M. M. Gooley and B. W. Wah, "Reordering of Prolog Programs with And-Parallelism," Proceedings of the 1988 Workshop on Languages for Automation, IEEE, Aug. 1988, pp. 219-244.

REORDERING OF PROLOG PROGRAMS WITH AND-PARALLELISM

Markian M. Gooley and Benjamin W. Wah

Coordinated Science Laboratory, University of Illinois
1101 West Springfield Avenue, Urbana, Illinois 61801

Abstract- We present a method, **order tracking**, that allows reordering of Prolog clauses without reordering their solutions. Reordering clauses lets us execute a program at lower cost and run more goals in AND-parallel; order tracking ensures that the reordered clauses behave exactly like their originals. We extend our Markov-chain model for estimating probability and cost, adding order tracking and a model for AND-parallel execution. This allows us to estimate when reordering and AND-parallelism will speed execution and when they will slow it.

1. Introduction

1.1. Motivation for our research

Various researchers have worked on the parallel execution of Prolog, proposing models and abstract machines [4,5,8], but they have done little on restructuring programs to improve their parallel behavior. Eventually something analogous to Parafrase [13], the Fortran vectorizer, should be possible for Prolog: a restructurer to let a program make better use of parallel hardware.

Parallel execution cannot make Prolog efficient. Recall that Prolog executes a program by traversing an AND/OR graph [14] depth first. Often it goes through several large subgraphs, each corresponding to a goal, only to fail later. On failure, Prolog backtracks, searching goals again to satisfy its conjunction in another way. We know how to reorder goals and clauses to reduce serial search [7]; we can also do this for parallel execution.

1.2. Defining the problem

We want to reorder the goals of Prolog clauses to minimize or at least reduce the cost of execution. We also want the new order to allow more goals to run in AND-parallel as a **parallel goal** (when this reduces cost). If a clause has multiple solutions, changing the order of goals usually changes the order of solutions; for some applications this does not matter [7], but in general we want to avoid or correct it.

1.3. Assumptions and definitions

We assume that the reader has some knowledge of Prolog, and perhaps of Warren's abstract Prolog machine [17]. We are concerned with AND-parallel execution of independent goals; familiarity with some scheme of AND-parallelism is useful but not necessary.

A predicate with name `name` and number of arguments `arity` is written `name/arity`. A variable not set to an atom or functor is **uninstantiated** or **free**; else it is **instantiated** or bound. A tuple of instantiations corresponding to the arguments of a goal is called a **mode**. If a predicate will not function in certain modes (*i.e.*, they cause a run-time error or an infinite recursion), it **demands** a particular mode. A mode different from one arising in the original program is legal for a predicate if it satisfies any demands of that predicate, and does not change its behavior in certain ways (which we describe later).

1988 Workshop on Languages for Automation, IEEE, University of Maryland, College Park, Maryland, August 1988

This research was supported in part by the National Aeronautics and Space Administration under grants NAG 1-613 and NCC 2-481.

1.4. Examples of improvement

Suppose that we call the clause `a(W,X) :- b(X,Y), c(Y,Z), d(W,Z).`, with both variables uninstantiated. Assume that the predicates `b/2`, `c/2`, and `d/2` instantiate their arguments, and calls to them in any mode are legal. Under AND-parallel models such as RAP [5], goals may run in parallel if they share no uninstantiated variables; therefore the clause in our example cannot, in this mode. Were the second goal first, the other two goals could be a parallel goal, because they share no variables.

Reordering also provide speedups without parallelism. Suppose we have sets of facts for `wife/2`, `mother/2`, and (for single females) `female/1`, and the predicates

```
gmother(GC,GM) :- gparent(GC,GM), female(GM).

gparent(GC,GP) :- parent(P,GP), parent(GC,P).

parent(C,P) :- mother(C,P).
parent(C,P) :- mother(C,M), wife(P,M).
```

The query `:- gmother(X,Y).` finds all grandmother-grandchild pairs; however, it finds a grandparent-grandchild pair first, instantiating the variables `GC` and `GM`, about half the time promptly rejecting it. Changing the first clause to

```
gmother(GC,GM) :- female(GM), gparent(GC,GM).
```

probably reduces the cost: `female/1` takes fewer calls than `gparent/2`. Note also that `female/1` instantiates `GM` and thus the mode of `gparent/2`: we consider fewer possibilities, making the goal less costly. Unless only a tiny fraction of the females in the database are grandmothers, the reordering pays.

1.5. Previous work

Warren [19] uses a simple heuristic to reorder goals of conjunctive queries to a Prolog database. Speedups are up to several orders of magnitude, but the method fails for Prolog programs in general. Li and Wah [14] model Prolog clauses as Markov chains, allowing one to compute approximately the cost and probability of success for a clause from their values for each goal; one need provide them only for ground clauses because they propagate upwards. Goals can be reordered to minimize the cost. We greatly extend this in our paper [7], giving restrictions on reordering; however, we do not show a way to prevent reordered goals from giving reordered results. Instead, we give "levels of equivalence" that different types of reordering preserve; these are still useful when exact equivalence is not essential and we want to avoid overhead.

Many Prolog dialects, execution models, and abstract machines support AND-parallelism. Parallel dialects include Parlog [2] and Concurrent Prolog [16]; with these, the programmer annotates variables to show the system how to avoid binding conflicts. Conery's AND/OR parallel execution [4] requires no annotations, but performs much run-time analysis. DeGroot's Restricted AND-Parallelism [5], extended by Hermenegildo [8,9,10], tests variables at run-time for groundedness and independence: if a putatively parallel goal passes the tests, it runs in parallel; otherwise its parts run serially. Our reordering method uses extensive static analysis that makes some run-time tests unnecessary. Chang *et. al.* [1] use static analysis expressly to find AND-parallel code.

1.6. Organization and goals of this paper

First we consider machines to support AND-parallelism: a simplified abstract machine and, after some necessary background on the Warren Abstract Machine (WAM) [17], Hermenegildo's [8]. Then we show how reordering goals reorders results, present order tracking, a method for solving the problem, and modify both the serial and parallel machines to support the method. Predicates with side-effects restrict both serial reordering [7] and AND-parallelism [6]; we describe how our method can ameliorate this problem. We then summarize other restrictions on reordering.

In our previous paper [7] we extended the Markov-chain model of clause execution to full serial Prolog; here we further extend it to include AND-parallelism and our new method. This should allow us to estimate at compile-time, when parallel execution pays, when reordering pays, and when it is wise to combine them. We discuss future work and give conclusions.

2. Machines for AND-parallelism

2.1. Properties of a machine

A machine has processors able to communicate with each other by some means. Each either runs a serial Prolog interpreter, consists of a Prolog engine such as a Warren machine, or emulates a Prolog engine; each has a copy of the program. Communication has some cost, not necessarily fixed. We can consider the machine either a set of processors or of processes running on them.

AND-parallel execution is straightforward at this level of abstraction. A processor, or unit, receives a goal to solve. It tries to match this to a clause of the corresponding predicate. Suppose that it succeeds and reaches, within the clause, a **parallel goal**: a conjunction of goals that is to run in parallel. For every goal of the conjunction, it sends information to another unit, letting it solve the goal. As each unit finishes its goal, it sends its results back to the "parent unit," the one working on the parent clause. If any constituent goal of the parallel goal fails, the unit running that goal informs the parent unit, which sends a "kill" signal to all units working on parts of the parallel goal; this is propagated to any descendants running parallel subgoals.

We need parameters for the costs of various operations, such as calling a remote goal (c_{send}), returning the result of a remote goal ($c_{receive}$), reporting failure or broadcasting a kill signal (c_{fail}, c_{kill}). Our unit of cost is the (local) predicate call, which we assume takes roughly constant time; therefore we can measure the cost of an operation as the number of calls that can be made during it. Because of our generalized model we assume single (fixed or average) values for these parameters, but for many methods of communication they would be functions.

Even this generalized model for an AND-parallel machine shows us how the cost of communication between processors affects the speed of execution. Expensive communication, clearly, makes running goals on remote processors impractical. As we discuss later, we must estimate the cost of each way of executing a set of goals: will parallel execution pay?

2.2. Specific machines

We now describe the Warren Abstract Machine (WAM) [17] and an AND-parallel extension of it designed by Hermenegildo [8]. We limit the details to those necessary for explaining our reordering techniques and cost model; the descriptions are taken largely from Hermenegildo's paper [8]. Note that our methods need not be limited to these particular machine models.

2.2.1. The Warren abstract machine

The WAM is Warren's extension of the execution model described in his doctoral thesis [18]. It has become the standard model for serial Prolog. Figure 2.1 shows the data areas of the WAM.

The **code** area holds the program, compiled into WAM object code. The **heap** holds data structures and global variables, which are created, altered, and discarded (on backtracking) as necessary. The **stack** contains **environments** and **choice points**. The WAM pushes an environment onto the stack on entering a clause. An environment has value cells for variables local to the clause, and pointers that show where within its parent clause execution will continue when the clause completes. Environments can be discarded when they are no longer needed [20], or can have variable cells removed [17].

The WAM pushes a choice point (CP) onto the stack upon entering a predicate. A CP contains copies of registers, sufficient to restore the state of the machine, and a pointer to the next untried clause of the predicate. If the current clause fails, the WAM checks the last CP (pointed to by register B), restores the saved state, and tries the alternative clause. Restoring the state reclaims the stack and heap space used by the failed clause, but it cannot undo instantiations to variables deeper in the data areas. the trail records instantiations which have to be undone on backtracking. The

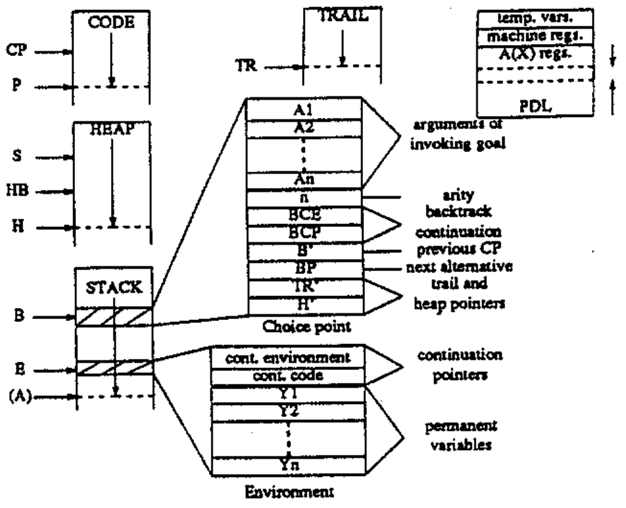

Figure 2.1: Data areas of the Warren abstract machine (WAM).

WAM also has **argument registers** to pass arguments in a predicate call, and a **push-down list** for unification.

2.2.2. The RAP abstract machine

Hermenegildo [8] presents a multiprocessor made of modified WAMs. He describes the additions a WAM needs to support AND-parallelism, and then proposes extra hardware and an altered execution model. A processor must be able to assign parallel work to other processors

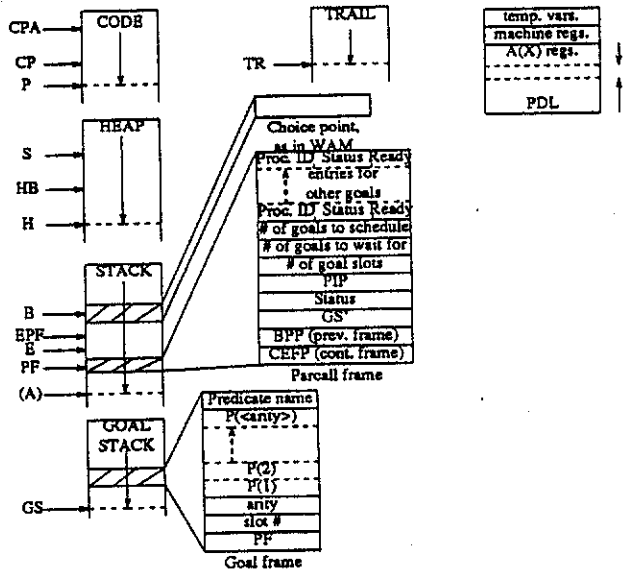

Figure 2.2: Data areas for one processor of the RAP machine.

and keep track of the state of this work, yet it should retain the useful features of the original WAM. Hermenegildo shows that his design does these things and supports AND-parallelism. Figure 2.2 shows the data areas for one processor of the design.

Records for parallel calls, **parcall frames**, now appear on the stack. Each processor also has a new **goal stack** onto which it pushes **goal frames** for goals which are ready to be executed in parallel. Each goal frame contains the information needed for remote execution, *viz.*: a pointer to the predicate being called, a copy of the argument registers for the call, the predicate's arity, a pointer to its "parent" parcall frame (EFP), and its position within that parcall frame. When execution of a clause reaches a parallel call, a goal frame is pushed onto the goal stack for each of the goals, and processors (including the one executing the clause) can "steal" these frames and start to execute the goals.

A processor creates a parcall frame on the stack for each parallel call. Within this frame, each goal has a three-field slot: the number of the processor that "stole" it for execution, a bit telling whether it still has alternatives, and a "ready" bit telling whether the goal's frame will actually be put on the goal stack (for when execution backtracks into the parallel set, the retry involves only goals with alternatives). The frame also holds the number of goals left to schedule, the number of goals executing but not completed, a pointer to the beginning of the first goal of the parallel call (the Put Instruction Pointer, used after backtracking to find parallel goals to push on the goal stack for the retry), a status bit to show if the parallel call has been backtracked into, the top of the goal stack when the parallel call began, a pointer to the previous parcall frame, and the previous parcall frame pointer (CEPF). There is also a new register, PF, which points to the last parcall frame -- the one to try in case of backtracking.

The parcall frame chiefly supports backtracking. The scheme [10], which we also follow though our cost models could support others as easily, works like this. In a clause with a parallel goal, goals can fail in three places: before the parallel goal, within it, and after it. In the first case, we backtrack as usual; in the second, the parallel goal fails and we backtrack to the goal before it; in the third, we backtrack as usual until we re-enter the parallel goal. There we consider only the goals that have alternatives (recall that bit in a parcall frame's slot). We backtrack between them in the usual way; when one succeeds, we lump those on its right into a smaller parallel goal and execute it (this is the use of the "ready" bits). If the goals all fail, we backtrack to the goal before the original parallel goal.

3. Order tracking

Suppose that we want to reorder the goals of a clause to promote AND-parallel execution or inexpensive execution of goals. Assume that no restrictions on reordering [7] (summarized later) interfere. If we reorder the goals of the clause, we usually reorder the solutions to the clause. Often this is unacceptable: we want the transformed program to behave just like its original.

3.1. Reordering goals reorders results

Consider this simple example; we have:

```
a(X,Y) :- b(X,Z), c(Y,Z).

b(m,n).    b(n,o).    b(o,p).

c(r,p).    c(s,o).    c(t,n).
```

The query `:- a(J,K).` gives answers `J=m, K=t; J=n, K=s; J=o, K=r` in that order. If we swap the goals of the clause for `a/2`, the order of the three answers is reversed.

3.2. How to correct this

Suppose that a predicate is non-deterministic. It or its descendents contain disjunctions, as multiple clauses or written explicitly. We can represent the predicate as an AND/OR graph (or tree, if there is no recursion), an AND node for each clause and an OR node for each disjunction. The sons of each AND node are goals; the sons of each OR node are members of disjunctions. Label the arcs from an OR node to its sons with numbers, in the order in which Prolog tries them: *viz.*, the order they are written in the program. OR nodes with one son need no labels.

We can use strings of bits as labels; a disjunction of n alternatives needs $\lceil \log_2 n \rceil$ bits. Figure 3.1 is a labeled graph for this program:

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

b:- l.    b:- m.    b:- n.    c:- o.
d:- p.    d:- q.    e:- r.    f:- s.
f:- t.    g:- u.    g:- v.    h:- w.
h:- x.    i:- y.    j:- z.    j:- a1.
j:- b1.   k:- c1.
```

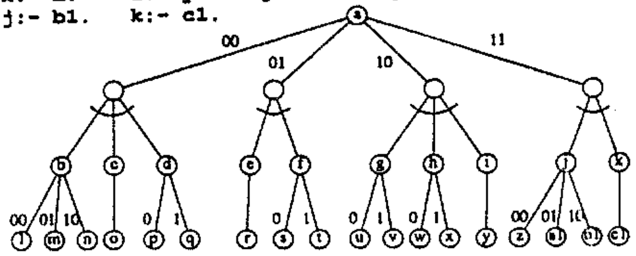

Figure 3.1: A labeled AND/OR graph

Now if we traverse the graph depth-first, as Prolog would execute its program, appending labels to a bit-string as we try alternatives and removing them when we backtrack, each solution yields a unique string. The strings are in order: a solution's string is always larger than its predecessor's, if we give the leading bit of two strings (whether ones or zeroes) the same significance and treat strings as binary numbers. Thus the strings tell us the original order of the answers to a predicate.

Now suppose that we include labels in compiled code. We also add to each choice point of the WAM two fields: one for a bit-string, the other for the bit-string's current length. When the WAM calls a predicate for which we want to keep track of order, it includes space at the end of the choice point for the bit string, copies the label of the first alternative, advances the length counter, and proceeds. The next choice point copies the string and length, and appends the next label; on backtracking to a choice point, the WAM replaces (within the string) the label of the current alternative with that of the new one. In this way we get a bit-string for each solution.

We have a reordered conjunction of goals, not necessarily an entire clause, for which we want to find the original order of results. We have a record of the original order of goals. At compile-time, we see which goals are responsible, in the original order, for instantiating variables; we have these generate bit-strings as they run. (Note that it is "safe" but wasteful to generate a bit-string for every goal of the conjunction; therefore we can include any goal that *might* instantiate a variable.) Running the reordered conjunction gives a set of bit-strings for each of its answers. We pad the strings with zeroes, so that all strings produced by the same goal have the same length. We arrange the strings of a solution into the original order of the goals, and concatenate them, forming what we call a **signature** for the solution.

Treating signatures as binary numbers and sorting solutions by increasing order of signatures gives us the same order of solutions as for the original conjunction. Our restrictions on reordering [7] guarantee that the set of solutions it unchanged. Each solution is made up of the solutions to the goals of the conjunction, and the order of the bit-strings produced by a goal is the order in which its AND/OR subtree is searched. A signature therefore represents a trace of execution for the original order of goals, and the higher the signature, the later the solution.

3.3. Details of the implementation

This method, **order tracking**, solves the problem of out-of-order results; however, the modified choice points described earlier are only part of the necessary mechanism. We describe first the additions to the WAM and then those for the RAP machine. (Note that Hermenegildo has independently proposed a similar labeling scheme [9] to prevent problems with memory management on the RAP machine; perhaps some of the overhead could be shared.)

3.3.1. Modifying the WAM

Once we reach a reordered conjunction, we must find all its solutions before we can proceed, as in a **bagof** [3], because in general we cannot know in what order they are found. We store each solution (and its signature), sort them, and return the first. When the WAM backtracks into the reordered conjunction, it gets the next stored solution, and so on until the solutions are used up and backtracking proceeds to an earlier point.

If a reordered conjunction does not take up an entire clause body, we create a new predicate for it, one having a single clause with the reordered goals for a body: *e. g.*, if we swap the two middle goals of `t:- a, b, c, d.`, we rewrite `t` as `t:- a, t1, d.` and `t1:- c, b.` (with appropriate variables). When we call the new predicate we create a **reordering frame** on the stack; this is a choice point with the **BP** (next alternative) pointer replaced by a pointer (**RP'**) to a new area called the **reordering heap**, and a new field to count the number of solutions. The reordering heap holds solutions as long as they are needed, and has an extra machine register (RP) pointing to its top. Figure 3.2 shows the features added to the WAM.

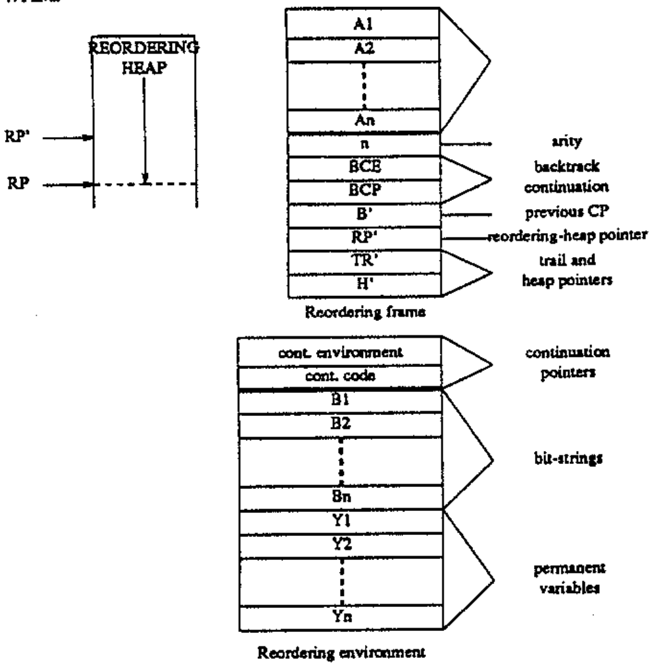

Figure 3.2: Additions to the WAM to support order tracking.

The machine copies its (new) RP register into the RP' field of the reordering frame, and makes a **reordering environment** on the stack for the reordered goals. This is an environment with fields for the bit-strings produced by each goal. When the conjunction produces a solution, the machine creates a signature from the bit-strings, combines it with the solutions to form a **solution frame** that it pushes onto the reordering heap, and increments the solution counter of the reordering frame. We force backtracking and find all solutions, discarding the environment as usual.

Now the RP' pointer of the reordering frame points to the first solution frame on the reordering heap. Extra hardware sorts the solution frames by signature; there are several possible ways to do this. We might create a sorted linked list of solution frames, which would make moving them unnecessary; instead, we sort them such that the first solution is at the top of the reordering heap and the last towards the bottom, which allows us to reclaim some space as solutions are used. When the sorting is done, the reordering frame returns the first solution and decrements its solution counter. On backtracking it fetches the next solution from the reordering heap, decrements its solution counter, and updates RP and RP' as needed. When no solutions remain, the machine backtracks to the previous choice point, as usual.

If a reordered conjunction is a descendant of another, the ancestor expects a label from it as part of a bit-string. We keep the bit-strings and signatures short by assigning a label to each of the descendant's solutions, rather than passing the signatures. We lose no information because these solutions are already in the correct order. This method is also useful if a descendant goal, not reordered, has such a large search tree that the bit string is too short to distinguish solutions.

3.3.2. Modifying the RAP machine

The processors of the RAP machine require the same modifications as the WAM, also, they need a field in the goal frame to hold a signature. Note that the parts of a parallel goal can be called in any order without changing the order of results, because they are independent. None of them will ever be a call to a new predicate of reordered goals, although they may have such calls as descendants. Similarly, a new predicate generated for a conjunction of reordered goals may include a parallel goal; in fact, if reordering has put into a parallel goal a goal that, in the original ordering, instantiated variables, the entire parallel goal must be in the new predicate.

3.4. Order tracking and side-effects

We have shown [7] that goals with side-effects (I/O, modifications to the database such as `asserta` and `retract`) cannot be reordered. For example, if we have the clause `a:- b, c, d.`, and `c` has a side-effect, putting `c` before `b` expresses the side-effect even if `b` fails; putting `d` before `c` prevents its being expressed if `d` fails. Goals with side-effects are *fixed*; No goal of a clause may be moved from one side of a fixed goal to the other. The effect propagates upwards: predicates calling fixed predicates are "responsible" for the side-effects of their descendants and are also fixed.

DeGroot [6] has studied how side-effects affect AND-parallelism: all goals that occur before a side-effect must complete before the side-effect goal can execute, and all goals that occur after one must wait until it is finished before they can execute. He divides side-effects into *hard* and *soft*; the latter (*e. g.*, `write`) cannot affect later goals, so that those later goals can begin execution before the side-effect goal finishes. Compile-time analysis should allow us to show that particular instances of `assert` and `retract` are also soft.

We can "cache" soft side-effects with solution frames on the reordering heap, thereby keeping the order of side-effects correct. This lets us reorder fixed goals, so long as their instantiations are preserved.

4. Other restrictions on goal-reordering

We have seen how certain built-in predicates demand particular modes, and how side-effects can fix goals; we now summarize other important restrictions. For details, see our earlier paper [7].

4.1. The cut

The *cut*, written `!`, is notorious for altering program behavior. Encountered during forward execution, a cut succeeds immediately; during backtracking, it makes its predicate fail without trying any of its later clauses.

In the body of a clause a cut is not mobile, but it does not fix its ancestor predicates. Instead it commits the earlier goals of the clause to their first solutions. Without order-tracking, we cannot reorder those goals or any of their descendents, because reordering would probably change that first solution. Order tracking works poorly with cuts, because it finds all solutions for each reordered conjunction of goals, though only one solution is needed.

4.2. Semi-fixity

Some predicates behave differently in different modes; for example, the built-in `var/1` succeeds only for an uninstantiated argument. We must preserve the modes of such predicates under reordering, or at least the instantiations of variables that cause this to happen, or the reordered program will be wrong. Usually the behavior involves a unification or test that always succeeds or fails in some mode, but due to a cut has no alternative. For example,

```
a(X,Y,b):- !.
a(X,Y,Z):- c(X,Y), d(Y,Z).
```

matches only its first clause if none of its arguments are instantiated, but probably only its second if the last argument is. Any reordering that changes the latter situation to the former is wrong, and that last argument is the "culprit." A way to preserve the mode of a culprit variable is to fix its goal with respect to other goals that might change its instantiation; hence we call such predicates *semi-fixed*. The only legal mode for a semi-fixed predicate is the one in which it was called in the original program. Semi-fixity propagates to ancestors if culprit variables also appear in the clause head.

5. The Markov-chain model

We want the expected cost and probability of success for calling a predicate, given those for the predicates its clauses call (recall that we measure cost as number of predicate calls). We model a Prolog clause as an absorbing Markov chain [7], after Li and Wah [14]. The cost and probability of the clause follow from properties of its chain; combining results for clauses gives them for the predicate. We summarize our earlier work, then present new models for order tracking and parallel goals.

5.1. Overview of earlier work

Consider `k:- a, b, c, d.` We know the probability p_i and cost c_i for each goal i. To find the expected probability and cost of a single solution to the clause, we model it as the Markov chain of Figure 5.1 We give each goal a state, label the arcs with success and failure probabilities, and add absorbing states S and F for success and failure.

Transitions of the chain mimic Prolog execution. We start in the state of b, proceeding to c if b succeeds, or going to F if it fails. In every i we move forward (probability p_i), or backtrack (probability $1-p_i$). Eventually we stop in an absorbing state: success or failure.

If we want the cost of finding all solutions to a conjunction of goals, as in a *bagof*, a *setof*, or a reordered conjunction under order tracking, we add an arc of probability 1 from S back to the last goal. When we find a solution we backtrack, looking for another, until we fail.

The success probability p_{clause} of the clause is that that the process ends in state S. $\sum c_i v_i$, for all goals i, is the expected cost of a solution, c_i being the expected cost of goal i and v_i the mean number of visits by the process to state i. Therefore we calculate p_{clause} and all the v_i: textbook [12] mathematics.

Begin with the transition matrix P. An element p_{ij} is the probability of a transition from state i to state j; S has index 1 and F index 2. The chain has r states, s of which are transient. Consider four submatrices of the $r{\times}r$

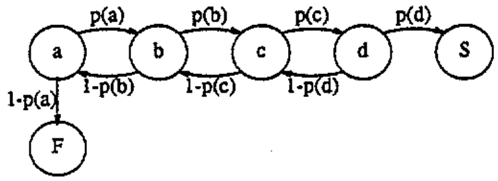

Figure 5.1: A clause body as a Markov chain.

matrix P:

$$P = \begin{bmatrix} I_{r-s} & 0 \\ R & Q \end{bmatrix}$$

I_{r-s}, the identity matrix, for transitions between absorbing states; R, $s{\times}(r-s)$, for transitions into absorbing states; Q, $s{\times}s$, for transitions between transient states. The zero matrix, $(r-s){\times}s$, shows that nothing leaves an absorbing state. Here is P for our example:

$$P_k = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1-p_a & 0 & p_a & 0 & 0 \\ 0 & 0 & 1-p_b & 0 & p_b & 0 \\ 0 & 0 & 0 & 1-p_c & 0 & p_c \\ p_d & 0 & 0 & 0 & 1-p_d & 0 \end{bmatrix}$$

The matrix $N = (I-Q)^{-1}$ provides our answers. Its first row contains the numbers of visits v_i for the transient states, because we start in the first state. The product NR gives the success probability p_{clause}: the first element of the column of state S (column 1); we take the dot product of that column of R and the first row of N. We find N_k numerically; if we have N goals, we invert an $N{\times}N$ matrix, and do $2N$ multiplications and sundry additions for the probability and cost. If we use an analysis program written in Prolog, we might call a C routine to generate and invert the matrix.

For each calling mode we find the least costly reordered version of a predicate. Different modes may benefit from different goal orders, and the cost of each goal varies with its own mode. After finding probability and cost for each clause, we combine the probabilities to find the conditional probabilities for the clauses in order, multiply these by the costs, and get the expected probability and cost for the entire predicate.

Cost and probability of a clause come from those of its goals; in database programs, these come from costs and probabilities of facts. In such programs only facts have non-variable arguments, so that their heads might fail. Evaluating a fact costs one call; its success probability is problematical unless the call is uninstantiated, when it is unity. We can, like Warren [19], find *domains* for each argument; a domain might be the set of constants in that argument of the predicate, or perhaps the set of all constants in the program. We take the probability as $\prod |domain_i|^{-1}$ for every position i that has a constant in both fact and call.

If we allow full Prolog with recursion, it becomes harder to tell the likelihood that a call unifies. if an argument position is non-variable in both the goal and the clause head, the probability is no longer 1. For constants, we can use domains, as with facts. For structures, the predicates are usually recursive, with a special case matching empty structures. The size of a structure affects both the probability and cost of a call; the system or user must estimate structure size before run-time. It might suffice to provide a set of parameters, giving the probability of a match for each type of term, the "typical" size of a data structure, and so forth.

5.2. A model for order tracking

Suppose we apply the Markov model to a clause body and find that some of its goals will run at lower cost if they are reordered. We construct a single-clause predicate from these goals. The clause head contains only variable arguments, so that it always matches a call. We model this clause using the "all-solutions" Markov chain mentioned earlier: the one with the added unit-probability arc from the success node to its predecessor. This tells us the least expensive order of the goals, as well as the expected number of solutions. (The additional arc keeps the success state S from being an absorbing state, so that we know the expected number of visits to it.) Putting this order into the usual chain gives an estimate of the probability of success.

The first solution of the order-tracked predicate is extremely expensive, but later solutions require only the cost of copying data from the reordering heap. Figure 5.2 shows a Markov chain for a clause that calls an order-tracking goal. We represent the goal by two nodes: the first (labeled f) bears the expected cost of finding all solutions (plus any overhead costs), and succeeds with the probability p_f that any solution is found; the second (labeled b), reached only on backtracking, bears the cost of looking up solutions, and the success probability $p_b = \frac{s-1}{s}$, where s is the number of solutions.

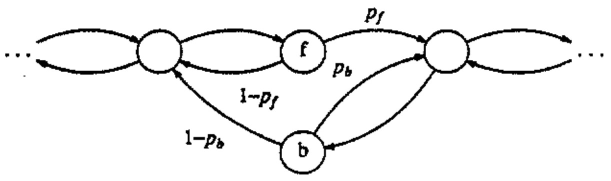

Figure 5.2: A clause that calls an order-tracking goal.

5.3. A model for AND-parallel goals

Suppose that we have n independent goals forming a parallel goal. We know the probability of success and the expected cost of each; we want the probability of success and the expected cost for the entire parallel goal. Because the subgoals are independent, the probability of success is simply the product of those for the subgoals.

For this model, we assume sufficient processors are available to run each subgoal of the parallel goal; therefore, when the parallel goal succeeds, we consider it to cost as much as its most expensive subgoal, because the parallel expression does not complete until all of its goals have completed. We are not counting the total number of calls executed on the processors running the goals, but only how soon the parallel goal will be completed.

The cost of a failing parallel goal is more difficult to find. For a subgoal i with expected cost c_i, we approximate the cost as a random variable U_i, exponentially distributed with mean c_i and therefore having parameter $\lambda = \frac{1}{c_i}$. (Note that the usual Markov chain gives us a single expected cost for completion rather than one for success and another for failure.) Each U_i, then, has density $f_i = \frac{1}{c_i} e^{-\frac{x}{c_i}}$ and distribution $F_i = 1 - e^{-\frac{x}{c_i}}$. Now we want the expected cost of the first failure; this is the first order statistic for the variables U_i [11]. A straightforward but tedious derivation based on a heuristic technique [11] shows that the first order statistic has an exponential density, just like those of the constituent goals, but with parameter $\lambda = \sum_{i=1}^{n} \frac{1}{c_i}$, so that the expected cost of failure for the parallel set is $\frac{1}{\lambda} = \frac{1}{\sum_{i=1}^{n} \frac{1}{c_i}}$.

Note that we can easily include the overheads of parallel execution in this model. Recall that sending a goal to a remote processor causes a delay, as does retrieving a result. Therefore we might take the cost of success as $\max_{i=1,n}(c_i) + c_{send} + c_{receive}$. We add such delays to the cost of each goal before we compute the expected cost of failure. To that cost of failure we also add the cost of sending a failure message to the parent unit, and the cost of the parent unit killing execution of the other goals, giving $\frac{1}{\sum_{i=1}^{n} \frac{1}{c_i}} + c_{fail} + c_{kill}$.

Figure 5.3 shows a chain for a clause that contains a parallel goal. It is so complex because we have to model two unusual features of a parallel goal: the differing costs of success and failure, and the backtracking scheme. We use a "dummy" node (labeled d) of no cost to select success or failure, and two deterministic nodes to bear their costs (s_i and f_i). Recall that when execution backtracks into the parallel goal after failure, backtracking proceeds in the usual reverse order (amongst goals with alternatives) until a goal succeeds. Then all goals to the right are reactivated in parallel. Using static analysis we can eliminate most of the goals that have no alternatives (again, this is not crucial: a few deterministic goals included

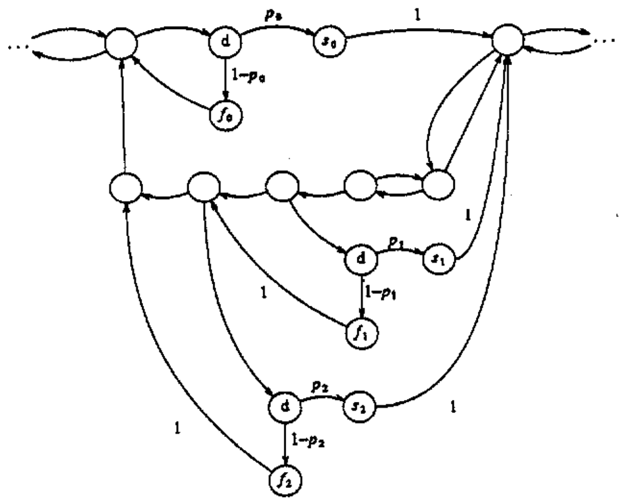

Figure 5.3: A clause containing a parallel goal.

erroneously will not greatly alter the estimates of probability and cost). We have, then, a chain of nodes that represents backtracking between goals of the parallel goal, with additional nodes that represent parallel retries of certain subsets of the parallel goal.

6. Conclusions

We presented a method, order tracking, for reordering clauses of both serial and AND-parallel Prolog without changing the order of solutions. We described serial and parallel Warren machines, and showed what must be added to them to make our method work. After summarizing our Markov-chain reordering method, we further extended it to cover order tracking and AND-parallelism, and showed an example of how reordering can promote AND-parallelism and faster execution.

Increasing Prolog's speed of execution will require a combination of restructuring methods and extensions to the Warren engine. We intend to study both OR- and AND-parallelism to see how they might be promoted by changing program structure. However, serial execution is still inefficient: restructing needs to ensure that each processor of a parallel machine runs efficiently.

We also need to study side-effects under restructuring and parallelism. Our earlier work [7] and that of DeGroot [6] will provide a good starting point, but it has been concerned with what reorderings and AND-parallel executions are forbidden by side-effects; we plan to look at more types of restructuring and parallelism, and try to circumvent the side-effects rather than be restricted by them.

7. References:

[1] J.-H. Chang, A. M. Despain, and D. DeGroot, "AND-Parallelism of Logic Programs Based on a Static Data Dependency Analysis," *Digest of Papers of COMPCON Spring '85*, pp. 218-225, February 1985.

[2] K. L. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 1-49, 1986.

[3] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, 1984.

[4] J. S. Conery, *Parallel Execution of Logic Programs*, Kluwer Academic Publishers, Boston, 1987.

[5] D. DeGroot, "Restricted AND-Parallelism," *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp. 471-478, ICOT, 1984.

[6] D. DeGroot, "Restricted AND-Parallelism and Side Effects," *Proceedings of the 1987 Symposium on Logic Programming* pp. 80-89, IEEE, August 1987.

[7] M. M. Gooley and B. W. Wah, "Efficient Reordering of Prolog Programs," *Fourth International Conference on Data Engineering*, to appear, IEEE, February 1988.

[8] M. V. Hermenegildo, "An Abstract Machine for Restricted AND-parallel Execution of Logic Programs," *Proceedings of the Third International Conference on Logic Programming*, Springer-Verlag, Berlin, 1986.

[9] M. V. Hermenegildo, "Relating Goal Scheduling, Precedence, and Memory Management in AND-parallel Execution of Logic Programs," *Proceedings of the Fourth International Conference on Logic Programming*, pp. 556-575, MIT Press, Cambridge, 1987.

[10] M. V. Hermenegildo and R. I. Nasr, "Efficient Management of Backtracking in AND-parallelism," *Proceedings of the Third International Conference on Logic Programming*, pp. 40-54, Springer-Verlag, Berlin, 1986.

[11] P. G. Hoel, S. C. Port, and C. J. Stone, *Introduction to Probability Theory*, Houghton Mifflin, Boston, 1971.

[12] J. G. Kemeny and J. L. Snell, *Finite Markov Chains*, Van Nostrand, New York, 1965.

[13] D. J. Kuck *et. al.*, "The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance," *Proceedings of the International Conference on Parallel Processing*, pp. 129-138, IEEE, August 1984.

[14] G.-J. Li and B. W. Wah, "How Good are Parallel and Ordered Depth-First Searches?" *Proceedings of the International Conference on Parallel Processing*, pp. 992-999, IEEE, August 1986.

[15] C. S. Mellish, "An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter," in: *Logic Programming*, pp. 99-106, Academic Press, New York, 1982.

[16] E. Y. Shapiro and A. Takeuchi, "Object Oriented Programming in Concurrent Prolog," *New Generation Computing*, vol. 1, pp. 25-48, 1983.

[17] D. H. D. Warren, An Abstract Prolog Instruction Set, Technical Note 309, October 1983, SRI International, Menlo Park, CA.

[18] D. H. D. Warren, Applied Logic -- Its Use and Implementation as a Programming Tool, Technical Note 290, June 1983, SRI International, Menlo Park, CA.

[19] D. H. D. Warren, "Efficient Processing of Interactive Relational Database Queries Expressed in Logic," *Seventh Conference on Very Large Data Bases*, IEEE, 1981.

[20] D. H. D. Warren, "Optimizing Tail Recursion in Prolog," in: *Logic Programming and its Applications*, pp. 77-90, Ablex, 1986.