A Unified Symbolic Evaluation Framework for Parallelizing Compilers *

T. Fahringer[†] B. Scholz[*]

[†]Institute for Software Science, University of Vienna

Liechtensteinstr. 22, A-1090, Vienna, Austria

tf@par.univie.ac.at

[*]Institute of Computer Languages, Vienna University of Technology

Argentinierstr. 8/4, A-1040, Vienna, Austria

scholz@complang.tuwien.ac.at

Abstract

The quality of many optimizations and analyses of parallelizing compilers depends significantly on the ability to evaluate symbolic expressions and on the amount of information available about program variables at arbitrary program points. In this paper, we describe an effective and unified symbolic evaluation framework that statically determines the values of variables and symbolic expressions, assumptions about and constraints between variable values and the condition under which control flow reaches a program statement. We introduce *program context*, a novel representation for comprehensive and compact control and data flow analysis information. Program contexts are described as first order logic formulas which enable us to use public domain software for standard symbolic manipulation. Computations are represented as algebraic expressions defined over a program's problem size. Our symbolic evaluation techniques comprise accurate modeling of assignment and input/output statements, branches, loops, recurrences, arrays and procedures. Efficiency is highly improved by aggressive simplification techniques. All of our techniques target both linear as well as non-linear expressions and constraints. A variety of examples, including program verification, dependence analysis, array privatization, communication vectorization and elimination of redundant communication is used to illustrate the effectiveness of our approach. We present results from a preliminary implementation of our framework which is used as part of a parallelizing compiler that demonstrate the potential performance gains achievable by employing symbolic evaluation to support program parallelization.

Index Terms: Symbolic analysis, symbolic evaluation, program context, data-flow and control-flow analysis, symbolic dependence testing, compiler optimizations, parallelizing compilers.

* This research is partially supported by the Austrian Science Fund as part of Aurora Project under contract SFBF1104.

1 Introduction

It is widely accepted [33, 53, 54, 9, 10, 52, 43] that current parallelizing compilers are not very effective in optimizing parallel programs that fully utilize the target multiprocessor system. The poor performance of many compiler analyses can be attributed to ineffective parallelization of programs (for instance, High Performance Fortran *HPF* [38] and Fortran 90 [44]) that have a strong potential for unknowns such as number of processors and sizes of allocateable arrays. Some compilers have a problem to make the relationship between a program's problem size (input data, array and machine sizes, etc.) and analysis information explicit. This relationship is crucial for performance driven optimization. Some others have poor capabilities to analyze dependences between references with linearized subscripts and non-linear terms in subscript expressions. Non-linear terms can stem from recurrences or tiling with symbolic block sizes [5]. Commonly, compilers do not employ interprocedural analysis at the cost of analysis accuracy [36]. Ineffective approaches to gather and propagate sufficient information about variables through the program continues to have a detrimental impact on many compiler analyses and optimizations [10, 53, 54]. As a consequence worst case assumptions are frequently made or program analysis is done at runtime which increases the execution overhead. Therefore, sophisticated symbolic analysis that can cope with program unknowns is needed to alleviate these compiler deficiencies.

Recent developments in parallelizing compilers have resulted in an increased use of symbolic analysis to support the detection of parallelism and optimization of programs. Parafrase-2 [45, 33] and Nascent [30] introduced symbolic analysis to recognize and compute closed forms of induction variables. Parafrase-2 also applies symbolic analysis for dependence testing, eliminating dead-code, and scheduling of parallel loops. The Parascope project [39] included interprocedural symbolic analysis. In the Polaris restructuring compiler [13] symbolic analysis supports range propagation (lower/upper bounds of symbolic expressions), array privatization and dependence testing. The Paradigm compiler [52] incorporates a commercial symbolic manipulation package to improve parallelization of programs for distributed memory architectures. The SUIF compiler [35] employs linear symbolic analysis for scalar, array and interprocedural analysis.

Previous approaches on symbolic compiler analysis frequently have several drawbacks associated with them:

- restricted program models (commonly exclude procedures, complex branching and arrays)
- analysis that covers only linear symbolic expressions
- insufficient simplification techniques
- memory and execution time intensive algorithms
- unstructured, redundant and inaccurate analysis information, and complex extracting of analysis information
- additional analysis is required to make the important relationship between problem size and analysis result explicit
- recurrences are frequently not supported at all or separate analysis and data structures are required to extract, represent, and resolve recurrence systems.

Furthermore, in order to build efficient symbolic analyses which can effectively support parallelization of programs, there is a need to express values of variables and expressions as well as the (path) condition under which control flow reaches a program statement. Some approaches [36, 34, 15] do not consider path conditions for their analysis. Without path conditions the analysis accuracy may be substantially reduced. More recent work [54] is based on an existing representation of analysis information. For instance, G-SSA form [3] commonly requires additional algorithms to determine variable values and path conditions. G-SSA form does not represent this information in a unified data structure. We are not aware of an approach that combines all important information about variable values, constraints between them, and path conditions in a unified and compact data representation. Moreover, it has been realized [52] that systems with interfaces for off-the-shelf software are critical for future research tool and compiler development. Approaches that rely on specialized representations for their analysis are commonly forced to re-implement standard symbolic manipulation techniques which otherwise could be taken from readily available software packages.

In order to overcome or at least alleviate many of the deficiencies mentioned above, we present a novel and unified compile time framework. It is based on symbolic evaluation and combines both data and control flow analysis. In the center of our framework is a new representation of analysis information, the *program context*, which includes the following three components:

- variable values
- assumptions about and constraints between variable values, and
- conditions under which control flow reaches a program statement.

A main advantage of our approach is that every component of program contexts can be separately accessed at well-defined program points without additional analysis. We describe an algorithm that can generate all program contexts by a single traversal of the input program. Program contexts are specified as first order logic formulas which is a general representation and enables us to use off-the-shelf software for standard symbolic manipulation techniques. Computations are represented as symbolic expressions defined over the program's problem size. By avoiding renaming of data objects our symbolic evaluation is more amenable to maintain the critical relationship between a program's problem size and resulting analysis information. This relationship is important for performance-driven program optimization. Our symbolic evaluation framework accurately models assignment and input/output statements, branches, loops, recurrences, arrays (including indirect accesses) and procedures. Recurrences are detected, closed forms are computed where possible and the result can be directly retrieved from well-defined program points. Detecting recurrences and finding closed forms for recurrences are decoupled which simplifies extending our recurrence solver (finds closed forms based on recurrence systems). All of our techniques target both linear as well as non-linear symbolic expressions and constraints.

We intensively manipulate and simplify symbolic expressions and constraints based on a system which we have built on top of existing software [47, 55]. We will describe a variety of new techniques to simplify program contexts. Furthermore, we have developed several novel algorithms [27, 28, 26] for comparing symbolic expressions, computing lower and upper bounds of symbolic expressions, counting the number of solutions to a system of constraints, and simplifying systems of constraints. While previous work mostly concentrated on symbolic analysis for shared memory architectures this research also supports symbolic compiler analysis for distributed memory architectures covering symbolic dependence testing, array privatizing, message vectorization and coalescing, communication aggregation and latency hiding.

We do not know of any other system that models a similar large class of program constructs based on a comprehensive and unified analysis representation (program context) that explicitly captures exact information about variable values, assumptions about and constraints between variable values, path conditions, recurrence systems, and side-effect information of procedure calls. Although we have examined our framework for Fortran programs, the underlying techniques are equally applicable to any similar imperative programming language.

Empirical results based on a variety of benchmark codes demonstrate among others the need for advanced analysis to handle non-linear and indirect array references, procedure calls and multiple exit loops. Furthermore, we will show the effectiveness of our approach for a variety of examples including program verification, dependence analysis, array privatization, communication vectorization and elimination of redundant communication.

We have implemented a prototype of our symbolic evaluation framework which is used as part of the Vienna High Performance Fortran Compiler (VFC) [5] – a parallelizing compiler for distributed memory architectures – and P^3T [24, 25] – a performance estimator – to parallelize and optimize High Performance Fortran programs [38, 5] for distributed memory architectures.

The organization of this paper is as follows. Preliminaries are presented in Section 2. In Section 3, we describe our symbolic evaluation framework. This includes techniques for computing path conditions and values of symbolic expressions. We will further illustrate how to model assignment and input/output statements, branches, loops, recurrences, arrays and procedures. In Section 4 we introduce an algorithm for automatically generating program contexts for every program statement as used by symbolic evaluation. Preliminary results of a prototype implementation are shown in Section 5 which demonstrate the benefits of the proposed method. Related work will be discussed in Section 6. Section 7 will give our conclusions.

2 Preliminaries

2.1 Program Model

A control flow graph (CFG) of a program is defined by a directed flow graph $G = (N, E, x, y)$ with a set of nodes N and a set of edges E. A node $n \in N$ represents a program instruction (statement), an edge $(m, n) \in E$ indicates transfer of control between instructions $m, n \in N$. x and y are respectively the unique *start* and *end* node of G, which are assumed not to possess any predecessors and successors. $succs(n)$ and $preds(n)$ correspond to the set of successor and predecessor nodes of n. A path in G is a sequence of nodes $[n_1, ..., n_k]$, where $\forall 1 \leq i < k$ the following holds: $n_{i+1} \in succs(n_i)$. Every node $n \in N$ is assumed to lie on a path from x to y.

Our loop model is based on natural loops that are detected by using dominator information [2]. We consider only reducible control flow graphs and loops. We assume that there are no jumps into the loop body from outside.

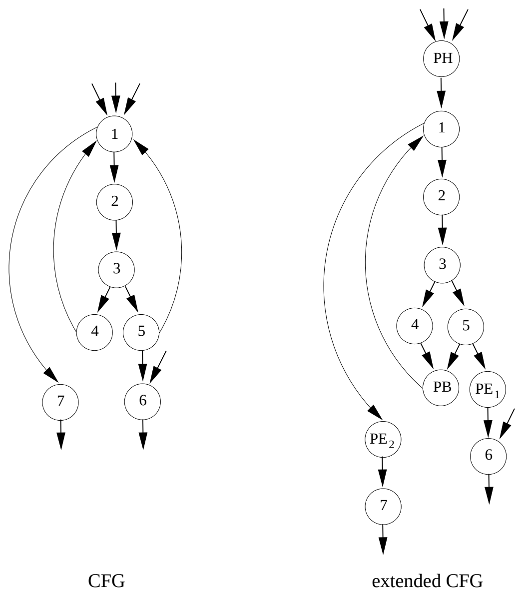

Figure 1: CFG and extended CFG

Each loop has a unique *header* which dominates all nodes in the loop body. Note that a loop may have several exits. We use an extended CFG similar to that proposed in [22, 30, 36] by inserting several artificial nodes without side-effects in the original CFG. This includes preheader, postbody and postexit nodes (see Figure 1). A *preheader* node, *PH*, is inserted such that it has only the header as successor and all edges which formerly entered the header from outside the loop instead enter the preheader. A *postbody* node, *PB*, is inserted such that it provides a target for all loop back edges. An edge is inserted from the preheader to the header and another edge from the postbody to the header. For each edge exiting the loop, a *postexit* node, *PE*, is inserted outside the loop, between the source (loop exit node within the loop body) of the exit edge and the target (outside the loop). These changes cause threefold growth in a CFG at worst. In the remainder of this paper a CFG refers to the extended CFG if not mentioned otherwise. We exclude alias analysis, as implied for instance by pointers and EQUIVALENT statements in Fortran 90 which goes beyond the scope of this work.

2.2 Manipulating and Simplifying Symbolic Expressions and Constraints

In order to provide a useful framework for manipulating symbolic expressions and constraints, we have implemented an interface [46] to *Mathematica*^{TM}[1] via *MathLink*^{TM} [55]. Our framework has been developed primarily based on Mathematica's support for pattern matching and transforming rules to rewrite expressions. Note that Mathematica does not provide explicit support for manipulating integer-valued symbolic expressions. We have implemented our own rules for manipulating such expressions on top of Mathematica. Furthermore, we have developed novel algorithms for computing the number of solutions to a system of constraints [26], simplifying a system of constraints

[1] Mathematica and MathLink are registered trademarks of Wolfram Research, Inc.

[27], determining the relationship between symbolic expressions and computing lower and upper bounds of symbolic expressions [28]. Our simplification techniques detect tautologies, contradictions and redundant constraints. If there are equalities in a given set of constraints, then they are solved for a specific variable included in the equality and substituted in the set of constraints. All techniques target wide classes of linear and non-linear symbolic expressions and constraints. For detailed information about our manipulation and simplification techniques, which goes beyond the scope of this paper, the reader may refer to [26, 27, 28, 46]. Furthermore, we have also integrated our framework with the Omega library [47] developed by Prof. W. Pugh at the University of Maryland, for simplifying first order logic expressions and constraints.

3 Symbolic Evaluation

Symbolic evaluation statically analyses a program at compile time. Every program statement is associated with a *context* C that describes the variable values, assumptions regarding and constraints between variable values and the condition under which the program statement is reached. A context C is defined by a triple $[S, T, P]$ which includes a state S, a state condition T and a path condition P.

- State S:

  The *state* S is described by a set of variable/value pairs $\{V_1 = e_1, \ldots, V_n = e_n\}$ where V_i is a program (scalar or array) variable and e_i a symbolic expression 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.

- State condition T:

  In traditional program execution without symbolic expressions each time a branch statement is reached the current state (variable binding) unequivocally enables the processor to determine which branch to follow. In the presence of symbolic expressions, however, the information in a state may not suffice to determine which branch to follow. Hence, some assumptions about variable values must be taken in order to symbolically evaluate branch nodes. These assumptions about variable values are described by a *state condition* T which is represented by a first order logic formula. Additional constraints on variable values such as those implied by loops (recurrences), variable declarations and user assertions (specifying relationships between variable values) are also added to the state condition.

- Path condition P:

  The path condition P describes the condition under which control flow reaches a given program statement. P is specified by a first order logic formula that comprises the conditional expressions of branches that are taken to reach the program statement.

Note that all components of a context – including state information – are represented as first order logic formulas. A sequence of statements or assertions Q_j $(1 \leq \mathrm{j} \leq \mathrm{r})$ is symbolically evaluated by a sequence $[S_0, T_0, P_0]$ Q_1 $[S_1, T_1, P_1]$ $\ldots Q_r$ $[S_r, T_r, P_r]$. $[S_0, T_0, P_0]$ represents the context before Q_1 and $[S_r, T_r, P_r]$ the context after Q_r. If Q_i in $\ldots[S_i, T_i, P_i]\ Q_i\ [S_{i+1}, T_{i+1}, P_{i+1}]\ldots$ does not contain side effects (implying 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. In the following, we describe how symbolic evaluation is used to model various program constructs including assignments, input/output, branches, loops, recurrences, arrays and procedures.

3.1 Initializations, Assignments and Input/Output Operations

The following code excerpt includes assignment and input/output statements. For every statement Q_i the corresponding context $[S_i, T_i, P_i]$ is displayed.

```
      integer D1,D2,A,B
            [S0={D1=⊥,D2=⊥,A=⊥,B=⊥},T0=true,P0=true]
Q1:  read(A,B)
            [S1 = δ(S0;A = ▽1,B = ▽2), T1=T0,P1=P0]
Q2:  D1 := (A-B)*A
            [S2=δ(S1;D1 = (▽1 - ▽2)*▽1), T2=T1, P2=P1]
Q3:  D2 := 2*(A-B)
            [S3=δ(S2;D2 = 2*(▽1 - ▽2)), T3=T2,P3=P2]
Q4:  read(B)
            [S4=δ(S3;B = ▽3), T4=T3, P4=P3]
Q5:  assert(D1/D2 > A/2 - 1)
            [S5=S4,T5=T4,P5=P4]
```

A statement in "[...]" denotes that this statement is not an executable statement but serves only for the purpose of illustrating the context of a given program statement.

At the beginning of the program, in principle all variable values are undefined ($\perp$). More elaborate initializations are conducted for constants or explicit initializations of variables. For instance, Fortran 90 contains named constants, of the form "*real, parameter :: A = 2.997*" which cannot be overwritten and initialization expressions such as "*real :: A = 0.0*" that assign an initial value to a variable in a type declaration statement which can be overwritten. In both cases the initial variable value is given by the value of the constant or the initialization expression. Additional constraints can be added to the state condition that are implied by type information (KIND function in Fortran 90) or provided by the user. The state and path conditions T_0 and P_0 are set to *true* which is the default case. The initialization of procedures is explained in Section 3.5.

In program statement Q_1 variables A, B are assigned an unknown value, which is specified by a special symbol ($\bigtriangledown$) for input data values as these are of special interest. Input data values are propagated through the program thereby maintaining the important relationship of variable values and input data. Similar is done for other problem sizes such as machine and array sizes.

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 whose value changes after evaluation of the associated statement are explicitly specified. 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 variable V_i $(1 \leq i \leq l)$. Variable V_i is assigned a new value e_i. 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 $\bigtriangledown_1$ and $\bigtriangledown_2$ in state S_1.

Q_2 and Q_3 change only variables D_1 and D_2, respectively. Q_4 is again a read statement that assigns a new input data value $\bigtriangledown_3$ to B. In order to verify whether the assert statement Q_5 is true at this statement we have to evaluate the assert expression with respect to the program context immediately preceding Q_5 which is $[S_4, T_4, P_4]$. We use a function

$$eval(e, [S, T, P])$$

that determines the value of an algebraic expression e under the constraints of program context $[S, T, P]$. Function *eval* is implemented based on the symbolic manipulation system described in Section 2.2. For the previous example

$$\begin{aligned} & eval(\frac{D1}{D2} > \frac{A}{2} - 1, [S_4, T_4, P_4]) = \\ = \quad & \frac{(\bigtriangledown_1 - \bigtriangledown_2) * \bigtriangledown_1}{2 * (\bigtriangledown_1 - \bigtriangledown_2)} > \frac{\bigtriangledown_1}{2} - 1 \\ = \quad & true \end{aligned}$$

The assert statement is true at statement Q_5 under the assumption that $\bigtriangledown_1 - \bigtriangledown_2$ never evaluates to 0 which would make the division operation in Q_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 SSA [21] combined with forward substitution can also determine that the assert statement is true at Q_5. Whereas conventional scalar forward substitution [57] of $D1$ and $D2$ cannot be used to determine that the assert statement is true because the definitions in Q_2 and Q_3 are killed by the read statement Q_4.

3.2 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. The impact of the control flow on the variable values is described by the state condition, whereas the condition under which the control flow reaches a program statement is given by the path condition. Both state and path conditions are specified as a first order logic formula. If a conditional statement is encountered the conditional expression is evaluated by using the variable bindings of the current context. If it cannot be statically determined whether for all input data 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

Q_1: read(Y)
  $[S_1 = \{X = x_1, Y = \bigtriangledown_1\}, T_1 = \text{true}, P_1 = \text{true}]$
Q_2: if (Y < 0) then
  $[S_2 = S_1, T_2 = T_1, P_2 = (P_1 \wedge \bigtriangledown_1 < 0) = \bigtriangledown_1 < 0]$
Q_3: $X := -2 * Y$
  $[S_3 = \delta(S_2; X = -2 * \bigtriangledown_1), T_3 = T_2, P_3 = P_2]$
Q_4: $Y := -X$
  $[S_4 = \delta(S_3; Y = 2 * \bigtriangledown_1), T_4 = T_3, P_4 = P_3]$

Q_5: else

$[S_5{=}S_1, T_5{=}T_1, P_5{=}(P_1 \wedge \nabla_1 \ \geq \ 0){=}\nabla_1 \ \geq \ 0]$

Q_6: $X{:=}2 * Y$

$[S_6{=}\delta(S_5; X \ = \ 2 * \nabla_1), T_6{=}T_5, P_6{=}P_5]$

Q_7: endif

$[S_7{=}\delta(S_1; \mathrm{X}{=}x_2, \mathrm{Y}{=}y_1), T_7{=}\gamma(\nabla_1{<}0; x_2{=}{-}2 * \nabla_1, \ y_1{=}2 * \nabla_1; \ x_2{=}2 * \nabla_1, y_1{=}\nabla_1), P_7{=}(P_4 \vee P_6){=}\mathrm{true}\ ]$

Q_8: if (Y $\geq$ 0) then

$[S_8{=}S_7, T_8{=}T_7, P_8{=}P_7 \wedge y_1 \ \geq 0] = [S_8'{=}\delta(S_7; \mathrm{Y}{=}\nabla_1), T_8'{=}\gamma(\nabla_1{<}0; x_2{=}{-}2 * \nabla_1; \ x_2{=}2 * \nabla_1), \ P_8'{=}\nabla_1{\geq}0]$

Q_9: $Y{:=}2 * Y$

$[S_9{=}\delta(S_8'; Y \ = \ 2 * \nabla_1), T_9{=}T_8', P_9{=}P_8']$

Q_{10}:endif

$[S_{10}{=}\delta(S_7; Y{=}2 * \nabla_1), \ T_{10} \ {=}\gamma(\nabla_1{<}0; \ x_2{=}{-}2 * \nabla_1; \ x_2{=}2 * \nabla_1), P_{10}{=}\mathrm{true}\ ]$

we assume that at the beginning of this code segment, X has the value x_1 and the state and path conditions T and P are *true*. In Q_2 a conditional expression 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 $\nabla_1 < 0$ where ∇_1 is the current value of Y. The corresponding state condition remains unchanged as the variable values are unequivocal. Similar accounts for all statements in the else-part whose path condition includes $\nabla_1 \geq 0$.

A confluence of two branches occurs after statement Q_7. Two different contexts $[S_4, T_4, P_4]$ and $[S_6, T_6, P_6]$ reach this point. In [17] a folding operator $\odot$ for collapsing analysis information at confluence nodes has been described as part of symbolic execution [40]. We present a modified version of this operator for folding contexts which takes as arguments two program contexts and produces a new program context resulting from the composition of its arguments. Given two contexts $[S', T', P']$ and $[S'', T'', P'']$ where

$$\begin{aligned} S' &= \{V_1 = e_1, \ldots, V_n = e_n\} \\ S'' &= \{V_1 = f_1, \ldots, V_n = f_n\} \end{aligned}$$

the result of the application of the $\odot$ operator is as follows: $[S', T', P'] \odot [S'', T'', P''] = [S''', T''', P''']$ where $[S''', T''', P''']$ is defined such that $\forall i, 1 \leq i \leq n$: $S''' = \{V_1 = g_1, \ldots, V_i = g_i, \ldots, V_n = g_n\}$ and

$$g_i = \begin{cases} e_i & : \text{ if } e_i = f_i \\ \text{new symbol} & : \ otherwise \end{cases}$$

$T''' = (T' \vee T'') \wedge ((P' \wedge \Omega') \vee (P'' \wedge \Omega''))$ such that

$$\begin{aligned} \Omega' &= \bigwedge_{1 \leq i \leq n, e_i \neq f_i} (g_i = e_i) \\ \Omega'' &= \bigwedge_{1 \leq i \leq n, e_i \neq f_i} (g_i = f_i) \end{aligned}$$

and $P''' = P' \vee P''$.

Therefore, at the confluence node immediately after Q_7, the value x_2 of X can be bound to two different expressions. First, $x_2 = -2 * \nabla_1$ if the conditional expression of Q_2 holds, otherwise $x_2 = 2 * \nabla_1$. Formally this is described by $(\bigtriangledown_1{<}0\wedge\ x_2 = -2 * \bigtriangledown_1) \vee (\bigtriangledown_1 \geq 0 \wedge x_2 = 2 * \bigtriangledown_1)$. Similar accounts for y_1, the value of Y, which can be bound two different expressions depending on which branch is taken at Q_2. In order to prevent extensive bracketing in path and state conditions in the remainder of this paper we assume conventional precedences for arithmetic expressions which have higher precedence than relational operators ($=,<,\leq$, etc.). Logical operators ($\wedge$ and $\vee$) have lowest precedence. Furthermore, we introduce a function γ to avoid long first order logic expressions,

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

where $\gamma(c;\ x_1 = e_1, \ldots, x_k = e_k;\ x_1 = f_1, \ldots, x_k = f_k)$ 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 conditional 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. Based on the definition of the γ function we can write the state condition T_7 for the previous code as $\gamma(\bigtriangledown_1{<}0;\ x_2{=}{-2} * \bigtriangledown_1,\ y_1{=}2 * \bigtriangledown_1;\ x_2{=}2 * \bigtriangledown_1, y_1{=}\bigtriangledown_1)$.

$$[S,\ T = T' \wedge \gamma(c;\ \mathrm{x}{=}e;\ \mathrm{x}{=}f),\ P] = \begin{cases} [S_{x\to e}, T'_{x\to e}, P_{x\to e}] & : \text{ if } (P_{x\to e}{\Rightarrow}c) \wedge (P_{x\to f}{\Rightarrow}c) \\ [S_{x\to f}, T'_{x\to f}, P_{x\to f}] & : \text{ if } (P_{x\to e}{\Rightarrow}\neg c) \wedge (P_{x\to f}{\Rightarrow}\neg c) \\ [S, T, P] & : \ otherwise \end{cases} \quad (1)$$

$$[S,\ T = T' \wedge \gamma(c;\ \mathrm{x}{=}e;\ \mathrm{x}{=}f),\ P] = \begin{cases} [S_{x\to e}, T'_{x\to e}, P_{x\to e}] & : \text{ if } P{\Rightarrow}c \\ [S_{x\to f}, T'_{x\to f}, P_{x\to f}] & : \text{ if } P{\Rightarrow}\neg c \\ [S, T, P] & : \ otherwise \end{cases} \quad (2)$$

$$[S{,}T = T' {\wedge} \gamma(c;\mathrm{x}{=}e;\mathrm{x}{=}f) {\wedge} \gamma(c';\mathrm{y}{=}e';\mathrm{y}{=}f'){,}P] = \begin{cases} [\mathrm{S,T'} \wedge \gamma(c;\ \mathrm{x}{=}e;\ \mathrm{x}{=}f) \wedge \gamma(c';\ \mathrm{y}{=}e'_{x\to e};\ \mathrm{y}{=}f'),\mathrm{P}] & : \text{ if } c' \Rightarrow c \\ [\mathrm{S,T'} \wedge \gamma(c;\ \mathrm{x}{=}e;\ \mathrm{x}{=}f) \wedge \gamma(c';\ \mathrm{y}{=}e';\ \mathrm{y}{=}f'_{x\to e}),\mathrm{P}] & : \text{ if } \neg c' \Rightarrow c \\ [\mathrm{S,T'} \wedge \gamma(c; x = e; x = f) \wedge \gamma(c'; y = e'_{x\to f}; y = f'),\mathrm{P}] & : \text{ if } c' \Rightarrow \neg c \\ [\mathrm{S,T'} \wedge \gamma(c; x = e; x = f) \wedge \gamma(c'; y = e'; y = f'_{x\to f}),\mathrm{P}] & : \text{ if } \neg c' \Rightarrow \neg c \\ [S, T, P] & : \ otherwise \end{cases} \quad (3)$$

Figure 2: Simplification of contexts with γ functions where c, c' are conditional expressions and e, f, e', f' are symbolic expressions.

P_8 is equal to $P_7 \wedge (y_1{\geq}0)$ where $y_1{\geq}0$ is the constraint induced by the conditional expression of Q_8. $[S_8, T_8, P_8]$ can be rewritten to $[S_8,\ \gamma(\bigtriangledown_1{<}0;x_2{=}{-2} * \bigtriangledown_1;\ x_2{=}2 * \bigtriangledown_1) \wedge \gamma(\bigtriangledown_1{<}0;y_1{=}2 * \bigtriangledown_1;y_1{=}\bigtriangledown_1){,}P_8]$ according to the following rewrite rule:

$$\gamma(c;\ x_1 = e_1, \ldots, x_k = e_k;\ x_1 = f_1, \ldots, x_k = f_k) =$$
$$\gamma(c;\ x_1 = e_1;\ x_1 = f_1) \wedge \gamma(c;\ x_2 = e_2, \ldots, x_k = e_k;\ x_2 = f_2, \ldots, x_k = f_k)$$

We can simplify $[S_8, T_8, P_8]$ to $[S'_8, T'_8, P'_8]$ based on simplification rule (1) of Figure 2. $S_{x\to e}$ means that all occurrences of x in S are simultaneously substituted by e. The first two rules of Figure 2 can simplify a context immediately after a branch whose conditional expression impacts the associated state condition. By using rules (1) or (2) we can eliminate a γ function $\gamma(c; ...; ...)$ in the state condition of a context C and simplify C if the path condition of C implies c or $\neg c$. The third rule may simplify a context at a confluence node that implies a new γ function. This rule can eliminate additional unknowns (variable values) in γ functions if the condition of one γ function implies the condition of another γ function. If we replace a γ function by its equivalent first order logic expression, then we can also use the simplification techniques described in Section 2.2 to achieve the same simplified result.

Another confluence of two branches occurs after statement Q_{10}. Two different contexts $[S_7, T_7, P_7 \wedge y_1 < 0]$ and $[S_9, T_9, P_9]$ reach this point. $[S_7, T_7, P_7 \wedge y_1 < 0]$ is a dummy context of the missing else branch of statement Q_8. The dummy context can be easily made explicit by adding an artificial else branch with an *empty* statement to every "*if then statement-sequence endif*" construct. $[S_7, T_7, P_7 \wedge y_1 < 0]$ can be rewritten to $[S_7'=\delta(S_7;\text{Y}=2*\nabla_1)$, $T_7'=\gamma(\nabla_1<0; x_2=-2*\nabla_1;\ x_2=2*\nabla_1)$, $P_7'=2*\nabla_1<0]$ based on the rewrite rule for the γ function and simplification rule (1) of Figure 2. At the confluence node after Q_{10} the value of X can have two different values depending on whether or not the true-branch of Q_8 has been taken. The value of Y is identical in both contexts that reach this point.

3.3 Loops and Recurrences

Modeling loops implies a problem with *recurrence (induction) variables*[2] 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. Instead, we will use functions 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.

The goal of our symbolic evaluation with respect to loops is to detect the recurrence variables, determine the recurrence system and finally find closed forms for recurrence variables at the loop exit by solving the recurrence system. The *recurrence system* is given by the *boundary conditions* (initial values for recurrence variables in the loop preheader), the *recurrence relations* (implied by the assignments to the recurrence variables in the loop body) and the *recurrence condition* (loop or exit condition). Consider the following example:

$[S_0=\{\text{M}=m,\text{F}=y,\text{I}=\perp,\text{B}=b\},\ T_0=b>0 \wedge m>0,\ P_0=\text{true}]$

Q_1: do I=B,M,B

$[S_1=\delta(S_0;\text{F=f(n)},\text{I=i(n)}),\ T_1=T_0 \wedge \text{f(0)=y} \wedge \text{i(0)=b} \wedge f(n+1)=2*f(n) \wedge i(n+1)=i(n)+b \wedge n \geq 0,$
$P_1=P_0 \wedge i(n) \leq m\ ]$

Q_2: F := 2*F

$[S_2=\delta(S_1; F=2*f(n)), T_2=T_1, P_2=P_1]$

Q_3: enddo

$[S_3=\delta(S_1;\text{F=f(z)},\text{I=i(z)}),\ T_3=T_1 \wedge z=min\{n \mid i(n)>m\},\ P_3=P_1 \wedge i(z)>m] =$
$= [S_3'=\delta(S_0;\text{F}=y*2^{\lceil\frac{m+1}{b}\rceil-1},\text{I=b}*\lceil\frac{m+1}{b}\rceil), T_3'=T_0,\ P_3'=P_0]$

There are two recurrence variables – F and I – whose values change during execution of the loop. We introduce functions $f(n)$ and $i(n)$ for F and I respectively to model their values at the end of a generic n-th iteration where $n \geq 0$. The boundary values for the recurrence variables are given by f(0)=y and i(0)=b which is part of the state

[2] All variables which are written inside a loop – including the loop variable – are defined as recurrence variables.

condition T_1. The recurrence relations are given by $f(n+1) = 2*f(n)$ and $i(n+1) = i(n)+b$. Note that in general the values $f(n+1)$ and $i(n+1)$ are known only after the last statement in the loop body has been symbolically evaluated. However, our techniques for generating contexts (see Section 4) insert the recurrence system at the loop header as soon as the postbody node has been traversed which enables extracting the recurrence system from a well defined statement (loop header).

In order to automatically compute a closed form for the recurrence variables F and I at the loop exit (after the *enddo*), we first need to find the recurrence relations and the boundary conditions for $n \geq 0$ as given in the loop header context $[S_1, T_1, P_1]$:

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

The solution (loop invariant) to the previous recurrences is given as follows:

$$f(n) = y * 2^n \tag{4}$$

$$i(n) = b*(n+1) \tag{5}$$

Based on (5) and $i(n) > m$ (recurrence condition) we can determine the last iteration of the loop which is equivalent to finding a z for which the following holds:

$$z = min\{n \mid i(n) > m\} \tag{6}$$

$$= min\{n \mid b*(n+1) > m\} \tag{7}$$

$$= \lceil \frac{m+1}{b} \rceil - 1 \tag{8}$$

The ceiling operation in (8) is 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 (4) and (5):

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

which specifies the variable binding of F and I at the loop exit $[S_3', T_3', P_3']$. As we can determine closed forms for the recurrence variables (I and F) of the loop, T_3' and P_3' are respectively identical with T_0 and P_0.

Our approach can detect all recurrences implied by recurrence variables and represent them by a recurrence system. We have implemented a recurrence solver [46] written on top of Mathematica. The recurrence solver tries to determine closed forms for recurrence variables based on their recurrence system which is directly obtained from the context at the loop header. The implementation of our recurrence solver (see also Section 5) is largely based on methods described in [30, 41] and improved by our own techniques as outlined in Section 6.

In general it is not possible to find closed forms for every recurrence [30]. We introduce new value symbols for every unresolved variable. An unresolved variable is a recurrence variable for which no closed form can be determined. Value symbols of unresolved variables are linked with their associated recurrence system. Evaluation of unresolved variables is then done based on their value symbols instead of their recurrence system. Furthermore, we have developed an algorithm [27] to determine the monotonic behavior of unresolved variables. This can be important for applied symbolic analysis (for instance, dependence analysis) to reason about expressions containing unresolved variables.

3.4 Arrays

In this section we will describe how to precisely model array operations which includes both linear and non-linear array accesses. In the case of indirect array accesses[3] such as $W(A(I))$ where an index array A is used to access a specific element of array W, our techniques commonly achieve better results (in terms of closed forms) for structural than for input dependent array accesses. Structural array references imply that array indices and index arrays are symbolic expressions whose domain is supplied by the program variables and integer constants. Array indices of input dependent array references depend on input data.

Let A be a one-dimensional array with t ($t \geq 1$) array elements. $A = \vec{v}$ specifies a generic value binding for every array element of A. The i-th ($1 \leq i \leq t$) element of vector $\vec{v}$ corresponds to the value of the i-th array element of A. The values of multi-dimensional arrays are described by matrices instead of vectors. Furthermore, let S_i={A=$\vec{v}$,...} be the state before an assignment statement $Q_{i+1} : A(e) = f$, where e and f are symbolic expressions and e can also be an index array. Then the value of A in the state after Q_{i+1} is given by a function φ such that

$$A = \varphi(\vec{v}, e, f)$$

where $\varphi(\vec{v}, e, f)$ denotes the same array element values of A as given in S_i except that when applied to the index e where it yields f. For instance, $\varphi(\vec{v}, e, e+1)$ denotes the same value binding as given by $\vec{v}$ except for the array element with the index e whose value is given by $e + 1$.

In general sequences of array assignments are modeled by chains of φ functions. For instance, $A(e_1) = f_1$ followed by $A(e_2) = f_2$ is modeled by $\varphi(\varphi(\vec{v}, e_1, f_1), e_2, f_2)$ where e_1, f_1, e_2, f_2 are symbolic expressions. Chains of φ functions are simplified by using the rules of Figure 3.

Array accesses are modeled similarly by a function ψ such that

$$\psi(\vec{v}, e)$$

yields the value of array A with index e where $\vec{v}$ is the value binding of A.

Consider the following code segment which consists of two loops. An array A is written in the first loop. 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={M=m,I=$\perp$,A=$\vec{v}$,W=$\vec{u}$}, $T_0 = 2 \leq m$, P_0=true]

Q_1: do I=1,M-1

[S_1=δ(S_0;A=$\vec{a}(n)$,I=i(n)), T_1=$T_0 \wedge \vec{a}(0)=\vec{v} \wedge$i(0)=1$\wedge$ i(n+1)=i(n)+1 $\wedge$ $\vec{a}(n+1)$=φ($\vec{a}$(n),i(n),m-i(n)) $\wedge$ $n \geq 0$, P_1=P_0 $\wedge$ $i(n) \leq m-1$]

Q_2: A(I):=M-I

[S_2=δ(S_1;A=φ($\vec{a}$(n),i(n),m-i(n))),$T_2 = T_1$,P_2=P_1]

Q_3: enddo

[S_3=δ(S_0;$A = \vec{a}(m-1)$,I=m), $T_3 = T_0$, $P_3 = P_0$]

Q_4: do I=1,M-1

[3] Note that indirect array accesses refer to a sub-class of non-linear array accesses.

$[S_4 = \delta(S_3; W = \vec{w}(n), I = i(n)),\ T_4 = T_3 \wedge \vec{w}(0) = \vec{u} \wedge i(0) = 1 \wedge i(n+1) = i(n)+1 \wedge$
$\wedge\ \vec{w}(n+1) = \varphi(\vec{w}(n), \psi(\vec{a}(m-1), i(n)), i(n)+1) \wedge n \geq 0, P_4 = P_3 \ \wedge\ i(n) \leq m-1]$

Q_5: W(A(I)):=I+1

$[S_5 = \delta(S_4; W = \varphi(\vec{w}(n), \psi(\vec{a}(m-1), i(n)), i(n)+1)),\ T_5 = T_4,\ P_5 = P_4]$

Q_6: enddo

$[S_6 = \delta(S_3; W = \vec{w}(m-1), I = m),\ T_6 = T_3,\ P_6 = P_3]$

Q_2 describes an array assignment *A(I) = M-I* which is described by $\varphi(\vec{a}(n), i(n), m - i(n))$. This means that A did not change due to Q_2 except for the element with index value $i(n)$ which received a new value $m - i(n)$. $\vec{a}(n)$ specifies the value binding of A at the end of the n-th iteration of the first loop. $\vec{a}(0) = \vec{v}$ specifies the value binding of A at $[S_0, T_0, P_0]$.

$$\psi(\varphi(\vec{v}, e, f), g) = \begin{cases} f & : \ e = g \\ \psi(\vec{v}, g) & : \ \text{otherwise} \end{cases} \tag{1}$$

$$\varphi(\varphi(\ldots(\varphi(\varphi(\vec{v}, e_i, f_i), e_{i-1}, f_{i-1}), \ldots, e_2, f_2), e_1, f_1) = \begin{cases} \varphi(\varphi(\ldots(\varphi(\vec{v}, e_{i-1}, f_{i-1}), \ldots, e_2, f_2), e_1, f_1) & : e_i = e_1 \\ \varphi(\varphi(\ldots(\varphi(\varphi(\vec{v}, e_i, f_i), e_{i-1}, f_{i-1}), \ldots, e_2, f_2), e_1, f_1) & : otherwise \end{cases} \tag{2}$$

Figure 3: Simplification rules for ψ and φ functions where e, f, g, e_i, f_i are symbolic expressions

Q_5 describes an indirect array access to W whose index expression is given by A(I). The associated value change is modeled by $\varphi(\vec{w}(n), \psi(\vec{a}(m-1), i(n)), i(n)+1)$ 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 $i(n)+1$. $\psi(\vec{a}(m-1), i(n))$ refers to the value of array A with index $i(n)$.

In order to determine $W(k)$, the value of the k-th element of array W, based on context $[S_6, T_6, P_6]$ we proceed as follows:

$$\begin{aligned} eval(W(k), [S_6, T_6, P_6]) &= \psi(\vec{w}(m-1), k) = \\ &= \psi(\varphi(\vec{w}(n), \psi(\vec{a}(m-1), i(n)), i(n)+1), k) \quad \text{where} \quad n \geq 0 \wedge i(n) \leq m-1 && (6) \\ &= \psi(\varphi(\vec{w}(n), \psi(\vec{a}(m-1), n+1), n+2), k) \quad \text{where} \quad 0 \leq n \leq m-2 && (7) \\ &= \begin{cases} n+2 & : \ k = \psi(\vec{a}(m-1), n+1) \\ \psi(\vec{w}(n), k) & : \ otherwise \end{cases} \quad \text{where} \quad 0 \leq n \leq m-2 && (8) \\ &= \begin{cases} n+2 & : \ k = \psi(\vec{a}(m-1), n+1) \\ \psi(\vec{u}, k) & : \ otherwise \end{cases} \quad \text{where} \quad 0 \leq n \leq m-2 && (9) \end{aligned}$$

(6) is determined from $[S_4, T_4, P_4]$. We derive (7) by computing the loop invariant for the recurrence of $i(n)$ which is given by $n+1$ according to Section 3.3 and consequently by substituting $n+1$ for $i(n)$ in (6). We determine (8) by applying rule (1) of Figure 3 to (7). If the index of W is not written in the first loop of the example code, then $W(k)$ accesses a value given by $\vec{u}$. By recursively applying rule (1) of Figure 3 we can simplify (8) to (9). We can further rewrite $k = \psi(\vec{a}(m-1), n+1)$ as follows:

$$
\begin{aligned}
k &= \psi(\vec{a}(m-1), n+1) && (10)\\
&= \psi(\varphi(\vec{a}(n), i(n), m-i(n)), n+1) && (11)\\
&= \psi(\varphi(\vec{a}(n), n+1, m-n-1), n+1) && (12)\\
&= m-n-1 && (13)\\
&\Rightarrow n = m-k-1 && (14)
\end{aligned}
$$

(11) is obtained from $[S_1, T_1, P_1]$. We derive (12) by computing the loop invariant $n+1$ for the recurrence of $i(n)$ (see Section 3.3) and by replacing it for $i(n)$ in (11). (12) is then simplified to (13) according to rule (1) of Figure 3.

Finally, substituting $m-k-1$ for n in (9) yields

$$
eval(W(k), [S_6, T_6, P_6]) = \begin{cases} m-k+1 & : \; 1 \leq k \leq m-1 \\ \psi(\vec{u}, k) & : \; otherwise \end{cases} \qquad (15)
$$

$1 \leq k \leq m-1$ is implied by substituting n with $m-k-1$ in $0 \leq n \leq m-2$.

3.5 Procedures

In order to model procedure calls, we first symbolically evaluate the associated procedures by using the techniques described in the previous sections. Generic variable bindings are assumed for all variables used at the procedure start. At the end of the procedure we determine the variable values as expressions of the variable values at the start. For the purpose of illustration we assume that each procedure has a single start and end which are respectively associated with a start and end context. Furthermore, our procedure model corresponds to Fortran where procedures are named *subroutines* or *functions* and all procedure parameters that are variables are passed by *call-by-reference*. Expressions other than variables are passed by *call-by-name*. Note that our procedure model is not restricted to Fortran but can be easily ported to a variety of programming languages with little effort. The values of procedure local variables are set to $\perp$ at the procedure start. Global variables are inherently modeled by our symbolic evaluation approach. These variables are always kept in the state of procedure contexts as additional variables. Currently, we assume an acyclic call graph which also excludes recursive procedure calls. For instance, in the following procedure *ADIFF*:

Q_1: subroutine ADIFF(A,B,S)

  $[S_1=\{A=a, B=b, S=s\}, T_1=\text{true}, P_1=\text{true}]$

Q_2: if (A $\geq$ B) then

  $[S_2=S_1, T_2=T_1, P_2=a \geq b]$

Q_3: S:=A-B

  $[S_3=\delta(S_2;S=a-b), T_3 = T_2, P_3=P_2]$

Q_4: else

  $[S_4=S_1, T_4 = T_1, P_4= a<b]$

Q_5: S:=B-A

  $[S_5=\delta(S_4;S=b-a), T_5 = T_4, P_5=P_4]$

Q_6: endif

  $[S_6=\delta(S_1;S=u),\ T_6=\gamma(a \geq b; u=a-b; u=b-a),\ P_6 = true]$

Q_7: endsubroutine

$[S_1, T_1, P_1]$ and $[S_6, T_6, P_6]$ are respectively the start and end context of procedure *ADIFF*. At the time where Q_1 is evaluated nothing may be known about the variable values of the procedure parameters, therefore, generic values are assumed. Similar considerations account for T_1 and P_1 which are set to *true*.

The following code segment contains a call to procedure *ADIFF*:

```
            [S'_0={X=m,Y=1,Z=⊥,I=⊥,J=⊥,M=m,W=w,...},T'_0=true, P'_0=m≥1]
Q'_1:  call ADIFF(X,Y,Z)
            [S'_1={X=m,Y=1,Z=u,I=⊥,...}, T'_1=γ(m≥1;u=m − 1;u=1 − m),P'_1=m≥1] =
            =[S''_1=δ(S'_1;Z=m-1), T''_1=true, P''_1=m≥1]
       do I=1,M
Q'_3:     W(M) := ...
          do J=1,M
            ...
            [S'_4={...,Z=m − 1,J=j(n),M=m,...},T'_4 =...,P'_4=m≥1∧...]
Q'_5:       W(M+J) := ...
            [S'_5={...,Z=m − 1,J=j(n),M=m,...},T'_5 =...,P'_5=m≥1∧...]
Q'_6        ... := W(Z+J)
          enddo
       enddo
```

We refer to A, B and S in *subroutine ADIFF(A,B,S)* as the *formal procedure variables*. 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 the following we present an algorithm that computes the context after the statement *call ADIFF(X,Y,Z)* in Q'_1.

In general, let $C_b^q = [S_b, T_b, P_b]$ and $C_a^q = [S_a, T_a, P_a]$ respectively specify the contexts before and after a call statement q to a procedure R. $(V_1, ..., V_k)$ is a vector of formal procedure variables of R and $(e_1, ..., e_k)$ the associated vector of actual procedure parameters of q. The associated formal procedure variable V_i of an actual procedure parameter e_i is denoted by *formal(e_i)* where $1 \leq i \leq k$. Let C^R be the context at the end of R. Then C_a^q is computed according to the following strategy:

1. In the first step, C^R is used to create a new context C'. The actual procedure parameter values are propagated into C^R. In other words, C^R is specialized for q by C_b^q which results in a new context $C' = [S', T', P']$:

$$C' \quad := \quad {C^R}_{v_1 \to eval(e_1, C_b^q), ..., v_k \to eval(e_k, C_b^q)}$$

   where v_i is the generic value of V_i at the beginning of procedure R for all $1 \leq i \leq k$. The symbolic expression eval(e_i, C_b^q) is the value of the i-th actual procedure parameter of q. $C_{v_1 \to e_1, ..., v_k \to e_k}$ specifies a simultaneous replacement of all v_i by e_i. ${C^R}_{v_i \to eval(e_i, C_b^q)}$ means that all occurrences of v_i are simultaneously substituted by $eval(e_i, C_b^q)$ for $1 \leq i \leq k$. Thereafter, C' is simplified by using the rules of Figure 2 and 3 including the techniques described in Section 2.2.

2. In the second step, the values of the formal procedure variables in C' are returned to their associated actual procedure parameters in $C_a'^q$. This step is only performed for those actual procedure parameters that are variables. Let $\{A_1, ..., A_l\}$ be the set of all vector components in $(e_1, ..., e_k)$ which are variables where $l \leq k$. Note that according to Section 2.1 we exclude aliasing of variables; therefore all actual parameters that are variables must be pairwise different.

$$\begin{aligned} S_a &:= \delta(S_b; A_1 = eval(formal(A_1), C'), ..., A_l = eval(formal(A_l), C')) \\ T_a &:= T_b \wedge T' \\ P_a &:= P_b \end{aligned}$$

Note that P_a is given by P_b as the path condition at the start and at the end of a procedure is always *true*.

Our strategy for handling procedures requires a single evaluation of every procedure. Only the end context of each procedure is re-evaluated for specialization of a specific call to this procedure which is in contrast to a strategy that would require full evaluation of the entire procedure for each call to it.

Furthermore, $[S_1', T_1', P_1']$ in the previous example can be rewritten as $[S_1'', T_1'', P_1'']$ according to simplification rule (2) of Figure 2.

Parallelization of the previous code example is commonly prevented due to memory-related dependences. We will demonstrate the usefulness of our analysis for array privatization [53, 54] which is an important optimization for parallel programs 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 safe 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 Q_6' is completely covered by the array section written in Q_3' and Q_5', we determine the index expression value of $W(M+J)$ (write access) at Q_5' based on $[S_4', T_4', P_4']$:

$$eval(M + J, [S_4', T_4', P_4']) = m + j(n)$$

as well as the index expression value of $W(Z+J)$ (read access) at Q_6' based on $[S_5', T_5', P_5']$:

$$eval(Z + J, [S_5', T_5', P_5']) = m - 1 + j(n)$$

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 Q_6' reads $W(M)$ which is written by Q_3'. Therefore, by using our symbolic evaluation it can be detected that privatizing W is safe for the previous example which enables parallelization of this code.

4 Generating Contexts for Symbolic Evaluation

The generation of contexts is explained based on the CFG $G = (N, E, x, y)$ of a program. Every $n \in N$ has a unique *entry context* $C_b(n) = [S_b, T_b, P_b]$ that describes the variable bindings and state and path conditions immediately before node n. Note that every n has only a single entry context although n may have multiple predecessors[4]. $C_b(x)$, the entry context of the CFG start node x includes the initial variable bindings and state and path condition of the CFG as described in Section 3.1.

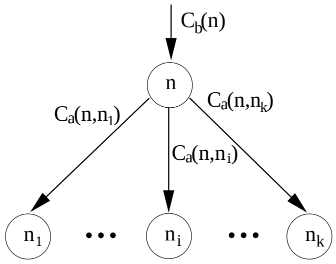

Figure 4: Exit and entry contexts for a generic node n with k $(1 \leq i \leq k)$ successor nodes.

Furthermore, every CFG node n has a unique *exit context* $C_a(n, n')$ for each of its successor nodes $n' \in succs(n)$. $C_a(y, undef)$, the exit context of the CFG end node y, describes the context at the end of the CFG. Note that y does not have a successor node; therefore, the second parameter of C_a is set to undef. Let *cond_expr(n,n′)* denote the conditional expression that governs traversal of the edge (n,n′). If n has only a single successor node n', then cond_expr(n,n′) = true. If n has two or more successor nodes, then every time n is executed at execution time, exactly one conditional expression of all outgoing edges from n is true, all others are false. Figure 4 illustrates the entry and exit contexts for a generic CFG node n.

```
procedure gen_context(CFG G)
    for every n ∈ N of G in reverse postorder traversal do
        gen_context_entry(n)
        gen_context_exit(n)
    endfor
endprocedure
```

Figure 5: Traverse CFG in reverse postorder to generate contexts

At the beginning of our algorithm we initialize the exit context of all postbody nodes to $\top$ which specifies an undefined context. This is important for computing the entry context of loop header nodes. Note that according to Section 2.1 all postbody nodes have only a single successor node (loop header).

Figure 5 shows our algorithm that generates the entry and exit contexts for every node $n \in N$ by traversing the CFG in reverse postorder. For every node n we call two procedures *gen_context_entry* and *gen_context_exit* that, respectively, generate the entry and exit contexts of n. Note that according to [37] the reverse postorder sorts a reducible flow graph in topological order which enables the algorithm to generate the entry and exit contexts of all CFG nodes – except for preheader and procedure end nodes – through a single traversal of the CFG. In other

[4] The folding operator $\odot$ is used to generate $C_b(n)$ based on the exit contexts of all the predecessor nodes of n.

```
procedure gen_context_entry(n)
    if |preds(n)| = 0 then
        init_start_node(n);
    else
        C_b(n) := ⨀_{n' ∈ preds(n)} C_a(n', n)
    endif
    if type(n) = PREHEADER then
        init_preheader(n)
    elseif type(n) = POSTEXIT then
        update_postexit(n)
    endif
endprocedure
```

Figure 6: Generate context for node entry

words, every time our algorithm traverses a node $n \in N$ it is guaranteed that the exit contexts of all nodes in $preds(n)$ are already generated. Only loop preheader and procedure end nodes must be traversed more than once.

We can generate contexts for an entire program (including main program and all procedures) by using gen_context. Every procedure including the main program is represented through a unique CFG. Based on the assumption of an acyclic procedure call graph we invoke gen_context for every procedure exactly once while traversing the call graph bottom up. This guarantees that the contexts of a procedure have already been generated at the time where gen_context encounters the associated procedure call. Moreover, according to Section 3.5 for every call statement q to a procedure R, our algorithm accesses the end context of R in order to compute the context after q.

In order to describe gen_context_entry and gen_context_exit we define for every $n \in N$ a predicate *type(n)* which specifies the type of the statement (ASSIGNMENT, GOTO, READ, CONDITIONAL, CALL, PREHEADER, POSTEXIT and POSTBODY) associated with n. Furthermore, for every loop exit node n, we introduce a predicate *recurrence_cond(n)* that specifies the (recurrence) condition under which the associated loop is exited at n.

Procedure *gen_context_entry* (see Figure 6) describes how to create the entry context of a node n. First, it is examined whether n is the start node of the CFG whose entry context is initialized by function *init_start_node* according to Section 3.1. Otherwise, n has at least one predecessor. In order to generate the entry context of n, the folding operator is applied to the exit contexts of all nodes in $preds(n)$. Note that if n corresponds to a loop header, the folding operation is defined due to initialization of the associated postbody's exit context. Figure 7 shows the properties that hold for the folding operator.

If a node n is traversed that corresponds to the preheader node of a loop L, then function *init_preheader(n)* is invoked which initializes $C_b(n)$ with the following information:

$S_b(n)$: The value of every recurrence variable V of L is set to a unique $v(i)$ where i corresponds to a generic loop iteration.

$T_b(n)$: Insert the boundary conditions (initial values, $i \geq 0$) – obtained from the current $C_b(n)$ – for every

$$\begin{aligned} C_1 \odot C_2 &= C_2 \odot C_1 \\ (C_1 \odot C_2) \odot C_3 &= C_1 \odot (C_2 \odot C_3) \\ C \odot C &= C \\ C \odot \top &= C \end{aligned}$$

Figure 7: Properties of the folding operator $\odot$ (C, C_1, C_2 and C_3 are contexts).

recurrence variable. Note that the recurrence relations are included in $T_b(n)$ when the postbody node of L is being traversed by algorithm gen_context_exit of Figure 8.

If a postexit node n of a loop L is traversed then a function *update_postexit* is invoked which tries to compute closed forms for all recurrence variables based on the recurrence system according to the strategy outlined in Section 3.3. The recurrence relations and boundary conditions are derived from the preheader of L and the recurrence condition from $P_b(n)$. The values of recurrence variables are then replaced in $C_b(n)$ by their associated closed forms. If a closed form for a recurrence variable V cannot be computed then the value of V is set to a unique $v(z)$ in $S_b(n)$. z corresponds to the last loop iteration where L is exited at n. All occurrences of values of V are replaced by $v(z)$ in $C_b(n)$. Furthermore, $z = min\{i \mid recurrence\_cond(n)\}$ is inserted through a conjunction in $T_b(n)$. As reasoning about recurrence variables that are represented by a recurrence system is a complex task, our current implementation introduces new value symbols for every recurrence variable for which no closed form can be determined. These value symbols are linked with their associated recurrence system. Evaluation of recurrence variables for which no closed form can be found is then done based on their value symbols instead of their recurrence system.

Procedure *gen_context_exit* (see Figure 8) generates the exit contexts of a node n based on the entry context of n and the semantics of the statement associated with n. If n corresponds to the postbody node of a loop L, then function gen_context_exit has traversed the entire loop body of L. A function *update_preheader(n)* is invoked to insert the recurrence relations – obtained from $C_b(n)$ – for all recurrence variables in the preheader node of L. Furthermore, the loop invariant is computed – based on the recurrence relations and the boundary conditions – and stored in the preheader node of L.

A function *eval_exit_context* is invoked in *gen_context_exit* in order to compute the exit context $C_a(n, n')$ of n with respect to every specific $n' \in succs(n)$. Function *side_effect(n, C)* evaluates all nodes n associated with an ASSIGNMENT, READ or CALL statement that may imply a side-effect. Let $C = [S = \{V_1 = e_1, \ldots, V_n = e_n\}, T, P]$ and n corresponds to

- an ASSIGNMENT statement of the following form $V_i := expr$ where $expr$ is a symbolic expression defined over variables $V_1, ..., V_n$, then

$$\text{side\_effect}(n, C = [S, T, P]) \quad = \quad [\{V_1 = e_1, ..., V_i = eval(expr, C), ..., V_n = e_n\}, T, P]$$

```
procedure gen_context_exit(n)
    for every n' ∈ succs(n) do
        C_a(n,n') := eval_exit_context(n, C_b(n), cond_expr(n,n'))
    endfor
    if type(n) = POSTBODY then
        update_preheader(n)
    endif
endprocedure

function eval_exit_context(n,C=[S,T,P],cond)
    switch type(n)
        case ASSIGNMENT | READ | CALL:
            C' := side_effect(n,C)
        default: /* PREHEADER, POSTEXIT, POSTBODY, GOTO, CONDITIONAL */
            S' := S
            T' := T
            if P ⇒ eval(cond,C) then
                P' := P
            elseif P ⇒ ¬ eval(cond,C) then
                P' := false
            else
                P' := P ∧ eval(cond,C)
            endif
    endswitch
    return C'
endfunction
```

Figure 8: Generate contexts for node exits

- a READ statement of the following form $read(V_i)$, then

$$\text{side\_effect}(n, C = [S,T,P]) \quad = \quad [\{V_1 = e_1, ..., V_i = \bigtriangledown, ..., V_n = e_n\}, T, P]$$

  Note that a read statement may imply additional constraints (for instance, input of positive integers instead of general integers) on the value of V_i which are added to the state condition of the associated exit context.

- a CALL statement, then side_effect(n, C) is computed according to the method described in Section 3.5.

The default case considers all statements without side-effects. For unconditional statements the exit context is given by the entry context. For branch statements we have to distinguish three cases: First, if the path condition of the entry context implies that *cond* is true, then the path conditions of both entry and exit contexts are identical. Second, P' is set to false if we can statically determine that the branch from n to n' can never be taken. Third, if it is not possible to statically determine whether a branch is taken or not, then we have to add the associated branch condition to P' through a conjunction.

In order to improve the efficiency of symbolic evaluation and to reduce the associated memory requirement our implementation only stores symbolic evaluation information in a node n that reflects the semantics of n. Information that has been propagated from predecessor nodes to n is accessible from n through linked lists.

The overall algorithm traverses every node and all its predecessors once and each loop preheader at a maximum three times (once each by the reverse postorder traversal, update_postexit and update_preheader). Furthermore, every end context of a procedure is accessed once for each call to it.

In general, all symbolic analyses – about which the authors are aware of – that compute exact values of variables, manipulate and simplify symbolic expressions, and resolve recurrences require exponential complexity. This accounts for our own approach as well.

5 Experiments

We have implemented a prototype of our symbolic evaluation framework which is used as part of *VFC*, a parallelizing compiler, and P^3T, a performance estimator for parallel programs. Our system for manipulating symbolic expressions and constraints, as well as our techniques for simplifying expressions and constraints, comparing symbolic expressions and computing the number of solutions to a system of constraints have been fully implemented. The current implementation of our recurrence solver handles recurrences of the following kind: linear recurrence variables (incremented inside a loop by a symbolic expression defined over constants and invariants), polynomial recurrence variables (incremented by a linear symbolic expression defined over constants, invariants and recurrence variables) and geometric recurrence variables (incremented by a term which contains a recurrence variable multiplied by an invariant). Our algorithm [28] for computing lower and upper bounds of symbolic expressions based on a set of constraints is used to detect whether a recurrence variable monotonically increases or decreases. Even if no closed form can be found for a recurrence variable, monotonicity information may be useful, for instance, to determine whether a pair of references can ever touch the same address. The current implementation of our symbolic evaluation framework models assignments, GOTO, IF, simple I/O and array statements, DO-loops and procedures. In the following we describe several experiments to measure the potential benefits of symbolic evaluation for improving the performance of parallel programs.

5.1 Empirical Results

In this section we present empirical results to demonstrate the need for more advanced symbolic analysis techniques. For this study we used several Fortran programs that have been taken from the Perfect, RiCEPS (Rice Compiler Evaluation Program Suite) and Genesis benchmark suites [6, 1] as well as from the WIEN95 software package [7]. The latter code has been developed by the chemistry department of the Vienna University of Technology and is used for quantum mechanical calculations of solids.

Table 1 provides a brief description, and the number of lines and subroutines for each program. In Table 2 we report on the frequencies of some important program constructs/features that are currently not adequately handled by most existing compiler techniques. The data is presented as a percentage of the total number of references,

code	benchmark	type	lines	subroutines
BDNA	molecular dynamics of DNA	Perfect Benchmarks	3980	43
MDG	molecular dynamics of water	Perfect Benchmarks	1238	16
MG3D	depth migration	Perfect Benchmarks	3038	28
OCEAN	2-dim ocean simulation	Perfect Benchmarks	3198	32
TRFD	2 electron integral transform	Perfect Benchmarks	485	7
TRACK	missile tracking	RiCEPS	3735	34
HNS	quantum mechanical calculations of solids	Wien95	495	5
PDE2	2-dim multigrid Poisson solver	GENESIS	551	14

Table 1: Program description

loops, variables, or expressions. For instance, 14 % of all array references in the BDNA code are non-linear. The two columns about "advanced dep. test" reflect the need of advanced dependence testing. Furthermore, many of the codes contain linear references with symbolic coefficients (reflected by the third column in Table 2). Without additional information about these coefficients conventional dependence analyses imply worst-case assumptions (assume a dependence). However, most of these symbolic coefficients in linear references in the codes of Table 2 – except for HNS and PDE2 – can be replaced with constants by using interprocedural constant propagation. The fraction of array references for which advanced dependence tests are required ranges from 7 - 78 % across all codes as shown in Table 2. Our techniques for comparing both linear and non-linear symbolic expressions and the ability of our symbolic evaluation to propagate constraints through a program enables us to handle the majority of these references. Note that there are a few sophisticated dependence tests [49, 4, 42, 11] that handle both linear and non-linear symbolic expressions. Although some of these approaches appear to be more powerful than our techniques, they also have some drawbacks associated with them, for instance, restricted program models (no GOTOs and procedure calls) [4], introduction of a large number of linear constraints to model non-linear constraints [42], etc. We believe that all of these techniques could benefit by interprocedural analysis and collection and propagation of constraints as done by our symbolic evaluation.

code	advanced dep. test		indirect array refs.		closed forms		bounds/subscript	loops	
	non-linear refs.	symbolic coeff.	non input depent.	input depend.	loop variables	other variables	expr. as a function of problem size	with procedure calls	with multiple exits
BDNA	14	23	14	0	95	55	95	11	1
MDG	3	15	0	0	100	71	91	10	0
MG3D	1	22	1	0	97	26	90	8	5
OCEAN	23	8	0	0	100	40	83	10	0
TRFD	3	4	3	0	100	57	99	3	1
TRACK	3	10	2	0	100	43	95	10	3
HNS	67	11	0	15	100	80	80	15	0
PDE2	18	8	12	0	100	84	90	12	0

Table 2: Program characteristics and effectiveness of symbolic evaluation. All figures are given as percentage values.

The next two columns ("indirect array refs.") in Table 2 summarize the frequencies of indirect array references which can go up to 15 %. For other codes of the perfect benchmarks, such as *spice* (not included in our study) up to 60 % of all array subscripts correspond to indirect array references as reported in [30]. All array accesses are precisely modeled by our symbolic evaluation techniques even in the case where we cannot find closed forms. For most array references excluding input dependent indirection arrays we can determine useful results in terms of closed forms. Although the number of indirection arrays appear to be small they may be significant in terms of performance. For instance, a single loop nest in TRFD code, that contains an indirect array reference, if not parallelized, accounts for close to 25 % of the overall execution time. Our symbolic evaluation techniques actually enable dependence testing of most non input dependent indirect array references as shown in Table 2. Similar accounts for all array references with linear subscript expressions. The two columns about "closed forms" tabulate for each program the percentage of all recurrences (loop and other variables) for which our symbolic evaluation finds a closed form. "Other variables" correspond to scalar variables that cannot be eliminated by scalar forward substitution and whose value changes inside of loops. In most cases it is very critical to find closed forms for recurrences as they commonly appear in array subscripts which are subject of dependence analysis.

The next column ("bounds/subscript expr. as a function of problem size") demonstrates that our symbolic evaluation techniques can express most variables appearing in loop bounds (lower and upper bound and stride expressions) and array subscript expressions as functions of the problem size.

Finally, the last two columns tabulate the percentage of loops with procedure calls or multiple exits. Although the number of these loops is small, in most cases they significantly impact the overall execution time. For instance, all loops with multiple exits of the TRACK code amount for approximately 40 % of the overall sequential execution time. D. Wonnacott [56] shows how to handle breaks from loops and gives rules for interprocedural analysis but apparently has not yet implemented these techniques. Except for Wonnacott's work and our symbolic evaluation framework we are not aware of any symbolic analysis technique that actually handles loops with multiple exits. We believe that modeling procedure calls is very important in order to detect and exploit sources of parallelism in particular in the case of procedure calls inside of loops.

We have examined the simplification techniques (for instance, those shown in Figures 2 and 3) of our symbolic evaluation framework when applied to the codes of Table 2. Without simplification techniques our framework's ability of

- dependence testing is reduced by at least 56 % for non-linear references, by 35 % for symbolic coefficients, and by 68 % for non-input dependent indirect array references.
- computing closed forms is reduced by at least 20 % for loop variables and 57 % for other variables.
- specifying variables appearing in loop bounds and array subscript expressions as functions of the problem size is reduced by at least 45 %.

Adding simplification techniques is, therefore, of paramount importance to further improve our symbolic evaluation framework.

5.2 Symbolic Evaluation to Optimize Communication

In the second experiment we demonstrate the effectiveness of symbolic evaluation to optimize communication. Consider the HPF code in Figure 9 which contains array index expressions that depend on a condition outside of a loop. The values for N and M are assumed to be unknown at compile time except that they are positive integers. Figure 10 shows the parallel code generated by VFC without symbolic evaluation assuming block distribution [38, 5] for array A onto NP processors. Block distribution specifies that the array is divided into contiguous blocks. The work distribution for this code under VFC is based on the "owner computes rule" which means that the processor that owns a datum will perform the computations that make an assignment to this datum. Non-local data referenced by a processor are buffered in so-called *overlap areas* that extend the memory allocated for the local segment (owned by a processor according to the data distribution strategy). VFC optimizes communication [29] by extracting single element messages from a loop and combining them to vector-form (*communication vectorization*), removing redundant communication (*communication coalescing*), and aggregating different communication statements (*communication aggregation*). VFC generates a communication (EXSR) statement for each right-hand side array reference in assignment statements. An *EXSR* [5] statement is syntactically described as $EXSR\ A(I_1, ..., I_n)\ [l_1/u_1, ..., l_n/u_n]$, where $v = A(I_1, ..., I_n)$ is the array element inducing communication and $[l_1/u_1, ..., l_n/u_n]$ is the overlap area for array A. For each i, l_i and u_i respectively specify the left and right extension of dimension i in the local segment. The dynamic behavior of an *EXSR* statement can be described as follows:

```
IF (executing processor p owns v) THEN
      send v to all processors p', such that
      (1) p' reads v and
      (2) p' does not own v
ELSEIF (v is in the overlap area of p) THEN
      receive v
ENDIF
```

VFC without symbolic evaluation cannot vectorize large portions of the communication inside of the loop (Figure 10) because it is unable to determine whether *A(M-I+MAX)* and *A(N-I+MIN)* are read before they are written by the assignment to *A(N-I)* in Q.

E_2 in Figure 10 implies that the processor which owns *A(N-I+MIN)* sends this element to the processor which owns *A(N-I)*. E_3 behaves similarly. E_2 and E_3 imply a large communication overhead, as they are executed inside a loop. If it can be proved that *N-I+MIN* and *M-I+MAX* (right-hand side array index expressions of Q) are both larger than or equal to *N-I* (left-hand side array index expression of Q), then E_2 and E_3 can be vectorized. This can be achieved by using our symbolic evaluation techniques as follows: First, eval(N-I+MIN-(N-I),$[S_4, T_4, P_4]$)= eval(MIN, $[S_4, T_4, P_4]$) = x where γ(n>m;x=m;x=n) according to T_2 of Figure 11. Based on T_1 we can conclude that $x \geq 0$ and consequently E_2 of Figure 10 can be vectorized. Second, eval(M-I+MAX-(N-I),$[S_4, T_4, P_4]$) = eval(M-N+MAX,$[S_4, T_4, P_4]$) = m-n+y where γ(n>m;y=n;y=m) according to T_2 of Figure 11. Consequently, if n>m then *m-n+y=m* otherwise n $\leq$ m and *m-n+y=2*m-n*. We can deduce that *m≥0* and *2*m-n≥0* based on T_1. Therefore, E_3 of Figure 10 can be vectorized. Note that the previous two evaluations of the function eval also determine the overlap areas associated with the corresponding communication statements. This yields the

```
!HPF$  PROCESSORS :: PROC(NP)
!HPF$  DISTRIBUTE A(BLOCK)
  if (N > M) then
    MIN = M
    MAX = N
  else
    MIN = N
    MAX = M
  endif
  do I=2,N-1
Q:  A(N-I)=A(M-I+MAX)-
        A(N-I+MIN)-A(N-I+1)
  enddo
```

Figure 9: Original HPF code

```
       if (N > M) then
         MIN = M
         MAX = N
       else
         MIN = N
         MAX = M
       endif
E1 :   EXSR A(:) [0/1]
       do I=2,N-1
E2 :     EXSR A(N-I+MIN) [</>]
E3 :     EXSR A(M-I+MAX) [</>]
Q:       A(N-I)=A(M-I+MAX)-
              A(N-I+MIN)-A(N-I+1)
       enddo
```

Figure 10: VFC (w/o SE) code

```
[S1={N=n,M=m,MIN=⊥,MAX=⊥,I=⊥,...},
   T1=n≥2∧m≥2, P1=true]
     if (N > M) then
       MIN = M
       MAX = N
     else
       MIN = N
       MAX = M
     endif
     [S2=δ(S1;MIN=x,MAX=y),
      T2=T1∧γ(n>m;x=m,y=n;x=n,y=m),
      P2=true]
Q3: if (N > M) then U=M else U=2*M-N endif
     [S3=δ(S2;U=u),
     T3=T2∧γ(n>m;u=m;u=2*m-n), P3=true]
E:   EXSR A(:) [0/U]
     do I=2,N-1
     [S4={N=n,M=m,MIN=x,MAX=y,...},
     T4=T3∧..., P4=i(n) ≤ n − 1]
Q:       A(N-I)=A(M-I+MAX)-
              A(N-I+MIN)-A(N-I+1)
     enddo
```

Figure 11: VFC (with SE) code

following communication statements which are placed immediately outside the loop (not shown in Figure 11) in the first step of communication optimization (vectorization):

- E_1 : EXSR A(:) [0/1]
- E_2 : EXSR A(:) [$0/U_1$]
- E_3 : EXSR A(:) [$0/U_2$]

where $U_1 = u_1$ such that γ(n>m;u_1=m;u_1=n) and $U_2 = u_2$ with γ(n>m;u_2=m;u_2=2*m-n). U_1 and U_2 in E_2 and E_3, respectively, specify the right extension of the associated overlap area. In order to eliminate redundant communication by communication coalescing [29] we have to verify whether the data transferred by one communication statement is already implied by another. By comparing the lower and upper bounds ([27] - see also Section 2.2) of the associated overlap areas our framework can determine that E_1 and E_2 are made redundant by E_3.

Figure 11 shows the final code as implied by VFC with symbolic evaluation. Note that E is equivalent to the previously shown E_3. Statement Q_4 is inserted to compute the different values of the overlap area U depending on whether or not $n > m$.

Table 3 tabulates the execution times on a Meiko CS-2 machine for various problem sizes and a fixed machine size (8 processors) for the codes shown in Figure 10 (VFC w/o symbolic evaluation) – and Figure 11 (VFC with symbolic

		Execution Time (secs)			
M	N	VFC	VFC	Improvement	Ovp
		(w/o SE)	(with SE)	(factor)	
256	256	0.11	0.00423	26	256
	512	0.22	0.00461	47	256
	768	0.31	0.00507	62	256
512	256	0.10	0.02210	4	768
	512	0.21	0.01051	20	512
	768	0.30	0.01255	23	512
768	256	0.11	0.03229	3	1280
	512	0.22	0.02918	7	1024
	768	0.33	0.02212	15	768

Table 3: Meiko CS-2 (8 processors) execution times for codes of Figure 10 and 11

evaluation). The resulting improvement (execution time of column 3 divided by execution time of column 4) that is gained by using symbolic evaluation varies from 3 to 62 and is shown in column 5. N significantly impacts the performance for the codes produced by VFC w/o symbolic evaluation as the loop iteration count is directly proportional to N and large portions of the communication remain inside the loop. This is also the reason why VFC w/o symbolic evaluation does not achieve any speedup for this code. For the code versions that are based on symbolic evaluation, it is the size of the overlap area (last column) that dominates the performance. For instance, the longest execution time has been measured for $M = 768$ and $N = 256$ which also implies the largest overlap area $2*768-256 = 1280$. The speedup figures range from 1.5 - 7 depending on the size of the overlap area. Note that the overlap column (Ovp) of Table 3 reflects only the overlap for EXSR statement E in Figure 11. Computation time as part of the total execution time for the codes that are optimized based on symbolic evaluation varies between 1 - 70 %. The smaller the overlap area the larger the percentage of computation time as part of the total execution time. Communication takes by far the largest portion (close to 99 %) of the total execution time for the VFC codes generated without symbolic evaluation.

5.3 Symbolic Evaluation to Optimize OLDA

In the second experiment we use OLDA which is the dominant routine in the TRFD program of the Perfect Benchmarks [6] to demonstrate more general improvements obtained by using symbolic evaluation. The OLDA code contains conditional assignments and GOTOs, recurrences, non-linear loop bounds and array index expressions. Table 4 displays the execution times as obtained on a Meiko CS-2 for various problem (N) and machine sizes (number of processors NP) for two different parallelization strategies. First, we optimized and parallelized OLDA by using VFC without symbolic evaluation (column 3). Second, we compiled the codes by using VFC with symbolic evaluation (column 4) which in particular reduced communication and also partially improved dependence analysis.

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

		Execution Time in Seconds		
N	NP	VFC	VFC	Improvement
		(w/o SE)	(with SE)	(in %)
4	1	0.412	0.393	4.7
	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	1	2.114	1.980	6.2
	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 4: Meiko CS-2 execution times for OLDA

Optimizations made possible by symbolic evaluation reduced the total execution time by up to 27 %. By resolving recurrences symbolic evaluation eliminates several dependences that cannot be handled by VFC without symoblic evaluation. By far the largest portion of the performance improvement obtained by our symbolic evaluation is caused by a reduced communication overhead. The ability to compare non-linear loop bound and index expressions allowed us to eliminate redundant communication which is not done by VFC without symbolic evaluation. Symbolic evaluation reduces only a small portion of OLDA's computation time. OLDA's computational overhead is of the order $O(N^4)$ and the communication overhead is of the order $O(N^2)$. 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.

5.4 Symbolic Evaluation to Optimize HNS

The HNS code (see also Section 5.1) is used for quantum mechanical calculations of solids as part of the WIEN95 software package [7]. HNS has 17 one-dimensional arrays, 14 two-dimensional arrays, 5 three-dimensional arrays, and 6 four-dimensional arrays. The computational overhead of HNS is of the order $O(N^2)$. All floating point operations are on double (eight bytes). All important arrays have been distributed CYCLIC [5, 38] due to non-linear array references and triangular loop iteration spaces.

Figures 12 and 13 plot the execution times of the HNS code for various problem (N) and machine sizes on a Meiko CS-2 for two different parallelization strategies. First, HNS has been parallelized and optimized by using VFC without symbolic evaluation (Figure 12). Due to many non-linear array references the compiler is unable to determine the absence of data dependences which results in sequentializing the core loops of HNS and as a consequence in a performance slowdown for increasing number of processors. Moreover, VFC inserts a mask for every array assignment statement in order to enforce the "owner computes rule". In addition, the compiler introduces for every right-hand-side array reference a function call that examines whether or not this reference must be communicated for a particular instance of the array assignment. The extra computational overhead implied by masks and function calls is the first order performance effect for smaller problem sizes. Communication becomes the predominated factor for increasing problem sizes.

Second, we used our symbolic evaluation to improve the parallelization and optimization techniques of VFC. Based on symbolic evaluation the compiler can parallelize all core loops of HNS without communication. Symbolic evaluation also enables the compiler to generate more efficient code as compared to compiling the program without symbolic evaluation. Instead of including function calls to ensure the "owner computes rule", loop bounds can now be parameterized [25] such that every processor is restricted to those loop iterations where local data (owned by the processor) is written. VFC with symbolic evaluation results in a performance gain (see Figure 13) that scales reasonable well for increasing processor numbers and, therefore, clearly outperforms the version without symbolic evaluation.

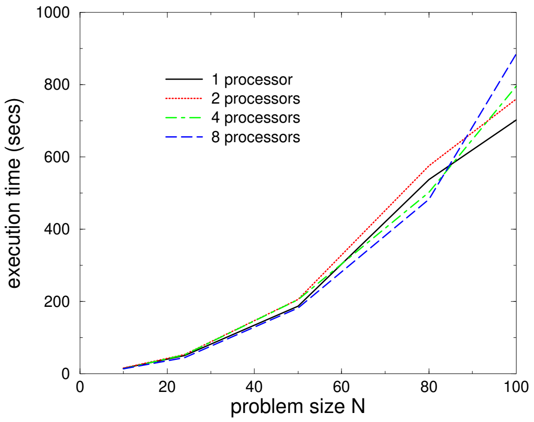

Figure 12: Execution times for parallel HNS code without symbolic evaluation for various problem and machine sizes on a Meiko CS-2.

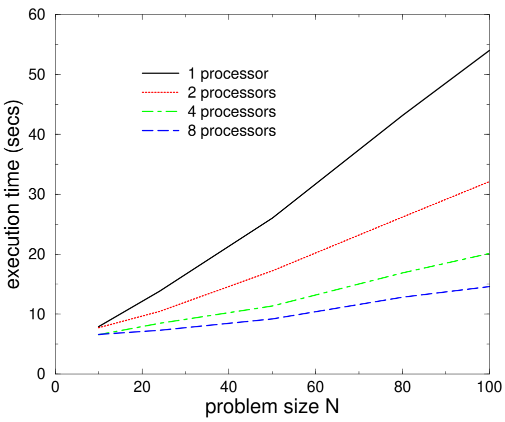

Figure 13: Execution times for parallel HNS code with symbolic evaluation for various problem and machine sizes on a Meiko CS-2.

The execution time and memory requirement to compile the HNS code by VFC without code generation and symbolic evaluation on a Sparc 10 workstation is 4.36 seconds and 2 MBytes, respectively. Whereas compiling HNS with symbolic evaluation takes 7.26 seconds and requires 3.1 MBytes memory space.

More experiments that show the effective usage of our symbolic evaluation framework can be found in [29, 8, 51].

6 Related Work and Discussion

P. and R. Cousot [18] pioneered abstract interpretation as a theory of semantic approximation for systematic data and control flow analysis of sequential programs. In abstract interpretation a program denotes computations in some universe of objects. According to [18] abstract interpretation can be considered as a mathematical theory to unify many program analyses. Moreover, abstract interpretation commonly achieves only useful results if substantially enriched for particular applications.

Despite its fundamentally incomplete results (for instance, due to approximations) abstract interpretation may allow to answer questions which tolerate an imprecise answer. Many program analysis systems that are used in modern parallelizing compilers are based on abstract interpretation. Blume and Eigenmann [12] use abstract interpretation to extract and propagate constraints about variable ranges in shared memory parallel programs. P. Cousot and N. Halbwachs [19] compute and propagate constraints through a sequential program based on abstract interpretation. Constraints between program variables are represented as n-dimensional convex polyhedrons and are restricted to linear inequality relationships. Neither interprocedural analysis nor effective techniques to eliminate redundant constraints are included in their approach. M. Haghighat and C. Polychronopoulos [33] describe a variety of symbolic analysis techniques based on abstract interpretation. The information of all incoming paths to a statement is intersected – at the cost of analysis accuracy – in order to reduce the amount of information to be maintained for each program statement. Their approach excludes an explicit model for arrays which prevents handling of array references as part of symbolic expressions (although array subscript expressions can be compared for dependence analysis). They do not propagate predicates guarding the conditional values of variables through a program.

The main differences between abstract interpretation and symbolic evaluation are as follows: First, symbolic evaluation precisely represents the values of program variables whereas abstract interpretation commonly approximates a program's computations. Second, path conditions or predicates guarding conditional variable values are not included in abstract interpretation. Third, applications of abstract interpretation are faced with a trade-off between the level of abstraction and the precision of the analysis. Although abstract interpretation may have a better computational complexity, its approximated information may not be accurate enough to be useful. Fourth, the underlying analysis representation is fundamentally different. Whereas we introduced program contexts (state, state condition, and path condition) as a novel basis for symbolic evaluation, abstract interpretation is primarily based on state (function from variables to abstract symbolic values) information only.

J. King [40] introduced symbolic execution of programs where a program is actually executed based on symbolic input. In [17] symbolic execution has been used for software specialization.

T. Cheatham, G. Holloway and J. Townley [15] discovered symbolic evaluation as a means of static program analysis to support verification of sequential EL1 programs. All of our program representations, techniques and algorithms are very different from their approach. They use a complex and costly program representation for their analysis. For instance, program statements, predicates of conditional assignments, and variable values are all separately represented and linked together via context numbers and time labels. The list of all values ever assigned to a variable is explicitly stored and retrieved via time stamps. Path conditions are not employed by their method. Our approach is integrated in a control flow graph and presents a compact association of each program statement with a state, state condition and path condition. Finally, we are applying symbolic evaluation to Fortran and use it to support parallelization of programs.

R. Cytron et al. [21] developed a new program representation called static single-assignment form (SSA) where each variable is written only once and which encodes DEF/USE information. Although it is not a target of our symbolic evaluation, DEF/USE information is implicitly contained in the program contexts. SSA form has been used by numerous researchers [30, 50] to statically examine programs at compile time.

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

Figure 14: Example 1

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

Figure 15: SSA or G-SSA form

read(A)
    [S_1={A=$\bigtriangledown$,B=b},P_1=true]
A=A+1
    [S_2=δ(S_1;A=$\bigtriangledown$+1),P_2=P_1]
B=A+2
    [S_3=δ(S_2;B=$\bigtriangledown$+3),P_3=P_1]

Figure 16: Symbolic evaluation form

R. Ballance et al. [3] extended SSA to gated SSA (G-SSA) form by introducing special gating functions to include predicates of conditional branches which is a major deficiency of SSA form. Extensive renaming of program variables in SSA and G-SSA form makes it more difficult to express symbolic values as functions over a program's problem size. Consider the code of Figure 14 where the value of B is computed as a function of the input value of A. SSA and G-SSA form (Figure 15) require explicit analysis to determine that the value of B depends on the input value of the read statement. By contrast the code based on symbolic evaluation (Figure 16) directly presents the value of B as a function of the input value $\bigtriangledown$ without additional analysis. For this reason G-SSA and SSA forms seem to be less amenable to performance oriented compiler optimizations and performance analysis where the resulting analysis information should be expressed as a function over problem sizes. For instance, it is very desirable to provide performance information as parameterized functions over the number of processors, array sizes and input data. These parameterized functions can then be employed to compute the optimal number of processors and array sizes to be used for a specific parallel machine [32, 14].

It has been shown [36] that G-SSA form can imply undefined and, therefore, redundant components in G-SSA gating functions. For instance, γ(X<N,X-2,γ(X<2*N,X-1,$\perp$)) is the G-SSA form of the value of X at the end of Q_4 in Figure 17 where $\perp$ is an undefined value implied by a path that can never be taken. Ballance et al. [3] need this information to drive the insertion of switches, creating a data-driven form of their PDW (Program Dependence Web). However, in most other G-SSA applications $\perp$ in a γ function implies redundant information and unnecessary overhead for analysis, simplification, and manipulation of such terms. Our symbolic evaluation technique does not imply redundancy for the context of Q_4 which is [S_4=δ(S_0;X=x_1),T_4=(x_0<N$\wedge x_1$=x_0-2)$\vee$($x_0 \geq$N$\wedge x_0$<2*N$\wedge x_1$=x_0-1),P_4=x_0<2*N]. The state immediately before Q_1 is S_0 and x_0 is the value of X in S_0. Although a method has been found to eliminate $\perp$ in G-SSA forms for well-structured IF-constructs [36], in general, it is not possible to eliminate all $\perp$ in G-SSA forms.

P. Havlak [36] used G-SSA form to build a global value graph and conduct interprocedural analysis to support symbolic dependence testing, array section analysis, and test elision. He does not address the issue of modeling recurrences.

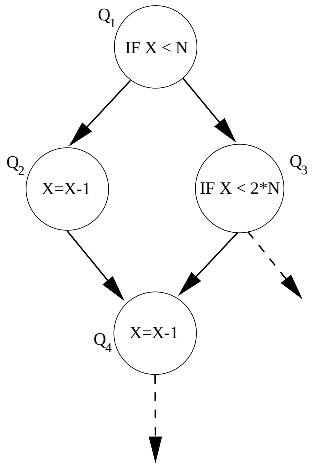

Figure 17: Example 2

P. Tu and D. Padua [53, 54] developed a system for computing symbolic values of expressions using a demand-driven backward analysis based on G-SSA form. Their backward-driven analysis 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. An additional algorithm and analysis is required to determine the path condition as G-SSA form does not explicitly incorporate this information. Symbolic evaluation directly represents path conditions in the context information. Furthermore, whereas in symbolic evaluation values of symbolic expressions and variables are determined based on local information of a program context, Tu's and Padua's method requires an additional backward-driven substitution algorithm based on G-SSA form. In contrast to our representation describing symbolic expression values based on G-SSA can imply significant redundancy. Consider the code of Section 3.2. Our symbolic evaluation represents the values of X and Y after statement Q_{10} by $[S_{10}{=}\delta(S_7;Y{=}2*\bigtriangledown_1),T_{10}{=}\gamma(\bigtriangledown_1{<}0;\ x_2{=}-2*\bigtriangledown_1;x_2{=}2*\bigtriangledown_1),P_{10}{=}\text{true}]$ which can be rewritten as $[S_{10}{=}\{X{=}x_2,Y{=}2*\bigtriangledown_1\},T_{10}{=}\gamma(\bigtriangledown_1{<}0;\ x_2{=}-2*\bigtriangledown_1;x_2{=}2*\bigtriangledown_1),P_{10}{=}\text{true}]$. G-SSA form with substitution yields y = $\gamma(\gamma(Y_0 < 0,\ 2{*}Y_0,\ Y_0) \geq 0,\ 2{*}\gamma(Y_0 < 0,\ 2{*}Y_0,Y_0),\ \gamma(Y_0 < 0,\ 2{*}Y_0,\ Y_0))$ and x=$\gamma(Y_0 < 0,\ {-2}{*}Y_0,\ 2{*}Y_0)$ where X_0 and Y_0, respectively, correspond to the variables of X and Y immediately after Q_1. Clearly, G-SSA form implies redundancy compared to symbolic evaluation as the gating function $\gamma(Y_0 < 0,\ 2{*}Y_0,Y_0)$ occurs three times and the gating condition $Y_0 < 0$ four times in the G-SSA form for X and Y.

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 (lambda calculus), 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 us to directly extract the system of recurrences from our data representation without additional analysis. Tu and Padua do not explain how symbolic expressions without gating functions are simplified and compared which is non-trivial. In [53, 54] they describe that two expressions e_1, e_2 without gating functions can be compared by their method iff

$e_1 - e_2$ is a constant. We developed highly efficient techniques [27, 28, 26] for comparing symbolic expressions, computing bounds of symbolic expressions, and simplifying systems of symbolic constraints which are considerably more effective. For instance, comparing symbolic expressions is done based on a system of constraints (contained in the context information). If the difference of $e_1 - e_2$ is not a constant it may still be possible to determine a useful result based on the constraints given and the underlying algorithm employed. In addition, their approach to compare symbolic expressions with gating functions can be substantially improved. For instance, for the purpose of comparing two expressions their techniques transform $i > \gamma(i > j, j, i-1)$ into $i > j$ and thus do not incorporate the predicate in the γ function to determine that the result of this expression is *true*. Our techniques [27, 28, 26] can easily determine that the result of this expression is *true* by considering the condition of the gamma function. Moreover, in Tu's and Padua's system expressions of the form $e(c)$ – expression value e at a statement with path condition c – are rewritten to e if e does not contain gating functions. Again important information is lost by their expression manipulation system which can imply critical approximations that may no longer be useful. They do not state how to handle division, multiplication, and exponentiation as part of operations defined over symbolic expressions which has been addressed by our expression manipulation and simplification system [27, 28, 26]. Tu's and Padua's approach avoids modeling of procedures by full in-line expansion.

M. Gerlek, et al. [30] developed a recurrence solver based on a classification system of linear induction variables as part of the Nascent compiler. We partially employ their techniques for resolving linear, polynomial and geometric recurrence variables. Their method uses SSA form without employing path conditions and propagation of constraints about program variables. Gerlek et al. cannot handle loops with multiple exits, conditional recurrences, array references as part of recurrences (for instance, I = I + A(I)), and indirect array references. They also have problems with interprocedural analysis. For instance, variables as procedure parameters cannot be handled in recurrences. An important difference between their work and our approach is the way in which recurrences are detected and represented. Our approach enables us to represent all recurrence systems implied by recurrence variables. Gerlek et al. classify recurrences without representing them. They couple detecting and resolving recurrences which makes their system more difficult to extend for new recurrence classes. Except for simple cases Gerlek does not state how to determine the monotonic behavior for unresolved variables (no closed form can be found) whereas we have developed a very effective algorithm [27] for this problem.

There is a variety of symbolic analyses that focus on array region analysis. For instance, W. Pugh and D. Wonnacott [48] use a set of constraints to describe linear array data flow problems and solve them by the Fourier-Motzkin variable elimination method. B. Creusillet and F. Irigoin [20] introduced an interprocedural array flow analysis for array privatization. Linear array subscript expressions and array regions are described by a single convex region. J. Gu, et al. [31] substantially improved array region analysis by conditional array regions which models control flow of programs and is used for array privatization. They also describe array regions by a set of convex regions which is more precise than using only a single convex region. This analysis can be applied both intra- and inter-procedurally. We believe that our symbolic evaluation framework can support all of these techniques by collecting constraints about program unknowns throughout the program.

Our symbolic evaluation framework has a promising potential for sparse representation of analysis information and demand driven analysis. For instance, analysis information can be stored at selected program statements such as at the entry and exit of loops and procedures. Analysis can be restricted to code segments, by assigning unknown symbolic values to all variables at the entry of the code segment selected. Modeling arrays, recurrences, procedures, etc. can be selectively turned on and off. Furthermore, if the values of a subset of variables 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. Finally, analysis information is implemented as a linked list of contexts. Only the semantic information implied by a statement Q is actually included in the context of Q. A link is included to the predecessor contexts that hold the remaining information valid at Q.

7 Conclusion

Existing compilers are unable to effectively parallelize and optimize programs on today's multiprocessor architectures primarily due to the lack of advanced and accurate compiler analysis. This paper introduces a novel and unified compile-time framework to support software tools in parallelizing and optimizing programs. Our framework is based on symbolic evaluation and combines both data and control flow analysis. The most important features and contributions of our approach are as follows:

- novel representation of analysis information, the *program context*, which includes three components: variable values, assumptions about and constraints between variable values, and path condition
- first order logic representation of program contexts (existing software for standard symbolic manipulation can be used)
- program context components can be separately accessed at well-defined program points
- single traversal of the input program is sufficient to compute all program contexts
- linear and non-linear symbolic expressions are supported
- representation of computations as symbolic expressions defined over the program's problem size (supports performance driven optimization)
- accurately modeling of assignment and input/output statements, branches, loops, recurrences, arrays (including indirect accesses) and procedures
- new techniques to simplify program contexts

We do not know of any other system that models a similar large class of program constructs based on a comprehensive and unified analysis representation that explicitly captures exact information about variable values, assumptions about and constraints between variable values, path conditions, recurrence systems, and side-effect information of procedure calls. Although we have examined our framework for Fortran programs, the underlying techniques are equally applicable to any similar imperative programming language.

Empirical results based on a variety of benchmark codes demonstrate among others the need for advanced analyses to handle non-linear and indirect array references, procedure calls, and multiple exit loops.

We have implemented a prototype of our symbolic evaluation framework which is used as part of a parallelizing compiler for distributed memory architectures. Our system has been used to support symbolic dependence testing and various optimizations (including communication vectorization and elimination of redundant communication) which can result in significant performance improvements of parallel programs.

Symbolic evaluation is also being used as part of P^3T [24, 25], a state-of-the-art performance estimator, in order to estimate the work distribution [26] of parallel programs as a parameterized function defined over unknown problem sizes.

Currently, we are extending several compiler optimizations for distributed memory architectures to exploit the prototype implementation of our symbolic evaluation framework under VFC. We are also working on modeling recursive procedures and adding alias and pointer analysis. In the future, we plan to conduct more extensive experiments, to study the performance impact of other optimizations captured by our framework such as reducing the number of cache misses and optimizing parallel input/output. Furthermore, our techniques are not restricted to compilers and tools for parallel architectures. They can be equally important to compiler analysis and tools for sequential architectures, program testing [40], program verification [15], software specialization [17], software reuse [16], pattern matching and concept comprehension [23], etc. We plan to investigate the application of our framework to some of these computer science areas.

References

[1] C. Addison, J. Allwright, N. Binsted, N. Bishop, B. Carpenter, P. Dalloz, D. Gee, V. Getov, A. Hey, R. Hockney, M. Lemke, J. Merlin, M. Pinches, C. Scott, and I. Wolton. The GENESIS distributed-memory benchmarks. Part 1: Methodology and general relativity bnechmark with results for the SUPRENUM computer. *Concurrency Practice and Experience (Ed. Geoffrey Fox), Wiley*, 5(1):1–22, 1993.

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

[3] 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.

[4] Denis Barthou, Jean-Francois Collard, and Paul Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40(2):210–226, February 1997.

[5] S. Benkner. VFC: The Vienna Fortran Compiler. *Scientific Programming, IOS Press, The Netherlands*, 7(1):67–81, 1999.

[6] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, pages 3(3): 5–40, 1989.

[7] P. Blaha, K. Schwarz, P. Dufek, and R. Augustyn. Wien95, a full-potential, linearized augmented plane wave program for calculating crystal properties. Institute of Technical Electrochemistry, Vienna University of Technology, Vienna, Austria, 1995.

[8] J. Blieberger, T. Fahringer, and B. Scholz. Symbolic Cache Analysis for Real-Time Systems. *Real-Time Systems, Kluwer Academic Publishers, Boston, USA, accepted for publication*, 2000.

[9] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the perfect benchmark programs. *IEEE Transactions on Parallel and Distributed Systems*, pages 3(6):643–656, November 1992.

[10] W. Blume and R. Eigenmann. An Overview of Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks. In *Proceedings of the 1994 International Conference on Parallel Processing*, St. Charles, IL, 1994.

[11] W. Blume and R. Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. CSRD (Center for Research on Parallel Computation) Report No. 1345, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, April 1994.

[12] 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.

[13] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David A. Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The Next Generation in Parallelizing Compilers. . In *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing, Ithaca, New York*, pages 10.1–10.18, August 1994.

[14] B. Chapman, T. Fahringer, and H. Zima. *Automatic Support for Data Distribution*, pages 184–199. Springer Verlag, Lecture Notes in Computer Science: Languages and Compilers for Parallel Computing, 6th International Workshop, (Ed.) U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Portland, Oregon, Aug. 1993.

[15] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic Evaluation and the Analysis of Programs. *IEEE Transactions on Software Engineering*, 5(4):402–417, 1979.

[16] Aniello Cimitile, Andrea De Lucia, and Malcom Munro. Qualifying Reusable Functions Using Symbolic Execution. In *Proceedings of the 2nd Working Conference on Reverse Engineering, IEEE Comp. Society*, Toronto, Canada, August 1995.

[17] Alberto Coen-Porisini, Flavio De Paoli, Carlo Ghezzi, and Dino Mandrioli. Software Specialization Via Symbolic Execution. *IEEE Transactions on Software Engineering*, 17(9):884 – 899, Sept. 1991.

[18] 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 238 – 252, Los Angeles, CA, January 1977.

[19] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84 – 97, Tucson, AZ, January 1978.

[20] Beatrice Creusillet and François Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, December 1996.

[21] 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.

[22] R. Cytron, J. Ferrante, and V. Sarkar. Experience using control dependence in PTRAN. In D. Gelernter, A. Nicolau, and D. Padua (Editors), editors, *Languages and Compilers for Parallel Computing. The MIT Press*, 1990.

[23] B. DiMartino. Algorithmic Concept Recognition Support for Automatic Parallelization: A Case Study for Loop Optimization and Parallelization. *Journal of Information Science and Engineering, Special Issue on Compiler Techniques for High-Performance Computing*, to appear in March 1998.

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

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

[26] 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. IEEE Computer Society Press.

[27] T. Fahringer. Efficient Symbolic Analysis for Parallelizing Compilers and Performance Estimators. *Journal of Supercomputing, Kluwer Academic Publishers*, 12(3):227–252, May 1998.

[28] T. Fahringer. Symbolic Analysis Techniques for Program Parallelization. *Journal of Future Generation Computer Systems, Elsevier Science, North-Holland*, 13(1997/98):385–396, March 1998.

[29] T. Fahringer and E. Mehofer. Buffer-Safe and Cost-Driven Communication Optimization. *Journal of Parallel and Distributed Computing, Academic Press*, 57(1):33–63, April 1999.

[30] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 17(1):85–122, January 1995.

[31] Junjie Gu, Zhiyuan Li, and Gyungho Lee. Experience with efficient array data flow analysis for array privatization. *ACM SIGPLAN Notices*, 32(7):157–??, July 1997.

[32] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, 1992.

[33] M. Haghighat and C. Polychronopoulos. Symbolic Analysis for Parallelizing Compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 18(4):477–518, July 1996.

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

[35] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler . *IEEE Computer*, 29(12):84 – 89, December 1996.

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

[37] M. S. Hecht. *Flow Analysis of Computer Programs* . Elsevier, North-Holland, 1977.

[38] High Performance FORTRAN Language Specification. Technical Report, Version 2.0.δ, Rice University, Houston, TX, October 1996.

[39] K. Kennedy, K. McKinley, and C.-W. Tseng. Interactive parallel programming using the ParaScope editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[40] James C. King. Symbolic Execution and Program Testing . *Communications of the ACM*, 19(7):385–394, July 1976.

[41] George S. Lueker. Some Techniques for Solving Recurrences. *ACM Computing Surveys*, 12(4):419–435, December 1980.

[42] Vadim Maslov. Delinearization: An efficient way to break multiloop dependence equations. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27, pages 152–161, New York, NY, July 1992. ACM Press.

[43] 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.

[44] M. Metcalf and J. Reid. *Fortran 90/95 explained.* Oxfor Science Publications, 1996.

[45] C.D. Polychronopoulos, M. Girkar, M.R. Haghighat, C.L. Lee, B. Leung, and D. Schouten. Parafrase-2: An Environment for Parallelizing, Paritioning, Synchronizing, and Scheduling Programs on Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II, pages 39 – 48. Pennsylvania State University Press, August 1989.

[46] Aleksandar Pozgaj and Thomas Fahringer. P3T+: A Performance Estimator for Parallel and Distributed Systems. Institute for Software Science, University of Vienna, AURORA Technical Report TR2000-03, February 2000.

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

[48] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. *Lecture Notes in Computer Science*, 768:546–??, 1994.

[49] William Pugh and David Wonnacott. Nonlinear array dependence analysis. In *In 3rd Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Troy, New York, May 1995.

[50] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 12 – 27, San Diego, CA, January 1988.

[51] Bernhard Scholz, Johann Blieberger, and Thomas Fahringer. Symbolic pointer analysis for detecting memory leaks. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 104–113, Boston, MA, USA, January 2000. ACM Press.

[52] 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 *Proc. of the 9th ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[53] P. Tu and D. Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. In *9th ACM International Conference on Supercomputing*, pages 414–423, Barcelona, Spain, July 1995.

[54] Peng Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis.* PhD thesis, University of Illinois at Urbana-Champaign, 1995.

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

[56] D. Wonnacott. *Constraint-Based Array Dependence Analysis.* PhD thesis, University of Maryland, 1995.

[57] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series. Addison-Wesley, 1990.