Restricted And-Parallelism and Side Effects

Doug DeGroot
Computer Science Center
Texas Instruments
Dallas, Texas

Introduction

Many novel approaches are being investigated today for executing logic programs in parallel. Although several approaches are being pursued, all involve variations of the basic mechanisms of and-parallelism or or-parallelism [Conery]. Although Prolog is the most successful and most-widespread logic programming language, many approaches to parallel logic programming have either abandoned the semantics of Prolog or have developed new language extensions or even new languages. Examples of these include Concurrent Prolog [Shapiro], Parlog [Gregory], GHC [Ueda], and Epilog [Wise]. Conery's model is a notable exception [Conery2]. In his And/Or Process model, a set of run-time tests and algorithms are executed in order to derive a run-time ordering of goals within a clause and produce an execution graph. Although this approach does retain the original Prolog syntax and semantics, it does so at considerable expense. A simpler and cheaper model was presented in [DeGroot]. This simpler model, called the Restricted And-Parallelism (RAP) model, also retains the full syntax and semantics of Prolog, but with significantly less run-time support required. A large portion of the work is performed at compile-time.

The present paper describes how this model can support side-effect computations and yet retain and-parallelism. It is shown how the execution graph expressions of the RAP model are able to express the necessary synchronizations for ensuring the correct execution sequence of goals with side-effects. With this model, maximal and-parallelism is still retained between two goals with side-effects, at least to the extent provided by the restricted and-parallelism model itself.

Overview of the RAP Model

The Restricted And-Parallelism (RAP) execution model of Prolog provides an efficient means of executing traditional Prolog programs in parallel [DeGroot]. The three main components of this model are 1) the typing algorithm, 2) the independence algorithm, and 3) the execution graph expressions. The **typing algorithm** monitors all terms during run-time in an attempt to maintain an accurate indication of the term's type. A term may be a variable (the dereferenced value is unbound), in which case it is assigned the type V; it may be a ground term, in which case it is assigned the type G; or it may be a non-ground, non-variable term, in which case it is assigned the type N. During execution, terms of type V (variables) may be bound to other terms of type V, G, or N; when they become bound, they assume the type of their dereferenced value. Terms of type N may become ground and therefore of type G. Terms of type G cannot change types (except through backtracking).

Several possible typing algorithms are possible. A run-time checking algorithm was described in [DeGroot]. Another typing algorithm based on a static data-dependency analysis is possible which incurs even less run-time overhead but which may occasionally have poorer performance [Chang,DeGroot2]. Other typing algorithms are possible. The RAP model is defined independently of any particular typing algorithm.

The **independence algorithm** is used to determine when two or more terms are independent, that is, when neither shares an unbound variable with the other. If two subgoals are called with arguments which are totally independent from each other, then the two subgoals may execute in parallel; otherwise, they must execute sequentially in order to prevent binding-conflicts. This constraint is sufficient but not necessary to ensure correct parallel execution.

The **execution graph expressions** are used to express the potential parallel execution sequence of the subgoals in a clause. Six expression types were originally defined, although others are possible:

1. G - a simple goal (or subgoal)
2. (SEQ E1 . . . En) - execute expressions E1 through En sequentially
3. (PAR E1 . . . En) - execute expressions E1 through En in parallel
4. (GPAR(V1...Vk) E1. . . En) - if all the variables V1 through Vk are ground (have type G), then execute expressions E1 through En in parallel; otherwise, execute E1 through En sequentially
5. (IPAR(V1...Vk) E1 . . . En) - if all the variables V1 through Vk are mutually independent, then execute expressions E1 through En in parallel; otherwise, execute E1 through En sequentially
6. (IF B E1 E2) - if expression B evaluates to true, execute expression E1; otherwise, execute expression E2

As examples of the types of execution graph expressions generated for typical Prolog clauses, consider

CH2472-9/87/0000/0080$01.00 © 1987 IEEE

the following examples. The clause:

```
    f(X) :- g(X), h(X), k(X).
```

may be compiled into the execution graph expression:

```
    f(X) :- (GPAR(X) g(X)
                    (GPAR(X) h(X) k(X)) ).
```

The clause:

```
    f(X,Y) :- p(X), q(Y), s(X,Y), t(Y).
```

may be compiled into the execution graph expression:

```
    f(X,Y) :- (GPAR(X,Y)
                    (IPAR(X,Y) p(X) q(Y))
                    (GPAR(Y) s(X,Y) t(Y)) )
```

or even into:

```
    f(X,Y) :- (GPAR(X) p(X)
                    (GPAR(Y) q(Y)
                          (GPAR(Y) s(X,Y) t(Y)))).
```

Even though every Prolog clause is compiled into a single execution graph expression, and this is done at compile-time, the actual execution graph may vary considerably depending on the results of any IPAR or GPAR tests embedded within it. This is a very significant advantage - a single compile-time expression can represent a number of actual run-time execution graphs. For example, the second execution graph expression above can actually result in any of the run-time execution graphs shown in Figure 1. It should be noted that the run-time tests are very simple and efficient [DeGroot]. They can easily be implemented in hardware or firmware. The expressions themselves can easily be compiled into an extended WAM model [Hermenegildo].

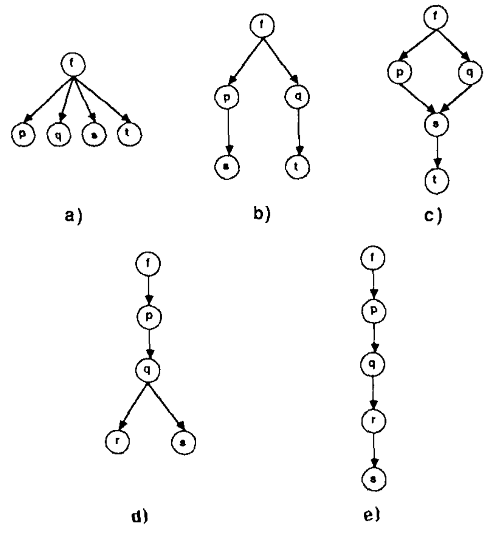

Figure 1. Examples of Execution Graphs.

Side-Effects

Unlike logic, Prolog has the curious aspect of occasionally having side-effects as a result of and during execution of a Prolog program. Goals, subgoals, and predicates that produce side-effects include those such as **assert** and **retract**, **read** and **write**, **cut**, and some other commonly used side-effect producing procedures. In addition, in the RAP model, goals which can produce possible machine-checks or system faults are also considered to be side-effect goals. Examples include **divide**, which might attempt to divide by zero and cause a program trap, and those that require a certain threshold in terms of the number of instantiated arguments, again, such as **divide**, which is required to have its first two arguments instantiated to non-variable values, and arithmetic comparison predicates, such as **gt**, **lt**, and so on. Other examples include **var**, **integer**, **plus**, **is**, and the like. All of these are classified as side-effect goals within the RAP model.

The particular operational semantics of Prolog involve a simple, sequential execution model in which subgoals within a clause are tried in order, from left to right. In addition, multiple clauses within a single procedure are tried one at a time, from top to bottom (at least, within the domain of the selected, indexed clauses [Warren]). These two orderings, when coupled with the observed side-effects, produce a given program's observable behavior. To execute a Prolog program in parallel, it is necessary to retain the order of observable side-effects. As long as this order is retained, the goals themselves may execute in any order. Thus it is not really the left-to-right and top-to-bottom execution orderings of all goals and clauses that give Prolog its unique semantics, it is really the induced ordering of the side-effects produced. If, for example, a clause contains only subgoals which are side-effect free, then the order of execution of these subgoals is irrelevant, and they may potentially execute in parallel. (Performance may differ, but the produced results will be identical.)

The semantics of Prolog then do not really require a sequential logic programming execution model, they only appear to with respect to the sequence of observable side-effects. The side-effect goals must retain their left-to-right and top-to-bottom, induced ordering, but all other goals are, to a certain extent (as explained below), order-independent. In fact, the goals in a side-effect free Prolog program can be executed in any order whatsoever. It is only when side-effect goals are introduced that order becomes important. (Order independence should not be confused with the ability to execute in parallel.)

Side-Effect Goals - Some Terminology

For ease of discussion throughout the remainder of this paper, an informal terminology is introduced. First, as previously mentioned there is a certain set of predefined, evaluable predicates in Prolog which when executed result in (or may result in) certain side-effects to the normal execution. This set includes such evaluable predicates as assert and retract, read and write, cut, and so on. These predicates are referred to below as side-effect built-ins (seb). A clause that contains a side-effect built-in is called a **side-effect-clause** (sec), and a procedure that contains at least one side-effect-clause is called a **side-effect procedure** (sep). A subgoal in a clause that calls a side-effect procedure is also considered a side-effect goal, although it is more appropriately called a "**potential side-effect goal.**" Any clause containing such a potential side-effect goal is called a **potential side-effect clause**, or simply, a **side-effect clause**. (The difference between and significance of whether a goal or clause has a side-effect or potentially has a side-effect is made clear below.) A goal or subgoal that is neither a side-effect goal nor a potentially side-effect goal is called a **pure goal** or **pure subgoal.** A clause that contains only pure subgoals is called a **pure clause**, and a procedure containing only pure clauses is a **pure procedure.** Clearly, a potential side-effect procedure may contain both side-effect clauses and pure clauses. A potential side-effect goal may unify with the heads of both side-effect clauses and pure clauses, but a pure goal may call only pure procedures and thus may unify with only the heads of pure clauses.

A compiler can easily make a recursive traversal of the program source code and determine the type of every goal, clause, and procedure with respect to side effects. The required algorithm is similar to Mellish's automatic mode detection [Mellish] and to the static data-dependency analysis traversal algorithm of [Chang] (see also [O'Keefe]). Once the compiler has ascertained the type of each goal, clause, and procedure, the required synchronization code can be added to the RAP execution graph expressions.

Sequencing Side-Effect Goals

A successful execution of a Prolog program given a particular query produces an and-execution tree in which inner nodes represent complex subgoals (clauses), and leaves represent final subgoals - either simple Prolog facts or built-in Prolog predicates. Whereas an internal node represents a clause, the successors of an internal node represent the subgoals of the clause. For example, Figure 2.a shows a simple Prolog program and query. Figure 2.b shows one possible and-execution tree. It should be remembered that a given program and query can result in more than one successful and-execution tree.

The leaves of a tree, when considered as a list, in order, from left to right, represent all final subgoals that had to eventually be proven in order to prove the initial query. In Figure 2.b, this list is (a,b,c,d,e,f).

In general, consider the list of leaves in an execution tree that results from proving a query. If all the goals represented by these leaves are pure, then they may possibly be executed simultaneously with full and-parallelism, or perhaps at least in any order using partial and-parallelism. In any event, it is clear that the order of execution and the degree of parallelism is irrelevant with respect to the program's observable behavior since the program has no observable behavior except in the top-level reply to the query. Whether the program is executed in order or out of order, sequentially or with restricted and-parallelism, the result is the same.

Consider now that some of the leaves represent side-effect goals, such as write predicates. Clearly, these write predicates must execute in their original, left-to-right, top-to-bottom order as in the execution tree. If not, the resulting stream of output might not appear in the same order as when the program is executed sequentially. So the write predicates must still be executed in order. Since the remainder of the goals are pure goals, it might appear that they could execute in any order. This is not so, as can be seen by considering the following example clause:

p(X) :- test(X), write(X).

Here, the first subgoal must execute successfully before the write goal can execute. If it fails, the write goal must not execute, for the write goal will exert an irrevocable side-effect upon the program's observable execution behavior. It can be seen then by extension that all pure subgoals preceding a side-effect goal must execute successfully before the side-effect goal can execute (at least those that may potentially fail).

```
p :- g, h.
p :- k.
g :- a, b, j.
g :- d, e, b.
h :- f.
j :- c, m, e.
m :- d.
a.
b.
c.
d.
e.
f.

?- p.
```

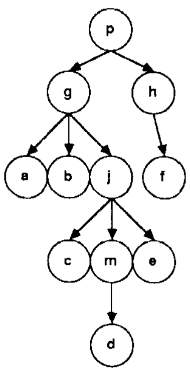

Figure 2. Execution Trees from a Query and Program

What about pure goals following a side-effect goal? Can they execute in any order? Consider the clause:

p(X) :- test(X), write(X), m(X).

Assume m/1 is pure. Then if X is ground, m(X) can execute in any order whatsoever, even before either of the first two goals. Because m(X) has no side-effects, it cannot affect the execution in any way whatsoever. Thus even if test(X) fails, m can still execute without changing the set of producible answers and observable behaviors.

But consider the following clause:

p :- m, assert(n), n.

Assume subgoals **m** and **n** are pure; clearly **assert** is not. Because **m** may be checking some set of preconditions for the assert subgoal, **m** must execute successfully before the **assert** subgoal can execute. But notice that the **assert** subgoal can affect the computation of the following pure subgoal **n**. In this example then, the pure subgoal following the side-effect subgoal may execute in order only after the preceding side-effect subgoal has successfully completed. Although we will see below how we can relax this constraint, for now we adopt the following rule:

Sequencing Rule:
All pure subgoals preceding a side-effect subgoal must complete successfully before the side-effect subgoal can execute. Further, all pure subgoals following a side-effect subgoal must not begin execution until the preceding side-effect subgoal has successfully completed.

Figure 3 illustrates this rule. Note that the leaves of the execution tree can be divided into alternating segments: pure, then a side-effect subgoal, then pure, then a side-effect subgoal, and so on. These segments must execute in order, but within a pure segment, the subgoals can execute in any order, and possibly in parallel. Because and-parallelism is used within these segments, care must be taken to ensure that no two subgoals execute in parallel if they share a common, unbound variable; but this is easily accomplished with the normal RAP execution graph expressions.

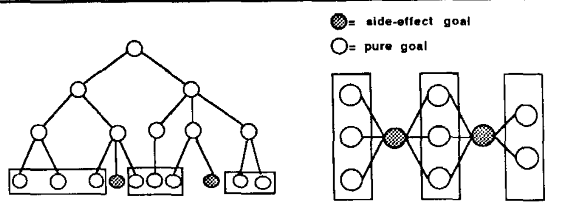

Figure 3. Alternating Pure and Side-Effect Segments

Compiling Side-Effect Expressions

Given a clause to compile, if the clause is pure and thus contains only pure subgoals, these subgoals may execute in parallel, in any order, within the limits imposed by the normal execution graph expressions. In other words, the subgoals can execute in parallel except when they are not independent. The graph expressions specify these conditions [DeGroot]. But suppose one of the subgoals is a side-effect built-in. In particular, consider the clause:

s2 :- a, s1, b.

where subgoals a and b are pure and s1 is a side-effect built-in. (For simplicity, assume also that these subgoals have no arguments and thus are independent.) Clearly, these three subgoals must execute sequentially (actually, as discussed above, we consider below cases where b can execute before s1, or even before a). Because these subgoals must execute sequentially, it is tempting to compile this clause into the graph-expression:

s2 :- (SEQ a s1 b).

And indeed, this is correct to a point. But note that because s1 is a side-effect built-in, the s2 clause becomes a side-effect clause, and thus s2 becomes a side-effect procedure.

Now consider the clause:

s3 :- c, s2, d.

where c and d are again pure but s2, from above, is a potential side-effect goal. Clearly then, the s3 clause is also a side-effect clause and s3 becomes a side-effect procedure. If we compile this clause in the same manner, yielding:

s3 :- (SEQ c s2 d).,

and if we then invoke the goal:

?- s3.

all subgoals in the two clauses will execute sequentially and not in parallel.

Note that the resulting execution tree is that shown in Figure 4. The left-to-right order of the leaves of the tree defines the normal execution sequence that is induced by a sequential Prolog. Note also that there are two pure subgoals preceding the single side-effect, leaf subgoal, and there are two following it. The two preceding it should be able to execute in parallel as should the two that follow it, as they are clearly independent. The compiled clauses above do not allow this, but they are certainly correct and do provide a proper sequencing of side-effects. But they unnecessarily limit the amount of parallelism. To correct this situation, the graph expressions need to be augmented with special synchronizing mechanisms. These mechanisms and their use are described below.[1]

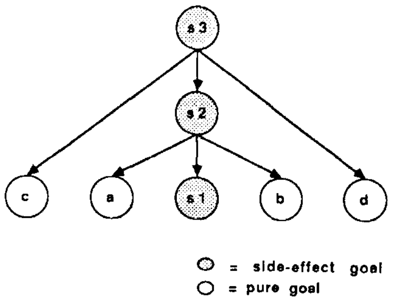

Figure 4. An Execution Tree

Synchronization Blocks

Only one memory data structure is required for implementing the synchronization between the disjoint, parallel executing parts. These structures are called simply enough synchronization blocks, or synch-blocks. They are two-word memory blocks; the first word is used to maintain a count of expressions involved in the synchronization, while the second word is used for signalling the completion of a side-effect predicate.

Five operations are defined for manipulating these synchronization blocks: create, inc, dec, signal, and wait. Each of these is defined below.

First, **create** simply creates a synch-block in free memory. Two types of creation are allowed, one for creating input synch-blocks and one for creating output synch-blocks. The created synch-blocks are identical in both cases, but the fields are initialized to different values depending on which type of creation is invoked.

create(sb,input) - creates a synch-block for the left part of the search tree preceding a side-effect predicate;
sb.ecnt := 0 {indicates no unfinished, preceding, pure expressions}
sb.signal := yes {indicates no unfinished, preceding, side-effect built-in}

create(sb,output) - creates a synch-block for the right part of the search tree following a side-effect predicate;
sb.ecnt := 1 {indicates one unfinished, preceding expression}
sb.signal := no {indicates that the preceding side-effect built-in has not completed}

The expression-count (**ecnt**) field of a synch-block can be either incremented or decremented atomically. The statement **inc(sb)** is shorthand for

sb.ecnt := sb.ecnt + 1;

[1]Actually, if not for soft side-effects, a single word synch-block could be used.

where the addition is performed atomicly. Similarly, the statement **dec(sb)** is shorthand for the statement

sb.ecnt := sb.ecnt - 1;

A signal field indicates whether the associated side-effect built-in has yet completed. A value of "no" indicates that the predicate has not yet completed, while a value of "yes" indicates that it has completed. A signal field may be changed to "yes" with the statement **signal(sb=yes)**.

A **wait(sb.ecnt=0)** statement indicates that the expression-count field of the designated synch-block is to be checked for a value of zero. If the value is zero, the processor continues execution of the current expression. If the value is non-zero, the expression evaluation is suspended and the expression is placed in a suspended expression list. The manner in which suspended expressions are reawakened is discussed in a following section.

Similarly, a **wait(sb.signal=yes)** tests the signal field of a synch-block for a value of "yes". If such a value is found, execution continues. Otherwise, execution is suspended, and the expression is placed in the suspended expression list.

Distributing Synchronization Blocks

As discussed before, the ordered list of leaves of an and-execution tree represents the list of final subgoals that were executed to solve the query. If this list contains only pure subgoals, then the subgoals in the list can potentially execute in any order as long as they are independent. The RAP graph expressions can be used to compile the list into execution graph expressions that will provide for parallel execution when subgoals are independent.

Consider now a list of subgoals which are all pure except for one subgoal in the middle of the list which has a side-effect. As described above, the list must now be broken into three segments: 1) the pure subgoals preceding the side-effect goal, 2) the side-effect goal itself, and 3) the pure subgoals following the side-effect goal. These three segments must execute in order. When one segment finishes, it must somehow "signal" the following segment that it has completed. Each segment must wait for the preceding segment to signal it. Figure 5 illustrates the use of synch-blocks between segments to provide a correct sequencing of subgoals.

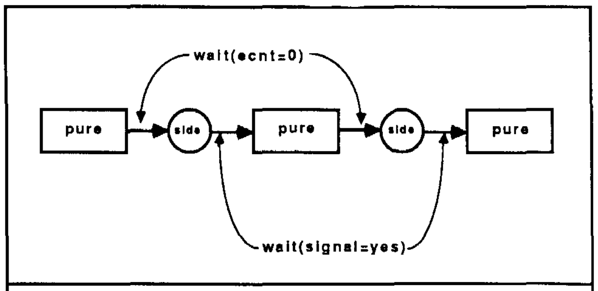

Figure 5. Synchronization Between Segments

Consider Figure 3 again. Notice that the pure subgoals in the first segment must all successfully complete before the "signal" can be sent to the following side-effect subgoal. Each pure subgoal must have some way of signalling the following side-effect goal that it has completed. If we use the above-defined synch-blocks to accomplish this, then each subgoal must have access to the shared synch-block. To to this, the RAP execution graph expressions must be extended to carry along the required synch-blocks and pass them down the and-execution tree in order to reach every goal. If not done properly, this can add significant additional overhead. In the technique discussed below, it is shown how this overhead is minimized.

First, consider the additional RAP expressions required. These expressions represent code macros that get expanded in-line before actual code-generation. In these expressions, **sep** stands for "side-effect procedure" and represents that the named procedure is a side-effect procedure; **seb** stands for "side-effect built-in".

1. (pure(SI) e)
   where e is a pure expression, gets expanded into the sequential code sequence:
   ```
   wait(SI.signal = yes)
   e
   dec(SI)
   ```
2. (seb(SI,SO) s)
   where s is a side-effect built-in or sequence of built-ins, gets expanded into the sequential code sequence:
   ```
   wait(SI.ecnt = 0)
   s
   signal(SO.signal=yes)
   dec(SO)
   ```
   If s is a sequence of side-effect built-ins, they execute sequentially.
3. (seg(SI,SO) p<args>)
   where p is a potential side-effect goal but is not a side-effect built-in, gets expanded into:
   ```
   call p(<args>,SI,SO)
   ```
4. f(<args>,SI,SO) :- <compiled graph expression for side-effect clause body>
   If f is a side-effect procedure and this is a side-effect clause, then the clause head is extended to accept the two additional synchronization block parameters.
5. f(<args>,SI,SO) :- (pure(SI) e)
   where f is a side-effect procedure but this clause is a pure-clause within f.
6. All pure procedures are compiled normally, without any induced overhead.

These additional expressions, along with the statements for manipulating synch-blocks, are sufficient to ensure the correct sequencing of pure code segments and side-effect goals.

Note that an expression receives one or two synch-blocks. One is generally an input synch-block, denoted as SI, and one is an output synch-block, denoted as SO. An input synch-block represents the preceding side-effect goal and the number of preceding pure expressions that follow the preceding side-effect goal. The synch-block's two components - ecnt and signal - represent these two pieces of information. Since the synch-block must be distributed to the preceding side-effect goal, to each preceding pure expression, and to the consuming side-effect goal, the ecnt field is used to count the number of times the synch-block has been distributed. Each time the synch-block is distributed, the ecnt field is incremented. When a pure expression or the preceding side-effect goal successfully completes, it decrements the ecnt field. When the ecnt field reaches zero, all preceding pure expressions and the preceding side-effect goal have completed. At this point, the next side-effect goal can begin execution. When a side-effect goal completes, it sets the signal field of its output synch-block to "yes", indicating that it has completed. At this point, the following pure expressions can begin execution.

Example 1

Consider the query:
?- a, s, b.
where a and b are pure, and s is a side-effect built-in. This query can be compiled as:

```
(SEQ  create(SI,input)
      create(SO,output)
      inc(SI)
      inc(SO)
      (PAR  (pure(SI)    a)
            (seb(SI,SO)  s)
            (pure(SO)    b)))
```

After macro expansion, this query becomes:

```
(SEQ  create(SI,input)
            SI.ecnt := 0;
            SI.signal := yes;
      create(SO,output)
            SO.ecnt := 1;
            SO.signal := no;
      inc(SI.ecnt)
      inc(SO.ecnt)
      (PAR  (pure(SI)  a)
                  wait(SI.signal = yes)
                  a
                  dec(SI.ecnt)
            (seb(SI,SO)  s)
                  wait(SI.ecnt = 0)
                  s
                  signal(SO.signal = yes)
                  dec(SO.ecnt)
            (pure(SO)  b)
                  wait(SO.signal = yes)
                  b
                  dec(SO.ecnt)))
```

When this expression is executed, two synch-blocks are created. The SI signal field is initialized to "yes" since there is no unfinished, preceding side-effect goal. The SI ecnt field is initialized to 0 since there are no unfinished, preceding pure expressions. The SO synch-block represents the side-effect goal and the following pure subgoals that precede the next side-effect subgoal, if any (here, there are none). The SO signal field is initialized to "no" since the side-effect goal **s** has not yet completed; the ecnt field is set to 1, representing the unfinished, preceding side-effect goal. Following creation and initialization, both synch-blocks have their ecnt fields incremented. This is because the query has both a left and a right pure part, and the corresponding synch-blocks will be passed into these expressions. These expressions must be counted before the expressions are entered and the synch-blocks passed in. This explains why the creation and incrementing occurs sequentially before the subgoal expressions begin.

Then the PAR expression is entered. Upon entry, the SI synch-block has SI.ecnt=1 and SI.signal=yes, while the SO synch-block has SO.ecnt=2 and SO.signal=no. Because a PAR expression was entered, all three subexpressions can begin executing in parallel. The second expression, the seb expression, cannot execute since SI.ecnt≠0, and the third expression cannot execute since SO.signal≠yes. Only the first expression can execute since SI.signal=yes. Thus subgoal **a** begins execution. When and if **a** successfully completes, SI.ecnt is decremented to 0. This is the condition the second expression is awaiting. When SI.ecnt is found to be zero, the side-effect subgoal **s** begins execution. When and if it successfully completes, SO.signal is set to yes, and SO.ecnt is decremented to 1. When SO.signal is found to be yes, the third subgoal can begin execution. Upon successful completion, SO.ecnt is decremented to 0. Since all three subexpressions completed successfully, the entire expression has completed, and so control can return to the caller. Both synch-blocks have their signal fields set to yes and their ecnt fields set to zero, indicating completion of all subgoals.

Note that the subgoal **a** can invoke an arbitrarily complex subtree of subgoals, one perhaps containing several thousands of nested subgoals. But these subgoals are all pure and so are compiled into normal execution graph expressions. These expressions do not have to consider any synch-blocks, and no synch-blocks are passed in as parameters. Thus no overhead is introduced into this subtree for the required synchronization. The same is true for the third subgoal, **b**. In fact, the only operations on the synch-blocks are those shown explicitly in the compiled expression. It can be seen then that the root node of a pure subtree in an and-execution tree handles the synch-block manipulation for the entire pure subtree below it. No overhead is introduced into the subtree except at the root node, and this overhead is minimal. This low overhead of passing synch-blocks is a significant advantage of this model.

The preceding example query could actually be compiled simply as a SEQ expression because **s** is a built-in. There is no sense in setting up synch-blocks, although if one wanted to, this is how it should be done. If **s** is not a built-in, then the seb expression is simply substituted with a seg expression, as shown in Example 3 below. Also note that the create and inc statements should be optimized into a single instruction so that SI.ecnt gets set to 1 and SO.ecnt gets set to 2 at creation time. A peephole optimizer can handle this.

Example 2

Suppose again that **a** and **b** are pure but that **s** is a side-effect built-in. Consider the following clause and query:

```
f :- a, s, b.
?- f.
```

Because **s** is a side-effect built-in, the clause is a side-effect clause, and **f** is a side-effect procedure. Thus the clause is compiled as:

```
f(SI,SO) :-
        (SEQ  inc(SI)
              inc(SO)
              (PAR  (pure(SI)     a)
                    (seb(SI,SO)   s)
                    (pure(SO)     b)))
```

which, after macro expansion, becomes the same as shown in Example 1 except for the two synch-block creation statements. Because f is a side-effect procedure, the query,

```
?- f.
```

is compiled into the sequential code:

```
(SEQ  create(SI,input)
      create(SO,output)
      call  f(SI,SO)).
```

First, the two synch-blocks are created and initialized, then the side-effect procedure **f** is called, and then the side-effect clause for **f** is entered. Upon entry, both synch-block ecnt fields are incremented, yielding SI.ecnt=1 and SO.ecnt=2. Additionally, SI.signal=yes and SO.signal=no.

Then the PAR expression is entered. As in example 1, only the first subexpression can begin execution. The second and third subexpressions must suspend since their entry conditions are not met. The three subexpressions will execute sequentially, just as in example 1.

Example 3

Now consider the following query and two clauses:

```
?- f.
f :- a, s1, b.
s1 :- c, s2, d.
```

Let subgoals **a**, **b**, **c**, and d be pure, and let **s2** be a side-effect built-in. Then **s1** is a side-effect procedure, making **f** a side-effect procedure as well. As before, the query is compiled into:

```
(SEQ  create(SI,input)
      create(SO,output)
      call  f(SI,SO)).
```

The two clauses are compiled nearly identically as:

```
f(SI,SO) :-
        (SEQ  inc(SI)
              inc(SO)
              (PAR  (pure(SI)     a)
                    (seg(SI,SO)   s1)
                    (pure(SO)     b))).
s1(SI,SO) :-
        (SEQ  inc(SI)
              inc(SO)
              (PAR  (pure(SI)     c)
                    (seb(SI,SO)   s2)
                    (pure(SO)     d))).
```

This example is possibly more interesting. Once again, when the query is activated, both an input and an output synch-block are created. Then the side-effect procedure **f** is called with these two synch-blocks, and the clause for **f** is entered. Upon entry, SI.ecnt is incremented to 1, counting the preceding pure expression for **a**, and SI.ecnt is incremented to 2, counting the following pure expression for **b** and the nested side-effect built-in. The PAR expression is then entered.

The third subexpression, for **b**, cannot execute since SO.signal≠yes. But the first two subexpressions can begin parallel execution. Suppose they do. The first subexpression invokes a pure execution subtree for subgoal **a** which begins immediate execution and does so without any overhead of the synchronization mechanism. The second subexpression begins execution and immediately calls the side-effect procedure **s1**, passing in the modified input and output synch-blocks, SI and SO. This call results in the clause for s1 being executed.

When this clause begins execution, both synch-blocks are again incremented, thereby counting the left and right pure subexpressions in the clause. Now, SI.ecnt=2 and SO.ecnt=3, and SI.signal=yes and SO.signal=no. Consequently, when the PAR expression is entered, only the first subexpression can begin execution. The second and third subexpressions must suspend. When the first subexpression, for **c**, begins execution, **c** may also invoke an arbitrarily complex execution subtree. Note that this subtree can execute in parallel with the subtree invoked by subgoal **a**.

Before the side-effect subgoal **s2** can execute, both subgoals **a** and **b** must complete. As the execution subtrees are executed, when successful, control returns to the pure subexpressions containing them. These subexpressions then proceed to decrement the SI.ecnt field. Since SI.ecnt=2 at this time, the two decrementings set SI.ecnt=0. Because both subexpressions may be executing in parallel and may possibly attempt to simultaneously decrement SI.ecnt, the decrement operation must be atomic, using semaphores, test-and-set instructions, replace-add instructions, or others. Eventually, however, SI.ecnt=0.

At that point, the second subexpression of the **s1** clause can begin execution. This subexpression, with its macro expansion, is:

```
(seb(SI,SO)  s2)
        wait(SI.ecnt = 0)
        s2
        signal(SO.signal = yes)
        dec(SO)
```

After the wait instruction succeeds, the side-effect built-in **s2** can execute. If successful, SO.signal is set to yes to indicate its completion to all following pure subgoal expressions, and SO.ecnt is decremented to 2.

When SO.signal is set to yes, the two subexpressions

```
        (pure(SO)  b)))
                wait(SO.signal = yes)
                b
                dec(SO.ecnt)
```

in the clause for **f**, and

```
        (pure(SO)  d)))
                wait(SO.signal = yes)
                d
                dec(SO.ecnt)
```

in the clause for **s1**, can both begin execution. This allows the two subgoals **b** and **d** to invoke their pure execution subtrees in parallel, and again without any overhead of the synchronization mechanism. This example shows how, in a sequence of nested clauses, pure subgoals preceding a side-effect goal are all allowed to execute in parallel, then the side-effect built-in executes, and then the following pure subgoals are allowed to execute in parallel. Obviously, this example extends to nested clauses of any depth.

Example 4

As mentioned before, a side-effect procedure can contain both side-effect clauses and pure clauses. When a pure clause is contained within a side-effect procedure, it is compiled as follows:

```
f(SI,SO) :-
        (pure(SI)  e)
                wait(SI.signal = yes)
                e
                dec(SI.ecnt)
```

Example 5

Consider the following example clauses and their compilations. In each clause, let **a**, **b**, and **c** be pure subgoals, let **s1** and **s2** be side-effect built-ins, and let **g1** and **g2** be non-built-in, side-effect subgoals. The clause:

```
        f :- a, s1, s2, b.
```

is compiled as:

```
f(SI,SO) :-
        inc(SI.ecnt)
        inc(SO.ecnt)
        (PAR  (pure(SI)     a)
              (seb(SI,SO)   s1 s2)
              (pure(SO)     b)).
```

Here, the two side-effect built-ins execute sequentially as required.

The clause:

```
        f :- a, g1, b, g2, c.
```

is compiled into:

```
f(SI,SO) :-
        create(SM,output)
        inc(SI.ecnt)
        inc(SM.ecnt)
        inc(SO.ecnt)
        (PAR  (pure(SI)     a)
              (seg(SI,SM)   g1)
              (pure(SM)     b)
              (seg(SM,SO)   g2)
              (pure(SO)     c)).
```

This example shows that some synch-blocks are created within a clause and not just at the query level.

Example 6

Finally, two examples are shown for clauses that do not have pure parts preceding or following a side-effect goal. First, the clause

```
        f :- a, s1.
```

is compiled as:

```
f(SI,SO) :-
        inc(SI)
        (PAR  (pure(SI)     a)
              (seg(SI,SO)   s1)).
```

The clause

```
        f :- s1, b.
```

is compiled as:

```
f(SI,SO) :-
        inc(SO)
        (PAR  (seg(SI,SO)   s1)
              (pure(SO)     b)).
```

Methods of compiling clauses of other types should be obvious.

Suspending Expression Execution

When a wait instruction is executed in an expression, a particular condition is checked. If the condition exists, execution of the expression may continue. If the condition does not exist, the expression execution must suspend. The expression is placed in a suspended expression list, and the processor looks for another expression to execute. These expressions will be found on the system'as active expressions stacks [Hermenegildo]. As long as the processor has active expressions it can execute, it will not be idle, and system utilization will not suffer. If a condition required by a suspended expression becomes true while the processor is executing other expressions, the condition is not noted by the waiting processor, that is, there are no interrupts to signal when a condition had become true. (It is, of course, certainly possible to extend this model to an interrupt or message-based system by having the waiting processor enqueue itself on a waiting queue attached to the synch-block. This does not appear to be necessary.)

When a processor runs out of active expressions to execute, it can then make a pass through the suspended queue, rechecking the required conditions for each suspended expression. As soon as one condition is found to be true, execution of the associated expression can resume. If no condition is found to be true, the processor has run out of expressions to execute, and it can then volunteer to assist other processors in executing their overload of active expressions, if any. Although this model is a polling model, it will not flood the communication system with condition-checking memory accesses except when a processor is idle and no other processor will send it additional work to do. But in such cases, the overall system activity will most likely be low, and the communication network will thus likely be underutilized.

Types of Side-Effect Built-Ins

It is possible to divide the set of side-effect built-ins into two different classes: those that can affect the following computation and those that cannot. Certainly, a **write** subgoal cannot affect the execution of the following pure subgoals. An **assert** subgoal, however, may very well affect their execution. Let the former type be called **soft side-effects** and the latter **hard side-effects**. With these definitions, the seb expression defined above is for hard side-effects. A soft side-effect subgoal **s** can be compiled as:

```
(soft-seb(SI,SO)  s)
        signal(SO.signal=yes)
        wait(SI.ecnt = 0)
        s
        dec(SO)
```

Here, because the signal operation occurs before the side-effect subgoal **s** begins execution, the pure subgoals following **s** can immediately begin execution. The side-effect goals still execute in sequential order, however, as required, and still, no-side-effect goal can execute until all preceding goals have completed.

It is even possible to classify side-effect procedures and clauses as either hard or soft. If a subgoal calls a soft side-effect procedure, then the output synch-block can even be created with signal=yes. In such cases, all pure subgoals following the soft side-effect subgoal can execute in parallel with those pure subgoals preceding the soft side-effect subgoal. For example, consider the two clauses and query below:

```
        f :- a, s1, b.
        s1 :- d, write(foo), e.
        ?- f.
```

Let **a**, **b**, **d**, and **e** be pure subgoals. Clearly, **write(foo)** is a soft side-effect subgoal, and thus **s1** is a soft side-effect procedure. The query would be compiled as:

```
        ? -     create(SI,input)
                        SI.ecnt := 0;
                        SI.signal := yes;
                create-soft(SO,output)
                        SO.ecnt := 1;
                        SO.signal := yes;
                call f(SI,SO).
```

The two clauses would be compiled as:

```
f :- a, s1, b.
f(SI,SO) :-
        (SEQ  inc(SI)
              inc(SO)
              (PAR  (pure(SI)     a)
                    (seg(SI,SO)   s1)
                    (pure(SO)     b)).

s1 :- d, write(foo), e.
s1(SI,SO) :-
        (SEQ  inc(SI)
              inc(SO)
              (PAR  (pure(SI)     d)
                    (seb-soft(SI,SO)  write(foo))
                    (pure(SO)     e)).
```

This will allow all of the pure subgoals **a**, **b**, **d**, and **e** to execute in parallel. However, the **write** subgoal can execute only when both **a** and **b** have successfully completed, as required.

Summary

A method of handling side-effect goals has been introduced into the Restricted And-Parallelism (RAP) model. This method ensures that the normal, observable sequence of side-effects that would occur in a traditional, sequential execution of a Prolog program is retained within the restricted and-parallel execution of the same program.

A provable query results in the definition of one or more and-execution trees. The leaves of the tree represent the list of all required, final subgoals. These subgoals are divisible into alternating segments of pure subgoals and then side-effect subgoals. The segments are executed in order, but within a pure segment, all the pure subgoals are potentially allowed to execute in parallel. Within a pure segment, little overhead is introduced for the maintenance of the required synchronization, thereby providing potentially greater performance.

It should be pointed out that the model is not yet complete. At present, the backtracking semantics and an efficient implementation model are being investigated. An automatic graph expression compiler is also being extended to incorporate the side-effect expressions.

Bibliography

[Chang] Jung-Herng Chang, Alvin Despain, and Doug DeGroot, "AND-Parallelism of Logic Programs Bases on a Static Data-Dependency Analysis," *Procs of the Spring Compcon 85*. IEEE Computer Society Press, 1985, pp. 281-225.

[Conery] John Conery and Dennis Kibler, "Parallel Interpretation of Logic Programs," *Procs. of the Conf. on Functional Programming Languages and Computer Architecture*, ACM, 1981, pp. 163-170.

[Conery2] John Conery, *Parallel Execution of Logic Programs*, Kluwer Academic Publishers, 1987.

[DeGroot] Doug DeGroot, "Restricted And-Parallelism," *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, North Holland, 1984, pp. 471-478.

[DeGroot2] Doug DeGroot and Jung-Herng Chang, "A Comparison of Two And-Parallel Execution Models," *Hardware and Software Components and Architectures for the 5th Generation, AFCET Informatique*, March 1985, Paris, pp. 271-280.

[Gregory] Steve Gregory, "PARLOG: A Parallel Logic Programming Language," Research Report DOC 83/5, Imperial College, London, March 1983.

[Hermenegildo] Manuel Hermenegildo, "A Restricted And-Parallel Execution Model and Abstract Machine for Prolog Programs," MCC Tech. Report Number:PP-104-85, MCC, Austin, Texas, Oct, 1985.

[Mellish] Chris Mellish, "The Automatic Generation of Mode Declarations for Prolog Programs," DAI Research Paper 163, Dept. of Artificial Intelligence, Univ. of Edinburgh, August 1981.

[O'Keefe] Richard O'Keefe, A Prolog Program to Detect Side-Effect Procedures, personal correspondence, May, 1987.

[Shapiro] Ehud Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," ICOT Tech. Report TR-003, ICOT, Tokyo, February, 1983.

[Ueda] Kazunori Ueda, *Guarded Horn Clauses*, Ph.D. Dissertation, University of Tokyo, March 1986.

[Warren] David H.D. Warren, "An Abstract Prolog Instruction Set," Tech. Note 309, SRI International, Oct. 1983.

[Wise] Michael Wise, *Prolog Multiprocessors*, Prentice/Hall International editions, 1986.