Symbolic Evaluation for Parallelizing Compilers*

Thomas Fahringer† Bernhard Scholz‡

†Institute for Software Technology and Parallel Systems, University of Vienna
Liechtensteinstr. 22, A-1090, Vienna, Austria
tf@par.univie.ac.at

‡Department of Automation, University of Technology
Treitlstr. 1/4, A-1040 Vienna, Austria

Published in Proc. of the 11th ACM International Conference on Supercomputing, Vienna, Austria, July, 1997

Abstract

In this paper we describe efficient symbolic evaluation techniques to compute the values of variables and symbolic expressions, and to determine the condition under which control flow reaches a program statement at compile time. Computations are represented as algebraic expressions over the input data which maintains the crucial relationship between input data and the resulting analysis information. Our symbolic evaluation techniques comprise accurate modeling of assignment and conditional statements, loops, recurrences, arrays (including indirect accesses) and procedures. Efficiency and accuracy is highly improved by aggressive usage of simplification techniques. Examples including program verification, dependence analysis, array privatization, communication vectorization, and elimination of redundant communication are used to illustrate how our symbolic evaluation techniques support program optimization in the context of a distributed memory parallelizing compiler.

1 Introduction

It is widely accepted [4, 14, 11] that current parallelizing compilers are not very effective in optimizing parallel programs that fully utilize the target multiprocessor system. Among others the poor performance of compilers can be attributed to ineffective parallelization of programs (for instance, HPF and Fortran 90) that have a strong potential for unknowns such as number of processors and sizes of allocateable arrays. Their inability to make the relationship between input data and analysis information transparent which is vital for performance driven optimization. Their lack of interprocedural analysis. Their poor capabilities to analyze dependences between references with linearized subscripts, and non-linear terms in subscript expressions arising from recurrences or tiling with symbolic block sizes. As a consequence worst case assumptions are frequently made or program analysis is moved into runtime at the cost of additional runtime overhead. Sophisticated symbolic analysis that can deal with program unknowns is needed to eliminate or at least alleviate these compiler deficiencies. Due to the complexity of symbolic analysis that is needed to tackle the problems mentioned above only few research efforts have been started in the field of symbolic analysis to support compilers and programming tools for parallel programs.

Previous work on symbolic compiler analysis has more or less severe drawbacks associated which them: restricted program models (no procedures or conditional branch statements), analysis that covers only linear symbolic expressions, insufficient simplification techniques, memory and runtime intensive algorithms, representation of analysis information which deteriorates readability of the underlying code, and lack of maintaining the crucial relationship between input data and analysis result information. It has been realized [13] that systems that are open for off-the-shelf software are critical for future research tool and compiler development. Approaches that rely on non-standard functions and data representation for their analysis are commonly forced into time-consuming re-implementation of standard symbolic manipulation techniques which otherwise could be taken from readily available software packages.

This paper describes a new approach that is based on compile time symbolic evaluation which tries to eliminate or at least alleviate many of the previously mentioned deficiencies. Our symbolic evaluation method combines both data and control flow analysis in a unified framework. Variable and expression values can be computed as algebraic functions defined over the input data at any given program point. The (path) condition under which the control flow reaches a program point is specified by a first order logic formula. We will describe how to model assignment and conditional statements, loops, arrays (including indirect accesses), and procedures as part of our symbolic evaluation. Recurrences are resolved and closed forms are computed. All of our techniques can handle both linear as well as non-linear expressions and constraints. We achieve good efficiency and high accuracy by aggressively using our techniques [9] for comparing symbolic expressions

* B. Scholz has been partially supported by FWF project nr. P10188-MAT.

and simplifying systems of constraints. Our approach fully maintains the crucial relationship between input data and analysis information. Our symbolic evaluation does not change the underlying code and therefore maintains readability. Finally, the representation of our analysis information is very general, which enables us to use off-the-shelf software for standard symbolic manipulation.

We will demonstrate the effectiveness of our approach for a variety of examples including program verification, dependence analysis, array privatization, communication vectorization, and elimination of redundant communication.

The organization of this paper is as follows. In Section 2, we describe our symbolic evaluation framework. This includes techniques for computing path conditions and values of variables and symbolic expressions. We will further illustrate how to model assignment and conditional statements, loops, recurrences, arrays, and procedures. Preliminary results that demonstrate the benefits of the proposed method are presented in Section 3. Related work will be discussed in Section 4. Section 5 will give our conclusions.

2 Symbolic Evaluation

Symbolic evaluation statically analyses a program at compile time. Each program variable is associated with a symbolic expression describing its value at a specific program point. In addition a path condition – represented by a first-order logic formula – describes the condition under which control flow reaches a given program point and the assumptions on variable values made for a given control flow branch.

The *state* S of a program is described by a set of pairs $\{< V_1, e_1 >, \ldots, < V_n, e_n >\}$ where V_i is a program variable and e_i an algebraic expression over the input data describing the value of V_i for $1 \leq i \leq n$. For each program variable V_i there exists exactly one pair $< V_i, e_i >$ in S. A state S and a path condition C specify a *program context* which is defined by a tuple $[S, C]$. A program context completely describes the variable bindings and path condition at a specific program point. A program consists of a sequence of statements or assertions P_j $(1 \leq j \leq r)$ that may change a program context. This can be described by a sequence $[S_0, C_0]$ P_1 $[S_1, C_1]$ $\ldots P_r$ $[S_r, C_r]$ where P_1 and P_r are respectively the entry and exit statement. $[S_0, C_0]$ and $[S_r, C_r]$ represent the program context at the entry and exit statement, respectively. For the sake of demonstration we assume that a program has only a single entry and exit statement. The actual implementation considers multiple entry and exit statements. If P_i in $\ldots [S_i, C_i]$ P_i $[S_{i+1}, C_{i+1}] \ldots$ does not contain side effects (implies a change of a variable value) then $S_i = S_{i+1}$ otherwise $S_i \neq S_{i+1}$. In the remainder of this paper "=" and ":=" respectively refer to equivalence and assignment.

For instance, in the following program

real D1,D2,A,B
  $[S_0=\{D1=\perp, D2=\perp, A=\perp, B=\perp\}, C_0=\text{true}]$
P_1: read(A,B)
  $[S_1 = \delta(S_0; A = \nabla_1, B = \nabla_2),\ C_1 = C_0]$
P_2: D1 := (A-B)*A
  $[S_2 = \delta(S_1; D1 = (\nabla_1 - \nabla_2) * \nabla_1),\ C_2 = C_0]$
P_3: D2 := 2*(A-B)
  $[S_3 = \delta(S_2; D2 = 2 * (\nabla_1 - \nabla_2)),\ C_3 = C_0]$
P_4: read(B)
  $[S_4 = \delta(S_3; B = \nabla_3),\ C_4 = C_0]$
P_5: assert(D1/D2 > A/2 - 1)
  $[S_5 = S_4,\ C_5 = C_0]$

for each statement P_i the corresponding context $[S_i, C_i]$ is displayed. A statement in "[...]" denotes that this statement is not an executable statement but serves only for the purpose of illustrating the context at a given program point. Note that for interactive tools (such as VFCS and P^3T) contexts can be optionally displayed. At the beginning of the program, all variable values are undefined ($\perp$). The path condition C_0 is set to *true* which is the default case. In program statement P_1 variables A, B are assigned an unknown value, which is specified by a special symbol (∇) for input values as these are of special interest. Input data values are propagated through the program which maintains the important relationship of program variable values and input data.

Most statements imply a change of only a single variable's value. In order to avoid large lists of variable values in state descriptions, only those variables are explicitly specified whose value changes after evaluation of the associated statement. For this reason we introduce a function δ:

$$S_i = \delta(S_j; V_1 = e_1, \ldots, V_l = e_l)$$

which specifies a state S_i whose variable binding is equal to that of state S_j except for variables V_i $(1 \leq i \leq l)$ which get new values e_i assigned. Therefore, in the previous example state S_1 is the same as state S_0 except for the values of A and B which are respectively ∇_1 and ∇_2 in state S_1.

P_2 and P_3 change only variable D_1 and D_2, respectively. P_4 is again a read statement that assigns a new input value ∇_3 to B. In order to verify whether the assert statement P_5 is true at this program point we have to evaluate the assert expression with respect to the program context immediately preceding P_5 which is $[S_4, C_4]$. We use a function

$$eval(e, [S, C])$$

that determines the value of a symbolic expression e under the constraints of program context $[S, C]$. For the previous example

$$\begin{aligned} & eval(\frac{D1}{D2} > \frac{A}{2} - 1, [S_4, C_4]) = \\ = \quad & \frac{(\nabla_1 - \nabla_2) * \nabla_1}{2 * (\nabla_1 - \nabla_2)} > \frac{\nabla_1}{2} - 1 \\ = \quad & true \end{aligned}$$

The assert statement is true at program point P_5 under the assumption that $\nabla_1 - \nabla_2$ never evaluates to 0 which would make the division operation in P_5 invalid. The path condition is true for all program statements. This is caused by the absence of control flow branches in this code example. Note that conventional scalar forward substitution of $D1$ and $D2$ cannot be used to determine that the assert statement is true, because the definitions in P_2 and P_3 are killed by the read statement P_4.

2.1 Path Condition

The path condition of a statement P describes the condition under which the control flow reaches P. Essentially all the branch statements of a program that reach P have to be examined in order to compute the path condition for P. There are several well-known choices to compute path conditions. Exact analysis must consider all incoming paths of a given statement which requires iterative data-flow analysis and implies exponential growth of the path condition. Instead we will incorporate control dependences [6] for computing path conditions.

A control flow graph (CFG) of a program is a directed graph whose nodes correspond to statements in the program. A CFG has a two specific nodes, *entry* and *exit*. There is an edge from *entry* to any statement at which the program can be entered, and there is an edge to *exit* from any statement that can exit the program. An edge exists between two nodes in a CFG if and only if the statement corresponding to the second node may be immediately executed after the statement corresponding to the first node. Let v and w be two nodes in CFG. If v appears on every path from w to *exit*, then v post-dominates w. If v post-dominates w but $v \neq w$, then v strictly post-dominates w.

A CFG node w is control dependent on a CFG node v if there is a non-null path[1] $v \rightarrow w$ in the CFG such that w post-dominates every node after v on this path and w does not strictly post-dominate v. In other words, there is some path from v that definitely causes w to execute, and there is also some path from v that avoids executing w. In [6] an efficient algorithm is presented to build the set of nodes that are control dependent on a node v.

The path condition of a statement P in our symbolic evaluation framework is computed as the intersection across all condition predicates of branch statements on which P is control dependent.

2.2 Conditional Statements

If the control flow of a program contains branches, the symbolic values for variables may not suffice to select a unique control flow branch. In such cases symbolic evaluation may have to follow multiple control flow branches. Each state is therefore associated with a control flow or *path condition* C in order to specify a program context. The path condition is specified as a first order logic formula. If a conditional statement is encountered the condition predicate is evaluated by using the variable bindings of the current context. If it cannot be statically determined whether for all input sets either the true or the false branch has to be followed then symbolic evaluation has to consider both branches.

For instance, in the following program

P_1: read(Y)
    $[S_1$={X=x_1,Y=$\bigtriangledown_1$},C_1=true$]$
P_2: if (Y < 0) then
    $[S_2$=S_1,C_2=$\bigtriangledown_1$ < 0$]$
P_3: X:=$-Y$
    $[S_3=\delta(S_1; X = -\bigtriangledown_1), C_3=C_2]$
P_4: endif
    $[S_4=\delta(S_1;\mathrm{X}=x_2), C_4=\gamma(\bigtriangledown_1<0; x_2=-\bigtriangledown_1; x_2=x_1)]$

P_5: if (Y < -10) then
    $[S_5=S_4, C_5=\bigtriangledown_1 < -10 \wedge\gamma(\bigtriangledown_1<0; x_2=-\bigtriangledown_1; x_2=x_1)] =$
    $= [S_5'=\delta(S_4;\mathrm{X}=-\bigtriangledown_1), C_5'=\bigtriangledown_1 < -10]$
P_6: X:=$2 * X$
    $[S_6=\delta(S_5'; X = -2 * \bigtriangledown_1), C_6=C_5']$
P_7: endif
    $[S_7=\delta(S_4; X = x_3),$
    $C_7=C_4\wedge\gamma(\bigtriangledown_1<-10; x_3=-2*\bigtriangledown_1; x_3=x_2)]$

we assume that at the beginning of this code segment X has the value x_1 and the path condition C is set to *true*.

In P_2 a condition predicate without side-effects is evaluated, which implies no change in the program state. However, the corresponding path condition of all statements within the then-part of this statement includes $\bigtriangledown_1 < 0$ where $\bigtriangledown_1$ is the current value of Y.

A confluence of two branches occurs after statement P_4. Two different contexts $[S_1, C_1]$ and $[S_3, C_3]$ reach this point. We use an operator for folding contexts which takes as arguments two program contexts and produces a new program context resulting from the composition of its arguments. If a variable has the same value in both the input contexts its value remains unchanged in the resulting context, otherwise it receives a new symbolic value. The path condition of the resulting context is the conjunction of two parts corresponding to the two input contexts. Each part is the conjunction of the path condition and a predicate binding the new symbolic values to the symbolic expressions the corresponding variables have in the resulting context.

Therefore, at the confluence node immediately after P_4, the value x_2 of X can be bound to 2 different expressions. First, $x_2 = -\bigtriangledown_1$ if the condition predicate of P_2 holds, otherwise $x_2 = x_1$. Formally this is described by C_4=$(\bigtriangledown_1<0\wedge x_2 = -\bigtriangledown_1) \vee (\bigtriangledown_1 \geq 0 \wedge x_2 = x_1)$ where $\vee$ and $\wedge$ have lower operator priority than $<, \geq$, and $=$. In order to avoid long first order logic formulas we introduce a function γ:

$$\gamma(c;\ x_1 = e_1, \ldots, x_k = e_k;\ x_1 = f_1, \ldots, x_k = f_k)$$

which is semantically equivalent to $(c \wedge x_1 = e_1 \wedge \ldots \wedge x_k = e_k) \vee (\neg\, c \wedge x_1 = f_1 \wedge \ldots \wedge x_k = f_k)$. c is a condition expression, $\neg c$ the negation of c, x_i $(1 \leq i \leq k)$ represent variable values and e_i, f_i are symbolic expressions. Therefore, the path condition C_4 for the previous code example can be written as $\gamma(\bigtriangledown_1<0; x_2=-\bigtriangledown_1; x_2=x_1)$.

C_5 is equal to $C_4 \wedge \bigtriangledown_1<-10$ where $\bigtriangledown_1<-10$ is the constraint induced by the condition predicate of P_5. $[S_5, C_5]$ can be simplified to $[S_5', C_5']$ according to the rewrite rules of Figure 1. $S_{x\rightarrow e_1}$ means that all occurrences of x in S are simultaneously substituted by e_1.

Note that if we replace a γ function by its equivalent first order logic formula, then we can also use our simplification techniques as described in [9] to achieve the same simplified result. We could also use systems written on top of Mathematica [15] or the Omega test [11] for simplifying first order logic expressions and constraints.

S_6 implies the same state as S_5' except for $X = -2 * \bigtriangledown_1$. At the confluence node after P_7 the value of X again can have two different values depending on whether or not the true-branch of P_5 has been taken. In order to prevent extensive bracketing in path conditions we assume – in the remainder of this paper

[1] A non-null path $v \rightarrow w$ implies that $v \neq w$

$$[S,\, C{=}C' \wedge \gamma(C'';\, \mathrm{x}{=}e_1;\, \mathrm{x}{=}e_2)] = \begin{cases} [S_{x\to e_1},\, C'_{x\to e_1}] & : \text{ if } C'{\Rightarrow}C'' \\ [S_{x\to e_2},\, C'_{x\to e_2}] & : \text{ if } C'{\Rightarrow}\neg C'' \\ [S,C] & : \textit{ otherwise} \end{cases}$$

Figure 1: Simplification of contexts with γ functions

– conventional precedences for arithmetic expressions which have higher precedence than relational operators (=,<,≤, etc.). Logical operators (∧ and ∨) have lowest precedence.

2.3 Loops and Recurrences

Modeling loops implies a slight problem with induction variables [1] under the constraints of first order logic. For instance, $i = i+1$ is a contradiction in first order logic. Note that "=" by definition is not an assignment but an equivalence. We will use functions instead to model recurrences as follows: $i(n+1) = i(n)+1$ where $i(n+1)$ is the value of a variable I at the end of iteration n+1. This is illustrated in the following example:

$[S_0{=}\{\mathrm{M}{=}m,\mathrm{F}{=}y,\mathrm{I}{=}\perp,\mathrm{B}{=}b\},\ C_0{=}b > 0 \wedge m > 0]$
P_1: do I=B,M,B
$[S_1{=}\delta(S_0;\mathrm{F}{=}\mathrm{f(n)},\mathrm{I}{=}\mathrm{i(n)}),$
$C_1{=}C_0 \wedge \mathrm{f(0)}{=}\mathrm{y} \wedge \mathrm{i(0)}{=}\mathrm{b} \wedge (\mathrm{n}{\geq}0 \wedge \mathrm{b}{\leq}\mathrm{i(n)}{\leq}\mathrm{m} \wedge$
$\mathrm{f(n+1)}{=}\mathrm{x} \wedge \mathrm{i(n+1)}{=}\mathrm{i(n)}{+}\mathrm{b})]$
P_2: F := 2*F
$[S_2{=}\delta(S_1;F = 2 * f(n)),C_2{=}C_1 \wedge \mathrm{x}{=}2{*}\mathrm{f(n)}]$
P_3: enddo
$[S_3{=}\delta(S_2;\mathrm{F}{=}\mathrm{f(z)},\mathrm{I}{=}\mathrm{i(z)}),$
$C_3{=}C_2 \wedge (\bigvee_{0 \leq k < z} (\mathrm{i(k)} \leq \mathrm{m}) \wedge \mathrm{i(z)}{>}\mathrm{m}) \vee (\mathrm{z}{=}0)] =$
$= [S_3'{=}\delta(S_2;\mathrm{F}{=}y * 2^{\lceil \frac{m+1}{b} \rceil - 1},\mathrm{I}{=}\mathrm{b}{*}\lceil \frac{m+1}{b} \rceil),C_3'{=}C_0]$

There are two variables – F and I – whose values change during execution of the loop. We introduce a function $f(n)$ and $i(n)$ for F and I respectively to model their values for a generic n-th iteration. The initial values for all variables are given by f(0)=y∧i(0)=b which is part of path condition C_1. Whereas the variable values at the beginning of the loop iteration are given by $n{\geq}0 \wedge$f(n+1)=x∧i(n+1)=i(n)+b. Note that x is a logic variable which will be bound to the value of F at the end of loop iteration $n+1$.

In order to automatically compute a closed form for variables F and I at the loop exit (after the *enddo*) we need to specify the corresponding system of recurrences for $n \geq 1$:

$$\begin{aligned} f(n) &= 2 * f(n-1) \\ i(n) &= i(n-1) + b \\ f(0) &= y \\ i(0) &= b \end{aligned}$$

which is obtained from $[S_2, C_2]$. Note that $f(n{+}1) = 2 * f(n)$ has been rewritten to $f(n) = 2 * f(n-1)$. Similar accounts for $i(n)$. We use a program written under Mathematica [15] to solve recurrence systems. For this purpose we implemented an interface to Mathematica via MathLink [15] which is described in detail in [12]. By using this system we can automatically compute a solution (loop invariant) to the previous system of recurrences for $n \geq 0$:

$$\begin{aligned} f(n) &= y * 2^n &(1)\\ i(n) &= b * (n+1) &(2) \end{aligned}$$

Based on (2) and $i(n) \leq m$ (loop condition) we can automatically determine the value of the loop variable at the exit of the loop which is equivalent to finding a z for which the following holds:

$$z = min(n \mid b * (n+1) > m) = \lceil \frac{m+1}{b} \rceil - 1 \qquad (3)$$

The ceiling operations in (3) are necessary as the loop condition is evaluated based on integer arithmetic. In order to compute the values of F and I at the loop exit we have to substitute z for n in (1) and (2):

$$\begin{aligned} F &= y * 2^{\lceil \frac{m+1}{b} \rceil - 1} \\ I &= b * \left\lceil \frac{m+1}{b} \right\rceil \end{aligned}$$

which specifies the variable binding of F and I at the loop exit $[S_3', C_3']$. As we can determine closed forms for the induction variables (I and F) of the loop, C_3' is equal to the path condition C_0 before the loop.

2.4 Arrays

In this section we will describe how to model array operations including indirect array accesses. In the case of indirect array accesses such as $W(A(I))$ where an index array A is used to access a specific element of array W, we restrict ourself to structural array accesses. This means that array indices and index arrays are symbolic expressions defined over integer constants and scalars. Currently we exclude data dependent arrays where array indices depend on input data.

In order to model array assignments we introduce a new function ξ_w:

$$A_2 = \xi_w(A_1, e_1, e_2)$$

where A_2 denotes the same array A_1 except that when applied to the index of e_1 it yields e_2. e_1 and e_2 are symbolic expressions which can also be index arrays. Array accesses are modeled similarly by a function $\xi_r(A, e)$ which yields the value of array A with index e.

Consider the following code segment, which consists of two loops. In the first loop, an array A is written. The array element values are functions over the loop index variable and a loop invariant. In the second loop, A is used as an index array for array W.

$[S_0{=}\{\mathrm{M}{=}m,\mathrm{I}{=}\perp,\mathrm{A}{=}v,\mathrm{W}{=}u\},C_0{=}2{\leq}\mathrm{m}]$
P_1: do I=1,M-1
$[S_1{=}\delta(S_0;\mathrm{A}{=}a(n),\mathrm{I}{=}\mathrm{i(n)}),C_1{=}C_0 \wedge a(0){=}v \wedge \mathrm{i(0)}{=}1 \wedge$
$(\mathrm{n}{\geq}0 \wedge 1{\leq}\mathrm{i(n)}{\leq}\mathrm{m}{-}1 \wedge \mathrm{i(n+1)}{=}\mathrm{i(n)}{+}1 \wedge a(n+1){=}\mathrm{x})]$

P_2: A(I):=M-I
[S_2=δ(S_1;A=ξ_w(a(n),i(n),m-i(n))),
C_2=$C_1 \wedge$ x=ξ_w(a(n),i(n),m-i(n))]
P_3: enddo
[S_3=δ(S_2;A=a(m-1),I=m), C_3=C_2]
P_4: do I=1,M-1
[S_4=δ(S_3;W=$w(n)$,I=i(n)), C_4=$C_3 \wedge w(0)=u \wedge$i(0)=1$\wedge$
(n$\geq$0$\wedge$1$\leq$i(n)$\leq$m-1$\wedge$i(n+1)=i(n)+1$\wedge$w(n+1)=y)]
P_5: W(A(I)):=I+1
[S_5=δ(S_4;W=ξ_w(w(n),ξ_r(a(m-1),i(n)),i(n)+1)),
C_5=$C_4 \wedge$y=ξ_w(w(n),ξ_r(a(m-1),i(n)),i(n)+1)]
P_6: enddo
[S_6=δ(S_5;W=w(m-1),I=m), C_6=C_5]

P_2 describes an array assigment *A(I) = M-I* which is described by $\xi_w(a(n), i(n), m - i(n))$. This means that A did not change due to P_2 except for the element with index value $i(n)$ which received a new value $m - i(n)$. $a(n)$ specifies the value binding of A at the end of the n-th iteration of the first loop. $a(0) = v$ specifies the value binding of A at $[S_0, C_0]$.

P_5 describes an indirect array access to W whose index expression is given by A(I). The associated value change is modeled by ξ_w(w(n),ξ_r(a(m-1),i(n)),i(n)+1) which describes the same array element values of W as those computed until the previous iteration except that at index $A(i(n))$ of W its value is $i(n) + 1$. $\xi_r(a(m-1), i(n))$ refers to the value of array A with index $i(n)$.

In order to determine the value of $W(k)$ based on context $[S_6, C_6]$ we proceed as follows:

$$eval(W(k), [S_6, C_6]) = \xi_r(w(m-1), k) \tag{4}$$

$$= \xi_r(\xi_w(w(n), \xi_r(a(m-1), i(n)), i(n)+1), k) \tag{5}$$

$$= \begin{cases} \xi_w(\text{w(n)},\xi_r(\text{a(m-1),i(n)}),\text{i(n)+1}) & : k \in \text{dom}(\xi_r(\text{a(m-1),i(n)})) \\ \xi_r(u,k) & : otherwise \end{cases} \tag{6}$$

where *dom(ξ_r(a(m-1),i(n)))={m-i(n) | 1≤i(n)≤m-1}* describes all possible values for $A(i(n))$ as written in the first loop. In order to examine the values for k the expression $\xi_r(a(m-1), i(n))$ must be further evaluated assuming *1 ≤ i(n) ≤ m-1* according to dom($\xi_r(a(m-1), i(n))$):

$$\begin{aligned} k &= \xi_r(a(m-1), i(n)) \\ &= \xi_r(\xi_w(a(n), i(n), m - i(n)), i(n)) \\ &= m - i(n) \\ &\Rightarrow i(n) = m - k \end{aligned}$$

This is based on the assumption that $i(n)$ is a strictly monotone function which can be automatically proven based on our techniques for comparing symbolic expressions as described in [9]. Substituting $i(n)$ by $m - k$ in (6) yields

$$= \begin{cases} \xi_w(\text{w(n)},\xi_r(\text{a(m-1),m-k}),\text{m-k+1}) & : k \in \text{dom}(\xi_r(\text{a(m-1),m-k})) \\ \xi_r(u,k) & : otherwise \end{cases} \tag{7}$$

$$= \begin{cases} \text{m-k+1} & : k \in \text{dom}(\xi_r(\text{a(m-1),m-k})) \\ \xi_r(u,k) & : otherwise \end{cases} \tag{8}$$

dom($\xi_r(a(m-1), m-k)$) = $\{m - i(n) | 1 \leq i(n) \leq m-1\}$ can be simplified to $\{m - (m-k) | 1 \leq m-k \leq m-1\} = \{k | 1 \leq k \leq m-1\}$. For this and other simplifications we developed a powerful algorithm [9] that simplifies symbolic expressions and constraints.

2.5 Procedures

In order to model procedure calls, we first symbolically evaluate the procedures itself by using the techniques described in the previous sections. Generic variable bindings are assumed for all variables used at the procedure entry. At the exit of the procedure we determine the variable values as functions of the variable values at the entry. For the sake of simplicity we assume that each procedure has a single entry and exit which are respectively associated with an entry and exit context. The actual implementation considers multiple procedure entries and exits.

A variable that is passed to a procedure Q is an *input variable* for Q if its value remains unchanged for all possible control flow paths between the procedure's entry and exit. In all other cases a variable is an *output variable* for Q. We assume a procedure model that corresponds to Fortran where procedures are named *subroutines* and all procedure parameters are passed by call-by-reference. Call-by-name procedure parameters – as occurring in many other imperative programming languages – as well as procedure local variables can be easily modeled by declaring them as *input variables*. The values of local variables are set to $\perp$ at the procedure entry. Currently we do not model recursive procedure calls. For instance, in the following procedure *ADIFF*

P_1: subroutine ADIFF(A,B,S)
[S_1={A=a,B=b,S=s},C_1=true]
P_2: if (A $\geq$ B) then
[S_2=S_1,C_2=a$\geq$b]
P_3: S:=A-B
[S_3=δ(S_1;S=a-b),C_3=C_2]
P_4: else
[S_4=S_1,C_4= a<b]
P_5: S:=B-A
[S_5=δ(S_1;S=b-a),C_5=C_4]
P_6: endif
[S_6=δ(S_1;S=u), C_6=γ(a$\geq$b;u=a-b;u=b-a)]
P_7: end

$[S_1, C_1]$ and $[S_6, C_6]$ is respectively the entry and exit context of procedure *ADIFF*. At the time where P_1 is evaluated nothing may be known about the variable values of the procedure parameters, therefore, generic values are assumed. Similar accounts for C_1 which is set to *true*. By comparing the variable bindings of S_1 with that of S_6 it can be easily determined that A and B are input variables whereas S is an output variable.

Let R be an arbitrary procedure call statement that calls a procedure Q. [S_b,C_b] denotes the program context immediately preceding R which is used to specialize and simplify the exit context of Q. The exit context of Q implies [S_a,C_a], the context immediately following R. Note that using this strategy requires a single evaluation of each individual procedure. Only the exit context of each procedure is then re-evaluated for specialization of a specific call to this procedure. This is in contrast to a strategy that would require full evaluation of the entire procedure for each call to it. The following code segment contains a call to procedure *ADIFF*:

[S'_0={X=m,Y=1,Z=$\perp$,I=$\perp$,J=$\perp$,M=m,W=w,...},
C'_0=m$\geq$1]
P'_1: call ADIFF(X,Y,Z)

```
        [S'_1={X=m,Y=1,Z=u,I=⊥,...},
        C'_1=m≥1∧γ(m≥1;u=m − 1;u=1 − m)] =
        =[S''_1=δ(S'_1;Z=m-1), C''_1=m≥1]
P'_2:  do I=1,M
P'_3:     W(M) := ...
P'_4:     do J=1,M
          [S'_4={...,J=j(n),...},C'_4=m≥1∧...]
P'_5:        W(M+J) := ...
          [S'_5={...,Z=m − 1,J=j(n),M=m,...},C'_5=m≥1∧...]
P'_6         ... := W(Z+J)
          [S'_6={...,Z=m − 1,J=j(n),M=m,...}, C'_6=m≥1∧...]
P'_7:     enddo
P'_8:  enddo
```

We refer to A, B, and S in *subroutine ADIFF(A,B,S)* as the *formal procedure variables*. Whereas X, Y, and Z in *call ADIFF(X,Y,Z)* are denoted the *actual procedure parameters*. Note that actual procedure parameters in principle can be arbitrary expressions whereas formal procedure variables are always variables. In order to compute $[S'_1, C'_1]$ the context after the call to procedure *ADIFF* in P'_1 we have to slightly modify our definition for *input variables*.

A formal procedure variable is an input variable if its value remains unchanged for all possible control flow paths between the procedure's entry and exit, or if the associated actual procedure parameter is not a variable. In all other cases a formal procedure variable is an output variable.

If an actual procedure parameter is not a variable, then it is either a constant or a more complex expression such as $X+Y$ which cannot be assigned a new value after return of the procedure call in our program model.

Let us assume that $[S_b, C_b]$ and $[S_a, C_a]$ are respectively the contexts before and after a call statement R to a procedure Q. In order to determine $[S_a, C_a]$ based on $[S_b, C_b]$ and the exit context of Q, we proceed as follows:

1. Copy $[S_b, C_b]$ to $[S_a, C_a]$ and the exit context of Q to an auxiliary context $[S_e, C_e]$.
2. Each value of a formal procedure variable that appears in $[S_e, C_e]$ is substituted by the associated actual procedure parameter value which is obtained from $[S_b, C_b]$.

   In this step the actual procedure parameter values are propagated into the procedure (exit). In other words, the exit context of Q is specialized by using the actual procedure parameter values.
3. Delete all local and input variables of Q from $[S_e, C_e]$. After this step only output variables remain in $[S_e, C_e]$.
4. Substitute the value of each variable in $[S_e, C_e]$ for the associated variable (actual procedure parameter) value in $[S_a, C_a]$. In this step the output variable values of Q are returned to $[S_a, C_a]$.
5. Simplify $[S_a, C_a]$ by using our simplification techniques including those described in [9].

Note that the symbolic value m (see state S'_0) is propagated into procedure *ADIFF* through the actual procedure parameter X. Furthermore, $[S'_1, C'_1]$ in the previous example can be rewritten as $[S''_1, C''_1]$ according to the rewrite rules of Figure 1.

Parallelization of the previous code example is prevented due to memory-related dependences. We will demonstrate the usefulness of our analysis for array privatization [14] which is an important program optimization that tries to eliminate such memory-related dependences. For the sake of brevity, only the most important contexts are shown. The outermost loop in the example code can be parallelized if a private copy of array W is allocated to each iteration of this loop. Array privatization is save if in all iterations of the outermost loop a read access to an element of W must be preceded by a write access to the same element, which holds for the previous code. In order to prove that the array section read in P'_6 is completely covered by the array section written in P'_3 and P'_5, we determine the index expression value of $W(M+J)$ (write access) at P'_5 under $[S'_5, C'_5]$:

$$eval(M+J, [S'_5, C'_5]) = m + j(n)$$

as well as the index expression value of $W(Z+J)$ (read access) at P'_6 under $[S'_6, C'_6]$:

$$eval(Z+J, [S'_6, C'_6]) = m - 1 + j(n)$$

which demonstrates that each array element is written before read within every iteration of the outer loop. Furthermore, in the first iteration of the outer loop P'_6 reads $W(M)$ which is written by P'_3. Therefore, by using our symbolic evaluation we detect that privatizing W is safe for the previous example which enables parallelization of this code.

3 Experiments

We are currently implementing the techniques described in this paper as part of *VFCS* (Vienna Fortran Compilation System [3]) and P^3T (a performance estimator for parallel programs [8]). The interface to Mathematica for solving systems of recurrences [12] as well as the simplifier [9] for linear and non-linear systems of constraints are already implemented.

Nevertheless, in the following we show an experiment to demonstrate the potential benefits of the application of symbolic evaluation techniques for improving the performance of programs on distributed-memory machines.

3.1 Symbolic Evaluation to Optimize OLDA

In this experiment we use OLDA which is the dominant routine in the TRFD program of the Perfect Benchmarks [7] to demonstrate more general improvements obtained by using symbolic evaluation. Among others the OLDA code contains conditional assignments and GOTOs, recurrences, non-linear loop bounds and array index expressions. Table 1 displays the execution times as obtained on a Meiko CS-2 for various problem (N) and machine sizes (number of processors $Proc$) for two different parallelization strategies. First, optimizing and parallelizing OLDA by using the current VFCS without symbolic evaluation (column 3). Standard optimizations under VFCS include communication vectorization, fusion, and aggregation.

Second, the codes of column 3 were hand-instrumented to reflect the optimizations that would be performed using symbolic evaluation as described in this paper (column 4). This includes resolving symbolic systems of recurrences, symbolic communication optimization, and symbolic dependence analysis.

		Execution Time in Seconds		
N	*Proc*	VFCS	VFCS	Improvement
		(w/o SE)	(with SE)	(in %)
4	4	0.165	0.145	13.8
	8	0.141	0.118	19.5
	16	0.127	0.101	25.7
	32	0.122	0.096	27.1
8	4	0.961	0.889	8.1
	8	0.801	0.711	12.6
	16	0.703	0.596	18.0
	32	0.654	0.546	19.8

Table 1: Meiko CS-2 execution times for OLDA

We ran our tests on 4, 8, 16 and 32 processors on a Meiko CS-2 for two different problem sizes $N = 4$ and $N = 8$.

Optimizations made possible by the improved symbolic evaluation reduced the total execution time by up to 27 % compared to the programs compiled with the current VFCS. By far the largest portion of the performance improvement obtained by our symbolic evaluation is caused by a reduced communication overhead. Resolving recurrences (including induction variables with closed forms) eliminated dependences which in turn enabled message vectorization. The ability of comparing non-linear loop bounds and index expressions that contain symbolics allowed us to further eliminate redundant communication which is not done by the current VFCS. Symbolic evaluation reduces only a small portion of OLDA's computation time. OLDA's computational overhead is of the order $\Theta(N^4)$, and the communication overhead is of the order $\Theta(N^2)$. Therefore, by increasing the problem size N, the communication takes a smaller portion of the total execution time, and the influence of our symbolic evaluation becomes less apparent. Similarly, when increasing the number of processors for a fixed problem size the percentage of reducing execution time increases which is caused by an increased communication overhead whereas the computation remains unchanged.

4 Related Work and Discussion

P. and R. Cousot [5] pioneered abstract interpretation as a theory of semantics approximation for systematic data and control flow analysis of sequential programs. Blume and Eigenmann [4] use abstract interpretation to extract constraints about variable ranges. M. Haghighat [10] described a variety of symbolic analysis techniques based on abstract interpretation. In order to reduce the amount of information to be maintained for each program statement Haghighat intersects the information of all incoming program paths to that statement.

R. Cytron et al. [6] developed a new program representation called static single-assignment (SSA) form where each variable is written only once. R. Ballance et al. [2] extended the SSA form to what is called gated SSA (G-SSA) form. Special gating functions are introduced to include predicates of conditional branches. Extensive renaming by SSA and G-SSA form destroys the vital relationship between problem sizes (program input data, number of processors, array sizes, etc.) and the resulting analysis information. Our analysis information representation does not change the code and can be optionally displayed together with the code. Consider the code of Figure 2 where the value of B is computed as a function of the input value of A. G-SSA form (Figure 3) requires explicit analysis to determine that the value of B depends on the input value of the read statement. Whereas the code based on symbolic evaluation (Figure 4) directly presents the value of B as a function of the input value $\bigtriangledown$ without additional analysis and code change. This is also the reason why G-SSA and SSA form seems to be only of limited use for performance oriented compiler optimizations and performance analysis where problem sizes must be highlighted (transparent) in the resulting analysis information. For instance, it is very desirable to provide performance information as parameterized functions over number of processors, array sizes, and input data. These parameterized functions can then be used to compute the optimal number of processors and array sizes to be used for a specific parallel machine. Note that our symbolic evaluation has a strong potential for sparse data representation. For instance, analysis information for loops and procedures can be stored only at their entry and exit points.

P. Tu and D. Padua [14] developed a system for computing symbolic values of expressions using a demand-driven backward analysis based on G-SSA form. Their method can be faster and less memory intensive than our approach if local analysis information suffices to obtain a result. However, if local information is not sufficient, they may have to examine large portions of a program. The application of our analysis can be limited to specific portions of a program (loops, procedures, etc.). Furthermore, if the value of a specific variable should be computed at a specific statement, we can use conventional USE/DEF information to reduce the list of variables to be maintained in a program context. For recurrences Tu and Padua cannot directly determine the corresponding system of recurrences from a given G-SSA form. Instead they need to apply a sequence of substitution rules to G-SSA form, interpret the result as a λ function, rewrite the λ function in the terms of recursive sequence in combinatorial mathematics, before they finally use standard techniques to resolve the system of recurrences. Our approach enables to directly extract the system of recurrences from our data representation without additional analysis. Finally, their approach avoids modeling of procedures by full in-line expansion.

5 Conclusion

This paper introduces symbolic evaluation to support parallelizing compilers in optimizing parallel codes. Our techniques compute values of variables and entire expressions (both linear and non-linear) as algebraic functions defined over the input data at any given program point. The condition under which the control flow reaches a program point is described by a first

read(A)
A=A+1
B=A+2

Figure 2: Original code

read(A_1)
A_2=A_1+1
B_1=A_2+2

Figure 3: SSA or G-SSA form

read(A)
[S_1={A=∇,B=b},C_1=true]
A=A+1
[S_2=δ(S_1;A=∇+1),C_2=C_1]
B=A+2
[S_3=δ(S_2;B=∇+2),C_3=C_1]

Figure 4: Symbolic evaluation form

order logic formula. Our approach has several advantages over much of previous work. Procedures and indirect array accesses are modeled. Large classes of linear and non-linear expressions can be processed. A compact representation of analysis information maintains the important relationship between problem sizes and analysis results. The underlying code is not changed due to our analysis which maintains readability. We can directly obtain the associated system of recurrences from our data representation without additional analysis. Our symbolic evaluation approach can be optimized by restricting our analysis to specific code sections and/or employ USE/DEF information (provided by the underlying compiler) in order to reduce analysis information. Symbolic evaluation has a strong potential for sparse data representation, for instance, by storing analysis information only for entry and exit points of loops and procedures.

Our preliminary experience indicates that optimizations (including communication vectorization, and elimination of redundant communication) based on our symbolic evaluation techniques can result in significant performance improvements.

We are currently in the process of implementing our symbolic evaluation framework and will use it as part of the Vienna Fortran Compilation System and P^3T (a performance estimator) to parallelize and optimize programs for distributed memory architectures.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools.* Series in Computer Science. Addison Wesley, 1988.

[2] R. Ballance, A. Maccabe, and K. Ottenstein. The program Dependence Web: a Representation Supporting Control-, Data-, and Demand Driven Interpretation of Imperative Languages . In *Proceedings of the SIGPLAN 90 Conference on Program Language Design and Implementation*, pages 257–271, White Plains, New York, June 1990.

[3] S. Benkner et al. *Vienna Fortran Compilation System - Version 2.0 - User's Guide*, October 1995.

[4] W. Blume and R. Eigenmann. Demand-driven, Symbolic Range Propagation. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing, Columbus, OH*, August 1995.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages pp. 238 – 252, Los Angeles, CA, January 1977.

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

[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. Symbolic Expression Evaluation to Support Parallelizing Compilers. In *IEEE Proc. of the 5th Euromicro Workshop on Parallel and Distributed Processing, London, UK*, pages 22–24, January 1997.

[10] Mohammad R. Haghighat. *Symbolic Analysis for Parallelizing Compilers.* Kluwer Academic Publishers, 1995.

[11] William Pugh. The Omega Test: a Fast and Practical Integer Programming Algorithm for dependence Analysis . *Communications of the ACM*, 8:102–114, August 1992.

[12] 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.

[13] 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.

[14] 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.

[15] S. Wolfram. *Mathematica - A System for Doing Mathematics by Computer.* The Advanced Book Program. Addison Wesley, 1991.