A Common Intermediate Language and its Use in Partitioning Concurrent Declarative Programs

Z. M. Ariola, B. C. Massey, M. Sami, and E. Tick

CIS-TR-94-05a
August 1994

Abstract

The plethora of concurrent declarative language families, each with subtly different semantics, makes the design and implementation of static analyses for these languages a demanding task. However, many of the languages share underlying structure, and if this structure can be exploited, static analysis techniques can be shared across language families. These techniques can thus provide a common kernel for the implementation of quality compilers for this entire language class.

The purpose of this paper is to exploit the similarities of non-strict functional and concurrent logic languages in the design of a common intermediate language (CIL). The CIL is introduced incrementally, giving at each step the rationale for its extension. As an application, we present, in CIL form, some state-of-the-art static partitioning algorithms from the literature. This allows us to "uncover" the relative advantages and disadvantages of the analyses, and determine promising directions for improving static partitioning.

An earlier version of this paper was presented at the ICOT-NSF Workshop on Parallel Logic Programming and its Programming Environments at the University of Oregon, Eugene, Oregon, March 1994 (proceedings available as UO CIS-TR-94-04). The current version was submitted to *New Generation Computing* for its Special Workshop Issue, August 1994.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

Contents

1 Introduction

"Great wits are sure to madness near allied,
And thin partitions do their bounds divide."
John Dryden
Absalom and Achitophel, 1680

The compilation of concurrent declarative languages, such as Id and Strand, for high-performance multiprocessors depends fundamentally on increasing task granularity (partitioning) and hiding various latencies such as memory accesses. The plethora of language families, each with subtly different semantics, makes the design and implementation of static analyses, the backbone of a good compiler, a demanding task. However, many of the languages share underlying structure, and if this structure can be exploited, static analysis techniques can be shared across language families.

The process of identifying portions of a program that can be executed as threads, referred to as *partitioning*, has been a subject of growing interest to the implementors of concurrent logic programming languages and functional languages, here referred to collectively as concurrent declarative languages. In this paper, the important issues concerning partitioning both concurrent logic programming languages and non-strict functional languages are presented, and some of the important partitioning techniques are discussed. The similarities between non-strict functional languages with single-assignment features and committed-choice languages with logic variables lead us to believe that partitioning techniques applicable in one language domain will be successful in the other. The partitioning algorithm and its associated static analyses are the kernel of any high-performance compiler for concurrent declarative languages. We believe that a good first step in understanding the state-of-the-art in partitioning and how to improve it is to define a common intermediate language (CIL) for both non-strict functional and concurrent logic languages, and evaluate partitioning algorithms with respect to the CIL.

This paper is organized as follows. Section 2 reviews the family of concurrent declarative languages. Section 3 defines a CIL covering a slightly restricted subset of this family. Section 4 reviews the partitioning problem and three algorithms proposed in the literature, using a CIL program as a comparative example. This comparison reveals strong and weak points in each approach, and suggests directions for future exploration. A summary of the research, conclusions, and discussion of future work is given in Section 5.

2 Review of Concurrent Declarative Languages

This paper focuses on the compilation of concurrent declarative languages containing what we refer to as *dynamic-single-assignment (DSA) variables.* This family includes Id [21], pH [5] (the parallel version of Haskell [14]), Strand [10], and many others. DSA variables correspond to the "I-structures" of Id [6], and to the "logic variables" of Strand. A DSA variable begins its life unbound, and may be bound only once during program execution. However, in contrast to static-single-assignment variables, the point at which a DSA variable is bound is not in general statically determinable: binding can occur at any time during program execution. Attempts to read a DSA variable before it is bound will typically cause the reader to suspend. Once bound, attempts to rebind a DSA variable will "fail," and will typically cause execution of the entire program to fail.[1]

In this section, we review two languages with DSA variables, chosen from the two language families of principle interest in this paper: the concurrent functional language Id, and the concurrent logic programming language Strand. We note some of the similarities between these languages, and thus motivate the introduction of our CIL.

Id is a higher-order, non-strict, strongly-typed functional language, augmented with I-structures [6] and M-structures [7]. I-structures are arrays whose elements get "refined" during the course of computation: that is, it is possible to create an I-structure without giving a definition for each of its elements. (In a purely functional language a variable is given exactly one binding or definition at creation time.) For our purposes, single-element I-structures are DSA variables [20]. M-structures add non-determinism to the languages; in this paper, however, we will not address this feature. Id has non-strict semantics, that is, a function is invoked even though its parameters may not be completely evaluated yet. However, unlike other modern functional languages, non-strictness is not implemented in terms of lazy evaluation. Id has a *parallel evaluation strategy*, that is, all computations are performed in parallel, not merely those whose results are sure to be needed. Thus, the parameters of a function are all evaluated in parallel with the evaluation of its body, even though the body might not need some parameter. For example, consider the following Id definitions:

$$\text{Def } nth\ (x : xs)\ n = \text{If } n == 1 \text{ then } x \text{ else } nth\ xs\ (n-1)$$
$$\text{Def } Ints\ x = x : (Ints\ (x+1))$$

and the main expression: $nth\ (Ints\ 1)\ 5$. (Note: The expression $x : xs$ denotes a list whose head is x and tail is xs.) The function nth is invoked even though the evaluation of the first parameter does not terminate. The evaluation of the main expression will return 5 as a result. This is accomplished by evaluating the expression $Ints\ 1$ in parallel with the body of the nth function. Note that the main expression returns a value even though its evaluation does not terminate. In this specific example, we can safely assume that 5 is the final answer. However, in the general case, due to the presence of single-assignment variables, the result must be interpreted as a partial answer, since the answer could be refined further. In particular, as mentioned before, the partial answer could be refined to failure in case of multiple bindings; this condition cannot be checked statically.[2]

All variables in a Strand program are DSA variables, although some may be bound immediately upon creation. Syntactically, a program consists of a sequence of procedures, each of which consists of a sequence of *clauses*. Each clause is of the form "*Head* :- *Guard* | *Body*" where the head is a term and the guard and body are sequences of terms. Execution is defined by *goals* corresponding to procedure invocations, also called tasks and processes. Each goal specifies a procedure and its arguments: the head and body of each clause of

[1]With regard to logic programs, we restrict ourselves to committed-choice languages with "eventual" tell unification [24] or assignment, rather than "atomic" tell unification, thus implying that binding failure causes program failure.

[2]The reader may refer to Ariola and Arvind [2] for further clarification. The parallel evaluation of Id has been adopted by the newly developed pH language [5], a parallel variant of the Haskell language [14]. Haskell is increasingly the standard language in the non-strict functional language research community.

the procedure are matched against the goal. If one or more clauses match the goal, one clause is selected for *commitment*. The committed clause's body will consist of a (possibly empty) sequence of new goals, which are all evaluated concurrently. If a goal fails to match because not enough information is available, the goal will suspend. If the goal fails to match for other reasons, the goal will cause failure of the program. A program may also fail by *deadlock* which occurs when all goals have suspended. Finally, as mentioned above, a program may fail if a DSA variable is assigned to a second time, thus informally there are three sources of program failure.

There are marked similarities among the members of this language family. Execution is described in terms of *eager concurrency*: all program units will be executed concurrently as soon as they are reached, the only source of delay being suspension due to an attempt to read an unbound DSA variable. The concurrency in the languages is very fine-grained — many concurrent tasks will be spawned, and each of these tasks will be relatively small, such as a goal or function evaluation. Finally, the fact that tasks suspend when trying to read an unbound DSA variable implies that the languages are subject to the possibility of deadlock, and thus the semantics are truly concurrent. If an attempt is made at compile time to decide the order of execution of tasks (*e.g.,* for uniprocessor execution), care must be taken to ensure that some latter task is not responsible for providing a binding which some earlier task needs to proceed.

This problem of static scheduling to increase task granularity while avoiding deadlock is a main component of compilers for executing these languages on high-performance multiprocessors [11, 16, 19, 22, 23, 29]. Comparative analysis of different partitioning techniques is best achieved by first developing an intermediate language common to the DSA language family. A common language allows us to accurately compare currently proposed analyses, as well as develop a single, unified analysis technique. The latter will help concurrent declarative language researchers to avoid unconsciously duplicating past research. The advantage of defining an intermediate *language* as opposed to an intermediate *form* (*i.e.,* a non-executable structure) is that the operational semantics of the language can assist us in formalizing and showing correctness of the static analyses.

The common intermediate language (CIL) or *kernel* language presented attempts to draw together the functional and logic programming communities. By sharing a CIL foundation, it will be easier to share ideas and implementations, exchange programs and reduce harmful diversity. However, we do not seek uniformity. There are clear differences between these languages. But having a common framework will allow us more easily to highlight and study these differences. In the next section, we develop the CIL incrementally, with a rationale for each extension.

3 Common Intermediate Language

In this section we introduce the main features of our kernel language through examples. In particular, we are interested in capturing the essential features needed to support DSA variables. At the outset we assume that a CIL program produces a unique result, independently of evaluation order. Thus, for instance, we disallow Prolog programs that produce multiple solutions, and concurrent logic procedures with non-mutually exclusive guards that are

```
sum(Xs, S) ← sum'(Xs, 0, S).                                 §1

sum'([ ], S, S).                                             §2
sum'([X | Xs], P, S) ← plus(X, P, P'), sum'(Xs, P', S).      §3

plus(0, 0, 0).
plus(0, 1, 1).
plus(1, 0, 1).
plus(2, 0, 2).
plus(2, 1, 3).
   ⋮
```

Figure 1: Sample Prolog Program

```
sum(Xs) → sum'(Xs, 0)                              §1
sum'([ ], P) → P                                   §2
sum'([X | Xs], P) → { P'  =  plus(X, P)            §3
                      S   =  sum'(Xs, P')
                       in  S }

plus(0, 0) → 0
plus(0, 1) → 1
plus(1, 0) → 1
plus(2, 0) → 2
plus(2, 1) → 3
   ⋮
```

Figure 2: Sample Program in Kernel Language

truly indeterminate, *e.g.*, stream mergers (which can easily be relegated to system-defined builtins).

Both concurrent logic languages and functional languages have gone through some evolution (and deevolution [27]) to support both parallelism and determinacy. Functional languages, on the one hand, restrict the handling of state, while concurrent logic languages restrict unification and backtracking. Consider the Prolog program in Figure 1.

Given a list of numbers,[3] sum computes the sum of its elements. Two selection rules are in play: clauses are selected top-down, and body goals are selected in left-to-right order. Backtracking undoes bindings that lead to local and nonlocal failures. In this example, even if the body goals of sum′ were reversed, or the clauses of plus were reordered, a query with a ground input list would be guaranteed by backtracking to produce a correct solution (although possibly very inefficiently!)

We can rewrite the above example program in a committed-choice language, removing

[3]We represent a list as [$X|Xs$], where X is the head and Xs is the tail of the list.

backtracking and restricting the clause selection to matching. In Parlog [8] or Strand the program remains textually identical to the one in Prolog, with the inclusion of *mode declarations*: sum(?,?), sum′(?,?,^), and plus(?,?,^). The input mode '?' indicates that the passed parameter must be bound and the '^' indicates that the argument binding is produced by the procedure itself. Other committed-choice languages indicate these *ask* and *tell* unifications in syntactically different ways, but the idea is the same.

In functional language terminology, we say that plus is *strict* in both of its input arguments. We can translate the above clauses into our kernel language as shown in Figure 2.

A typical query is written as $\{Res = \mathsf{sum}([1,2])\ \mathsf{in}\ Res\}$. The kernel language semantics allows a variable to appear on the left-hand side of only one equation (the output instance), and on the right-hand side of multiple equations (the input instances). The key point enabling this translation is the determination of variable modes. This is difficult to do in many cases, and impossible in general for an unrestricted language, as we discuss in the next section.

Note that the kernel language is essentially a *graph rewriting system* [1, 4]: each clause is translated into a rewrite rule. Note also that in order to respect the communication pattern expressed in a logic program via logical variables, the kernel language must allow assignment of a name to each computation. For example, in the atom $\mathsf{plus}(X, P, P')$, P' denotes the result of the operation. We make this explicit by saying $P' = \mathsf{plus}(X, P)$. Thus the right-hand side of our rewrite rules are not just simple terms, as in rule §1, but can be a set of equations of the form $\alpha_1 = t_1, \cdots, \alpha_n = t_n$, where all left-hand-side variables are distinct. The in keyword denotes the result of the term: for example, the result of rule §3 is S. Note that in the term $\{\alpha_1 = t_1, \cdots, \alpha_n = t_n\ \mathsf{in}\ \alpha_i\}$ the order of the equations is irrelevant. We discuss graph rewriting semantics in Section 3.2, after first dealing with mode analysis.

3.1 Mode Analysis

In logic programs, including committed-choice programs, no syntactic distinction is made between variables and their subterms which have *output mode* and are thus being bound in an occurrence, and those which have *input mode* and are thus being read in an occurrence. Thus, some form of mode analysis (*e.g.*, [28, 31]) or mode declarations [26] are required for translation into our kernel language. However, these methods are not perfect. For example, array elements can be aliased by indexing accesses, which means we might not be fully knowledgeable about all modes in an array. Furthermore, without good general aliasing analysis, these unknown array modes could "contaminate" the remainder of the program (to date, no mode analyzers do the sort of aliasing analysis required to effectively control this contamination).

Another point is that a certain class of automatic mode analyzers require language restrictions based on the concept of *permissible modes* [25, 31]. Sundararajan [25] defines a static analysis technique for determining a minimal set of permissible modes for a Prolog program. This is complicated by the lack of restrictions on Prolog unification. As discussed earlier, committed-choice languages restrict unification to ask and tell, making mode analysis considerably less complex. To further simplify the analysis, Ueda and Morita [31] restrict the language family to a "fully-moded" subset wherein each logic variable can have only a single producer. Furthermore, the subset requires equal modes in corresponding argument positions of a procedure's defining clauses (and all corresponding subterms *within* those arguments). These restrictions enable efficient, practical mode analysis [28].

Note that in the concurrent logic program of Figure 1 the processes spawned in clause §3 communicate through the logical variable P', while in our setting the communication is rendered via a data dependency, as shown in clause §3 of Figure 2. This provides a hint for later phases of compilation that it is safe to sequentialize plus and sum'. However, if the implementation desires to execute $\mathsf{plus}(X, P)$ and the call to sum' in parallel then P' will have to be implemented as a DSA variable. The key point is that instead of restricting programs to use only DSA variables (as in concurrent languages), programs written in the kernel language may explicitly indicate points where synchronization is not needed to respect the language semantics. The kernel language implementation may still eliminate some synchronization points through static analysis, as well as inserting synchronization points if desired.

There are programs that cannot be consistently moded, and thus cannot be translated into the kernel language as it now stands. Furthermore, even fully-moded programs can have variable occurrences in which some portion of the occurrence's term structure is read, and another portion written. This condition also precludes translation. We discuss an extension to deal with both of these problems in Section 3.4, after first discussing the basic rewriting semantics.

3.2 Rewrite Semantics

Let us now present the execution of a query to the previous example following a rewrite semantics. We start out with the term $\{Res = \mathsf{sum}([1,2]) \ \mathsf{in} \ Res\}$, by applying rule §1 we obtain:

$$\{Res = \mathsf{sum}'([1,2],0) \ \mathsf{in} \ Res\}$$

by applying rule §3 we have:

$$\begin{array}{lcl} \{ \ Res & = & \{ \ \begin{array}{rcl} P'' & = & \mathsf{plus}(1,0) \\ S' & = & \mathsf{sum}'([2],P'') \\ & & \quad \mathsf{in} \ S'\} \end{array} \\ \quad \mathsf{in} \ Res\} & & \end{array}$$

By pattern matching $\mathsf{sum}'([1,2],0)$ with the left-hand side of rule §3 we have created the bindings $X = 1$, $Xs = [2]$: we then create an instance of the right-hand side of the rule. This corresponds to the renaming done before matching in concurrent logic languages, and can also be seen as the allocation of a frame in the context of a procedure call. In the kernel language we want to eliminate all syntactic sugar, and thus we will consider the

$$
\begin{array}{lcl}
\mathit{SE} & \in & \text{Simple Expression} \\
\\
\mathit{Variable} & ::= & X \mid Y \mid Z \mid \cdots \\
\mathit{Constant} & ::= & \mathit{Integer} \mid \mathit{Boolean} \mid \cdots \\
\mathit{SE} & ::= & \mathit{Variable} \mid \mathit{Constant} \\
\mathit{Term} & ::= & \mathit{SE} \mid \mathsf{f}^n(\mathit{SE}_1, \cdots, \mathit{SE}_n) \\
\mathit{Block} & ::= & \{[\mathit{Variable} = \mathit{Term}]^* \ \mathsf{in}\ \mathit{SE}\} \\
\mathit{Clause} & ::= & \mathit{Term} \rightarrow \mathit{Block} \mid \mathit{Term} \rightarrow \mathit{Term} \\
\mathit{Program} & ::= & [\mathit{Clause}]^*
\end{array}
$$

Figure 3: The Grammar of Kernel$_0$

above program equivalent to:

$$
\begin{array}{lcl}
\{\ \mathit{Res} & = & S' \\
\ \ P'' & = & \mathsf{plus}(1,0) \\
\ \ S' & = & \mathsf{sum}'([2], P'') \\
\ \ \mathsf{in}\ \mathit{Res}\}
\end{array}
$$

That is, the internal nesting of blocks does not matter. In the above program we can now execute, for example, the third expression, obtaining:

$$
\begin{array}{lcl}
\{\ \mathit{Res} & = & S' \\
\ \ P'' & = & \mathsf{plus}(1,0) \\
\ \ S' & = & \mathsf{sum}'([\,], P''') \\
\ \ P''' & = & \mathsf{plus}(2, P'') \\
\ \ \mathsf{in}\ \mathit{Res}\}
\end{array}
$$

No conflict of names will arise, because at each step of the execution new names are introduced. Note also that $\mathsf{plus}(2, P'')$ cannot be executed until P'' gets a value, that is, until $\mathsf{plus}(1,0)$ has executed.

There is a major difference between the computational model presented here and the one usually employed to give the operational semantics of concurrent languages. In our approach we do not keep a separate component describing the current value of each variable. Instead, these "substitutions" are kept directly in the term itself. Our preliminary kernel language is basically a system of recursive equations over a first-order signature Σ, as illustrated in Figure 3. Note that each functional symbol is applied to simple expressions only: that is, we assume that each expression has been given a name. It is possible to show that the term-rewriting system of Figure 3 is *confluent*, though this is outside the scope of this paper.

3.3 Multiple Values

Concurrent logic procedures can also return multiple values, as shown in the following Strand example:

$$\begin{aligned}
&\mathsf{distribute}([\mathsf{send}(1,X) \mid Xs], Out1, Out2) \leftarrow \\
&\qquad Out1 := [X \mid Out1'], \mathsf{distribute}(Xs, Out1', Out2). \\
&\mathsf{distribute}([\mathsf{send}(2,X) \mid Xs], Out1, Out2) \leftarrow \\
&\qquad Out2 := [X \mid Out2'], \mathsf{distribute}(Xs, Out1, Out2'). \\
&\mathsf{distribute}([\,], Out1, Out2) \leftarrow Out1 := [\,], Out2 := [\,].
\end{aligned}$$

In order to translate the above clauses we need to extend Kernel_0 by allowing multiple return values. The need for multiple-return-value facilities in intermediate languages has been illustrated in the context of functional languages [3]. A new language, Kernel_1, is obtained by changing the production of the syntactic category *Block* as follows:

$$Block ::= \{[Variable \cdots Variable = Term]^* \ \mathsf{in}\ SE \cdots SE\}$$

Thus the translation of the above program will be:

$$\begin{array}{lrcl}
\mathsf{distribute}([\mathsf{send}(1,X) \mid Xs]) \rightarrow \{ & Out1 & = & [X \mid Out1'] \\
 & Out1'\ Out2 & = & \mathsf{distribute}(Xs) \\
 & & \multicolumn{2}{l}{\mathsf{in}\ Out1\ Out2\}} \\
\mathsf{distribute}([\mathsf{send}(2,X) \mid Xs]) \rightarrow \{ & Out2 & = & [X \mid Out2'] \\
 & Out1\ Out2' & = & \mathsf{distribute}(Xs) \\
 & & \multicolumn{2}{l}{\mathsf{in}\ Out1\ Out2\}} \\
\mathsf{distribute}([\,]) \rightarrow [\,]\ [\,]. & & &
\end{array}$$

According to the semantics of the previous section, an invocation of a procedure with multiple return values will be rewritten into multiple expressions. These expressions correspond to the return values and are evaluated concurrently. If a concurrent program is fully moded, *i.e.*, if no occurrence of a variable corresponds to multiple modes, then the program can be translated into a Kernel_1 program. In the next section, we tackle the problem of translating programs that do not meet these constraints.

3.4 Dynamic-Single-Assignment (DSA) Variables

A more powerful way to synchronize concurrent expression evaluation in the CIL is via dynamic-single-assignment (DSA) variables. Consider the following Strand program:

$$\begin{aligned}
&\mathsf{counter}(X) \leftarrow \mathsf{counter}'(X, 0). \\
\\
&\mathsf{counter}'([\mathsf{clear} \mid Xs], C) \leftarrow \mathsf{counter}'(Xs, 0). \\
&\mathsf{counter}'([\mathsf{add} \mid Xs], C) \leftarrow C' is C + 1, \mathsf{counter}'(Xs, C'). \\
&\mathsf{counter}'([\mathsf{read}(X) \mid Xs], C) \leftarrow X := C, \mathsf{counter}'(Xs, C). \\
&\mathsf{counter}'([\,], C).
\end{aligned}$$

Mode analysis can detect that the (first argument) stream is *input* and also that the variable C is *input*; however, the stream contains further variables that need to be instantiated. Kernel$_1$ does not allow this because procedure formal parameters are required to be bound by the caller. In other words, a variable is simply a name of a value and every variable is bound at the point where it is allocated. In order to translate the above program into the CIL we now extend the kernel language with DSA variables.

A DSA variable can be allocated and bound at two different program points. It is allocated via the Mk-Var primitive. Associated with a DSA variable there are two operations: *read* and *write*. The *read* operation fetches the value of a DSA variable: however, if no value has yet been given to the DSA variable, the *read* is suspended until a *write* operation gives the variable a value. Moreover, if an attempt is made to *write* a DSA variable that has already been written, an error will be raised and execution of the program will fail. We thus extend the set of predefined function symbols with two new operators: Store of arity two (a DSA variable and the value to be stored), and Select, of arity one (the DSA variable). Finally, we will relax the restriction that a block must return a value, since some blocks may be invoked only for their side-effects. We use the special identifier "_" as indication that the expression does not return a meaningful result.

The translated program will be:

$$\mathsf{counter}(X) \rightarrow \mathsf{counter}'(X, 0).$$

$$\begin{array}{lcl}
\mathsf{counter}'([\mathsf{clear} \mid Xs], C) \rightarrow \mathsf{counter}'(Xs, 0). & & \\
\mathsf{counter}'([\mathsf{add} \mid Xs], C) \rightarrow \{\ C' & = & C+1 \\
\qquad\qquad\qquad\qquad\qquad\quad \_ & = & \mathsf{counter}'(Xs, C')\} \\
\mathsf{counter}'([\mathsf{read}(X) \mid Xs], C) \rightarrow \{\ \_ & = & \mathsf{store}(\mathsf{X}, \mathsf{C}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \_ & = & \mathsf{counter}'(Xs, C)\} \\
\mathsf{counter}'([\,], C) \rightarrow \{\,\}. & &
\end{array}$$

For example, we can invoke the counter with the following expression:

$$\begin{array}{lcl}
\{\ A & = & \mathsf{Mk\text{-}Var} \\
\ \ B & = & \mathsf{Mk\text{-}Var} \\
\ \ \_ & = & \mathsf{counter}([\mathsf{add}, \mathsf{add}, \mathsf{read}(A), \mathsf{add}, \mathsf{read}(B)])\}
\end{array}$$

As with Kernel$_1$, it is possible to show that the extended term-rewriting system is *confluent*.

In order to express the notion of guards present in concurrent logic languages we need to further extend the kernel language with the concept of conditional rewrite rules. Thus for example, we will allow a rule to be expressed as: $l \rightarrow r\ P$, where P is a set of "flat" predicates, *e.g.*, builtins such as $\leq/2$. We will say that the above rule is applicable only if P is true. The syntax of our final kernel language, Kernel$_2$, is given in Figure 4.

4 Partitioning Techniques

Compile-time partitioning of concurrent declarative languages is difficult, because any partitioning technique must be able to significantly increase the thread size while avoiding

Variable	::=	$X \mid Y \mid Z \mid \cdots$
Constant	::=	$\mathit{Integer} \mid \mathit{Boolean} \mid \cdots$
SE	::=	$\mathit{Variable} \mid \mathit{Constant}$
Term	::=	$\mathit{SE} \mid \mathsf{f}^n((\mathit{SE})_1, \cdots, (\mathit{SE})_n) \mid \mathsf{Select}(\mathit{SE}) \mid \mathsf{Mk\text{-}Var}$
Block	::=	$\{[\mathit{Statement}]^* \ \mathsf{in}\ \mathit{SE} \cdots \mathit{SE}\}$
Statement	::=	$\mathit{Binding} \mid \mathit{Command}$
Binding	::=	$\mathit{Variable} \cdots \mathit{Variable} = \mathit{Term}$
Commands	::=	$\mathsf{Store}(\mathit{SE}, \mathit{SE})$
Clause	::=	$\mathit{Term} \rightarrow \mathit{Block}\ [\mathit{Predicate}]^* \mid \mathit{Term} \rightarrow \mathit{Term}\ [\mathit{Predicate}]^*$
Predicate	::=	$<(\mathit{SE}, \mathit{SE}) \mid >(\mathit{SE}, \mathit{SE}) \mid \cdots$
Program	::=	$[\mathit{Clause}]^*$

Figure 4: The Grammar of Kernel$_2$

the accidental introduction of deadlock. Sometimes the correct order of execution of two processes can only be determined dynamically. Such processes have to be identified and placed in separate threads. This is especially intricate for concurrent logic programming languages, because of the prevalence of logic variables, which can cause hidden cycles through *aliasing*.[4] Non-strict functional languages have analogous problems with hidden cycles through I-structures.

Several researchers have explored the problem of thread partitioning of non-strict functional languages [11, 16, 22, 23, 29]. While most of the techniques produce threads within a function and in isolation from the rest of the program, the analysis of Traub *et al.* [30], referred to in this paper as *DD analysis*, attempts to improve the thread size within a function by propagating global dependence information. The analysis we call *AB analysis* (after the sets A and B used in the analysis), proposed by Coorg [9], is a modification of DD analysis which achieves better partitioning across recursive calls. Partitioning techniques have also been proposed in the context of granularity analysis for concurrent logic programming languages, most significant of which is the analysis proposed by King and Soper [18], referred to in this paper as *coloring analysis.* This technique also uses global dependence information to produce partitions.

Higher granularity threads lead to lower process-creation, management, and communication overheads. However excessive sequentialization could be detrimental to performance, because useful parallelism may be lost. Unlike DD and AB analyses, coloring analysis attempts to control the granularity of the threads by performing a rather crude granularity analysis [17]. This is done by classifying terms as constant, linear, or non-linear depending upon the estimated cumulative difference between the growth of computation and communication during the execution lifespan. The constant class corresponds to terms which are builtins, or are defined in terms of constant terms. The linear class corresponds to either non-recursive or linearly recursive functions. These are fine-grained, whereas the non-linear class (neither constant nor recursive) corresponds to functions which are typically coarse-grained. This classification gives a simple prescription for controlling granularity: two terms

[4]Aliasing occurs when two or more identifiers are simultaneously bound to the same variable

which are non-linear are allocated to different partitions, whereas any other pair of terms can potentially be allocated to the same partition.

The similarities of non-strict functional languages with I-structures and committed-choice languages with logic variables lead us to believe that partitioning techniques applicable in one language domain will be successful in the other. While the techniques developed for different language families differ in their methodology and input, there are also some similarities. The translation of both non-strict functional languages and committed-choice languages into the common intermediate language (Kernel$_2$, introduced in Section 3) provides a common framework for examining the different analysis techniques. This facilitates "uncovering" the relative advantages and disadvantages of the analyses, and determining future directions for developing improved partitioning analyses.

In Section 4.1, we present the definitions of thread which are used in the analyses. We present the three analyses discussed above in Sections 4.2, 4.3, and 4.4. These descriptions are simplified from the originals for expository purposes. The similarities and differences between the three partitioning techniques are discussed in Section 4.5. In order to compare and contrast the different techniques, a single example will be used as an illustration throughout. Figure 5 presents a Strand program with variables renamed apart. Figure 6 presents the program's translation into the CIL. Though no DSA variables are needed for the translation, the variables I' and L' in the function f are defined as DSA variables for the sole purpose of illustration.

4.1 Thread Definitions

The proposed techniques for partitioning differ in their formalizations of the notion of "thread." This difference has significant influence on the complexity of the techniques, and on the results produced by them. It is thus important that the notion of thread used in each technique be clearly defined.

- *Liberal Thread:* A thread is defined to be a totally ordered set of atomic actions of a program. Every function may have one or more threads. Thus a thread can itself include calls to other threads, and also *split-phase*[5] memory accesses, which implies that a thread may have to suspend and wait for other threads to complete before it can continue.[6]

  The liberal definition is commonly used in partitioning concurrent logic programming languages, because it corresponds well to their operational semantics: since the binding of logical variables is typically used for thread synchronization, threads will typically suspend until some logical variable is bound, and then resume execution.

- *Conservative Thread:* A thread is defined to be a subset of the atomic actions comprising a program, such that: 1) the thread is contained within a single function; 2) a compile-time ordering can be determined for the thread which is valid in all contexts

[5]Split-phase memory accesses involve separate request and response messages, potentially long latencies, and potentially out-of-order responses.

[6]The inclusion of function calls in a thread could lead to more than one thread being spawned concurrently in order to execute the callee. A stricter definition might require the callee to be completely sequentialized.

```
f(A,B,G) ← true |
        H is L+1,
        J is I+2,
        g(A,H,I,M),
        h(A,J,L),
        B is I+M,
        G is M+L.

g(C,D,P,Q) ← C > 0 |
        E is C−D,
        F is C+D,
        g(E,F,R,S),
        P is R+1,
        Q is R+S.

g(C,_,P,Q) ← C ≤ 0 |
        P is C−2,
        Q is C+3.

h(K,_,T) ← K > 0 |
        T is K−4.

h(K,N,T) ← K ≤ 0 |
        T is K−N.
```

Figure 5: A Sample Strand Program

```
f(A) → { H    =  Select(L')+1     §1
         J    =  Select(I')+2     §2
         I M  =  g(A,H)           §3
         L    =  h(A,J)           §4
         B    =  I+M              §5
         G    =  M+L              §6
         _    =  Store(I,I')      §7
         _    =  Store(L,L')      §8
                 in B G}

g(C,D) → { E    =  C−D
           F    =  C+D
           R S  =  g(E,F)
           P    =  R+1
           Q    =  R+S
                   in P Q} C > 0.

g(C,_) → { P  =  C−2
           Q  =  C+3
                 in P Q} C ≤ 0.

h(K,_) → { T  =  K−4
                 in T} K > 0.

h(K,N) → { T  =  K−N
                 in T} K ≤ 0.
```

Figure 6: The Sample Program in the CIL

in which the containing function can be invoked, and 3) once the first instruction in a thread is executed, it is always possible to execute each of the remaining instructions in the compile-time ordering without pause, interruption, or execution of instructions from other threads.

The conservative definition is used in partitioning "dataflow" languages. Synchronization only occurs at the start of the thread, and a thread, once scheduled, runs to completion. Thus, function calls and split-phase memory accesses result in multiple threads. Also, multiple clauses result in separate threads for each clause.

Both conservative and liberal threads wait until their input variables have values. Thus synchronization is required for the input variables. In the case of liberal threads, synchronization may also be required for variables accessed inside the thread. This synchronization is achieved with a family of strict identity operators S_n (after Coorg [9]). S_n waits until all of its n arguments are defined, and then returns the arguments as its result.[7]

4.2 DD (Demand-Dependence) Analysis

The *DD analysis* technique proposed by Traub *et al.* [30] uses dependence and demand information to reduce modified forms of dataflow graphs. Their partitioning technique is based on a greedy algorithm which attempts to make the threads as large as possible. This approach avoids the danger of oversequentializing because of the size limits imposed by the conservative definition of the thread, the language model, the use of split-phase accesses, and the control paradigm. The program to be partitioned is expressed as a structured dataflow graph, which consists of a collection of *directed acyclic graphs* (DAGs) describing CIL blocks and *interfaces* which describe how the blocks relate to one another. The predicate used to select a clause of a multiple-clause CIL function is encoded in the interface.

A CIL program can be translated into this representation using a methodology similar to the one described by Nikhil [21]. Figure 7 shows the structured dataflow graph representation of the sample CIL program shown in Figure 6. The representation of function h is not given in Figure 7 due to space limitations. The vertices of the DAGs describing the basic blocks represent primitive operators, and edges indicate dependences. These are augmented with annotations which encode information about some operators' behavior and interrelationships: these annotations are used extensively during analysis. In addition to the arithmetic and logical operators, the operator set includes *send* and *recv* ("receive") operators which communicate values between basic blocks corresponding to function linkage in the CIL. I-Fetch and I-Store are synchronizing split-phase memory operators, where the response to an I-Fetch is not received until a value is written at that location. The response of an I-Fetch is received by an I-F-Resp ("I-Fetch Response") operator. Every Select term in the source CIL program is replaced by an I-Fetch and I-F-Resp pair. Nodes 1 and 2 of Figure 7 have been transformed in this way. Similarly, a Store command in the source CIL program is translated into an I-Store operator.

Three forms of dependences are expressed in the graph representation: *certain*, *indirect*, and *potential* [30]. A *certain dependence*, indicated by a straight edge between two nodes, represents an ordering constraint due to a data dependence which holds for every invocation of the function. In Figure 7 the edge connecting nodes 3 and 4 indicates a certain dependence. An *indirect dependence*, represented by a squiggly edge, is a dependence completed through one or more vertices in a different basic block. An indirect dependence exists between nodes 1 and 2 of Figure 7. A *potential dependence* represents an ordering constraint due to a data dependence that may either differ depending on the context of function invocation (say because of a query), or is otherwise unknown at compile time. All vertices that may be end-points of incoming potential dependences are annotated with *inlet* annotations and those that may be outgoing potential dependences are annotated with *outlet* annotations. These annotations, represented by sets of names, link potential corresponding outlets and inlets. In Figure 7 there is a potential dependency between nodes 29 and 23, and between nodes 40 and 23 (via variable P'). These dependencies are completed through

[7]Thus, an S_n operator is equivalent to n S_1 operators, one for each variable. An S_1 operator is similar to the Select operator of the CIL (Section 3.4).

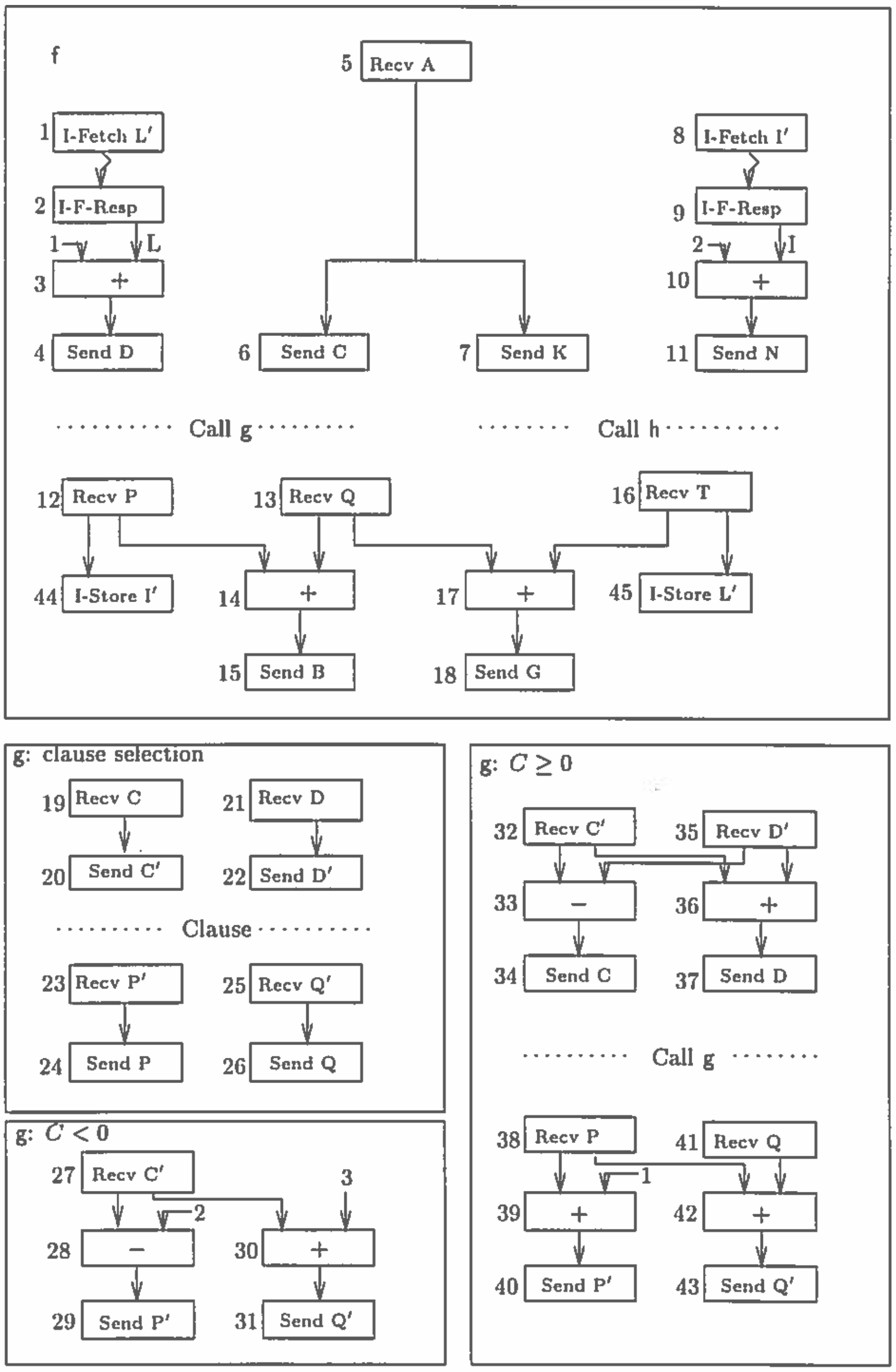

Figure 7: Structured Dataflow Graph Representation of the Sample Program

the two clauses of function g. The dependencies are potential because it is not known at compile time which of the two clauses will be selected by a particular call.

The overall partitioning technique consists of the iterative application of two stages: *basic-block partitioning* and *global propagation*. We now explain each of these in turn.

4.2.1 Basic-Block Partitioning

Basic-block partitioning is used to group the terms of a CIL block into disjoint subsets that are trivially mapped into threads. The basic-block partitioning algorithm, which is applied to the DAG corresponding to each of the CIL blocks, consists of *dependence-set* partitioning and *demand-set* partitioning. Dependence-set partitioning forms partitions by grouping together all vertices that depend on the same set of inlets. Demand-set partitioning groups together all vertices which are demanded by the same set of outlets. The basic-block partitioning algorithm iteratively applies dependence-set and demand-set partitioning. This is done by considering each thread formed during a partitioning pass as a node of a reduced graph, and applying the next iteration to the reduced graph. Thus dependence-set and demand-set partitioning algorithms are repeatedly applied, forming a reduced graph at each stage, until the number of partitions does not change.

Since no distinction is made between certain and indirect edges, there may be two vertices in the same partition which are connected by an indirect edge. *Subpartitioning* is applied to divide such a partition into subsets so that the nodes connected by an indirect edge are in different partitions. Subpartitioning is performed after each iteration of the dependence-set and demand-set partitioning algorithms.

As an example of basic-block partitioning and subpartitioning, consider the graph representation of function f in Figure 7 leading to the partitioning shown in Figure 8. A detailed trace of propagation is shown in Figure 9. Initially the dependence set of nodes 14 and 15 is $\{P, Q\}$. Similarly each pair of nodes {12,44}, {17,18}, and {16,45} have equal dependence sets. Hence these pairs are placed in the same partitions during the dependence iteration shown in the lower portion of Figure 9. These partitions are labeled T_6, T_8, T_{17}, and T_{18} in Figure 8. Node 13 is not placed in any of these partitions because its dependence set is different from the others. Partitioning this portion of the graph does not require additional iterations; however, in general multiple passes may be needed.

Simple demand-set partitioning would place node 1 which fetches L, in the same partition as the nodes dependent upon it, *i.e.*, nodes 2, 3, and 4. Since a split-phase access cannot be placed in a conservative thread, subpartitioning is used to identify the presence of an indirect edge in a partition and then split the partition across the indirect edge. Hence node 1 is placed in a separate partition, and not with nodes 2, 3, and 4.

4.2.2 Global Propagation

The effectiveness of the basic-block partitioning algorithm is limited by its inability to propagate dependence and demand information across function call boundaries. Global propagation derives dependence information across function call boundaries and uses it to improve partitioning. The information gained during the partitioning of a function p may be used to form larger threads when partitioning another function q which calls p, in contrast

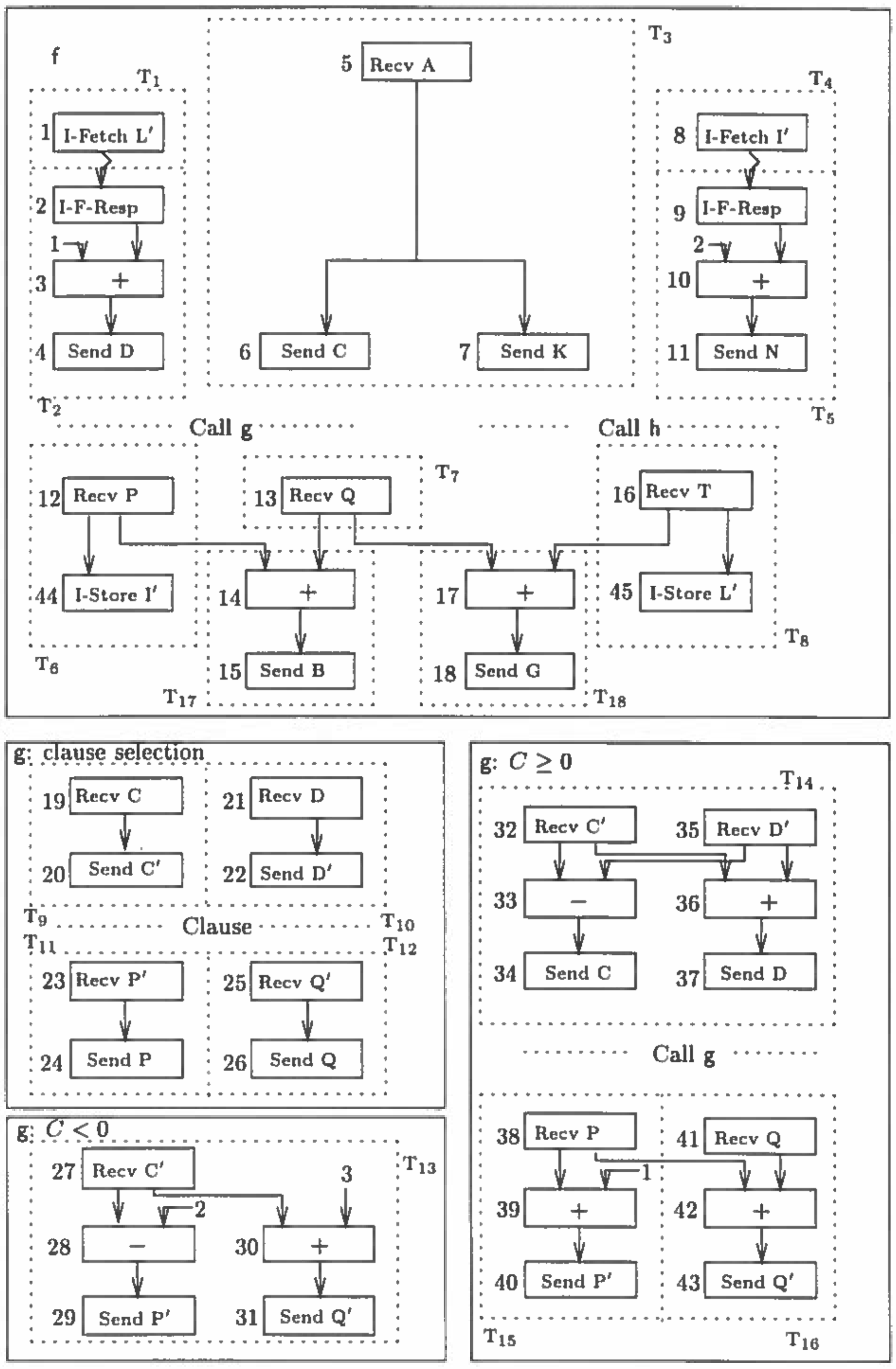

Figure 8: Sample Program After Basic-Block Partitioning

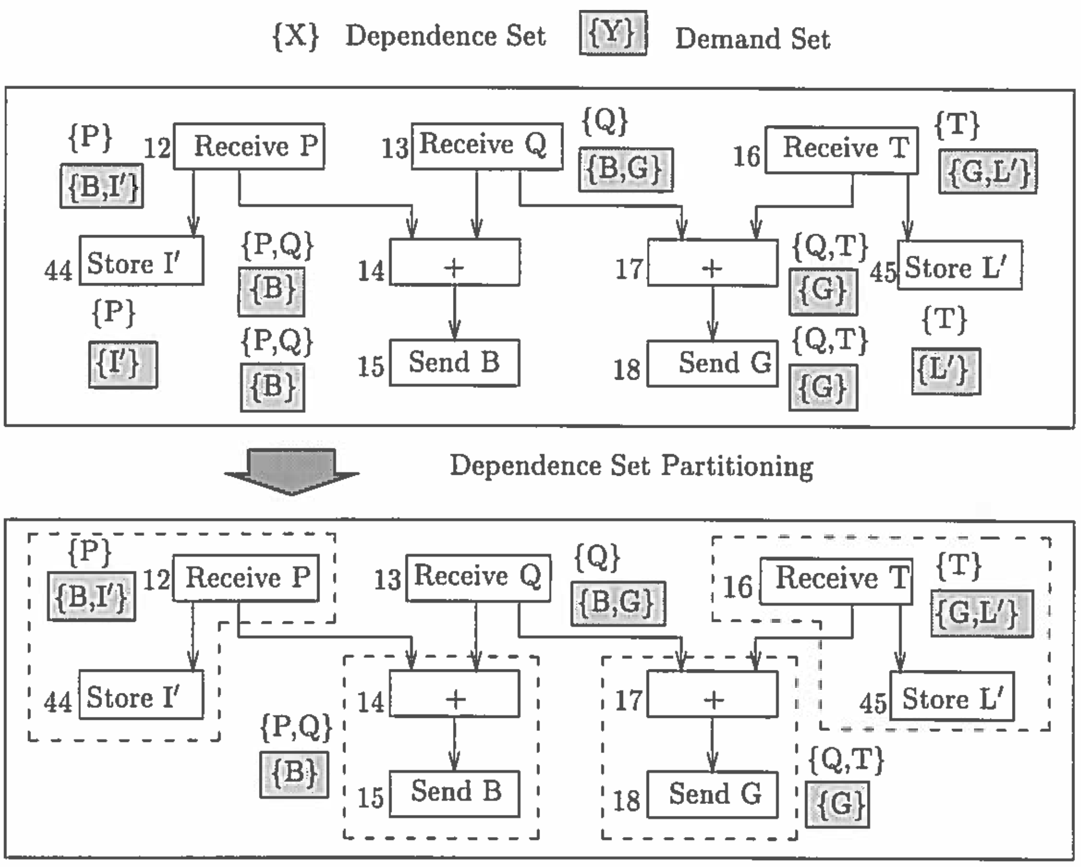

Figure 9: Detailed Propagation of Dependence Sets

to partitioning q in isolation. If q is known to be the only caller of p, then the partitioning information from q can likewise improve partitioning of p. This process is iterated to arrive at mutually improved versions of both q and p.

As an abstract illustration, consider the partitions for function g in Figure 8. The input variable P for partition T_{15}, and the input variable Q for partition T_{16} are always produced together, either by partition T_{13} or by partition T_{14}. If this dependence information were to be propagated across the recursive call to function g, it would lead to merging of partitions T_{15} and T_{16} into one partition, say T_{17}. Global propagation would then infer that the two result arguments of the call to g from f are produced together. This new dependence information would then be used to improve the partitioning of function f by merging partitions T_6 and T_7. In a similar manner, demand information could also be globally propagated.

Unfortunately, DD analysis is not sophisticated enough to perform the above optimization for recursive functions (it does fine for non-recursive functions). The problem lies in the following: to combine basic-block partitioning and global propagation, a selection strategy is needed to decide whether to repartition the basic blocks, or to globally propagate. The selection strategy chosen by DD analysis is to form a call tree and alternately sweep from the leaves to the root and from the root to the leaves. At each node, representing a function composed of blocks, the node is repartitioned and the results are propagated alternately upwards or downwards. The motivation behind using a tree is to facilitate termination. For recursive programs, first a call graph is constructed. Next, a spanning tree is built from the call graph. Finally, propagation is limited to function call interfaces in the spanning tree, since propagating over a general call graph might result in non-termination because of cycles.

For example, consider function g in Figure 8. In DD analysis, no information can be propagated across the recursive call to g in the $C > 0$ clause. This prevents improvement of the partitioning of blocks across the recursive call, because the dependence/demand information has to be propagated through the certain and indirect edges of the callee, which is function g itself. In contrast, the effect of this global optimization is obtained by AB analysis (Section 4.4).

The CIL representation of the partitioned function f produced by DD analysis is given in Figure 10. The program uses S_n synchronization operators. There are 12 threads noted in the function corresponding to those partitions outlined in Figure 8 (and also threads T_{19} and T_{20} representing the procedure calls). We purposely did not simplify the code to emphasize its correspondence with the dataflow graph; however, assuming simple peephole optimizations,[8] the code shown in Figure 11 can be produced. Although the CIL semantics does not guarantee sequential execution within each thread, sequential execution is acceptable. Certainly sequential code will be generated by most implementations, and the synchronization operators specify just that.

4.3 Coloring Analysis

King and Soper's *coloring analysis* [18] is an example of a partitioning technique that was developed specifically for the family of concurrent logic programming languages. It was motivated by the serial combinator proposed as the grain for parallelism in the context of lazy graph reduction [13], which uses both data dependence information and granularity information to build threads.

In all of the analyses considered in this paper, individual clauses of a program are considered in isolation, *i.e.*, partitions do not include more than one clause. Coloring analysis produces liberal threads, *i.e.*, the partitions may include functions calls. In contrast, in DD and AB analyses function calls break partitions.

Coloring analysis needs global information to infer the directionality of dataflow between variables. To obtain this information, we perform mode analysis during the translation of committed-choice logic programs to CIL programs (Section 3.1). King and Soper obtained this information by means of abstract interpretation [18]. In DD analysis, global analysis is performed during the actual partitioning step by propagating dependence information

[8]For example, there is no need to issue a split-phase memory access for L (T_1) because the result (L_3) will be strictly synchronized (T_2).

$$\begin{array}{lllll}
\mathsf{f}(A) \to \{ & L_3 & = & \mathsf{Select}(L') & T_1 \\
 & L_2 & = & \mathsf{S}_1(L_3) & T_2 \\
 & H & = & L_2 + 1 & \\
 & A_1 & = & \mathsf{S}_1(A) & T_3 \\
 & A_2 & = & A_1 & \\
 & A_3 & = & A_1 & \\
 & I_3 & = & \mathsf{Select}(I') & T_4 \\
 & I_2 & = & \mathsf{S}_1(I_3) & T_5 \\
 & J & = & I_2 + 2 & \\
 & I_1 & = & \mathsf{S}_1(I) & T_6 \\
 & \_ & = & \mathsf{Store}(I_1, I') & \\
 & I_4\ M_2 & = & \mathsf{S}_2(I_1, M_3) & T_{17} \\
 & B & = & I_4 + M_2 & \\
 & M_3 & = & \mathsf{S}_1(M) & T_7 \\
 & L_1 & = & \mathsf{S}_1(L) & T_8 \\
 & \_ & = & \mathsf{Store}(L_1, L') & \\
 & L_4\ M_1 & = & \mathsf{S}_2(L_1, M_3) & T_{18} \\
 & G & = & M_1 + L_4 & \\
 & I\ M & = & \mathsf{g}(A_2, H) & T_{19} \\
 & L & = & \mathsf{h}(A_3, J) & T_{20} \\
 & & & \mathit{in}\ B\ G\} &
\end{array}$$

Figure 10: Partitions of f After DD Analysis

$$\begin{array}{lllll}
\mathsf{f}(A) \to \{ & L_3 & = & \mathsf{Select}(L') & T_1 \\
 & L_2 & = & \mathsf{S}_1(L_3) & T_2 \\
 & H & = & L_2 + 1 & \\
 & A_1 & = & \mathsf{S}_1(A) & T_3 \\
 & I_3 & = & \mathsf{Select}(I') & T_4 \\
 & I_2 & = & \mathsf{S}_1(I_3) & T_5 \\
 & J & = & I_2 + 2 & \\
 & I_1 & = & \mathsf{S}_1(I) & T_6 \\
 & \_ & = & \mathsf{Store}(I_1, I') & \\
 & B & = & I_1 + M_3 & T_{17} \\
 & M_3 & = & \mathsf{S}_1(M) & T_7 \\
 & L_1 & = & \mathsf{S}_1(L) & T_8 \\
 & \_ & = & \mathsf{Store}(L_1, L') & \\
 & G & = & M_3 + L_1 & T_{18} \\
 & I\ M & = & \mathsf{g}(A_2, H) & T_{19} \\
 & L & = & \mathsf{h}(A_3, J) & T_{20} \\
 & & & \mathit{in}\ B\ G\} &
\end{array}$$

Figure 11: Peephole Optimized f

across function call boundaries. However in coloring analysis (and in AB analysis) the global analysis phase *precedes* the actual partitioning step.

A data dependence exists between two terms of a CIL function if one (or more) of the results produced by one term may be consumed by the other term. This idea is formalized by the data dependence relation $Depend_c$, which is a relation defined over the set of terms of a clause c of a function. A tuple of terms $<t_i, t_j> \in Depend_c$ if t_j depends on t_i (directly or indirectly) for some invocation of clause c. Thus $Depend_c$ is a closure. Figure 12 presents the unclosed version (for clarify) of relation $Depend_{\mathsf{f}}$ of the sole clause of function f of Figure 6 as a cyclic directed graph.

A separation relation is derived from the data dependence relation, to isolate pairs of terms which have to be allocated to different partitions because their data dependence can only be resolved at run-time. Pairs of terms which have to be separated in this way are indicated by a relation $Separate_c$. Specifically, $<t_1, t_2> \in Separate_c$ if $<t_1, t_2> \in Depend_c$ and $<t_2, t_1> \in Depend_c$. In function f of the sample program, closure of dependencies create edges among nodes {1, 2, 3, 4, 7, 8}. Thus for instance $<3, 4> \in Separate_{\mathsf{f}}$. Figure 13 illustrates $Separate_{\mathsf{f}}$.

King and Soper [17] define a similar separation relation $Granular_c$ based on the granularity of terms, so that two sufficiently coarse-grained terms can be placed in different

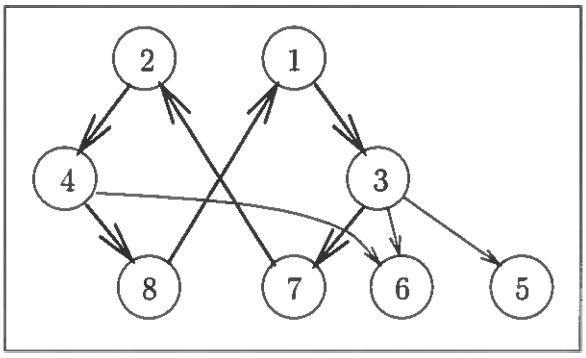
Figure 12: Dependence Relation for f (Not Yet Closed)

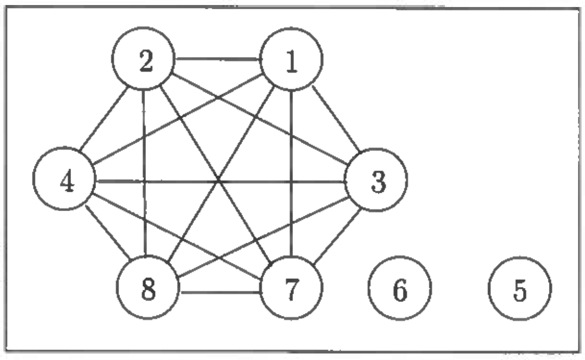
Figure 13: Separation Relation for f

$$\begin{array}{llllll}
\mathsf{f}(A) \rightarrow \{ & H & = & \mathsf{Select}(L') + 1 & \S1 & T_1 \\
 & J & = & \mathsf{Select}(I') + 2 & \S2 & T_2 \\
 & I\ M & = & \mathsf{g}(A, H) & \S3 & T_3 \\
 & L & = & \mathsf{h}(A, J) & \S4 & T_4 \\
 & B & = & I + M & \S5 & \\
 & G & = & M + L & \S6 & \\
 & \_ & = & \mathsf{Store}(I, I') & \S7 & T_5 \\
 & \_ & = & \mathsf{Store}(L, L') & \S8 & T_6 \\
 & & & \mathit{in}\ B\ G\} & &
\end{array}$$

Figure 14: Partitions of f After Coloring Analysis

partitions. Their original analysis determined this relation by abstract interpretation over a coarse domain, and proved somewhat inaccurate. However, their overall framework, discussed next, is quite intriguing.

Partitioning a clause of a function corresponds to computing a k-coloring with respect to $Tot_c = Separate_c \cup Granular_c$, each color being interpreted as a separate partition. This leads to partitioning the set of terms of the clause into

$$Partition_c = \{Partition_{c,1}, \cdots, Partition_{c,k}\}$$

in which $t_i \in Partition_{c,m}$ and $t_j \in Partition_{c,n}$ for $m \neq n$ if $<t_i, t_j> \in Tot_c$. A minimum of six colors (partitions) are needed to color Figure 13.

To turn a partition $Partition_c$ into threads, a total order is assigned to each $Partition_{c,m}$, chosen not to contradict any data dependency in $Depend_c$. This is done by topologically sorting each partition. This step is common to all the techniques discussed. King and Soper [19] describe a more general approach wherein a given partition can be split into subpartitions, creating more threads. They warn that if the subpartitions are split such that cycles are introduced in the data dependencies among the subpartitions, deadlock can occur. We add that if subpartitioning is desired, subpartitions produced by a preorder traversal of the dependency graph are exactly those which do not contain such cycles. Furthermore, simple analysis should be able to identify subpartitions of this form which are both "safe" by King's definition and contain maximal concurrency. Thus unless there are compelling reasons to disobey the preorder, we suggest following it and thereby avoiding the trial-and-error use of King's safety check.

Considering an example, function f of the sample program, application of the k-coloring with respect to Tot_{f} results in six partitions, only the fourth of which has multiple terms, as shown in Figure 14.[9] The two clauses of function g result in two separate partitions because their Tot_c relations are empty. Clearly coloring does not produce an optimal solution for f because a single cut through the dependency graph (Figure 12) can produce two DAGs (see Figure 16). While DD analysis produces 12 partitions (Figure 10) for function f of the sample program, coloring analysis produces only 6 partitions. However, this difference is due to the different types of threads produced by the analyses, as further discussed in Section 4.5.

4.4 AB (Above-Below) Analysis

As was discussed in Section 4.2, DD partitioning technique does not handle recursion satisfactorily. *Strictness analysis* [32] deals with recursion using fixed-point iteration. The AB partitioning technique proposed by Coorg [9] attempts to alleviate this drawback of DD analysis by incorporating the recursion handling property of strictness analysis.

4.4.1 Paths

The AB technique uses paths as a way of representing dependence-demand information. Intuitively, a path of a term gives the set of inputs to a term which are required to produce an output of that term. For example, in the term $Z = X + Y$, the path of Z is $\{X, Y\}$. In the case of terms producing multiple outputs, a path is computed for each value returned. In a function call, the outputs might be produced from different combinations of the inputs. We use the special path $\perp_i$ to denote the case where a function call might infinite loop in the i^{th} clause of the program (where the clauses are numbered in some order). In such cases, a set of possible paths would be computed, only one of which would be valid during any run-time call.

Figure 15 shows the set of paths of each term of the sample CIL program. Note that the variables in paths resulting from function-call terms are the actuals of the call, rather than the formals: thus in §4 the variables A and J appear rather than K and N. Call §3 illustrates that terms with multiple output variables require multiple paths (although in this case the paths happen to be the same). Call §3 also illustrates an interesting case of path computation. From the second clause of g, we deduce that A is in the path set of §3. The first clause of g, however, may not terminate due to the recursive call. In such a case, we simply say that the path set should contain $\perp_2$, rather than $\{\perp_2, A, H\}$. In other words, any path corresponding to a computation which "might compute $\perp$" is itself $\perp$. This

[9] No granularity relations were assumed in this example.

$$\begin{array}{rlllll}
\mathsf{f}(A) \rightarrow \{ & H & = & \mathsf{Select}(L')+1 & \{\{L'\}\} & \S1 \\
 & J & = & \mathsf{Select}(I')+2 & \{\{I'\}\} & \S2 \\
 & I\ M & = & \mathsf{g}(A,H) & \{\perp_2,\{A\}\}\ \{\perp_2,\{A\}\} & \S3 \\
 & L & = & \mathsf{h}(A,J) & \{\{A\},\{A,J\}\} & \S4 \\
 & B & = & I+M & \{\{I,M\}\} & \S5 \\
 & G & = & M+L & \{\{M,L\}\} & \S6 \\
 & \_ & = & \mathsf{Store}(I,I') & \{\{I\}\} & \S7 \\
 & \_ & = & \mathsf{Store}(L,L') & \{\{L\}\} & \S8 \\
 & & & in\ B\ G\} & &
\end{array}$$

$$\begin{array}{rllll}
\mathsf{g}(C,D) \rightarrow \{ & E & = & C-D & \{\{C,D\}\} \\
 & F & = & C+D & \{\{C,D\}\} \\
 & R\ S & = & \mathsf{g}(E,F) & \{\perp_2,\{E\}\}\ \{\perp_2,\{E\}\} \\
 & P & = & R+1 & \{\{R\}\} \\
 & Q & = & R+S & \{\{R,S\}\} \\
 & & & in\ P\ Q\}\ C>0. &
\end{array}$$

$$\begin{array}{rllll}
\mathsf{g}(C,\_) \rightarrow \{ & P & = & C-2 & \{\{C\}\} \\
 & Q & = & C+3 & \{\{C\}\} \\
 & & & in\ P\ Q\}\ C\leq 0. &
\end{array}$$

$$\begin{array}{rllll}
\mathsf{h}(K,\_) \rightarrow \{ & T & = & K-4 & \{\{K\}\} \\
 & & & in\ T\}\ K>0. &
\end{array}$$

$$\begin{array}{rllll}
\mathsf{h}(K,N) \rightarrow \{ & T & = & K-N & \{\{K,N\}\} \\
 & & & in\ T\}\ K\leq 0. &
\end{array}$$

Figure 15: Sample Program with Paths

approximation is useful because it avoids nontermination of path analysis in recursive cases. For an algorithm to construct paths from a program, see Coorg [9].

4.4.2 Preprocessing: Handling Cyclic Dependencies

The presence of cyclic dependencies in the program makes the task of propagating dependence information difficult. Instead of handling cyclic dependencies in the step where dependence information is propagated, AB analysis performs an initial step, dividing the function into *acyclic blocks*: artificial clauses whose dependencies are well-ordered.

In the preprocessing stage, the terms of a block are formed into a block-graph, which reflects the possible dependencies that can arise during execution. Nodes represent the terms of the function block, and edges represent the dependencies between them. Edges which represent dependencies through function calls are specially marked, so that subpartitioning can be performed during analysis, by breaking threads across function calls. Figure 16 shows the block-graph of the function f. Node 9 corresponds to the input variable A, while

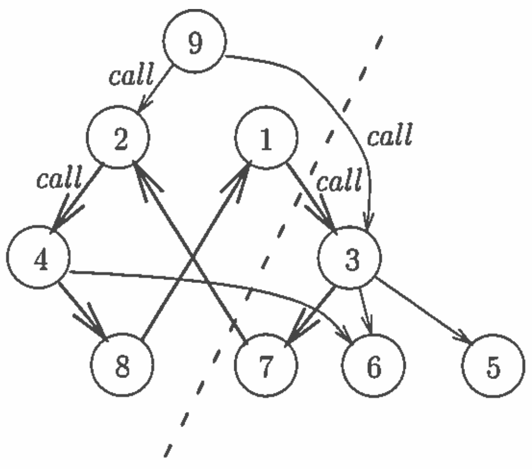

Figure 16: Dependency Graph of the Function f

$$\begin{array}{llcllr}
\mathsf{f}_2(A,H,L) \rightarrow \{ & I\ M & = & \mathsf{g}(A,H) & \{\perp_2,\{A\}\}\ \{\perp_2,\{A\}\} & \S 3 \\
& B & = & I+M & \{\{I,M\}\} & \S 5 \\
& G & = & M+L & \{\{M,L\}\} & \S 6 \\
& \_ & = & \mathsf{Store}(I,I') & \{\{I\}\} & \S 7 \\
& & & \mathit{in}\ B\ G\ I'\} & &
\end{array}$$

Figure 17: Example Acyclic Block With Paths

the other nodes correspond to the labeled terms shown in Figure 15.

Some technique of converting a cyclic graph to a DAG is used, e.g., standard depth-first search [12]. For the function f, one partition of its graph contains the sets of nodes $\{1,2,4,8,9\}$ and $\{3,5,6,7\}$. This step, in which the call graph is transformed into two DAGs, matches the graph-coloring step of coloring analysis. The DAGs produced may be different than those produced by coloring; however, it is guaranteed that liberal threads will be produced by both techniques. Nevertheless, for AB analysis, our goal is to produce conservative threads, and therefore we need to do partitioning analysis over each DAG, as explained in the next section. The analysis we will describe is different than DD analysis and in practice seems to produce larger threads than DD analysis.

4.4.3 AB Partitioning Algorithm for Acyclic Blocks

The AB partitioning algorithm is applied to each acyclic block created by the preprocessing step. As noted in in the previous section, one acyclic block resulting from the preprocessing stage of AB analysis consists of terms {3,5,6,7} of function f. We display this block in CIL clause format in Figure 17 and use if for illustration purposes throughout. The AB partitioning algorithm groups together terms under one of two conditions:

Term	Term Outputs	Term Inputs	Path Tuples
§3	$\{I, M\}$	$\{A, H\}$	$\{ \langle I, \{\perp_2, \{A\}\}\rangle, \langle M, \{\perp_2, \{A\}\}\rangle \}$
§5	$\{B\}$	$\{I, M\}$	$\{ \langle B, \{\{I, M\}\}\rangle \}$
§6	$\{G\}$	$\{M, L\}$	$\{ \langle G, \{\{M, L\}\}\rangle \}$
§7	$\{I'\}$	$\{I\}$	$\{ \langle I', \{\{I\}\}\rangle \}$

Table 1: Abstraction of Acyclic Block

A: All the terms must completely execute before some particular set of outputs can be produced.

B: Some particular set of inputs must be present in order to begin execution of any of the terms.

To achieve these conditions, the variables of the block are first divided into three categories: *input variables* corresponding to inputs to the block, *local variables* which are visible only within the block, and *output variables* corresponding to results of the block. Next, each term of the block is replaced by an abstraction consisting of a set of *term output* variables, a set of *term input* variables, and a set of *path tuples*, one tuple for each term output variable. Each *path tuple* has as its first element a term output variable, and as its second element the path computed for this output variable. Our sample function $\mathfrak{f}_2$ has input variables A, H and L, output variables B, G, and I', and local variables I and M. The abstract form of $\mathfrak{f}_2$ is illustrated in Table 1.

For each variable in the block, both A and B sets are iteratively computed using a standard dataflow technique.

The A **set** of an (block) output variable is the *label* corresponding to the variable itself (a complete definition of label is given below). The A set of a local or input variable V contains the union of the A sets of all output variables of terms whose paths contain V, together with labels introduced due to non-termination.

The B **set** of an (block) input variable is the A set of that variable. The B set of a local or output variable V contains the A set of V, and, if the path set associated with V does not contain any $\perp$ paths, the intersection of the B sets of all the variables in V's path.

A label is either a variable name, or a variable name subscripted by a sequence of clause numbers. The A and B sets are sets of labels. In the calculation of A and B sets of the example, the labels I'_2, G_2, and B_2 arise. The subscripts of a label indicate that the production of the variable may depend on the termination of the program clauses corresponding to those subscripts. When combining two labels with the same name, e.g., G and G_2, the "extra" subscript is removed yielding G. In other words, $\{G\} \cup \{G_2\} = \{G\}$. In more generality, if one subscript is a prefix of another, e.g., G_2, and $G_{2,3,4}$ only the prefix is retained, yielding G_2. Furthermore, $\{G_4\} \cup \{G_3\} = \{G_3, G_4\}$.

Var	A	B
A	$\{I', B, G\}$	$\{I', B, G\}$
H	$\{I'_2, B_2, G_2\}$	$\{I'_2, B_2, G_2\}$
L	$\{G\}$	$\{G\}$
I	$\{I', B\}$	$\{I', B\}$
M	$\{B, G\}$	$\{B, G\}$
I'	$\{I'\}$	$\{I', B\}$
B	$\{B\}$	$\{B\}$
G	$\{G\}$	$\{G\}$

Table 2: A and B Sets for the Example

Intuitively, the A set is very similar to the demand set of DD analysis. The main difference is that it contains labels rather than raw variables, so there is a bit more information content. The B set serves the same function as the dependence set, but it is computed indirectly from the A set rather than directly from the graph.

As an example of A sets, consider Table 2. (The reader is invited to trace through the derivation of the B sets in a similar manner.) Output variables G, B and I' have A sets $\{G\}$, $\{B\}$ and $\{I'\}$ respectively by the first case of the A set definition. Input and local variables are more complex. Consider M for example. Viewing Table 1, M is contained in the term paths of path tuples belonging to terms §5 and §6. The corresponding term outputs of these path tuples are $\{B\}$ and $\{G\}$ respectively. Thus we compute the union of the A sets of B and G, which is $\{B\} \cup \{G\} = \{B, G\}$. Notice I and L sets are computed in a similar manner. However, variables A and H require more heavy machinery. Because these variables are consumed by a term that may not terminate (§3 of Table 1 has $\perp$ in its path tuples), we need to include outputs I and M in their A set calculation. Thus in this example, the A sets of variables A and H are the same modulo label subscripts. H derives its label subscript from the $\perp_2$ subscript, whereas A loses this subscript during the label union operation.

Subpartitioning is used to guarantee that partitions themselves do not introduce a cycle in the block, and that partitions do not contain any function call. Partitioning could introduce a cycle within a thread in a block B given the following situation. Suppose there are two terms T_j executing after T_i within the thread. It is possible that T_i depends on T_j indirectly through another thread or threads. Necessarily these threads will not be in block B, but can arise nonetheless. Such scenarios are identified and the partitions are divided, using the marked edges mentioned earlier. Also, partitions are broken at every call to a function.

The AB partitioning algorithm proceeds iteratively, as follows:

1. The A sets corresponding to the current partitioning are computed.

2. Partitions that share the same A sets are combined.

3. Subpartitioning is performed.

4. The B sets corresponding to the now-current partitioning are computed (using A from step 1).

5. Partitions that share the same B sets are combined.

6. Subpartitioning is reperformed.

7. If the now-current partition is different from the initial partition (step 1), then go to step 1.

The A and B sets calculated for our example during the first iteration of the AB analysis algorithm are given in Table 2. The A set was calculated first, starting at the bottom of the table, with the block output variables G, B, and I', and proceeding upward. From the A sets, it was determined that the consumption of the block input L (not shown) could be placed in the same partition with §6, so this merge was performed. Subpartitioning was then performed, but had no effect. Next, the B set was calculated, starting at the top of the table, with block input variables A, H, and L. From the B set it was determined that §3 and §7 could be merged, since their B sets $\{I', G\}$ were identical. Subpartitioning was again performed, and split §3 and §7 because of the function call to g in §3. Thus, after one iteration, the partitions produced were identical with the initial partitions, except for the combination of §7 and §6. Another iteration of the algorithm (not shown) produced no change in this partitioning, and so analysis terminated.

Thus, the partitions shown in Figure 11 were produced by this analysis, *i.e.,* the same partitions produced by DD analysis with peephole optimization. Although no significant improvement over DD analysis is seen here, AB analysis will partition g with threads T_{15} and T_{16} (Figure 8) merged, effectively doing global propagation optimization across recursive calls that eludes DD analysis (Section 4.2.2).

4.5 Discussion

DD and AB partitioning techniques on one hand, and coloring analysis on the other, represent very different approaches developed for different programming language paradigms. In order to accurately evaluate these approaches it is necessary to understand the issues they address. These issues include the number of threads produced for a given program, prevention of deadlock, simplicity, practicality, and the ability to tune the thread size to the underlying implementation.

The translation of programs from various language families into the CIL provided us with the same starting point for all the techniques discussed, facilitating a fair comparison. All three partitioning techniques share the same basic analysis steps: collecting the data dependencies between the terms of a function, grouping the terms into partitions, and performing topological sort on the partitions to obtain threads.

The different stages in the three partitioning techniques are shown in Figure 18. Unlike DD and AB analyses, coloring analysis attempts to control the granularity of the threads (Section 4.3). DD analysis infers the data dependencies in a program through dataflow analysis. Coloring analysis can use dependence information derived from a variety of sources, such as mode analysis. AB analysis uses path analysis to derive dependence information.

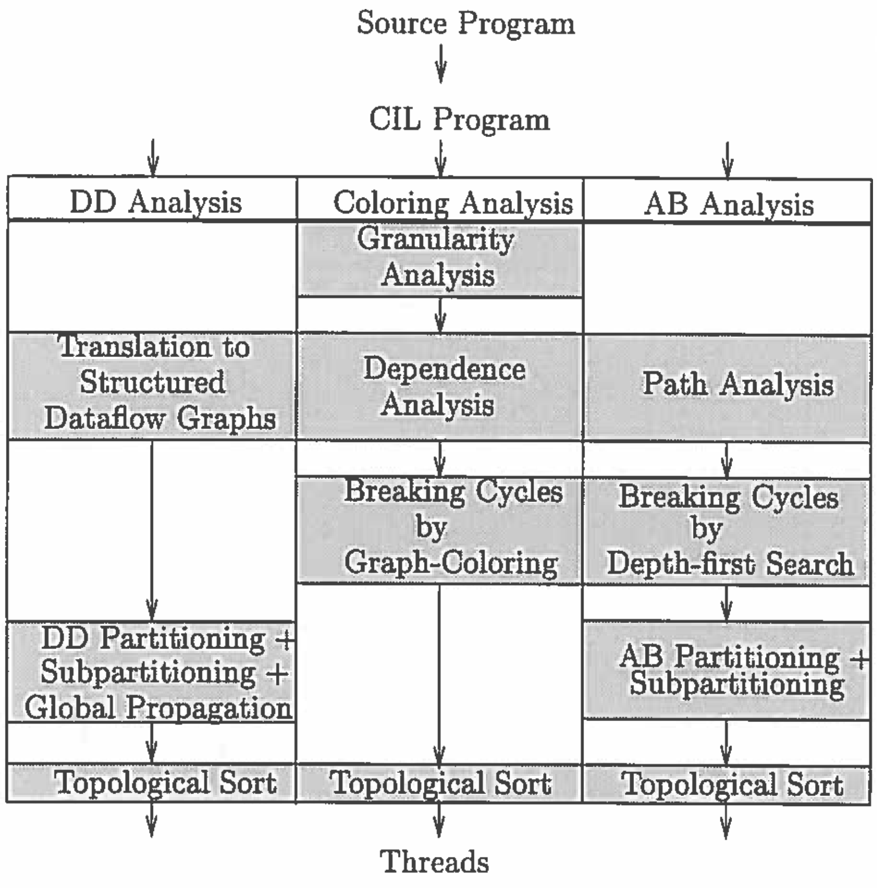
Source Program
CIL Program
DD Analysis
Coloring Analysis
AB Analysis
Granularity Analysis
Translation to Structured Dataflow Graphs
Dependence Analysis
Path Analysis
Breaking Cycles by Graph-Coloring
Breaking Cycles by Depth-first Search
DD Partitioning + Subpartitioning + Global Propagation
AB Partitioning + Subpartitioning
Topological Sort
Topological Sort
Topological Sort
Threads

Figure 18: A Comparison of the Different Stages of DD, Coloring, and AB Analyses

DD analysis breaks threads at all points which *could* be involved in a cyclic dependency. Coloring and AB analyses handle cyclic dependencies explicitly. Once the cycles have been broken, coloring analysis proceeds to produce liberal threads. However, DD and AB analyses, which produce conservative threads, attempt to improve the thread size by propagating dependence information across function call boundaries. Once partitions are produced, threads are obtained in all cases by topologically sorting the partitions.

Different definitions of "thread" used in the analysis lead to a large difference in the number of threads produced for the same program. In coloring analysis, a liberal thread can include more than one function, and can have an arbitrary number of synchronizing and long latency accesses (which may lead to suspensions). In DD and AB analyses, a thread is a collection of instructions which does not contain procedure calls or synchronizing or long latency accesses — synchronization occurs only at the beginning of the thread. Thus, in DD and AB analyses, the requirement that split-phase accesses be kept in separate threads results in extra threads, and function calls split threads as well.

Another important reason for differences in the number of threads is that in DD and AB analyses, a term will not be placed in a thread unless there is a dependency between that term and some other term in that thread. Coloring analysis removes this constraint, potentially leading to threads containing terms with no interdependencies. It is unclear which of these two strategies produces optimal-size threads for practical programs: further study is required.

DD analysis is based on a worst-case scenario. Inclusion of more than one function call in a thread may cause deadlock, if the order of execution of the functions is determinable only dynamically. To prevent introduction of deadlock, DD analysis breaks threads at all function call boundaries. Similarly, inclusion of synchronizing accesses in threads may cause deadlock, if each of two threads blocks waiting for bindings which can only be provided by the other. In order to prevent any such possibility, DD analysis splits threads across all synchronizing accesses. In real programs, not all functions require their scheduling order to be determined at run time, not every synchronizing access results in deadlock, and not every conditional produces cyclic dependencies. Instead of splitting threads only when failing to split them might compromise the termination characteristics of the program, DD analysis applies the same strategy in both safe and unsafe situations. This keeps the analysis technique simple, but results in small thread size.

Moreover, in DD analysis there is no attempt to control the size of the thread. This is based on the realization that, given the strict definition of a conservative thread, there is practically no danger of over-sequentialization. In addition, the specialized hardware typically used for dataflow execution typically has low thread management overheads. However, to efficiently utilize a conventional multiprocessor, thread size may need to be controlled and load distribution may need to be performed.

AB analysis attempts to improve DD analysis while still retaining its simplicity. The propagation of dependence information across recursive calls is improved by use of path analysis, which identifies the presence of strict arguments in recursive calls using fixpoint iteration, and uses this information to improve partitioning. To keep the partitioning technique simple, DD analysis breaks threads at all function call boundaries and synchronizing accesses before applying the partitioning algorithm. AB analysis initially breaks threads to remove cyclic dependencies. Later, it groups threads but ensures that function call boundaries and synchronizing accesses are not grouped during subpartitioning. This separation facilitates better propagation of dependence information. AB analysis, like DD analysis, does not include mechanisms for controlling the granularity of threads.

Coloring analysis alone produces liberal threads. The inclusion of multiple functions and long latency accesses in liberal threads results in increased thread size. This can result in inefficiency, for example thread state has to be stored and context switching has to be performed on every suspension. If suspension is frequent, this overhead places a burden on the underlying implementation. Coloring analysis provides no control over the number of suspension points in a thread.

To summarize our results: DD analysis is a straightforward and very conservative method of producing threads. The resulting thread size is small, especially given the global nature of the analysis. AB analysis offers improvements over DD analysis in the handling of cyclic dependencies and recursion, while keeping its basic control structure. In contrast, coloring analysis obtains larger threads and has an integrated granularity control mechanism, but the liberal threads it produces may have high overheads.

5 Conclusions and Future Work

Non-strict functional and concurrent logic languages seem to be strongly similar, although this has not been widely recognized by researchers in these communities. Thus, solutions to problems in one domain may well transfer to the other. In particular, the static analysis problems encountered during compilation of these languages seem to be quite general: comparable solutions to these problems have been adopted independently in the two language communities.

We thus suggest that translation of programs from both language families into a common intermediate language has a number of attractive features. In particular, we note that 1) a clear description of CIL semantics will permit easy definition of the semantics of traditional languages in terms of the CIL; 2) optimizations performed on the CIL form will provide compilation improvements for both language families, and 3) comparison between static analysis algorithms for CIL programs is much easier than comparisons between algorithms for static algorithms of programs written in a variety of different languages.

We propose a particular CIL, Kernel$_2$ (Figure 4), as a candidate. We note that its semantics are easy to define via term rewriting, and that it seems to contain the important features necessary for the straightforward translation of non-strict functional and concurrent logic languages. While this language seems useful for illustrative purposes, one important area for future work is in obtaining a final CIL design which satisfies all of our criteria, yet is simple to understand, analyze, and implement.

By way of evaluation of our technique, and because it is important in its own right, we consider a particular static analysis example, that of partitioning or threading concurrent programs. A good partitioning strategy has to address several sometimes conflicting goals [15]. The thread length has to be maximized in order to decrease thread switches, decrease explicit synchronization, and increase pipeline utilization and locality. This however should not be at the expense of exploitable parallelism, as the objective of partitioning is to create threads containing little or no easily exploitable parallelism. It is also necessary to identify portions of a program whose execution order is dynamically determined, so that they are not statically scheduled in a single thread. The thread size should be flexible: the compiler should be able to tune the size to the underlying hardware for efficiency.

We discuss three important thread-building algorithms. Our comparison was made a great deal easier by the use of the CIL. We note some of the relative advantages and disadvantages of these techniques, and discuss their domains of applicability. The comparison emphasizes that different thread definitions, motivated by different needs, lead to different (though comparable) static analysis performance under various metrics.

Future work in the development of partitioning algorithms will have to address the issue of obtaining optimal-sized threads based on a model that accounts for local *vs.* non-local execution and communication for a given architecture. The resulting threads can then be efficiently executed on architectures designed to execute threaded programs as well as on standard multiprocessors.

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award, with matching funds from Sequent Computer Systems Inc. M. Sami and B. Massey were supported by a grant from the Institute of New Generation Computer Technology (ICOT).

References

[1] Z. M. Ariola. *An Algebraic Approach to the Compilation and Operational Semantics of Functional Languages with I-Structures.* PhD thesis, Harvard University, June 1992.

[2] Z. M. Ariola and Arvind. P-TAC: A Parallel Intermediate Language. In *Conference on Functional Programming Languages and Computer Architecture*, pages 230–242. London, ACM Press, September 1989.

[3] Z. M. Ariola and Arvind. A Syntactic Approach to Program Transformation. In *ACM SIGPLAN Symposium on Parallel Evaluation and Semantics Based Program Manipulation*, pages 116–129, New Haven, June 1991. Yale University, ACM Press.

[4] Z. M. Ariola and Arvind. Graph Rewriting Systems For Efficient Compilation. In *Term Graph Rewriting: Theory and Practice*, pages 77–90. John Wiley and Sons, 1993.

[5] Arvind, L. Augusston, J. Hicks, R. S. Nikhil, S. Peyton-Jones, J. Stoy, and W. Williams. pH: a Parallel Haskell. Technical report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, September 1993. Unpublished.

[6] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. In *Workshop on Graph Reduction*, number 279 in Lecture Notes in Computer Science, pages 336–369, Santa Fe, September/October 1986. Springer-Verlag.

[7] P. S. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-Strict, Functional Language with State. In *Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, pages 538–568. Springer-Verlag, 1991.

[8] K. L. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[9] S. Coorg. Partitioning Non-strict Functional Languages, 1994. MIT Masters Thesis. Unpublished.

[10] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[11] J. E. Hoch, D. M. Davenport, V. G. Grafe, and K. M. Steele. Compile-Time Partitioning of a Non-Strict Language into Sequential Threads. In *Symposium on Parallel and Distributed Processing*, pages 180–189. Dallas, IEEE Computer Society Press, December 1991.

[12] E. Horowitz and S. Sahni. *Fundamentals of Data Structures.* Computer Science Press, Woodland Hills, CA, 1976.

[13] P. Hudak and B. Goldberg. Serial Combinators: Optimal Grains for Parallelism. In *Conference on Functional Programming Languages and Computer Architecture*, pages 382–399, Nancy, France, September 1985. Springer-Verlag.

[14] P. Hudak, S. Peyton-Jones, and P. Wadler. Report on Programming Language Haskell: A Non-Strict, Purely Functional Language, Version 1.2. *ACM SIGPLAN Notices*, 27(5):1–164, May 1992.

[15] R. A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *International Symposium on Computer Architecture*, pages 131–140. Honolulu, IEEE Computer Society Press, May 1988.

[16] R. A. Iannucci. *Parallel Machine Languages.* Kluwer Academic Publishers, Boston, 1990.

[17] A. King and P. Soper. Heuristics, Thresholding and a New Technique for Controlling the Granularity of Concurrent Logic Programs. Technical Report CSTR 92–08, Department of Electronics and Computer Science, University of Southampton, 1992.

[18] A. King and P. Soper. Schedule Analysis: A Full Theory, A Pilot Implementation, And A Preliminary Assessment. Technical Report CSTR 92–06, Department of Electronics and Computer Science, University of Southampton, February 1992.

[19] A. King and P. Soper. Schedule Analysis of Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 478–492. Washington D.C., MIT Press, November 1992.

[20] G. Lindstrom. Functional Programming and the Logic Variable. In *SIGPLAN Symposium on Principles of Programming Languages*, pages 266–280. New Orleans, ACM Press, January 1985.

[21] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo 284-a, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 1990.

[22] R. S. Nikhil. A Multithreaded Implementation of Id Using P-RISC Graphs. In *Languages and Compilers for Parallel Computing.* Portland, Springer-Verlag, August 1993.

[23] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-Controlled Multithreading for Lenient Parallel Languages. In *Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, pages 50–72. Cambridge MA., Springer-Verlag, 1991.

[24] E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[25] R. Sundararajan. *Data Flow and Control Flow Analysis of Logic Programs.* PhD thesis, Department of Computer Science, University of Oregon, 1994. Also available as Technical Report CIS-TR-94-08.

[26] Z. Symogyi. A System of Precise Modes for Logic Programs. In *International Conference on Logic Programming*, pages 769–787. University of Melbourne, MIT Press, May 1987.

[27] E. Tick. The Deevolution of Concurrent Logic Programming Languages. *Journal of Logic Programming*, 1995. In Press. (Also available as University of Oregon Technical Report CIS-TR-94-07).

[28] E. Tick, B. C. Massey, F. Rakoczi, and P. Tulayathun. Concurrent Logic Programs *a la Mode.* In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems*, pages 239–244. Kluwer Academic Publishers, 1994.

[29] K. R. Traub. Compilation as Partitioning: A New Approach to Compiling Non-Strict Functional Languages. In *Conference on Functional Programming Languages and Computer Architecture*, pages 75–88, London, September 1989. ACM Press.

[30] K. R. Traub, D. E. Culler, and K. E. Schauser. Global Analysis for Partitioning Non-Strict Programs into Sequential Threads. In *Conference on Lisp and Functional Programming*, pages 324–334, San Francisco, 1992. ACM Press.

[31] K. Ueda and M. Morita. Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, May 1994.

[32] P. Wadler. Strictness Analysis on Non-Flat Domains (by Abstract Interpretation Over Finite Domains). In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 181–198. Ellis Horwood Ltd, Chichester, 1987.