Compilation of Constraint Programs with Non-Cyclic and Cyclic Dependencies to Procedural Parallel Programs

Ajita John
Bell Labs Innovations
Lucent Technologies
101 Crawfords Corner Rd, 4E-622
Holmdel, NJ 07733
ajita@research.bell-labs.com

J.C. Browne
Dept. of Computer Sciences
University of Texas
Taylor Hall 2.124
Austin, TX 78712
browne@cs.utexas.edu

Abstract

This paper reports on a compiler for translation of constraint specifications into procedural parallel programs. A constraint program in our system consists of a set of constraints and an input set containing a subset of the variables appearing in the constraints. The compiler described in this paper successfully compiles a substantially larger class of constraint specifications to efficient programs than did its predecessors. In particular the compiler has been extended to generate processor and memory efficient programs for cyclic constraints which can be resolved by computational relaxation methods. The paper first details the basic compilation process for non-cyclic constraints. It then describes the additional steps in the compilation process which enable resolution of cyclic constraints to iterative computational processes and illustrates the process using derivation of a parallel program for solution of the Laplace equation as the example.

1 Introduction and Overview

Constraint systems are a natural means of expressing a family of computations. Specification of some subset (inputs) of the type instances (variables) appearing in the constraint system leads to a constraint program which can be evaluated to determine the values for the non-input variables. Constraint programs are attractive as a representation to be compiled for parallel execution since constraint systems do not specify control flow and thus allow the compiler full freedom to optimize the compiled program to a specific target execution environment. But previous evaluation methods for constraint systems have largely employed interpretive evaluation methods [20, 30] and have been quite slow in execution. Also, previous use of compilation of constraint systems [12, 11] to procedural programs have not considered parallelism.

This paper defines and describes a compilation process which translates constraint programs specifying matrix computations to efficient parallel programs. A constraint program is a system of constraints and a specification of some subset of the type instances (variables) appearing in the system of constraints as being known. The compilation process converts the constraint program to a dependence graph where the nodes implement operations over the variables appearing in the constraint system. The dependence graph is then mapped to a procedural program for execution. This program computes the variables designated as unknown. In the case where the dependence graph has no cycles mapping of the dependence graph to an efficient procedural program is straightforward. But the compilation process leading to dependence graphs naturally produces single assignment variables. Resolution of cyclic dependence graphs of single assignment variables to efficient parallel programs requires additional phases of compilation including both introduction of iterative algorithms for resolution of cyclic dependence graphs and translation of the single assignment variables in the cyclic portion of the dependence graph to be mutable variables.

Previous papers [15, 16] gave preliminary descriptions of the compilation process through generation of procedural parallel programs for non-cyclic dependence graphs. The compiler has now been extended to generate efficient parallel programs for cyclic dependence graphs which can be resolved by iterative (relaxation) algorithms. This paper gives a complete description of the compilation process through development of the dependence graph and defines and describes the additional compilation phases necessary to generate efficient parallel executables for cyclic dependence graphs. Solution of the Laplace equation is used as the illustration and the vehicle for measurement of performance.

Section 2 defines the constraint specification language used as the basis for compilation. Section 3 defines the compilation process through generation of dependence graphs and mapping of non-cyclic dependence graphs to procedural parallel programs. Section 4 defines the additional compilation phases needed to resolve cyclic dependence graphs. A constraint specification does not contain the implicit information about the target execution environment present in most procedural language programs. But this information can be specified to the compiler separately from the program. Section 5 gives provisions for describing the target execution environment to the compiler. Section 6 gives the execution behavior of the parallel program for the Laplace equation. Section 7 is a review of related research. Conclusions and directions for future work are presented in Sections 8 and 9, respectively.

2 Language Description

This section describes the components of our programming system. It explicates the type system, the rules for expressing constraints, and the structure of a complete program in the system. The section concludes with the constraint specifications for a few sample programs. The notations used are similar to those in the C programming language.

2.1 Type System

Our approach relies on a rich hierarchical type system where types at higher levels are constructed from those at lower levels in the hierarchy. The schematic for the layout of the type system is shown in Figure 1. The lowest level of the type hierarchy contains integers, reals, and characters. At the next level of the hierarchy are arrays to which we give semantic structure to construct the base matrix types, which define matrices of scalar elements. In addition to dense matrices, the base matrix type currently supports specialized matrix types such as lower and upper triangular enabling the flexibility to invoke specialized algorithms based on the structure of the matrix for the operations defined on the matrix subtypes. Other specialized types can also be easily incorporated. At the highest level of the type system are hierarchical matrices, whose individual elements are matrices.

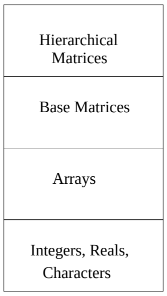


Figure 1: The Type System Layout

The entities in the type system are integers, reals, characters, arrays, base matrices and hierarchical matrices. The operators of addition, subtraction, multiplication, and division are defined on integers and reals. The operator sets for characters and arrays are empty. The base matrix type has associated operators of addition, subtraction, scalar multiplication, matrix multiplication and inverse defined for matrices over integers and reals. The operator set for hierarchical matrices is empty since operations are only defined on the blocks which compose it.

2.2 Expressions

Expressions can be formed by applying defined operators on instances of types in the type system and through calls to library and user-defined functions. Functions must have defined inverses, otherwise only a limited form of compilation can be done. Examples of library functions are mathematical functions such as *sqrt* and *sqr*.

In addition to defined applications of operators, expressions of the following form using *indexed operators* are allowed.

$$< op > FOR \ (< index > < b1 > < b2 >) \ \{ \ X \ \}$$

An indexed operator applies a binary operator *op* to an expression X through a range of values $b1 \ldots b2$ for an integer variable *index*. The values of *b1* and *b2* have to be bounded at compile-time. An indexed operator allows for the compact representation of expressions and is useful in large systems. For example, the construct

$$+ \ FOR \ (i \ 1 \ 3) \ \{ \ + \ FOR \ (j \ 1 \ i) \ \{ \ A[i][j] \ \}\}$$

expresses the sum of the lower partitions of a 3×3 matrix A:

$$\begin{array}{l} A[1][1]+ \\ A[2][1] + A[2][2]+ \\ A[3][1] + A[3][2] + A[3][3] \end{array}$$

The "+" operator refers to scalar addition or matrix addition depending on whether A is a base matrix or a hierarchical matrix, respectively.

2.3 Constraints

Rules which govern the specification of constraints are enumerated in this section. In designing these rules we have the motivation of capturing the entire set of constraints a programmer would wish to impose upon a system. Rule 1 allows for the expression of simple conditions, using relational operators, on expressions involving type instances. Rule 2 allows propositional connectives AND/OR/NOT to be applied on constraints to express conditions using compositions of constraints. Rule 3 is a generalization of Rule 2 through which large compositions of constraints using AND/OR operators can be compactly represented. Rule 4 introduces modularity to enable large bodies of constraints to be replaced by calls to reusable modules.

Rule 1:

(i) $X_1 \ \mathcal{R} \ X_2$, is a constraint,
where $\mathcal{R} \in \{ <, <=, >, >=, == , != \}$,
X_1, X_2 are expressions over instances of scalar types.

(ii) $M_1 == M_2$ is a constraint,
where M_1, M_2 are expressions involving matrices and matrix operators.

Rule 1(ii) allows a mix of scalars and matrices. Although we do not currently allow relations of the form $M_1 < M_2$, these could be easily defined to extend expressibility.

Rule 2:

(i) A AND/OR B
(ii) NOT A
are constraints, where A and B are constraints.

Rule 3: Constraints over *indexed sets* have the form:

AND/OR FOR (<index> <b1> <b2>) { $A_1, A_2, \ldots, A_n$ }

An indexed set groups a set of constraints { $A_1, A_2, \ldots, A_n$ } to be connected by an AND/OR connective through a range of values $b1 \ldots b2$ for an integer variable *index*. The values of $b1$ and $b2$ have to be bounded at compile-time. This condition will be relaxed in later versions of the compiler. Indexed sets allow for the compact representation of large constraint systems.

An application of Rule 3 is

$$\text{AND FOR (i 1 2) } \{\ A[i] == A[i-1],\ B[i+1] == A[i]\ \}.$$

This construct represents the constraint

$$A[1] == A[0] \text{ AND } A[2] == A[1] \text{ AND } B[2] == A[1] \text{ AND } B[3] == A[2].$$

Another application of Rule 3 is

$$\text{OR FOR (i 1 2) } \{\ A[i] == 0,\ B[i+1] == A[i]\ \}.$$

This construct succinctly captures the constraint

$$A[1] == 0 \text{ OR } A[2] == 0 \text{ OR } B[2] == A[1] \text{ OR } B[3] == A[2].$$

Rule 4: Calls to user-defined constraint modules are constraints. They have the form:

$$< ModuleName > (P_1, P_2, \ldots, P_n)$$

where $ModuleName$ is the name of a defined constraint module (Section 2.4 describes definition of constraint modules), which encapsulates constraints between its formal parameters, local variables, and global variables within its scope. $P_1, P_2, \ldots, P_n$ are the actual parameters for the constraint module call.

Constraints constructed from applications of Rule 1 are referred to as *simple constraints*, which form the building blocks for constraints constructed from applications of Rules 2-4. Both linear and non-linear constraints can be expressed using these rules. Each rule has an analog in the procedural world - Rule 1 maps to simple conditionals and simple computations such as assignments, Rule 2 to sequencing and conditional statements, Rule 3 to loops and Rule 4 to procedures.

Our basic compilation algorithm can be applied to both linear and non-linear constraints without cycles. Cyclic constraints such as simultaneous systems of equations cannot be resolved by the basic compilation algorithm. The extended compilation process described in Section 4 generates programs which resolve cyclic systems through iterative solution algorithms.

The implemented compiler handles all types of linear and non-linear constraints where the initialization results in all non-linear terms being known at runtime. A detailed discussion is given in Section 3. This restriction could be alleviated by an extension to the compiler to incorporate higher order solvers for unknown non-linear terms into the compiled program.

All invoked functions must have defined inverses, otherwise compilation is only successful for cases where all parameters of the functions are known at runtime.

Constraint systems involving inequalities must be cast by the compilation process to conditional expressions where the values of all of the variables are known at runtime.

2.4 Program Structure

A program in our system has the following constituents.

(i) Program name.

(ii) Global variable declarations: list of global variables with associated types.

(iii) Global input variables: input set $\mathcal{I}$.

(iv) User-defined function signatures: signatures of C functions, which may be invoked in expressions. For example, the user-defined function *max* in the constraint $max(a, b) < 5$ may have the function signature *int max*(*int x*, *int y*). The actual function definitions are provided in a separate file which is linked with the compiled executable for the constraint program.

(v) Constraint module definitions: module name, formal parameters and their types, local variable declarations, and a *constraint module body* constructed from applications of Rules 1-4 in Section 2.3. Constraints within a module can involve local variables, formal parameters, and global variables. Name scoping and type matching are similar to those implemented for procedures in C programs.

(vi) *Main body* of the program: constraints on global variables expressed through applications of Rules 1-4 in Section 2.3.

2.5 Sample Programs

This section presents four example programs written using the language constructs presented in Section 2.3. While the first one is a toy example, the others have been successfully executed with good performance results.

2.5.1 The Quadratic Equation Solver

Figure 2 shows a constraint specification for the non-complex roots of a quadratic equation $ax^2 + bx + c == 0$. *sqr*, *sqrt*, and *abs* are library functions. The main body specifies the conditions on values of the roots $r1$ and $r2$ when $a == 0$ and when $a \, != 0$. The condition on the values of $r1$ and $r2$ when $a \, != 0$ is expressed by a call to a constraint module *DefinedRoots*. The definition for the module expresses the relationship between the parameters $a, b, c, r1$, and $r2$ in the event that the discriminant (t) is greater than or equal to 0. The specification can be enhanced for imaginary roots. The input set could be $\{a, b, c\}$, $\{a, b, r1\}$, or $\{a, b, r2\}$. Other input sets will not lead to dependence graphs through the compilation process described in Section 3.

2.5.2 The Block Triangular Solver(BTS)

The example chosen is the solution of the $AX == B$ linear algebra problem for a known lower triangular matrix A and vector B. The matrix and vectors can be divided into blocks as shown in Figure 3. $S_0 \ldots S_3$ represent lower triangular sub-matrices along the diagonal of A and $M_{10}, M_{20}, \ldots M_{32}$ represent dense sub-matrices within A.

A constraint specification (excluding declarations) for a problem instance split into 4 blocks is shown in Figure 4. The input set can be chosen as $\{ \ S_0, \ldots, S_3, M_{10}, M_{20}, \ldots, M_{32}, B_0, \ldots B_3 \ \}$. The constraint specification closely

```
PROGRAM QUAD_ROOTS

VAR real a, b, c, r1, r2;    /* Global Variables */
INPUTS a, b, c;    /* Input Variables */


/* Constraint Module */
DefinedRoots(a:real; b:real; c:real; r1:real; r2:real)
real t;    /* Local Variable */

/* Constraint Module Body */
t == sqr(b) - 4 * a * c AND t >= 0 AND
2 * a * r1 == (-b + sqrt(abs(t))) AND 2 * a * r2 == -(b + sqrt(abs(t)))

/* Main Body */
a == 0 AND b! = 0 AND r1 == r2 AND b * r1 + c == 0
OR
a! = 0 AND DefinedRoots(a, b, c, r1, r2)
```

Figure 2: Constraint Specification for the Quadratic Equation Solver

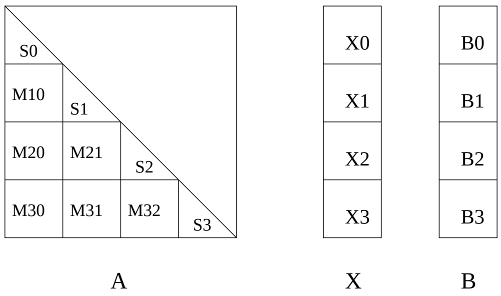


Figure 3: BTS: Partitioned Lower Triangular Matrix A, Vectors X and B

imitates the mathematical representation of the partitioned version of the problem $AX == B$.

Using an indexed set of constraints and an indexed operator, an alternate compact program is shown in Figure 6 using partitions on A as shown in Figure 5. The input set can be chosen as $\{ A, B \}$ to yield a solution for X. Alternatively, $\{ A, X \}$ can be chosen as the input set to yield a solution for B. Choosing $\{ B, X \}$ as the input set will not yield a solution for A through the compilation process described in Section 3.

2.5.3 The Block Odd-Even Reduction Algorithm(BOER)

This is an example deliberately chosen by us to demonstrate that constructing the constraint specification by inspecting a given algorithm and processing it through the compiler extracts the original algorithm if an appropriate input set is chosen (shown later in the paper). Consider a linear tridiagonal system $Ax == d$ where

$$A = \begin{bmatrix} B & C & 0 & 0 & \dots & 0 & 0 & 0 \\ C & B & C & 0 & \dots & 0 & 0 & 0 \\ 0 & C & B & C & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & C & B & C \\ 0 & 0 & 0 & 0 & \dots & 0 & C & B \end{bmatrix}$$

PROGRAM BTS_1

($S_0 * X_0 == B_0$ *AND*
$M_{10} * X_0 + S_1 * X_1 == B_1$ *AND*
$M_{20} * X_0 + M_{21} * X_1 + S_2 * X_2 == B_2$ *AND*
$M_{30} * X_0 + M_{31} * X_1 + M_{32} * X_2 + S_3 * X_3 == B_3$)

Figure 4: Constraint Specification for the BTS System

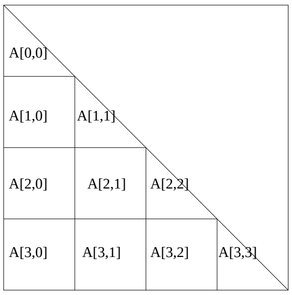


Figure 5: BTS: Partitioned Lower Triangular Matrix A

is a block tridiagonal matrix and B and C are square matrices of order $n \geq 2$. It is assumed that there are M such blocks along the principal diagonal of A, and $M = 2^k - 1$, for some $k \geq 2$. Thus, $N = Mn$ denotes the order of A. It is assumed that the vectors x and d are likewise partitioned, that is, $x = (x_1, x_2, \ldots, x_M)^t$, $d = (d_1, d_2, \ldots, d_M)^t$, $x_i = (x_{i1}, x_{i2}, \ldots, x_{in})^t$, and $d_i = (d_{i1}, d_{i2}, \ldots, d_{in})^t$, for $i = 1, 2, \ldots, M$. It is further assumed that the blocks B and C are symmetric and commute ($B \times C == C \times B$).

A version of the parallel algorithm given in [18] has a reduction phase in which the system is split into two subsystems: one for odd-indexed (reduced system) and another for even-indexed (eliminated system) terms. The reduction process is repeatedly applied to the reduced system. After $k-1$ iterations the reduced system contains the solution for a single term. The rest of the terms can be obtained by back-substitution.

The constraint specification (excluding declarations) for the problem is shown in Figure 7. The variable names BP, CP, and dP correspond to the indexed terms B, C, and d in [18] and are examples of the hierarchical data type in our system (elements of BP, CP and dP are matrices). The inputs to the system are $BP[0]$, $CP[0]$ and $dP[i][0]$, $1 \leq i \leq M$. *pow* is a C function implementing the arithmetic power function. The constraints have been constructed by mapping assignments (=) in the algorithm [18] to equality (==) in the constraint specification and loops to indexed sets. Also, the three constraints corresponding to the reduction, single-solution, and back-substitution phases have been reordered to demonstrate the independence of a constraint specification on the expressed order of the constraints.

PROGRAM BTS_2

AND FOR (i 0 3) { + *FOR (j 0 i)* { $A[i][j] * X[j]$ } $== B[i]$ }

Figure 6: Alternate Notation for the Constraint Specification for the BTS System

```
PROGRAM BOER

/* SINGLE-SOLUTION PHASE */
BP[k-1] * x[pow(2,k-1)] == dP[pow(2,k-1)][k-1]

AND

/* REDUCTION PHASE */
AND FOR (j 1 k-1) {

    2 * CP[j-1] * CP[j-1] == BP[j] + BP[j-1] * BP[j-1] ,

    CP[j] - CP[j-1] * CP[j-1] == 0 ,

    AND FOR (i 0 pow(2,k-j)-2) {
        CP[j-1] * ( dP[i*pow(2,j) + pow(2,j-1)][j-1] +
                        dP[i*pow(2,j) - pow(2,j-1)][j-1] ) ==
        dP[i*pow(2,j)][j] + BP[j-1] * dP[i*pow(2,j)][j-1] }}

AND

/* BACK-SUBSTITUTION PHASE */
AND FOR (j k-1 1) {

    AND FOR (i 0 pow(2,k-j)-1) {

        CP[j-1] * ( x[(i+1)*pow(2,j)] + x[i*pow(2,j)] ) ==
            dP[(i+1)*pow(2,j)-pow(2,j-1)][j-1] -
            BP[j-1] * x[(i+1)*pow(2,j)-pow(2,j-1)] }}
```

Figure 7: Constraint Specification for the BOER System

2.5.4 The Laplace Equation

Consider the Laplace equation for a 4-point stencil on an $N \times N$ grid indexed by $(0..N-1)(0..N-1)$ as shown in Figure 8 for $N = 10$. The boundary elements (shaded) are inputs to the problem. Every element not on the boundary is the average of its four neighbors. Since there are $(N-2) \times (N-2)$ non-boundary elements, there are $(N-2) \times (N-2)$ constraints to satisfy.

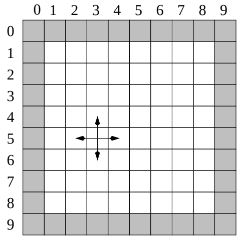


Figure 8: The Laplace Equation Grid

A constraint specification (excluding declarations) for the problem is shown in Figure 9. x is an array of dimensions ranging in $(0..N-1,\ 0..N-1)$. The simple constraint 4 * x[i][j] - x[i-1][j] - x[i+1][j] == x[i][j-1] + x[i][j+1] in the specification can be expressed in many equivalent representations including 4 * x[i][j] == x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1]. This program is an example of constraints over scalar elements of a structured type.

```
PROGRAM LAPLACE

    AND FOR (i 2 N-2) {
        AND FOR (j 2 N-2) {
            4 * x[i][j] - x[i-1][j] - x[i+1][j] == x[i][j-1] + x[i][j+1] } }
```

Figure 9: Constraint Specification for the Laplace Equation System

3 Compilation

The constraint compiler transforms a textual program given in the format outlined in Section 2.4 to a sequential or parallel C program for a selected architecture such as a Sparc, Cray, PVM, or MPI configuration. This section discusses the basic compilation algorithm [15] which handles constraint systems without cycles. We discuss an enhancement to the basic algorithm for constraint systems with cycles in Section 4. The compilation algorithm consists of the following phases.

Phase 1. The textually expressed constraint specification is transformed to an undirected graph representation as for example given by Leler [20].

Phase 2. A depth-first traversal algorithm transforms the undirected graph to a directed graph.

Phase 3. With a set of input variables $\mathcal{I}$, the directed graph is traversed in a depth-first manner to map the constraints in the constraint specification to conditionals and computations for nodes of a generalized dependence graph.

Phase 4. Specifications of the execution environment are used to optimally select the communication and synchronization mechanisms to be used by CODE [24].

Phase 5. The dependence graph is mapped to the CODE parallel programming environment to produce sequential and parallel programs in C as executable for different parallel architectures.

Phases 1-5 are described in detail in the rest of this section. Phases 1-3 will be illustrated through the quadratic equation solver introduced in Figure 2.

3.1 Phase 1: Generation of Constraint Graphs

A parser transforms the textual source program to a source graph for the compiler. Starting from an empty graph, for each application of Rules 1-4 in Section 2.3 an undirected constraint graph can be constructed by adding appropriate nodes and edges to the existing graph. For each instance of a simple constraint (Rule 1) a node is created with the constraint attached to it as shown in Figure 10(a). For each application of Rule 2 (A AND/OR B, NOT A) the graph is expanded as shown in Figures 10(b),(c). Figure 11(a) illustrates the expansion of the constraint graph for each application of Rule 3 (AND/OR FOR (<index> <b1> <b2>) { $A_1, A_2, \ldots, A_n$ }). For each application of Rule 4 ($< ModuleName >$ $(P_1, P_2, \ldots, P_n)$) a node is created with the constraint module call and the actual parameters attached to it as shown in Figure 11(b).

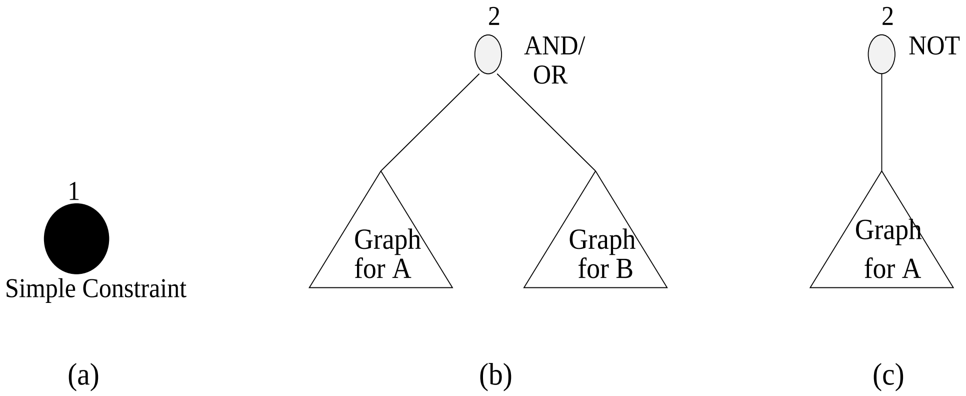


Figure 10: Constraint Graphs for (a) Rule 1 (b),(c) Rule 2

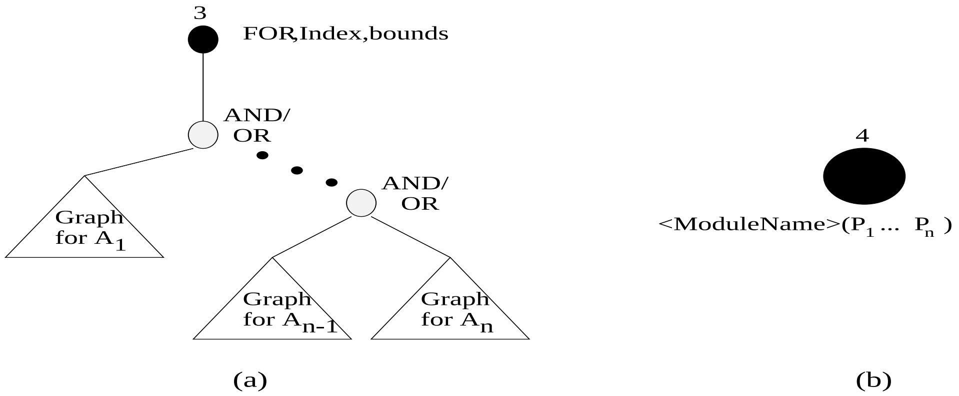


Figure 11: Constraint Graphs for (a) Rule 3 (b) Rule 4

The different kinds of nodes in the constraint graph are (i) *simple constraint* nodes (1 in Figure 10(a)) (ii) *operator* nodes corresponding to AND/OR/NOT connectives (2 in Figures 10(b),(c)), (iii) *for* nodes corresponding to indexed sets (3 in Figures 11(a)), and (iv) *call* nodes corresponding to constraint module calls (4 in Figure 11(b)). The index and its range information for an indexed set are attached to the corresponding *for* node.

A constraint graph is constructed for the main body and for each of the constraint module bodies giving rise to a set of constraint graphs. Each graph is constructed in a hierarchical fashion. *Simple constraint* and *call* nodes occur at lower levels, and *operator* and *for* nodes connect one or more subgraphs at higher levels. There will be a single node at the highest level. The constraint graph obtained for a particular constraint specification is unique. The constraint graphs for the quadratic equation solver are shown in Figure 12.

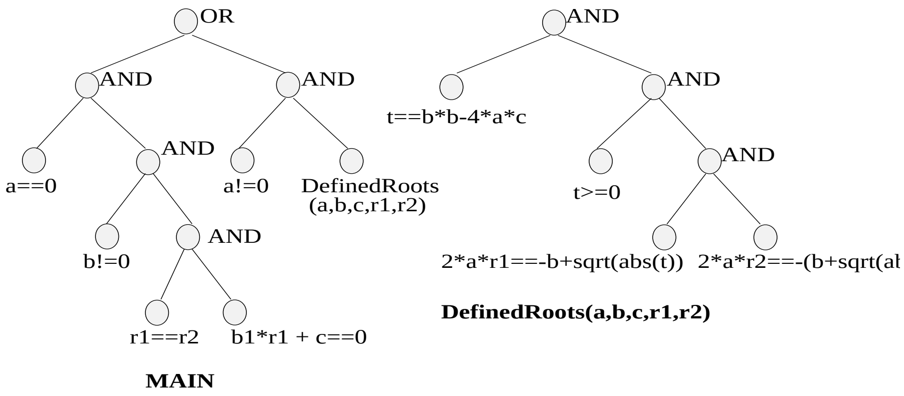


Figure 12: Constraint Graphs for the Quadratic Equation Solver

3.2 Phase 2: Translation of Constraint Graphs to Directed Graphs

A depth-first traversal of each graph in the set of constraint graphs obtained from the main body and the constraint module bodies constructs a set of directed graphs. The directed graph corresponding to the main body is referred to as the *main tree*. The traversal assigns constraints connected by AND operators in a constraint graph to the same node in the corresponding directed graph and constraints connected by OR operators in a constraint graph to nodes on diverging paths in the corresponding directed graph.

Figure 13 illustrates phase 2 for four base cases, where a, b, c, and d are simple constraints. There is a potential for combinatorial explosion in case 4 which corresponds to the applying the distributive law: (a OR b) AND (c OR d) $\equiv$ (a AND c) OR (a AND d) OR (b AND c) OR (b AND d).

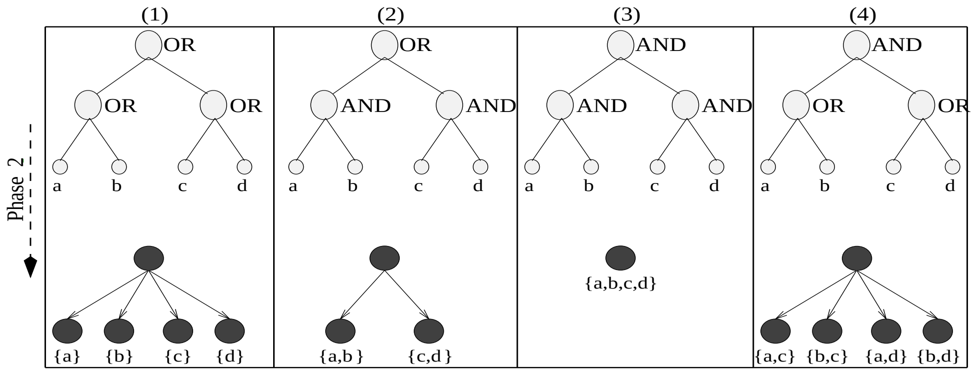


Figure 13: Phase 2 for four base cases

The resulting directed graphs in this phase do not contain any AND/OR nodes. Instead, a node in a directed graph may contain a list of simple constraints, indexed sets, or constraint module calls. However, AND/OR nodes are implicitly represented in a directed graph since all constraints along a path are connected by the AND operator and constraints on different paths are connected by the OR operator. The satisfaction of all the constraints along a path from the root to a leaf in a directed graph represents a satisfaction of the constraint system represented by the directed graph. Different paths, being implicitly connected by the OR operator, represent different ways of satisfying the constraint system.

The algorithm *dft* is a generalization of Figure 13. Let v_1 be the unique node at the highest level of the input constraint graph G. Each output directed graph G^* is initialized to a root v_1^*. Each node in G^* can hold a list of constraints. An indexed set of constraints within a node in G^* has an associated directed graph obtained from the depth-first traversal of the constraint graph corresponding to constraints in the indexed set. v_c and v_c^* are the nodes currently being visited in G and G^*, respectively. dft is initially invoked with the call dft(v_1, v_1^*).

The case of the *operator* node NOT has been omitted from the description of *dft*. However, it is implemented in the system as follows. A NOT operator node operates on a single constraint subgraph. It is moved down all the levels of the subgraph by changing nodes - AND to OR and OR to AND - traversed in its path until it reaches a simple constraint or another NOT node. If it reaches a simple constraint, the NOT node is removed by negating the simple constraint. If it reaches another NOT node, both NOT nodes are removed from the graph.

ALGORITHM dft (v_c, v_c^*)

begin

visited[v_c] = true;

Case type(v_c) **of**

OR : **for** each unvisited neighbor u of v_c **do**

if type(u) == OR dft(u, v_c^*)

else create node u^* in G^* as child of v_c^*;

dft(u, u^*);

AND : **if** there is an unvisited OR neighbor u_1 of v_c

let u_2 be the other neighbor of v_c;

let u_{11} and u_{12} be the two unvisited neighbors of u_1;

/*(u_{11} OR u_{12}) AND $u_2 \equiv$ (u_{11} AND u_2) OR (u_{12} AND u_2)*/

visited[v_c] = false;

change type of v_c to OR, remove u_1, u_2 as neighbors of v_c;

create two unvisited AND neighbors and_1 & and_2 for v_c;

make u_2 and u_{11} the neighbors of and_1;

make u_2 and u_{12} the neighbors of and_2;

dft(v_c, v_c^*);

else for each unvisited neighbor u of v_c **do** dft(u, v_c^*);

Simple_constraint : attach constraint to v_c^*;

Call Node : attach constraint module call to v_c^*;

For Node : attach indexed set with index and bounds to ${v_c}^*$

create new root ${v_i}^*$ for tree corresponding to indexed set;

let v_i be node at highest level of constraint graph in indexed set;

dft(v_i, ${v_i}^*$);

end;

The directed graphs obtained for the quadratic equation solver through phase 2 are shown in Figure 14.

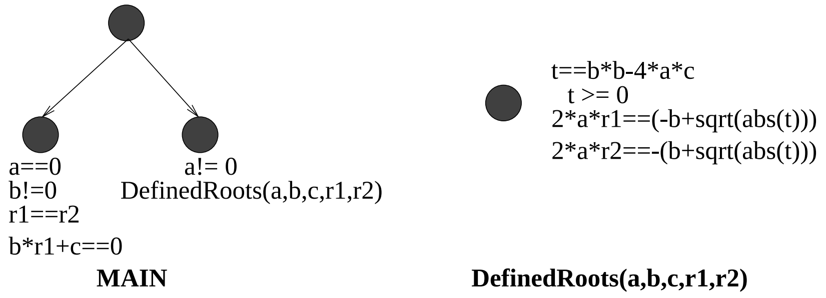


Figure 14: Directed Graphs from Phase 2 for the Quadratic Equation Solver

3.3 Phase 3: Generation of Dependence Graphs

Using the input set $\mathcal{I}$, a depth-first traversal of the main tree T_{main} from phase 2 attempts to generate a dependence graph. The generated dependence graph is a directed graph in which nodes are computational elements and arcs between nodes express data dependency. It has a unique *start* node which has no arcs directed into it and whose inputs are in $\mathcal{I}$. The start node can be executed exactly once at the initiation of the computation. A path from the start node in the graph is a computation path. A node in the dependence graph has the form: firing rule, computation, routing rule (see Figure 15). A firing rule is a condition that must hold before the node can be enabled for execution. The computation at a node is performed when the node is executed. A routing rule is a condition that must hold for the node to send data on its outgoing paths.

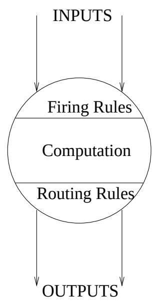


Figure 15: Generalized Dependence Graph Node

At the initiation of phase 3, a dependence graph G is constructed which is similar in structure to T_{main}, i.e., there is a 1-1 mapping between nodes and arcs in T_{main} and the nodes and arcs in G, respectively. The node in G corresponding to the root of T_{main} is designated as the start node. The structure of G may change later as detailed in Sections 3.3.2 and 3.3.3. The nodes in G are initially empty.

A *known* set is associated with each node in the dependence graph G. The variables in the known set at a node are *knowns* at that node. (The values of these variables are known at runtime at that node.) All variables not in the known set at a node are *unknowns* at that node. The input set is cast as the *known* set for the start node. A child node in the dependence graph inherits the known set of its parent when the node in T_{main} corresponding to the child node is visited during the depth-first traversal.

When a node in T_{main} is visited, constraints at that node may be *resolved* through processes detailed in Sections 3.3.1, 3.3.2, and 3.3.3 into computations or conditionals (firing/routing rules) of the corresponding node in G. Any constraint which cannot be resolved is retained in an unresolved set of constraints which is propagated down T_{main} to other nodes through the depth-first traversal in the hope that it may get resolved later. A number of passes may be made through each constraint at a node and the propagated unresolved set of constraints for resolution of these constraints. A new pass is initiated if at least one constraint was resolved in the previous pass; otherwise the depth-first traversal proceeds to visit the next node. Treatment of constraints remaining unresolved at the leaves of T_{main} is described in Section 3.3.8.

3.3.1 Resolution of Simple Constraints

Each node v in the directed graph from phase 2 may have a set of simple constraints attached to it. Additionally, the depth-first traversal may have a list of unresolved constraints propagated down from v's parent. Each simple constraint at v or in the unresolved set of constraints can be *resolved* as one of the following for the corresponding node v^* in the dependence graph.
(i) Firing Rule: To be so classified a constraint must have no unknowns at v^* before the first pass through the list of constraints at v and the unresolved set of constraints.
(ii) Computation: To fall into this category a constraint must involve an equality and have a single unknown at v^*. The constraint is cast as a computation at v^* for the unknown which is added to the known set for v^*.
(iii) Routing Rule: To be a routing rule all unknown variables in the constraint must become knowns through computations at v^*.

Constraints involving inequalities must be resolved as firing/routing rules. When a constraint is classified as a computation it is mapped to an equation. All terms involving the single unknown in the computation are moved to the left-hand side of the equation. If the unknown occurs in an actual parameter of a function, the inverse of the function may be applied to extract a computation for the unknown. Currently, our system solves equations in linear unknown terms. Thus non-linear constraints can be currently resolved if the unknown terms are linear. In the future we plan to incorporate solvers for scalar types that will solve for higher powers of the unknown. If the variables in the computation are matrices, the computation is replaced by calls to specialized matrix routines written in C. For example, the statement $A * x + b1 == b2$ with x as the unknown is first transformed into $A * x == b2 - b1$ and then a routine is invoked to solve for x. If A is lower (upper) triangular, then forward (backward) substitution is used to solve for x. Otherwise x is solved through an LU decomposition of A.

3.3.2 Resolution of Indexed Sets

An indexed set AND/OR FOR (<index> <b1> <b2>) $\{A_1, A_2, \ldots, A_n\}$ is resolved if every constraint A_i, $1 \leq i \leq n$, is resolved for all values of *index* in $b1 \ldots b2$. Resolved indexed sets are compiled to loops which iterate over values of *index* in $b1 \ldots b2$. If every constraint in a set $S_1 \subseteq \{A_1, A_2, \ldots, A_n\}$ is resolved as a computation, every constraint in a set $S_2 \subseteq \{A_1, A_2, \ldots, A_n\}$ is resolved as a firing/routing rule and every constraint in a set $S_3 \subseteq \{A_1, A_2, \ldots, A_n\}$ remains unresolved, the indexed set is split into the following three indexed sets.
(1) An indexed set AND/OR FOR (*index* <b1> <b2>) S_1 resolved as a computation
(2) An indexed set AND/OR FOR (*index* <b1> <b2>) S_2 resolved as a firing/routing rule
(3) An unresolved indexed set AND/OR FOR (*index* <b1> <b2>) S_3
Note that $S_1 \cup S_2 \cup S_3 == \{A_1, A_2, \ldots, A_n\}$ and
$S_i \cap S_j == \phi$ (null set) where $1 \leq i, j \leq 3$ and $i \neq j$.

The restrictions for a constraint $A_i, 1 \leq i \leq n$, in an indexed set structure to be compiled successfully in our system are as follows. For all values of *index* in $b1 \ldots b2$ (a) A_i has to have the same classification (computation/firing rule/routing rule), (b) if A_i is a simple constraint and is classified as a computation, a unique term in the constraint has to be the unknown (a term can be a simple variable x or an indexed term such as $X[< list\ of\ indices >]$, where X is a structured data type). An example of a construct that will be compiled successfully is

$$X[0] == 0 \text{ AND (AND FOR (i 1 5) } \{ X[i-1] == X[i] + Y[i] \} \text{)}$$

with Y known and X unknown.
It will be compiled to the computations

$X[0] = 0;$

for i=1 to 5 do

$X[i] = X[i-1] - Y[i];$

Note that the indexed set is compiled to a loop which computes the value of $X[i]$ in successive iterations.

An example of a construct that will not be compiled successfully is

$$\text{AND FOR (i 1 5) } \{\ X[1] == X[i] + Y[i]\ \}$$

with X unknown and Y known. This is because in the first iteration both the terms $X[1]$ and $X[i]$ are unknown whereas subsequent iterations have only $X[i]$ as an unknown (violates (b)).

Resolution of AND Indexed Sets

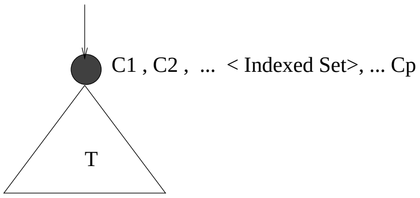


Figure 16: Indexed Set at a Node in a Directed Graph from Phase 2

Let an AND indexed set AND FOR (i <b1> <b2>) $\{A_1, A_2, \ldots, A_n\}$ occur among constraints $C_1, C_2, \ldots, C_p$ at a node in a directed graph as shown in Figure 16. Evaluate constraints $A_1, A_2, \ldots, A_n$ for classification as firing/routing rules or computations for $i = b1 \ldots b2$. Let $k(1) \ldots k(n)$ be a reordering of the subscripts $1 \ldots n$. Let $\{A_{k(1)},\ A_{k(2)},\ \ldots\ A_{k(m1)}\}$ be the constraints which evaluate to firing rules for all $i = b1 \ldots b2$. Let $\{A_{k(m1+1)}\ \ldots\ A_{k(m2)}\}$ be the constraints which evaluate to computation for all $i = b1 \ldots b2$. Let $\{A_{k(m2+1)}\ \ldots\ A_{k(m3)}\}$ be the constraints which evaluate to routing rules for all $i = b1 \ldots b2$. Let $\{A_{k(m3+1)}\ \ldots\ A_{k(n)}\}$ be the constraints which remain unresolved.

Similarly, evaluate constraints $C_1, C_2, \ldots, C_p$. Let $r(1) \ldots r(p)$ be a reordering of the subscripts $1 \ldots p$. Let $\{C_{r(1)},\ C_{r(2)},\ \ldots\ C_{r(l1)}\}$, $\{C_{r(l1+1)}\ \ldots\ C_{r(l2)}\}$, and $\{C_{r(l2+1)}\ \ldots\ C_{r(l3)}\}$ be the constraints which evaluate to firing rules, computations, and routing rules, respectively and $\{C_{r(l3+1)}\ \ldots\ C_{r(p)}\}$ be the unresolved constraints. The generated dependence graph is shown in Figure 17. The unresolved constraints are propagated down T.

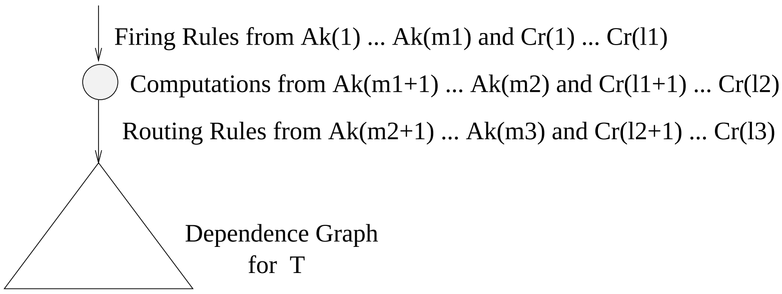


Figure 17: Generated Dependence Graph for an AND Indexed Set

The firing rule corresponding to $A_{k(1)},\ A_{k(2)},\ \ldots\ A_{k(m1)}$ is $A_{k(1)}$ AND $A_{k(2)}$ AND ... AND $A_{k(m1)}$ for all $i = b1 \ldots b2$. A similar construct is set up for the routing rule corresponding to $A_{k(m2+1)}\ \ldots\ A_{k(m3)}$. The computations for $A_{k(m1+1)}\ \ldots\ A_{k(m2)}$ are expressed as

for $i = b1$ *to* $b2$ *do*

Computation corresponding to $A_{k(m1+1)}$;

Computation corresponding to $A_{k(m1+2)}$;

$\vdots$

Computation corresponding to $A_{k(m2)}$*;*

Resolution of OR Indexed Sets

Let an OR indexed set OR FOR (i <b1> <b2>) $\{A_1, A_2, \ldots, A_n\}$ occur among constraints $C_1, C_2, \ldots, C_p$ at a node in a directed graph from phase 2 as shown in Figure 16. Evaluate constraints $A_1, A_2, \ldots, A_n$ for classification as firing/routing rules or computation for $i = b1 \ldots b2$. Let $k(1) \ldots k(n)$ be a reordering of the subscripts $1 \ldots n$. Let $\{A_{k(1)},\ A_{k(2)},\ \ldots\ A_{k(m1)}\}$ be the constraints which evaluate to firing rules for all $i = b1 \ldots b2$. Let $\{A_{k(m1+1)}\ \ldots\ A_{k(m2)}\}$ be the constraints which evaluate to computations for all $i = b1 \ldots b2$. Let $\{A_{k(m2+1)}\ \ldots\ A_{k(m3)}\}$ be the constraints which evaluate to routing rules for all $i = b1 \ldots b2$. Let $A_{k(m3+1)}\ \ldots\ A_{k(n)}\}$ be the constraints which remain unresolved.

Similarly, evaluate constraints $C_1, C_2, \ldots, C_p$. Let $r(1) \ldots r(p)$ be a reordering of the subscripts $1 \ldots p$. Let $\{C_{r(1)},\ C_{r(2)},\ \ldots\ C_{r(l1)}\}$, $\{C_{r(l1+1)}\ \ldots\ C_{r(l2)}\}$, and $\{C_{r(l2+1)}\ \ldots\ C_{r(l3)}\}$ be the constraints which evaluate to firing rules, computations, and routing rules, respectively and $\{C_{r(l3+1)}\ \ldots\ C_{r(p)}\}$ be the unresolved constraints. The generated dependence graph is shown in Figure 18. The unresolved constraints are propagated down T.

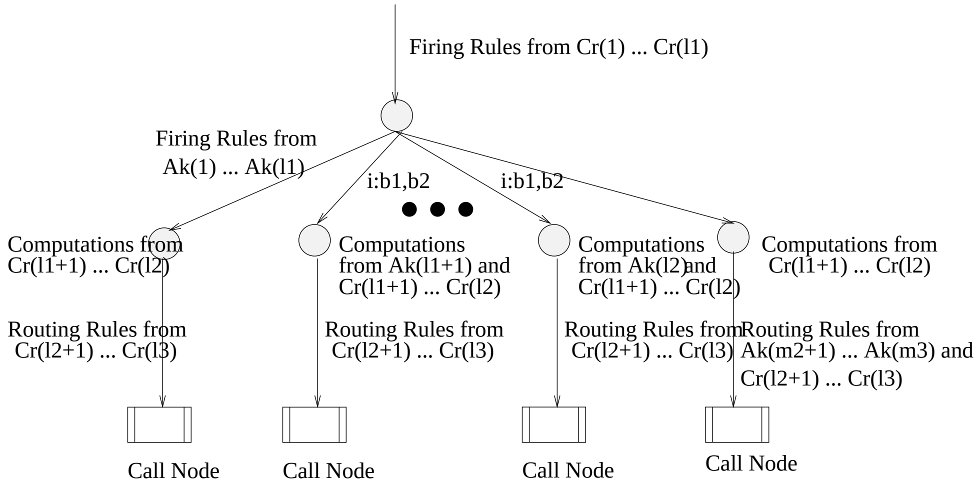


Figure 18: Generated Dependence Graph for an OR Indexed Set

The "Call Node" invokes the dependence graph corresponding to T. $i : b1, b2$ shows that the associated arc and its destination node are replicated for values of i from $b1 \ldots b2$. The firing rule for $A_{k(1)},\ A_{k(2)},\ \ldots\ A_{k(m1)}$ is $A_{k(1)}$ *OR* $A_{k(2)}$ *OR ...* $A_{k(m)}$ *for any i=b1 ... b2.* A similar construct is set up for the routing rule for $A_{k(m2+1)}$ ... $A_{k(m3)}$.

3.3.3 Resolution of Constraint Module Calls

A constraint module call has the form $ModuleName(e_1, e_2, \ldots, e_n)$ where e_i, $1 \leq i \leq n$, is an actual parameter. Actual parameters may be expressions. Let the formal parameters corresponding to $e_1, e_2, \ldots, e_n$ be $f_1, f_2, \ldots, f_n$, respectively. Let K be the known set at that node in G (dependence graph) which corresponds to the node in T (directed graph from phase 2) where the constraint module is invoked. If all the variables in $e_1, \ldots, e_n$ and all the global variables occuring in the constraint module body are in K and no local variable occurs in the constraint module body, the call to the constraint module is cast as a firing/routing rule which tests whether the body of the constraint module is satisfied or not.

If the constraint module call cannot be cast as a firing/routing rule, an attempt is made to generate a dependence graph from the constraint module definition. A new dependence graph G_{mod} is created which is similar in structure to the directed graph T_{mod} from phase 2 for the constraint module, i.e., there is a 1-1 mapping between nodes and arcs in G_{mod} and nodes and arcs in T_{mod}, respectively. T_{mod} is traversed with a new known set K_{module} which is initialized to $\{ f_i \mid \{\text{all variables in } e_i\} \subseteq K,\ 1 \leq i \leq n\} \cup \{ x \mid x \in K$ and x is a global variable in the scope of the module $\}$. The unknowns are considered to be all formal parameters not in K_{module}, the local variables in the constraint module, and all the global variables not in K but in the scope of the module.

The resolution of constraints in the constraint module is similar to that for the main module with one difference. The dependence graph G_{mod} is retained with only the set of paths with the maximal output set for formal parameters and global variables. For example, let there be 5 paths numbered 1 through 5 with the following computed formal parameters and global variables. 1: $\{ a, b \}$, 2: $\{ a \}$, 3: $\{ a, b, c \}$, 4: $\{ a, b \}$, 5: $\{ a, b, c \}$. Paths 3 and 5 have the maximal output set $\{ a, b, c \}$ and are the only ones retained in the dependence graph; paths 1, 2, and 4 are deleted. If there is more than one distinct maximal set, any one maximal set is chosen at random. This technique of deleting paths not having the maximal output set is not implemented in the dependence graph generation of the main module where all paths need not have the same set of computed variables. The reason for imposing this condition in a constraint module is that the actual parameters are bound to the formal parameters at the point of call. If different sets of variables are computed in different paths of the dependence graph corresponding to a constraint module it is not possible to determine statically the actual parameters and global variables computed in the constraint module call, which have to be added to K.

If the dependence graph generation is successful (there are no unresolved constraints at the end of the paths with the maximal output set), a new set of constraints is generated as follows.
$e_{k1} == Z_1$, $e_{k2} == Z_2$, $\ldots$, $e_{kp} == Z_p$, where Z_i, $1 \leq i \leq p$, are new variables generated by the compiler and $e_{k1} \ldots e_{kp}$ are the actual parameters corresponding to the set of computed formal parameters in the maximal output set. An attempt is made to resolve this set of constraints with $Z_1 \ldots Z_p$ in the known set K. If the constraints in this set are resolved as computation for all the unknowns in $e_{k1} \ldots e_{kp}$, a call node which invokes the dependence graph for the constraint module call G_{mod} is generated as shown in Figure 19. A child node of the call node receives values computed for the formal parameters by the call node and binds them to $Z_1 \ldots Z_p$ and performs the computation generated from the new set of constraints.

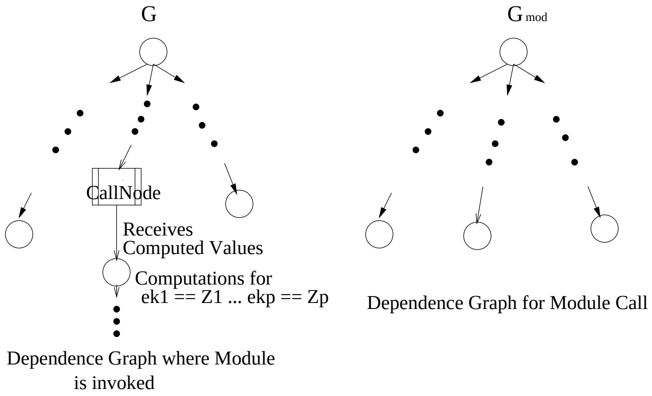


Figure 19: Dependence Graphs for a Constraint Module Call

If the dependence graph generation is not successful, the constraint module call is considered to be unresolved.

For a constraint module with n parameters there are 2^n possible input parameter sets and consequently, there are 2^n potential translations for a particular constraint module. Of course, not all translations might be successful. Constraint module invocations, with the same set of formal parameters and global variables as inputs, reuse the same dependence graph.

3.3.4 The Quadratic Equation Solver through Phase 3

The dependence graphs for the quadratic equation solver with the input set $\{ a, b, c \}$ are shown in Figure 20 where computations for $r1$ and $r2$ are extracted.

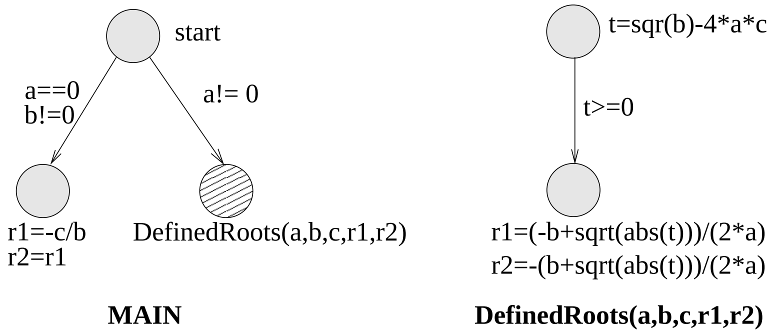


Figure 20: Dependence Graphs for the Quadratic Equation Solver with $\mathcal{I}$ = {a, b, c}

The dependence graphs for the quadratic equation solver with a different input set $\{ a, b, r1 \}$ are shown in Figure 21. The dependence graphs compute values for variables c and $r2$. The inverses of the functions *sqrt* and *abs* have been applied to derive the computations for t. The compiler can be optimized to detect that the path starting from the node computing $t = -sqr(2 * a * r1 + b)$ can never be traversed to completion.

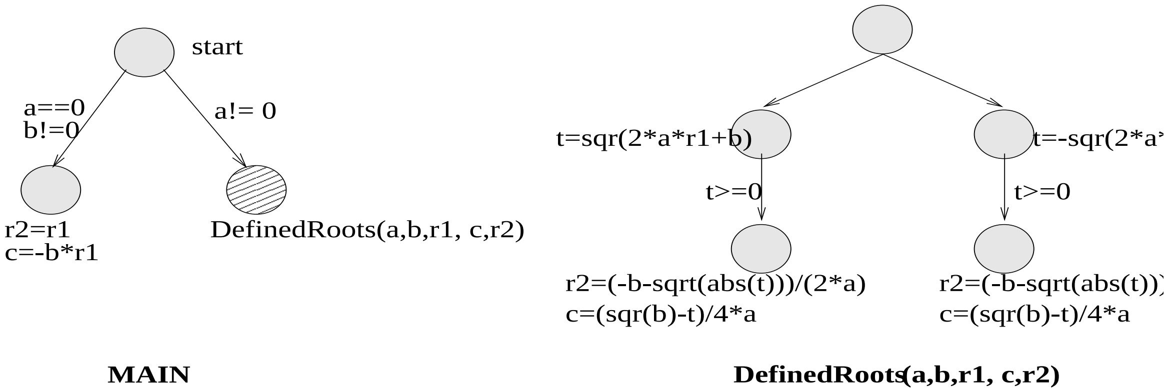


Figure 21: Dependence Graphs for the Quadratic Equation Solver with $\mathcal{I}$ = {a, b, r1}

Figures 20 and 21 show that the same constraint program specification can be reused to derive the dependence graphs for different input sets. However, not all input sets can lead to dependence graphs where no constraints remain unresolved. For example, no dependence graph can be generated with the input set $\{ a, r1 \}$ because the simple constraint $b * r1 + c == 0$ and the constraint module call $DefinedRoots$ remain unresolved in the *main tree* (The module call $DefinedRoots$ remains unresolved because all the constraints in the directed graph for the module remain unresolved.). Note that phases 1 and 2 need not be repeated when a new input set is supplied for a constraint specification.

Part of the appeal of a constraint programming language is its multi-directional nature - the facility to extract values for different sets of variables depending on the composition of the input set. While this is a secondary aspect for parallel programming concerns as compared to the ease of use aspect, this is still important in many application domains.

3.3.5 Single Assignment Variable Programs

The compilation process generates dependence graphs with single assignment variables. This occurs because a child node in the dependence graph G inherits the known set of its parent as its initial known set and no deletions are made to the known set of a node. Hence, once a variable is added to the known set of a node it is retained in the known sets of all nodes in the subgraph rooted at that node. If a path in G contains nodes in the order $v_1{}^* \ldots v_n{}^*$, where $v_1{}^*$ is the start node, $v_n{}^*$ is the leaf and n is the length of the path, exactly one node $v_i{}^*$, $1 \leq i \leq n$, can contain a computation for a variable x. There will be no occurrence of x in any computation or firing/routing rule for nodes in $v_1{}^* \ldots v_{i-1}{}^*$ or in any firing rule for $v_i{}^*$. While single assignment variables are appropriate for parallel programs, they can lead to excessive use of memory in some circumstances. Section 4 details our approach to introduction of mutable variables where they are necessary.

3.3.6 Generation of either Effective or Complete Programs

The presence of the OR operator in a constraint system results in the possibility that there exists more than one assignment of values to the variables which will result in satisfaction of the constraint system. (A given input set for a program with an OR operator may or may not allow multiple assignments which satisfy the constraint system.) A program which is effective generates exactly one set of assignments of values to variables which satisfies the constraint system. A program which is complete generates all of the sets of assignments of values to variables which will satisfy the constraint system. The compilation process can be directed to generate the executable either for exactly one "OR branch" of the dependence graph or to generate the executable for all paths which lead to valid assignments. Thus, the compilation process can produce programs which are either effective or complete. A program which is complete realizes OR parallelism, as will be further discussed in Section 3.3.7. Non-determinism arises if the compiler randomly chooses a path for execution in effective programs.

3.3.7 Extraction of parallelism

Our constraint representation maps to a dependence graph which is a parallel computation structure because all nodes that are enabled for execution may be executed in parallel. The constraint representation allows the targeting all types of parallelism (AND/OR, task and data parallelism) through a single representation. AND/OR parallelism refers to parallelism in computations extracted from terms connected by AND and OR operators, respectively. Task parallelism refers to parallelism in computations for different data. Data parallelism refers to parallelism in computations for different parts of a structured data item. In our system data parallelism arises from the hierarchical representation of our type system. For example, matrices can be represented as blocks of sub-matrices and constraints over sub-matrices are translated to data-parallel conditionals/computations.

The different sources of parallelism and their respective types in the representation are enumerated as follows. While 1-4 are extracted by the current compiler, 5 has not yet been implemented.

1. **OR, Task**: OR parallelism corresponds to executing the different paths in the dependence graph in parallel. These paths have resulted from the extraction of computation from constraints connected by OR operators.

2. **AND, Task**: The computational statements that are assigned to a node have the potential for parallel execution. For instance, the assignments $r1 = (-b + r)/2 * a$ and $r2 = -(b + r)/2 * a$ in Figure 20 can be done in parallel. Parallelism is exploited by keeping in mind that the compiler generates a single-assignment system and the lone write to a variable will appear before any reads to it. A particular node may be split into several nodes to exploit the parallelism in the computations at the node. The granularity of such a scheme depends on the complexity of the functions and the operators invoked in the statements.

We illustrate AND-OR parallelism in 1 and 2 through a simple example. Consider the constraint specification in Figure 22 for a program involving variables { a,b,c,x,y }.

a<b AND b==x AND y==c
OR
a<c AND x==c AND b==y

Figure 22: Constraint Specification for a Simple Example

The dependence graph for the input set { a,b,c } and output set { x,y } for the specification in Figure 22 is shown in Figure 23. Since a, b, c are inputs, $a < b$ and $a < c$ are classified as conditionals. The constraints involving equalities ($b == x$, $y == c$, $x == c$, and $b == y$) are classified as computations for the single unknown in them. OR parallelism comes into play in the parallel execution of the two paths branching out from the start node in the event that $a < b$ and $a < c$. This also implies that this program can be compiled to be either complete or effective, as discussed in Section 3.3.6. AND parallelism is extracted from the computations for x and y.

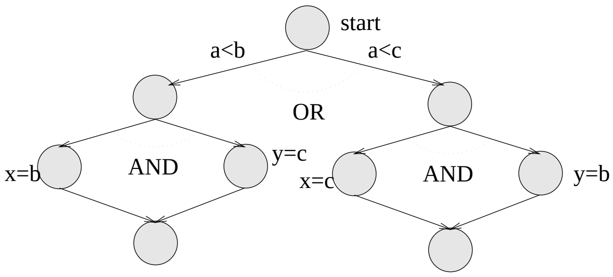


Figure 23: Dependence Graph showing AND-OR Parallelism

3. **Task**: We have further exploited the complexity of matrix operations by splitting up the specifications, performing computations in parallel, and composing them. For example, if $x = m * y + m * z$, where x, m, y, z, and b are matrices, $m * y$ and $m * z$ can be done in parallel. This leads to significant speedup since multiplication of matrices is an $\mathrm{O}(N^3)$ operation (m, y, z being order $N \times N$). In a later version of the compiler, provision will be made for user specification of parallelism for operations over structures.

4. **Data (Parallelism in AND indexed sets)**: The computations within the compiled loop structures corresponding to *AND* indexed sets have the potential for parallel execution. We first discuss the case of loops with a single computation. The discussion is then generalized to the case of loops with multiple computations. Throughout this discussion the case of array accesses will be detailed. The case of scalar accesses in loops will follow trivially since they do not involve indexed terms.

(i) If the array corresponding to the computed term is not accessed anywhere in the computation, all iterations of the loop can be executed in parallel. The compiled parallel structure for such a loop is shown in Figure 24(a). The node performing the computation and the arc connecting the parent to it are replicated N times, where N is the range of the loop index. The results of the computation performed by the parallel nodes are merged (not shown in figure).

(ii) If the array corresponding to the computed term is accessed in the computation and the set of accessed indices of the array are disjoint from the set of computed indices of the array, all iterations of the loop can be executed in parallel. The compiled structure is again as shown in Figure 24(a).

(iii) If cases (i) and (ii) do not hold, the loop iterations are inter-dependent and are executed sequentially. The compiled structure for this case is shown in Figure 24(b). The node performing the computation is invoked repeatedly in succession.

A similar analysis is done for the loop structure compiled from an indexed set with more than one constraint. In such a case there is more than one computation within the loop and interdependencies between different com-

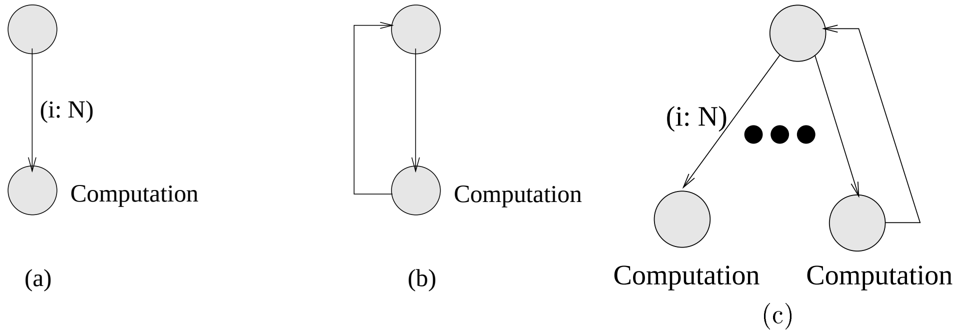


Figure 24: (a) Parallel Execution of Loop (b) Sequential Execution of Loop (c) Generalized Compiled Loop Structure

putations for all the iterations have to be checked in addition to dependencies between iterations of the same computation. If there are no dependencies between the iterations of a computation (cases (i) and (ii)) and no iteration of the computation is dependent on an iteration of another computation, then all iterations of the computation are executed in parallel; otherwise, the iterations of the computation are executed sequentially. In general, the loop structure will be a combination of parallel and sequential loop executions as shown in Figure 24(c).

5. **Data**: Finally, primitive operations in the base types like matrix-matrix multiply can be executed in parallel by invoking appropriate parallel algorithms.

3.3.8 Unresolved Constraints

Any path P from the root to a leaf in the directed graph T from phase 2 consists of nodes, each containing a set of constraints. P represents one way of satisfying the constraint system since constraints on different paths are implicitly connected by the OR operator. Every constraint on P must be resolved to either a computation or to a conditional (firing/routing rule) for P to satisfy the constraint system. The depth-first traversal described in Section 3.3 collects any unresolved constraint on P at its leaf. An unresolved constraint can be of the following types.

(i) A simple constraint involving an equality and at least two unknowns.

(ii) A simple constraint involving a relational operator other than an equality and at least one unknown.

(iii) An unresolved call to a constraint module. This would imply that there is more than one unknown in the set of actual parameters, local variables, and global variables in the body of the constraint module. (Unknowns in an actual parameter imply that the corresponding formal parameter is unknown.)

(iv) An unresolved indexed set of constraints AND/OR FOR (<index> <b1> <b2>) $\{A_1, A_2, \ldots, A_n\}$ where each A_i, $1 \leq i \leq n$, is unresolved due to one of the following reasons.

(a) A_i may be an unresolved indexed set.

(b) If A_i is a simple constraint or a constraint module call, there is no unique unknown term for all values of i in $b1 \ldots b2$ (See Section 3.3.2).

(c) During the resolution process A_i is classified as a computation for some values of i in $b1 \ldots b2$ and as a conditional (firing/routing rule) for other values of i in $b1 \ldots b2$.

In case (c) we may be able to split the indexed set into several resolved indexed sets with different index bounds. Assume that
$S_1 \subseteq \{A_1, A_2, \ldots, A_n\}$ is resolved as computations and conditionals in the ranges
$B_{s_1(1)}, B_{s_1(2)}, \ldots, B_{s_1(p_1)}$,
$S_2 \subseteq \{A_1, A_2, \ldots, A_n\}$ is resolved as computations and conditionals in the ranges $B_{s_2(1)}, B_{s_2(2)}, \ldots, B_{s_2(p_2)}$,

$\vdots$

and $S_q \subseteq \{A_1, A_2, \ldots, A_n\}$ is resolved as computations and conditionals in the ranges $B_{s_q(1)}, B_{s_q(2)}, \ldots, B_{s_q(p_q)}$,
where B_i, $s_j(1) \leq i \leq s_j(p_j)$, $1 \leq j \leq q$, is a subrange in $b1 \ldots b2$,
$S_i \cap S_j ==$ the null set, ϕ, $1 \leq i, j \leq q$, $i \neq j$, and
$S_1 \cup S_2 \cup \ldots \cup S_q == \{A_1, A_2, \ldots, A_n\}$.

The indexed set can be split into the following resolved indexed sets.
AND/OR FOR ($i < B_{s_1(1)} >$) S_1
AND/OR FOR ($i < B_{s_1(2)} >$) S_1

$\vdots$

AND/OR FOR ($i < B_{s_1(p_1)} >$) S_1
AND/OR FOR ($i < B_{s_2(1)} >$) S_2
AND/OR FOR ($i < B_{s_2(2)} >$) S_2

$\vdots$

AND/OR FOR ($i < B_{s_2(p_2)} >$) S_2

$\vdots$

AND/OR FOR ($i < B_{s_q(p_q)} >$) S_q

There are several options available for resolution of each type of unresolved constraint. We shall enumerate some of these.

1. Since there is a 1-1 mapping between nodes in T and the dependence graph G, there is a unique leaf in G corresponding to the leaf in P containing the unresolved constraints. In Figure 25 the two corresponding leaves in T and G are shaded. The shaded leaf in G can be deleted. This virtually removes P from T and corresponds to not attempting to satisfy the constraint system through the path P. If deletion of the leaf in G results in its parent becoming a leaf, the parent must be deleted too. This must be continued in a recursive fashion until the deletion of a leaf does not result in its parent becoming a leaf. Then a new path descending T is chosen and pursued to see if a usable G can be obtained. This approach can be applied to unresolved constraints of any type ((i)-(iv)). Of course, there is the danger of getting an empty dependence graph if all leaves in T contain unresolved constraints.

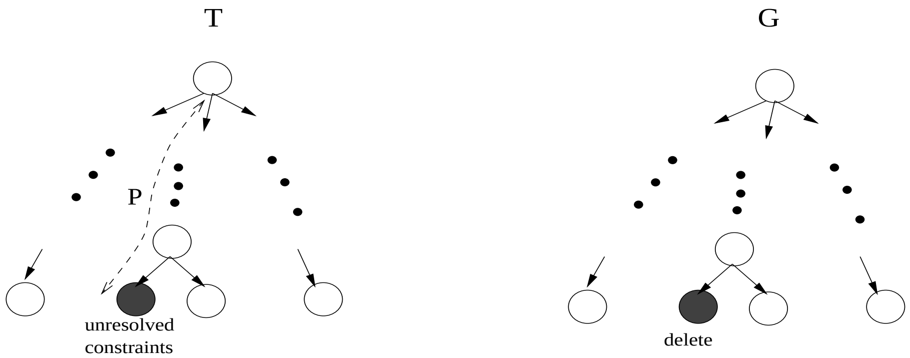


Figure 25: Deletion of a Path with Unresolved Constraints

2. The user may have incorrectly specified the initial input set by overlooking the inclusion of some variables or including the wrong variables and may be helped in the choice of a new input set through display of the unresolved constraints and the unknowns in them. The depth-first traversal in phase 3 can be performed again with the new input set. This can be done repeatedly until all constraints in the system are resolved. While selection of unknowns to be added to the initial input set may be easy for constraints of types (i), (ii) and (iii), it may be quite difficult to do for constraints of type (iv) since unknowns could typically be of the form $A[fn_1(i_1, \ldots, i_k)] \ldots [fn_l(i_1, \ldots, i_k)]$ where $i_1 \ldots i_k$ are indices for nested indexed sets containing the unresolved constraints, $fn_1 \ldots fn_l$ are arithmetic functions over the indices, and A is any structured data type. Some parts of A may be known and other parts of A may be unknown forcing the user to identify the regions that the term $A[fn_1(i_1, \ldots, i_k)] \ldots [fn_l(i_1, \ldots, i_k)]$ refers to and denote them as known.

3. Commercial solvers such as MATLAB can be invoked to solve the unresolved constraints by providing a wrapper around the invocation to the MATLAB solver in the form of a constraint module call. This technique can be most beneficial for the resolution of constraints of types (i) and (ii).

4. Iterative solutions can be attempted for unresolved constraints of types (i), (iii) and (iv) through several relaxation methods. This process is described in detail in Section 4.

3.4 Phases 4 and 5: Specification of Execution Environment and Mapping to Code

Apart from the textual constraint program the programmer is encouraged to specify an execution environment specification which is used by the compiler to optimally select certain execution environment characteristics used by CODE [24] to generate programs. The execution environment specification merits a separate discussion and is described in Section 5.

Our target for executable for constraint programs is the CODE parallel programming environment. CODE takes a dependence graph as its input. The form of a node in a CODE dependence graph is given in Figure 15. It is seen that there is a natural match between the nodes of the dependence graph developed by the constraint compilation algorithm and the nodes in the CODE graph. The arcs in the dependence graph in CODE are used to bind names from one node to another. This is exactly the role played by arcs in the dependence graph generated by our translation algorithm. CODE produces sequential and parallel C programs for a variety of architectures.

The control flow for the entire compiler is shown in Figure 26.

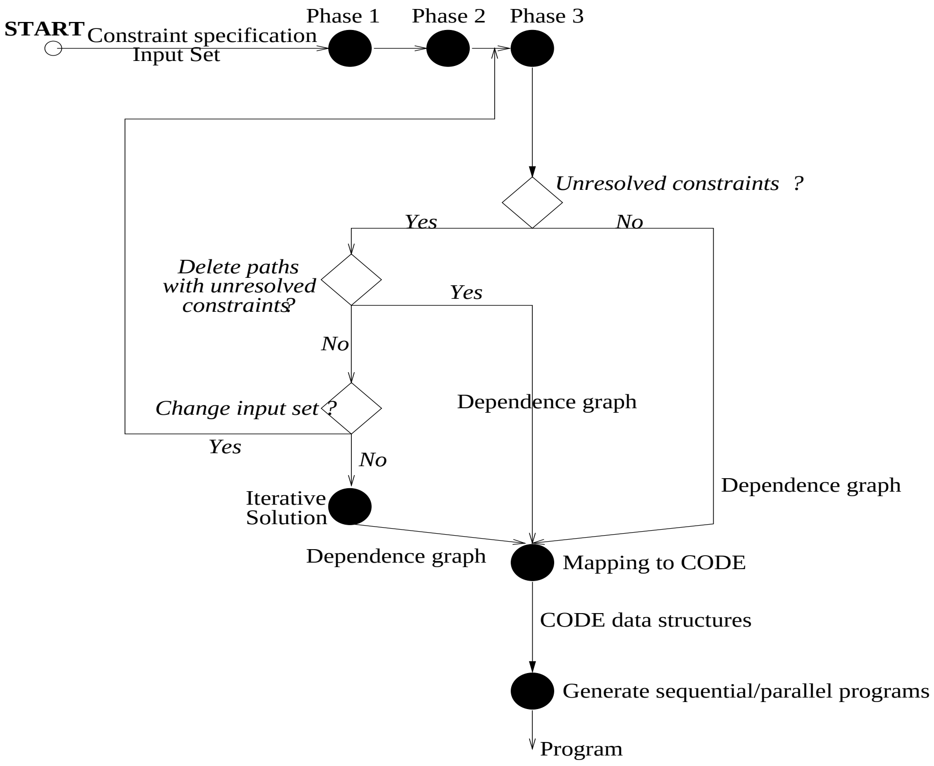


Figure 26: Control Flow for the Constraint Compiler

3.5 Procedural Parallel Programs for the BTS and BOER Systems

In this subsection we show how all of the parallelism in the BTS (Figure 4) and BOER (Figure 7) examples can be extracted by the compiler.

3.5.1 The BTS System

Consider the specification for the BTS system being compiled with the input set $\{ S_0, \ldots, S_3, M_{10}, M_{20}, \ldots, M_{32}, B_0, \ldots B_3 \}$. The specification has been repeated in Figure 27 with certain terms ($\{X_0, X_1, X_2, X_3\}$) in bold-faced to indicate terms that are chosen as outputs by the compiler.

PROGRAM BTS_1
$(S_0 * \mathbf{X_0} == B_0 \; AND$
$M_{10} * X_0 + S_1 * \mathbf{X_1} == B_1 \; AND$
$M_{20} * X_0 + M_{21} * X_1 + S_2 * \mathbf{X_2} == B_2 \; AND$
$M_{30} * X_0 + M_{31} * X_1 + M_{32} * X_2 + S_3 * \mathbf{X_3} == B_3 \;)$

Figure 27: Constraint Specification for the BTS System with Computed Terms in bold

By applying technique 3 in Section 3.3.7 the compiler splits up the specifications to perform the multiplications in series such as $M_{10} * X_0$, $M_{20} * X_0$, and $M_{30} * X_0$ in parallel. Thus the vector multiplications for all Ms within a column may be done in parallel. Figure 28 shows the form of the extracted dataflow and exactly corresponds to the parallel algorithm in [7]. Data parallelism could be used on the block level operations and captured in our representation with an appropriate type structure, if desired.

3.5.2 The BOER System

The constraint specification for the BOER system has been repeated in Figure 29 with certain terms in bold-faced to indicate computed terms. Each indexed set is compiled to a loop iterating over values of the index. Each simple constraint is compiled to a computation for a term (bold in Figure 29).

Analysis of the computations extracted shows that, in the reduction phase, the computations for $BP[j], CP[j]$, and $dP[i * pow(2, j)][j]$ can be executed in parallel. However, different iterations of the loop enclosing these computations (for index j) cannot be done in parallel due to interdependencies between the three computations. The different iterations of the nested loop for index i enclosing the computation for $dP[i * pow(2, j)][j]$ can be performed in parallel. The nested loop for index i in the back-substitution phase enclosing the computation for $x[\ldots]$ can be performed in parallel. However, the iterations for the outer loop for index j enclosing the computation for $x[\ldots]$ cannot be parallelized. Our compiler detects all the dependencies for this analysis and correctly extracts all the existing parallelism in the specification.

The resulting dependence graph is shown in Figure 30 and exactly corresponds to the dataflow in the algorithm in [18]. The *START* and *STOP* nodes initiate and terminate the program, respectively. A *FOR* node initiates the different iterations of a loop. The two such nodes in the figure correspond to the two outer indexed sets for index j in the reduction and back-substitution phases in the constraint specification. The annotation “Replicated” on the arcs specify that the annotated arc and the destination node (shaded in Figure 30) are dynamically replicated for parallel execution. The two such annotated arcs correspond to the two nested indexed sets (for index i) in the constraint specification and are instances of data parallelism. The nodes annotated by *BP, CP, dP*, and x compute values for parts of the corresponding variable. The parallel execution of the computations for *BP, CP*, and *dP* is an instance of task parallelism. The nodes annotated by “Merge” collect computed results from parallel executions. It is to be noted that our compiler automatically detects the parallelism in the *for* loops in

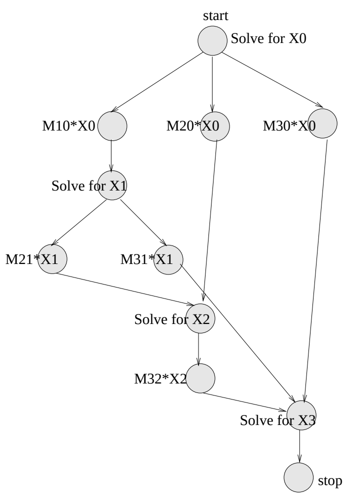


Figure 28: Dependence Graph for the BTS Program

the reduction and back-substitution phases. Furthermore, it is capable of extracting the parallelism within the expression $2*CP[j-1]*CP[j-1]-BP[j-1]*BP[j-1]$ in the computation for $BP[j]$ by computing the products $2*CP[j-1]*CP[j-1]$ and $BP[j-1]*BP[j-1]$ in parallel. By incorporating calls to BLAS routines (technique 5 in Section 3.3.7) which invoke parallel algorithms, incorporating data parallelism, for matrix-matrix multiply the compiler would have extracted all the existent parallelism in the example.

4 Iterative Solutions from Constraint Systems

Section 3 detailed the basic compilation algorithm for translating a constraint specification along with an input set to a dependence graph. The basic compilation algorithm cannot resolve constraint specifications with input sets that give rise to dependencies with cycles.

To illustrate dependencies with cycles, consider the constraint program shown in Figure 31. Phase 2 of the compiler collects constraints connected by AND operators at the same node and since there is only a single AND operator in the specification in Figure 31, phase 2 will generate a single node with the two simple constraints: $a+b==x$ and $x+b==y$ (shown in Figure 32). When this node is traversed in phase 3 with the input set $\{ a, y \}$ both simple constraints remain unresolved because there are two unknowns b and x in each of them (simple constraints are resolved as conditionals if they have no unknowns and as computations if they involve an equality and only one unknown; otherwise they are unresolved). The term *cyclic* is used to refer to this situation because a cycle exists in the low-level constraint graph representation (where variables and operators have associated nodes) for this constraint program as shown in Figure 33. Note that the arcs connected to the input variables a and y have directions on them to denote that the values for these variables are available. The non-input variables x and b are in a cycle and neither of the two "+" operator nodes can "fire" for computed values to be propagated along the arcs until either x or b is given a value. The constraints involved in such a situation are sometimes referred to as *cyclic constraints*. In fact, cyclic constraints give rise to *cyclic dependencies*.

This section discusses the augmentation to the basic compiler for handling constraints with cyclic dependencies. We opt to use the technique of relaxation whereby iterative solutions to cyclic constraints are sought. Relaxation attempts to satisfy all the constraints in the system within a certain degree of accuracy by making an initial assignment of values to the unknowns, computing the value of one unknown in each constraint and then estimating the error in the current value. Further iterations of computing the value of the unknown variables are initiated if the errors are not sufficiently small. In each iteration, the values computed in previous or current iterations are used to recompute the values of the unknowns in an attempt to achieve convergence where the difference in

```
PROGRAM BOER
BP[k-1] * x[pow(2,k-1)] == dP[pow(2,k-1)][k-1] AND


AND FOR (j 1 k-1) {
      2 * CP[j-1] * CP[j-1] == BP[j] + BP[j-1] * BP[j-1] ,

      CP[j] - CP[j-1] * CP[j-1] == 0 ,

      AND FOR (i 0 pow(2,k-j)-2) {
            CP[j-1] * ( dP[i*pow(2,j) + pow(2,j-1)][j-1] +
                              dP[i*pow(2,j) - pow(2,j-1)][j-1] ) ==
            dP[i*pow(2,j)][j] + BP[j-1] * dP[i*pow(2,j)][j-1] }} AND


AND FOR (j k-1 1) {
      AND FOR (i 0 pow(2,k-j)-1) {
            CP[j-1] * ( x[(i+1)*pow(2,j)] + x[i*pow(2,j)] ) ==
                  dP[(i+1)*pow(2,j)-pow(2,j-1)][j-1] -
                  BP[j-1] * x[(i+1)*pow(2,j)-pow(2,j-1)] }}
```

Figure 29: Constraint Specification of the BOER System with Computed Terms in bold

computed values in two consecutive iterations is reasonably small. The solutions extracted for the unknowns in the system are often approximate.

The class of numerical applications which can be solved through iterative methods is quite large. Many such applications are also quite amenable to parallelization. Relaxation is not, however, a universally satisfactory solution. Iterative methods may suffer from numerical stability problems. Systems using these methods might fail to terminate. Even for systems guaranteed to converge, these methods may be very slow.

A number of issues arise with respect to implementation of relaxation as an algorithm for resolution of cyclic dependencies: (i) Since there will be more than one unknown term in an unresolved constraint, how is the term to be computed selected from among all the unknown terms ? (ii) How does the compiler deal with the memory requirement for single assignment variables (Section 3.3.5) in iterative solutions ? (iii) How is the choice between the different kinds of relaxation methods (Jacobi, Gauss-Seidel etc.) made ? In the rest of this section we trace the design of the compiler for iterative solutions to cyclic constraint systems.

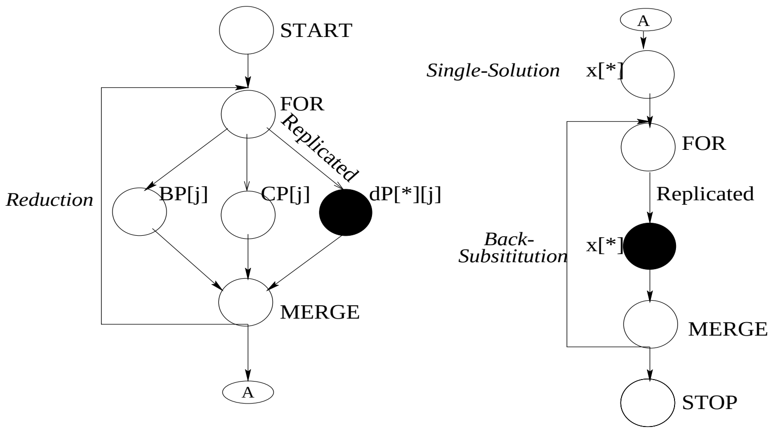


Figure 30: Dependence Graph for the BOER Program

```
PROGRAM CYCLIC_DEP1

VAR int a, b, x, y;
INPUTS a, y;

a + b == x AND x + b == y
```

Figure 31: Constraint Specification and Input Set with a Cyclic Dependency

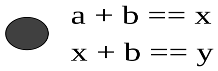


Figure 32: Directed Graph from Phase 2 for Constraint Specification in Figure 31

4.1 Selection of Term to be Computed

Constraints that remain unresolved through the basic constraint compiler are collected at the leaves of the directed graph from phase 2 and will involve more than one unknown term (except in the case of simple constraints not involving an equality). The compiler chooses one of the unknown terms in an unresolved constraint as the term to be computed and either assigns default initial values to other unknown terms or accepts such values as inputs from the user. Unresolved constraints can be of three types: a simple constraint, a constraint module call, or an indexed set of constraints (See Section 3.3.8 for a detailed description of the causes for these constraints being unresolved). The following subsections detail the selection of the computed term for the three types of unresolved constraints.

4.1.1 Unresolved Simple Constraints

Relaxation can be attempted only for simple constraints involving an equality since other types of simple constraints must be resolved as firing/routing rules. An unresolved simple constraint involving an equality has more than one unknown variable and any such variable is randomly chosen as the term to be computed. For example, consider the unresolved constraints in Figure 32. There are two unknowns b and x in both the constraints. b can be chosen as the term to be computed in the first constraint $a + b == x$. Subsequently, second constraint $x + b == y$ has just one unknown x, which is chosen as the term to be computed.

4.1.2 Unresolved Constraint Module Calls

An unresolved constraint module call has more than one unknown in its set of actual parameters, local variables and global variables in the body of the constraint module. An unknown in an actual parameter implies than the corresponding formal parameter is unknown. A constraint module call could be unresolved for either of the two following reasons.

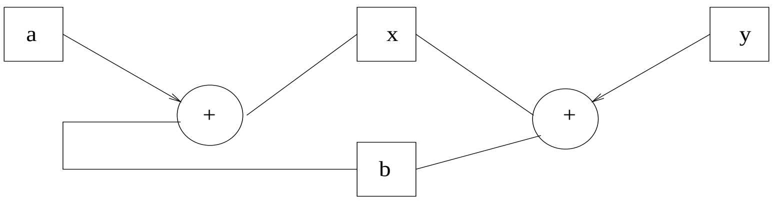


Figure 33: A Constraint Graph with a Cycle

(a) The directed graph from phase 2 for the constraint module call has unresolved constraints at the leaf of at least one path. This situation is shown in Figure 34(a) where the unresolved constraint C contains unknown variables $\{ f_1 \dots f_p, l_1, \dots l_q, g_1 \dots g_r \}$, where f_i, $1 \leq i \leq p$, is a formal parameter for the constraint module, l_i, $1 \leq i \leq q$, is a local variable for the constraint module, and g_i, $1 \leq i \leq r$, is a global variable in the body of the constraint module. Depending on the structure of C (simple constraint/constraint module call/indexed set) an unknown variable will be chosen for computation and other unknown variables will be given initial values.

(b) Some subset of the set of constraints $e_{k1} == Z_1$, $e_{k2} == Z_2$, ..., $e_{kp} == Z_p$ (See Section 3.3.3 for a description of the terms and notation) to be resolved as computations for the child node of the call node in the dependence graph which invokes the constraint module remain unresolved (see Figure 34(b)). Again, depending on the structure of each unresolved constraint a computed variable is chosen and other unknown variables are initialized.

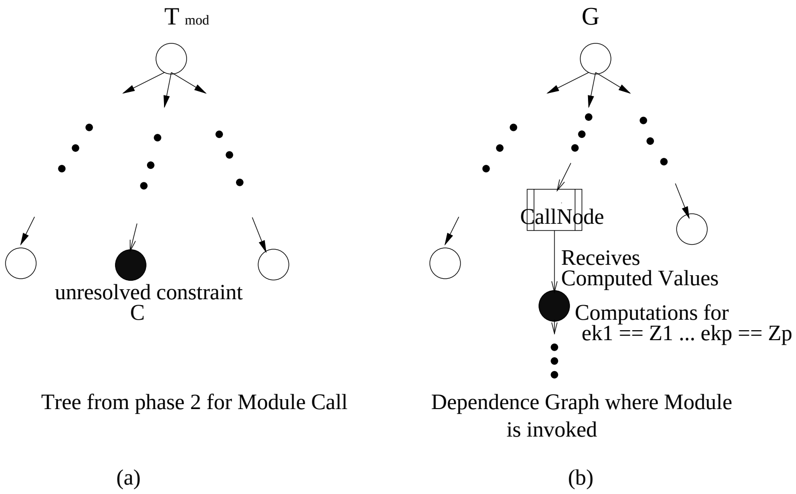


Figure 34: An Unresolved Constraint Module Call

4.1.3 Unresolved Indexed Sets

Relaxation cannot be used when an indexed set AND/OR FOR (i <b1> <b2>) $\{A_1, A_2, \dots, A_n\}$ is unresolved for the following reason (See (c) in Section 25).

- During the resolution process a constraint A_i, $1 \leq i \leq n$, is resolved as a computation for some values of i in $b1 \dots b2$ and as a conditional (firing/routing rule) for other values of i in $b1 \dots b2$.

We showed in Section 3.3.8 how the compiler can generate a closed form solution in the preceding situation. If this case does not arise, the failure to resolve an indexed set of constraints can be recursively traced to "culprit" (unresolved) simple constraints and constraint module calls nested in it (A_i, $1 \leq i \leq n$).

Consider any unresolved (simple constraint/constraint module call) constraint C nested within an unresolved indexed set. A term in C is typically of the form $A[fn_1(i_1, \dots, i_k)] \dots [fn_l(i_1, \dots, i_k)]$ where $i_1 \dots i_k$ are indices for nested indexed sets containing C, $fn_1 \dots fn_l$ are functions over the indices, and A is any structured data type. Some parts of A may be known and other parts may be unknown depending on the initial input set and the preceding computations in the current path in the dependence graph. The compiler evaluates each term in C to determine the term that accesses the largest unknown region in the structured data type. To illustrate this, consider an example constraint specification involving a $1 \times N$ array x in Figure 35. The end elements of A (shaded in Figure 36) are the inputs to the system. The values for the index i in the indexed set are in the range $2 \dots N-1$. The constraint $x[i-1] == x[i] - x[i+1]$ remains unresolved because there is no unique unknown term for all values of i in $2 \dots N-1$ (Reason (b) in Section 25). The term $x[i-1]$ accesses the region between indices $1 \dots N-2$

in the array x, the term $x[i]$ accesses the region between indices $2 \ldots N-1$ in the array x, and the term $x[i+1]$ accesses the region between indices $3 \ldots N$ in the array x (see Figure 36). Hence, the term $x[i]$ accesses the largest unknown region in x, i.e., $x[2], x[3], \ldots, x[N-1]$ and is selected as the term to be computed in the iteration process while other terms have to be given initial values for the first iteration.

```
PROGRAM CYCLIC_DEP2

VAR x;
INPUTS x[1], x[N];

AND FOR (i 2 N-1) {
      x[i-1] == x[i] - x[i+1] }
```

Figure 35: Example of an Unresolved Constraint Specification

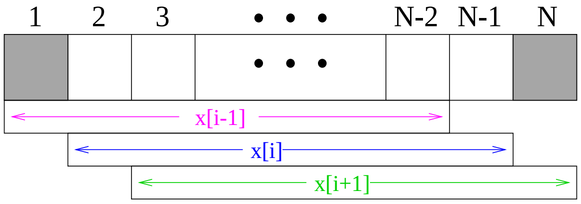


Figure 36: Regions of Access by Terms in Figure 35

The motivation behind using the heuristic of selecting the term accessing the largest unknown region as the computed term is due to the following reasons.

- Since the selected (computed) term accesses the largest unknown region, the largest number of values will be computed in each iteration of the relaxation process.
- Since the other terms access smaller unknown regions, fewer initializations will have to be done.

If the selected term does not access the entire unknown region in the data, the iterative process will not converge because certain locations in the data will not be computed. The compiler can abort the process after a fixed number of iterations, which can be a parameter in the system. Also, if the selected term accesses a location that is an initial input to the system, convergence may not be reached because that location will be overwritten in the first iteration.

4.2 Mapping single assignment variables to mutable variables

To satisfy a constraint within some degree of accuracy, the values for the selected unknown terms have to be computed over some number of iterations t. Since the basic compilation process generates single assignment variables, iterative computations would require t memory locations for each computed term. Such a memory requirement can be quite prohibitive when the values of large data structures are being computed iteratively.

To overcome the large memory requirement for computing iterative solutions with single assignment variables, a procedure for local introduction of mutable variables is required. For each variable x being computed iteratively, the compiler may keep two locations: x and old_x. Any computed value is stored in the location x. Accessed values may come from either x or old_x, depending on the relaxation scheme being used. This will be detailed in Section 4.3. At the end of each iteration, a check is done to see if the difference between values in x and old_x is greater than the specified degree of accuracy for solution of the constraints. If it is, x is copied to old_x and further iterations are initiated. The parallel functional language SISAL employs a variant of this technique [32]. In our system, the user may choose to supply a value for the degree of accuracy or accept the default value assigned by the system.

Using only single assignment variables, any computed variable with N memory locations would require $t \times N$ memory locations for t iterations. By transforming single assignment variables to be mutable variables, the memory requirement is reduced to $2 \times N$.

4.3 Relaxation Methods

Relaxation methods such as Jacobi and Gauss-Seidel [10] can be used for iterative solutions to constraints. The Jacobi method is a stationary, iterative, method typically used for solving a partial differential equation on a numerical grid. The update of each grid point depends only on the values at neighboring grid points (defined by a *stencil*) from the previous iteration. In the Gauss-Seidel method the most recent grid values are used in performing updates. To generalize these two techniques to an iterative system, the Jacobi method can be implemented by using values from the previous iteration and the Gauss-Seidel method can be implemented by using the most recent values (some possibly from the current iteration). The Jacobi method yields more parallelism since all computations in a current iteration are independent. However, convergence is typically slower than the Gauss-Seidel method.

The user should be able to choose the method of relaxation to be used by the constraint compiler. As mentioned in Section 4.2, two locations for each computed variable x are kept: x and old_x. If the chosen method of relaxation is Jacobi, the compiler restricts all accessed values of the variable x to be retrieved from the location old_x, which stores the values of variable x computed in the previous iteration. If the chosen method of relaxation is Gauss-Seidel, the compiler restricts all accessed values of variable x to be retrieved from location x which stores the most recently computed value. The compiler currently implements only the Jacobi relaxation technique.

4.4 The Laplace Equation Example

Consider the Laplace equation for a 4-point stencil on an $N \times N$ grid indexed by $(0 \dots N-1)(0 \dots N-1)$ as shown in Figure 8. A constraint specification for the problem was presented in Figure 9.

The Laplace equation specification with the input set (boundary elements) constitutes a cyclic dependency. Applying the technique described in Section 4.1, x[i] will be chosen as the term to be computed since it accesses the largest unknown region, i.e., all interior elements in the grid x. The two indexed sets in the specification are compiled to loops and the simple constraint $4 * x[i][j] - x[i-1][j] - x[i+1][j] == x[i][j-1] + x[i][j+1]$ is compiled to a computation for $x[i] : x[i][j] = (x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1])/4$.

If the Jacobi method of relaxation is chosen by the user, the constraint specification can be compiled to the procedural code shown in Figure 37. If the Gauss-Seidel method of relaxation is chosen by the user, the constraint specification can be compiled to the procedural code shown in Figure 38. The user may supply initial values for the interior (non-shaded) points of the grid or choose to accept the default initial values assigned by the compiler. Variable x is initialized to the initial values and the input boundary values. Variable old_x is initialized such that at least one point differs in value from its corresponding point in x by more than the degree of accuracy so that the first iteration can be initiated. The function *check_accuracy(x, old_x)* returns 1 if the difference between any value in x and old_x is greater than the degree of accuracy; otherwise it returns 0. The function *copy_values(old_x,x)* copies values from locations in x to corresponding locations in old_x.

4.4.1 The Dependence Graph for the Laplace Equation

Compilation of cyclic dependencies for an iterative solution has been implemented in the constraint compiler for the Jacobi method of relaxation. The Gauss-Seidel method has not yet been implemented.

In the Jacobi method of relaxation, both loops (for i and j) surrounding the computation can be executed

```
while (check_accuracy(x,old_x)) {
      copy_values(old_x,x);
      for (i 2 N-2) {
            for (j 2 N-2) {
                  x[i][j] = (old_x[i-1][j] + old_x[i+1][j] + old_x[i][j-1] + old_x[i][j+1])/4
} } }
```

Figure 37: Jacobi Relaxation for the Laplace Equation

```
while (check_accuracy(x,old_x)) {
      copy_values(old_x,x);
      for (i 2 N-2) {
            for (j 2 N-2) {
                  x[i][j] = (x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1])/4
} } }
```

Figure 38: Gauss-Seidel Relaxation for the Laplace Equation

in parallel. A naive parallelization of the loops will lead to $(N-2)^2$ computation nodes, each executing an instance of the computation $x[i][j] = (old\_x[i-1][j] + old\_x[i+1][j] + old\_x[i][j-1] + old\_x[i][j+1])/4$. This is highly undesirable since the computations are too fine-grained. To overcome this, the compiler detects instances of computation extracted from constraints specified at the scalar level. Simple data partitioning techniques are applied to partition the data involved in the computation over a specified number of computation nodes P. The data partitioning techniques are detailed in Section 5. In the Laplace equation, the grid x is partitioned in a row-wise manner across P nodes in the extracted dependence graph. Each partitioned slice in a computation node contains locations that the node computes through each iteration and any overlapping regions with other computation nodes that it accesses. For computations specified at the scalar level, as in this case, the region of overlap between computation nodes is determined by examining the terms in the computation. In the Laplace equation, the accessed terms are $x[i-1][j], x[i+1][j], x[i][j-1]$, and $x[i][j+1]$. The indices for the accessed terms specify a maximum displacement of 1 in the four directions of north, south, east, and west. Since x has been partitioned in a row-wise manner, the overlap is 1 row in the north and south directions. The row-wise partitioning of a 10×10 matrix across 4 nodes numbered $0 \ldots 3$ is illustrated in Figure 39(a). Each node i, $0 \leq i \leq 3$, gets rows in the range $2*i \ldots 2*(i+1)+1$.

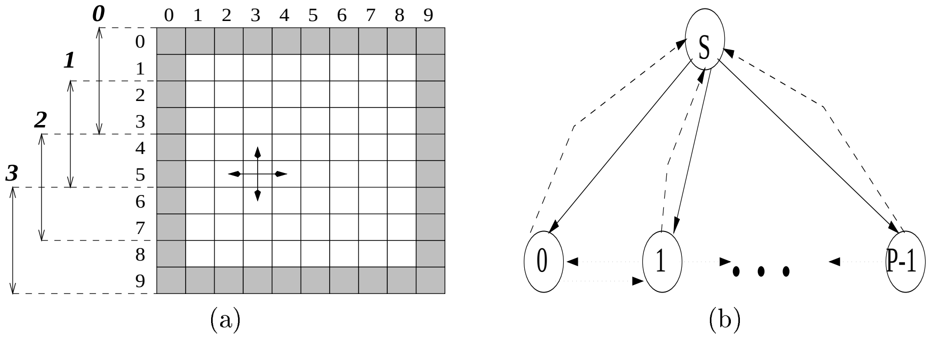


Figure 39: (a) Data Partitioning for the Laplace Equation (b) Dependence Graph for the Laplace Equation

In Figure 39(b) we show the dependence graph extracted by the compiler for a Laplace equation system executing on P nodes. The super node S initiates new iterations. The computation nodes numbered $0 \ldots P-1$ each have a slice of approximate size $\frac{N}{P} \times P + 2$ (overlap between slices) of the matrix x. In each iteration the code in Figure 37 is executed by each computation node on its local slice. At the end of each iteration overlapping regions are exchanged between computation nodes and the super node is informed by each computation node whether the degree of accuracy has been reached for the values in the local slice. Computation is terminated when all the nodes achieve convergence on individual slices.

5 Execution Environment Specification

The advantage of using a program specification that is independent of the execution machine is portability - the ability to create executables for different architectures without changing the program specification. The constraint program specifications in our system are translated to an intermediate architecture-independent dependence graph which can be mapped to many different parallel machines. However, there are many architectural mechanisms which can be exploited by an executable program if it is directed to do so. This usually leads to an improvement in performance. Without violating the "sanctity" of our architecture-independent program specification, we propose an execution environment specification, separate from the constraint program, that allows the user to provide useful hints to the compiler about the underlying execution machine. The compiler can use these hints to produce programs that may be more optimized for performance.

This section discusses the design of the execution environment specification for our compiler. Several features are discussed in individual subsections. While some of them have been implemented in our system, there are several others which could be added in the future.

5.1 Shared Variables

In shared memory architectures such as the Sparc and Cray J90, a vast improvement in performance can be obtained if some variables are declared as shared because it avoids the copying of large data across computation nodes. This was demonstrated through the BTS example [15] where the program using shared variables shows a dramatic improvement in performance over the one not using shared variables.

The user has to be cautious when declaring shared variables in a program containing constraints connected by OR operators. OR operators translate to multiple paths in the dependence graph and hence, give rise to the potential for multiple solutions. In a program not using shared variables, each path can compute a solution independent of other paths. However, a path in the dependence graph for a program using shared variables may overwrite the value computed for a variable in another path. Hence, the user should not declare variables as shared if there is the potential for multiple solutions for them, which can be determined from the constraint specification by inspection.

5.2 Number of Available Processors

This piece of information can be used to determine the number of nodes to be created when spawning off a computation to be executed in parallel. For example, the N iterations in a loop structure can be partitioned across P processors such that each processor gets approximately $\frac{N}{P}$ iterations to execute or the data computed within a loop structure can be partitioned equally across P processors. Since CODE allows the dynamic creation of nodes, the number of processors can determine the number of computation nodes at runtime.

5.3 Data Partitioning with Overlap Sections

It has been amply demonstrated by many parallel programming experiments that data partitioning techniques play a significant role in improving performance. While, currently, we have only implemented simple mechanisms for data partitioning, we show in this section that other sophisticated mechanisms can be specified too.

In many applications such as the Laplace equation, the computations are specified at a very fine granularity, say, at the scalar level. When the compiler detects that the operations involved in the computations are over scalar types or over small-sized data (the threshold size is fixed by a parameter to the system), it partitions the computed variables over a number of nodes. This is especially important if the computation is nested within loops because the computation is executed repeatedly and the overhead in executing scalar operations repeatedly can severely degrade performance.

The form of the partition depends on the data accesses in the computation. For any matrix, if the accesses are only in the north and/or south directions the data is partitioned column-wise. If the accesses are only in the east and/or west directions the data is partitioned row-wise. If there are accesses in mixed directions, say north and east, the data is partitioned such that there is minimum overlap between the partitioned slices. This scheme minimizes the overhead in the synchronizations necessary when data is shared across computation nodes. Each node gets approximately $\frac{N \times M}{P} + overlap$, where the matrix being partitioned is of size $N \times M$ and P is the number of nodes. The amount of overlap between partitioned slices must be determined by the user or by the compiler by inspecting the terms in the computation. The mechanism of partitioning data involved in scalar computations has been used for the Laplace equation.

The user may specify the partitioning mechanism, instead of allowing the compiler to select it, by indicating the actual regions in the data type to be distributed across the nodes. (The user must specify the actual overlap between the partitions to determine the regions to be synchronized.) This mechanism is yet to be implemented in our compiler.

5.4 Option of not Parallelizing a Module

A constraint module may have very fine-grained operations in the constraints for the constraint module body. Parallelizing such a module may lead to degradation in performance due to the overheads involved. For this reason, a user can denote that the dependence graph for a particular module call should be mapped to a sequential procedure rather than a parallel one. This feature has not yet been implemented in our compiler. However, since CODE allows the generation of sequential programs, this would be simple to incorporate.

5.5 Selecting Operations to be Executed in Parallel

Operations over structured data types are primitives in the type system. But parallel execution can be selected for these primitive operations. The complexity of some of these operations may be larger than others. An example is the matrix-matrix multiplication operation. In the interests of performance, it would be beneficial to extract such operations out of a computation to execute in parallel. For example, if there is a computation $(d_1 \triangle d_2) \bigtriangledown (d_3 \triangle d_2)$, where $\triangle$ and $\bigtriangledown$ are primitive operations, to be executed and the operation $\triangle$ is very computation-intensive, the specification can be split into two computations to be executed in parallel: $(d_1 \triangle d_2)$ and $(d_3 \triangle d_2)$. The results can then be merged and operator $\bigtriangledown$ can be applied on them. We use this technique in the BTS example where multiple matrix-matrix multiply operators in a computation are executed in parallel.

The execution environment specification provides a platform for the user to indicate that some operations be selected for extraction from a computation for subsequent parallel execution.

5.6 Choices among Parallel Algorithms to execute some of the Operations

A variety of choices exist among parallel algorithms to execute operations on data instances under a type system. The user should be able to select one among a number of implemented algorithms in the system to execute an operation. We have not yet implemented this feature in our system.

6 Performance Results

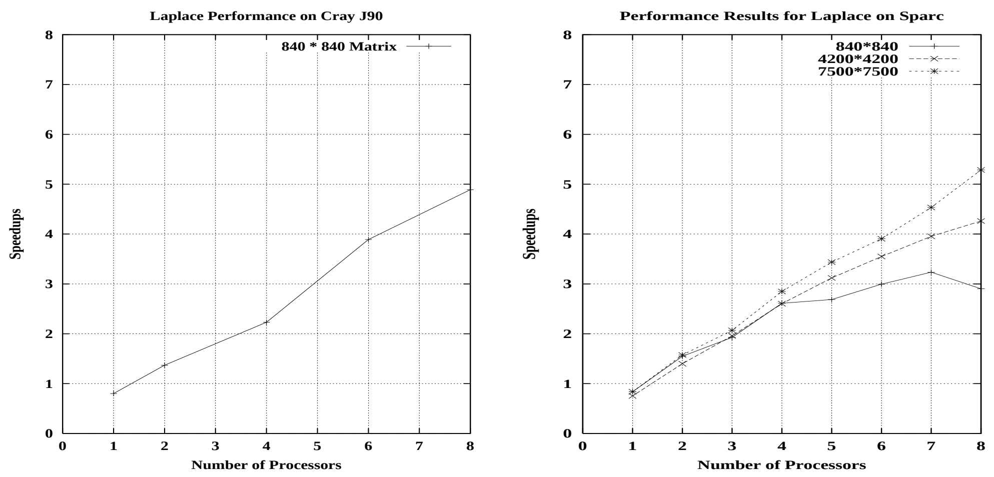
A prototype of the constraint compiler has been implemented in C++ using object-oriented techniques. A number of examples have also been programmed and executed on the Cray J90, SPARCcenter 2000, Enterprise 5000, Sequent Symmetry machine [26], and the PVM system. [15] reported the performance results for the BTS program on the Sequent Symmetry and Sparc machines and the BOER program on a Sparc machine. In this section we present the performance results for the Laplace Equation.

The execution times reported in this section are wall clock times. Whenever possible, timings have been taken for executions during either dedicated CPU access or when the loads on the machines were low.

6.1 The Laplace Equation

Figure 40(a) presents speedups for the solution of the Laplace Equation for an 840×840 grid over a sequential implementation of the solver on a CRAY J90. Figure 40(b) presents speedups for the solution of the Laplace Equation for different sizes of the grid over a sequential implementation of the solver on a Sparc (Enterprise 5000) with 8 processors. Both executions used the Jacobi method of relaxation. As expected, the number of iterations for convergence was very high. Consequently, the time taken for experiments ran into several hours. The results presented here were obtained for a fairly small degree of accuracy so that the number of iterations were small. This is a reasonable setup for the experiments since each iteration took approximately the same amount of time.

It can be seen that the speedups obtained for the CRAY are fairly good considering the small size (approximately 800×800) of the grid which is partitioned across the nodes. The results on the Sparc show that as the size of the problem increases, corresponding to an increase in the amount of computation at each node, the speedups improve. Page faults marred the results for grid sizes greater than 7500 and hence are not reported.



(a) (b)

Figure 40: Performance Results for Laplace Equation Program on (a) a CRAYJ90 (b) an Enterprise 5000

7 Related Work

Our research is related to work in two fields: parallel programming and constraint programming. The goals in these areas have been quite different. Constraint programming research has focussed on providing constraints as an attractive platform for sequential and concurrent programming both for general and special purpose environments. Not much emphasis has been placed on performance in concurrent constraint systems. In contrast, the parallel programming community has mainly concerned itself with performance, although many programming models have been proposed over the years. We have attempted to bridge the motivations in the two areas through this research.

Parallel programming has a vast spectrum of research to its credit over the years. In comparison, constraint programming research has been modest in scope. One of the earliest constraint systems was Sketchpad [31] developed by Ivan Sutherland in 1963. But, most of the work in the field did not begin until the 1980s. However, since then there has been a flurry of activity which has continued until the present day. Out of the many outstanding research projects in the two fields, we have selected those most closely related to this research for discussion.

7.1 Constraint Programming

7.1.1 Consul

The goal of Consul [1] resembles ours in that it is to extract parallelism from constraints. But the approach is different in that interpretative local propagation is used to find values satisfying the system of constraints. This approach has little hope of extracting efficient programs and offers performance only in the range of programs written in logic languages. A noteworthy feature of the language is that it offers sets (with set operations) as a primitive data type.

7.1.2 Thinglab

Thinglab [12] transforms constraints to a compilable language rather than to an interpretive execution environment. It is a constraint-oriented graphic simulation laboratory where constraints are compiled to sequential procedural code. The compilation of simple constraints is done by storing all possible transformations for a single constraint. No compilation for structures involving indexed sets or constraint modules is done. Also, the system is not concerned with the extraction of parallel structures, which is our major concern.

7.1.3 Kaleidoscope

Kaleidoscope [11] integrates two different paradigms - constraint programming and imperative programming - within the same programming language. It could be basically seen "either as adding constraints to imperative programs, or as adding control flow to constraint programs" [13]. Programs in the language mix object-oriented constructs with constraints. As in Thinglab, compilation to procedural code has been attempted. However, Kaleidoscope does not target parallelism and the compiled constraint structures are simple (as defined in Section 2.3).

7.1.4 Concurrent Constraint Programming

Vijay Saraswat described a family of concurrent constraint logic programming languages, the cc languages [30]. The logic and constraint portions are explicitly separated with the constraint part acting as an active data store for the logic part. The logic communicates with the constraint part only through constraints either by a "tell" operation (a new constraint is added to the store) or an "ask" operation (to check if a constraint is consistent with the store).

A number of systems based on the cc model were developed. The most notable among them is Oz which is a concurrent programming language based on an extension of the basic concurrent constraint model provided in [30]. The performance reported for the system is comparable with commercial Prolog [22].

7.2 Parallel Programming

7.2.1 Automatic Parallelization of Sequential Programs

Attempt to parallelize Fortran with additional specifications have not been broadly successful [8, 27]. There have also been attempts to parallelize languages such as Scheme and other dialects of Lisp [19, 14].

There are several algorithms for generation of efficient (optimal or near optimal) parallel code from straight-line code [23]. There are a number of algorithms for fast parallel evaluation of expressions [17]. These algorithms can, in some cases, be applied to the sequential code segments in the macro-parallel computation structures generated by our compilation process as further optimizations or to obtain micro-level parallelism when it can be exploited by the underlying execution environment.

7.2.2 Extension of Procedural Languages with Directives for Parallelism

There are many extensions of sequential languages with directives for parallelism. We consider only two of them in this subsection.

Declarative extensions have been added as part of High-Performance Fortran(HPF) [29]. This is a data-parallel programming language designed for portability and an implicitly parallel programming style with some optimization directives. In these respects, it is similar to our system. But, the difference lies in the program specification styles: procedural versus declarative. Also, HPF does not address task parallelism.

CC++ [3] is a parallel programming language designed by providing extensions to the object-oriented language C++. It incorporates the fundamental ideas from compositional programming: synchronization variables and parallel composition.

7.2.3 Parallel Logic Programming

PCN [5] and Strand [9] are two parallel programming representations with a strong component of logic specification. Both require the programmer to provide explicit operators for specification of parallelism and the dependence graph structures which could be generated were restricted to trees.

7.2.4 Parallel Functional Programming

Functional languages are unidirectional (a single variable is computed) as opposed to constraint languages which are multi-directional (any variable in a constraint can be computed). A number of parallel systems have emerged from the functional programming area. Most prominent among them are SISAL, EPL, Id, Crystal, and the PTRAN system. A description of these systems can be found in [32]. While SISAL and Id are pure implicitly parallel languages, EPL provides mechanisms for expressing mapping and scheduling constraints. PTRAN provides explicitly parallel control structures.

7.2.5 Equational Systems

Equational specifications of computations [25] are a restriction of constraint specifications. Unity [4] is an equational programming representation around which Chandy and Misra have built a powerful paradigm for the design of parallel programs. Unity requires addition of explicit specifications for parallelism.

7.2.6 Miscellaneous Systems

Other related work includes technical computing environments like MATLAB [21] which integrate numerical analysis and matrix computation.

We, of course, use the methods of parallel compilation [2] to derive procedural programs from constraint systems. The approach of Pandey and Browne [28] expresses parallel computation as constraints on the order of execution of units of computations. We derive this ordering from the constraints and an input set of variables. Collins [6] expresses matrix computations as hierarchical type declarations and translates to implementations which maximizes use of type information. We use hierarchical matrices as a tool for controlling granularity.

Some reasons why our approach has greater potential for obtaining efficient parallel programs than parallel logic languages, compilation of functional languages to parallel execution and/or concurrent constraint logic programming languages include (i) A constraint specification system can provide a richer set of primitives than is given in current logic languages. (ii) Functional languages restrict dataflow to be unidirectional whereas constraint systems impose no constraints on dataflow. (iii) Data parallelism is more readily expressed in constraint specifications than in pure functional languages. (iv) Concurrent constraint satisfaction systems currently rely on interpretive methods for evaluation of constraint satisfaction whereas we compile to direct procedural code. (v) Narrowing the target domain and direct use of semantic domain knowledge enable the compiler to choose efficient algorithms for the derived computations.

8 Conclusions

In this paper we enhance the capability of our primary constraint compiler to target constraint systems which can be solved iteratively. This capability allows our programming system to encompass many parallel iterative numerical applications like the Laplace Equation and the Partial Differential Equation solvers. Our primary compiler generates single assignment variables which pose a large memory requirement for iterative solutions. We discuss techniques to overcome this requirement for different relaxation methods used for the iterative solutions. Simple data partitioning techniques have been incorporated. A prototype compiler for the system has been developed. We present performance results for several examples including the Laplace Equation.

9 Future Work

Our research has established the feasibility of expressing computations for parallel execution as constraints. It also paves the way for a number of future research endeavors. In this final section, we present a few of the possible future directions for research.

Constraint specifications do not explicitly specify algorithms. However, algorithms can be extracted from them by supplying appropriate input sets of variables. Our system opens up the possibility of extracting algorithms for applications where algorithms are difficult to specify. One such application is a Partial Differential Equation Solver using domain decomposition methods where it is far easier to specify the constraints that must be maintained within a particular domain and across the boundaries of domains than to specify how to maintain these constraints. A number of issues such as the factors determining the optimality of the extracted algorithms require research.

An issue that has been raised is "how to write 'good' constraint specifications". One could write a specification in which a large number of function calls and very few constraints are used. Take, for example, the Barnes Hut problem in parallel programming. One could express the building of the quad-tree as a call to a procedural function or a constraint module. Of course, the decision to choose one representation over the other might be based on whether or not to parallelize the particular computation. On the other hand, we can provide the capability (through the execution environment specification) of not parallelizing a particular constraint module. So, this decision might be based on other factors such as expressibility.

In Section 3.3.6 we explained how effective programs (programs with a single solution) can be extracted by the compiler through choosing only a single path in the dependence graph for execution. The choice of a path among a number of paths is based on factors such as maximal output (computed) set of variables. This needs further investigation.

Other application areas such as boundary matching for decomposed domains, data mediation, and back tracing in circuits each offer opportunities for constraint specification systems.

The current representation and compilation system are a solid feasibility demonstration. But much more could be done in the domain of matrix computations. Complete implementation of the execution environment specification and the full feature set for the hierarchical type system need to be done. Additionally, further steps in this research are to define the semantics of recursion in constraint module calls, to relax the requirement that all of the bounds in indexed sets of constraints be statically found, and to incorporate higher-order solvers for unknowns in computations.

10 Acknowledgments

This work was supported in part through grants from the Advanced Research Projects Office/CSTO (subcontract to Syracuse University #3531427) and DARPA (grant #DABT63-92-0042). The experiments were run on E5000 UltraSPARC machines which were a donation from SUN Microsystems.

References

[1] Doug Baldwin. A status report on CONSUL. In Nicolau Gelernter and Padua, editors, *Languages and Compilers for Parallel Computing*. MIT Press, 1990.

[2] Utpal Banerjee. *Loop Parallelization.* A Book Series on Loop Transformations for Restructuring Compilers. Kluwer, 1994.

[3] K. M. Chandy and Carl Kesselman. CC++: A declarative concurrent object oriented programming notation. Technical Report CS-TR-92-01, California Institute of Technology, 1992.

[4] K.M. Chandy and J. Misra. *Parallel Program Design : A Foundation.* Addison-Wesley, 1989.

[5] K.M. Chandy and S. Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett, 1992.

[6] T.S. Collins and J.C. Browne. Matrix++: An object-oriented environment for parallel high-perfomance matrix computations. In *Proc. of the Hawaii Intl. Conf. on Systems and Software*, 1995.

[7] J.J. Dongarra and D.C. Sorenson. SCHEDULE: Tools for developing and analyzing parallel fortran programs. Technical Report 86, Argonne National Laboratory, November 1986.

[8] R. Eigenmann and W. Blume. An effectiveness study of parallelizing compiler techniques. In *Proc. Intl. Conf. Par. Proc.*, volume II, pages 17–25, 1991.

[9] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, 1990.

[10] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume I. Prentice Hall, 1988.

[11] B. Freeman-Benson and Alan Borning. The design and implementation of Kaleidoscope '90, a constraint imperative programming language. In *Computer Languages*, pages 174–180. IEEE Computer Society, April 1992.

[12] Bjorn N. Freeman-Benson. A module compiler for Thinglab II. In *Proc. 1989 ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, October 1989.

[13] B.N. Freeman-Benson. Constraint imperative programming. Technical Report 91-07-02, Department of Computer Science and Engineering,University of Washington, August 1991.

[14] Luddy Harrison and David A. Padua. Parcel: Project for the automatic restructuring and concurrent evaluation of Lisp. In *Proceedings of the 1988 International Conference on Supercomputing,*, pages 527–538, 1988.

[15] Ajita John and J.C. Browne. Compilation of constraint systems to procedural parallel programs. In D. Sehr, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Workshop on Languages and Compilers for Parallel Computers*, number 1239 in Lecture Notes in Computer Science, pages 501–518. Springer-Verlag, 1996.

[16] Ajita John and J.C. Browne. Extraction of parallelism from constraint specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume III, pages 1501–1512, August 1996.

[17] R.M. Karp and V. Ramachandran. *Handbook of Theoretical Computer Scince*, chapter Parallel algorithms for shared memory machines, pages 869–941. Elsevier Science, 1990.

[18] S. Lakshmivarahan and Sudarshan K. Dhall. *Analysis and Design of Parallel Algorithms: Arithmetic and Matrix Problems.* Supercomputing and Parallel Processing. McGraw-Hill, 1990.

[19] James R. Larus and Paul N. Hilfinger. Restructuring Lisp programs for concurrent execution. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 81–90, 1989.

[20] William Leler. *Constraint Programming Languages.* Addison-Wesley, 1988.

[21] The MathLab Group, Laboratory for Computer Science, MIT, Cambridge, Mass. *MACSYMA Reference Manual*, January 1983.

[22] Michael Mehl, Ralf Scheidhauer, and Christian Schulte. An abstract machine for Oz. In *Programming Languages, Implementations, Logics and Programs, Seventh Intl. Symposium, LNCS*, number 982. Springer-Verlag, September 1995.

[23] Gary L. Miller, Vijaya Ramachandran, and Erich Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM Journal on Computing*, 17(4):687–695, 1988.

[24] P. Newton and J. C. Browne. The CODE 2.0 graphical parallel programming environment. In *Proc. of the Intl. Conf. on Supercomputing*, pages 167–177, July 1992.

[25] Michael J. O'Donnell. *Equational Logic as a Programming Language.* MIT Press, 1985.

[26] A. Osterhaug. *Guide to Parallel Programming on Sequent Computer Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[27] D. A. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 12(29):1184–1201, December 1986.

[28] Raju Pandey and J. C. Browne. Event-based composition of concurrent programs. In *Workshop on Languages and Compilers for Parallel Computation*, August 1993.

[29] Harvey Richardson. High Performance Fortran: History, overview and current developments. Technical Report TMC 261, Thinking Machines Corporation.

[30] Vijay A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie Mellon, School of Computer Science, Pittsburgh, 1989.

[31] I. Sutherland. SKETCHPAD: A man-machine graphical communication system. In *IFIPS Proc. of the Spring Joint Computer Conference*, January 1963.

[32] B. K. Szymanski. *Parallel Functional Languages and Compilers.* ACM Press, 1991.