bounds for J_2 which are given by $\{J_1 * N_1, UB\}$. Based on S4.3. we split the inequalities of $\mathcal{I}$ into I_1 and I_2.

- $I_1 = \{1 \leq J_1 \leq N_1,\ J_1 * N_1 \leq UB,\ LB \leq J_1 * N_1\}$ with $\sum_{J_2=LB}^{J_1*N_1} 1 = J_1 * N_1 - LB$.

  In S1 of the algorithm (see Figure 2) we detect that $1 \leq J_1$ is made redundant by $\frac{LB}{N_1} \leq J_1$. We now eliminate J_1 based on two upper bounds $\{\frac{UB}{N_1}, N_1\}$ for this variable. This yields two solutions C_1 and C_2

- $C_1 = \{\frac{UB}{N_1} \leq N_1,\ LB \leq UB\} = \{UB \leq n_1^2,\ P \leq N_2\}$ with
  $E_1(k) = \sum_{J_1=\frac{LB}{N_1}}^{\frac{UB}{N_1}} J_1 * N_1 - LB = \frac{(N_1+UB-LB)*(LB-2*N_1+2*LB*N_1+UB)}{2*N_1^2}$.
- $C_2 = \{\frac{UB}{N_1} > N_1,\ \frac{LB}{N_1} \leq N_1\}$ with $E_2(k) = \sum_{J_1=\frac{LB}{N_1}}^{N_1} J_1 * N_1 - LB = (N_1 - \frac{LB}{N_1} + 1) * (\frac{N_1^2}{2} - \frac{LB}{2} + 1)$.

- $I_2 = \{1 \leq J_1 \leq N_1,\ J_1 * N_1 > UB,\ LB \leq UB\} = \{1 \leq J_1 \leq N_1,\ J_1 \geq \frac{UB+1}{N_1},\ LB \leq UB\}$ with $\sum_{J_2=LB}^{UB} 1 = \frac{N_2}{P}$.

  The algorithm detects in S1 that $1 \leq J_1$ is made redundant by $\frac{UB+1}{N_1} \leq J_1$. Therefore, we can eliminate J_1 for $C_3 = \{N_2 \geq P,\ N_1^2 \geq UB+1\}$ which yields $E_3(k) = \sum_{J_1=\frac{UB+1}{N_1}}^{N_1} \frac{N_2}{P} = \frac{N_2}{P} * (N_1 - \frac{UB+1}{N_1} + 1)$.

Finally, statement S in the example code is approximately executed $\overline{work(k)} = \sum_{1 \leq i \leq 3} \gamma(C_i) * E_i(k)$ times by a specific processor k $(1 \leq k \leq P)$ for the parameters N_1, N_2 and P. $\gamma(C_i)$ is defined by (10). Note that most conventional performance estimators must repeat the entire performance analysis whenever the problem size or the number of processors used are changing. However, our symbolic performance analysis provides the solution of the above problem as a symbolic expression of the program unknowns (P,N_1,N_2, and k). For each change in the value of any program unknown we simply re-evaluate the result, instead of repeating the entire performance analysis. Table 3 shows an experiment which compares measured against estimated values for a 4 $(= P)$ processor version of the example code by varying the values for N_1 and N_2. The measurements have been done by executing the example code on an iPSC/860 hypercube system and enumerating for each processor the number of times it writes array A in the loop of the example code. It can be seen that the estimates $(\overline{work(k)})$ are very close to the measurements $(work(k))$. In the worst case the estimates are off by 1.19 % for a relative small problem size $(N_1 = 200, N_2 = 100)$. Moreover, we also observe that for larger problem sizes the estimation accuracy consistently improves.

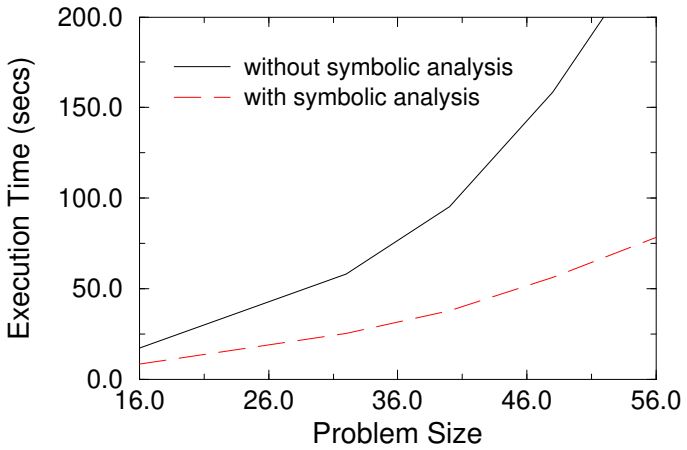

Figure 3: Measured execution times of FTRVMT with and without symbolic analysis for varying problem sizes (NEX) on a MEIKO CS-2 with 16 processors

7.3 Optimizing FTRVMT

In this section we study the potential performance benefits as implied by our advanced symbolic analysis for FTRVMT which is the core subroutine of the OCEAN perfect benchmark (simulates 2-dimensional fluid flow – [7]). FTRVMT accounts for close to 50 % of the overall execution time of OCEAN. FTRVMT contains non-linear

array subscript expressions and 5 DO-loop nests which iterate over a one-dimensional array DATA. We created two parallel program versions for varying problem sizes specified through parameter NEX (ranging from 16 to 56) in the OCEAN benchmark:

Firstly, a version (solid line curve in Figure 3) as created by *VFCS* without symbolic analysis. A true dependence that affects array DATA for all 5 loops of FTRVMT is assumed which prevents communication optimization.

Secondly, the VFCS code was hand-instrumented (dashed line curve in Figure 3) to reflect the communication optimizations that are enabled through a symbolic dependence test. The symbolic dependence test requires to compare non-linear symbolic expressions which is based on our algorithm EXPR_BOUND (see Section 3.1). For the first 3 loops of FTRVMT our algorithm can be used to detect the absence of a true dependence on DATA which enables communication vectorization and aggregation.

For both program versions we applied induction variable substitution [2] and block-distributed array DATA.

Figure 3 shows the execution times of both versions for varying problem sizes. Clearly, the version based on a symbolic dependence test outperforms the original VFCS code. Our techniques for comparing symbolic expressions enable hoisting communication out of loop nests. The current VFCS without symbolic analysis implies that communication remains inside of the innermost loop which drastically increases the number of network messages. Although our symbolic analysis detects the absence of a true dependence with respect to array DATA for only three loop nests, it makes a big difference in the overall execution time ranging by a factor of roughly 2 - 3 for the problem sizes measured and consistently improves for increasing problem sizes.

8 Conclusions

Numerous researchers have shown the importance of symbolic compiler analysis [14, 3, 17, 15, 4, 20, 21, 11, 22, 6] for optimizing and analyzing performance of parallel programs. A crucial problem of many symbolic compiler analyses is to determine the relationship between symbolic expressions.

We have described an algorithm for computing lower and upper bounds of wide classes of linear and non-linear symbolic expressions based on a set of constraints defined over loop variables and loop invariants. This algorithm supports comparing symbolic expressions for equality and inequality ($=, <, >$) relationships, simplifying systems of constraints, examining non-linear array subscript expressions for data dependences, and optimizing communication.

Furthermore, we presented a symbolic sum algorithm that estimates the number of integer solutions to a system of constraints. This algorithm supports detection of zero-trip loops, elimination of dead code, and performance prediction of parallel programs.

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 and more complex (due to the lack of aggressive simplification techniques) results.

Preliminary results demonstrate 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. Furthermore, it has been demonstrated that our symbolic sum algorithm can estimate the work distribution of a parallel program with high accuracy. Finally, we showed how our techniques can be used to significantly improve the communication costs and consequently the execution time of a code taken from the perfect benchmarks.

We are currently implementing a variety of performance analysis and compiler optimizations that aggressively use our symbolic analysis techniques. Another area of ongoing work concentrates on collecting constraints about program variables that are used to further improve symbolic analysis.

References

[1] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers. In *Proceedings of the First International Workshop on Parallel Processing*, pages 322–330, Bangalore, India, December 1994.

[2] 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.P. Zima. *Vienna Fortran Compilation System - Version 2.0 - User's Guide*, October 1995.

[3] W. Blume and R. Eigenmann. An Overview of Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks. In *Proceedings of ICPP'94*, St. Charles, IL, 1994.

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

[5] I. Bronstein and K. Semendjajev. *Taschenbuch der Mathematik.* Verlag Harri Deutsch, Frankfurt, Germany, ISBN 3 87144492 8, 1989.

[6] M.J. Clement and M.J. Quinn. Symbolic Performance Prediction of Scalable Parallel Programs. In *Proc. of 9th International Parallel Processing Symposium*, St. Barbara, CA, April 1995.

[7] M. Berry et al. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, pages 3(3): 5–40, 1989.

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

[9] T. Fahringer. Automatic Estimation of Communication Costs for Data Parallel Programs. *Journal of Parallel and Distributed Computing, Academic Press*, 39(1):46–65, Nov. 1996.

[10] T. Fahringer. On Estimating the Useful Work Distribution of Parallel Programs under P^3T: A Static Performance Estimator. *Concurrency Practice and Experience (Ed. Geoffrey Fox), Wiley*, 8(4):261 – 282, May 1996.

[11] 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 15 - 19, 1996.

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

[13] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics - A foundation for Computer Science.* Addison-Wesley Publishing Company, ISBN 0-201-14236-8, 1988.

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

[15] Paul Havlak. *Interprocedural Symbolic Analysis.* PhD thesis, Department of Computer Science, Rice University, 1994.

[16] High Performance FORTRAN Language Specification. Technical Report, Version 1.0, Rice University, Houston, TX, May 1993.

[17] Vadim Maslov and William Pugh. Simplifying Polynomial Constraints Over Integers to Make Dependence Analysis More Precise. UMIACS-TR-93-68.1 and CS-TR-3109.1, Univ. of Maryland, Dept. of Computer Science, February 1994.

[18] William Pugh. Counting Solutions to Presburger Formulas: How and Why. In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 121–134, Orlando, FL, June 20-24 1994.

[19] M. Scheibl, A. Celic, and T. Fahringer. Interfacing Mathematica from the Vienna Fortran Compilation System. Technical Report, Institute for Software Technology and Parallel Systems, Univ. of Vienna, December 1996.

[20] E. Su, A. Lain, S. Ramaswamy, D. Palermo, E. Hodges, and P. Banerjee. Advanced Compilation Techniques in the Paradigm Compiler for Distributed-Memory Multicomputers. In *9th ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[21] N. Tawbi. Estimation of nested loop execution time by integer arithmetic in convex polyhedra. In *Proc. of the 1994 International Parallel Processing Symposium*, April 1994.

[22] 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.