*Technical Report*

UCAM-CL-TR-212
ISSN 1476-2986

Number 212

UNIVERSITY OF CAMBRIDGE

**Computer Laboratory**

# A distributed and-or parallel Prolog network

K.L. Wrench

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

*https://www.cl.cam.ac.uk/*

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

*https://www.cl.cam.ac.uk/techreports/*

ISSN 1476-2986

# A Distributed And-Or Parallel Prolog Network

K.L. Wrench
Computer Laboratory
New Museums Site
University of Cambridge
Cambridge, CB2 3QG

**Abstract**

A model is proposed for the parallel execution of Prolog, exploiting both *dependent and-* and *full or-parallelism*. The model is implemented on a distributed network of loosely-coupled processors and has no need of shared memory nor multiprocessor hardware.

Known as the APPNet, the model makes use of *oracles* to partition the search space dynamically, thereby enabling processing elements to be allocated a unique portion of the computation. No communication takes place between processing elements. In executing problems that do not exhibit any and-parallelism, all solutions found represent final answers to the query. When an and-parallel problem is executed, the solutions generated are only partial solutions. The sets of partial solution are then *joined* to produce consistent final solutions. *Back-unification* is the process whereby partial solutions are unified according to a *template* derived from the program.

Prolog source programs need not be modified by the user. Static analysis is, however, carried out automatically on all programs by a preprocessor before their execution in the APPNet to ensure that clauses are not distributed before it is feasible to do so. Side-effecting constructs are identified and the appropriate restrictions are placed on the parallel execution strategy.

---

[1]This report is a shortened version of the dissertation submitted for the degree of Doctor of Philosophy at the University of Cambridge.

# 1 Introduction

In the past decade, Prolog has developed from a relatively unknown language to one which is widely used for increasingly large applications. Allied to this newfound attention is the possibility of a parallel logic language based on Prolog. The confidence with which Prolog has been accepted by the Japanese Fifth Generation Computer Systems Project [Fuchi 1983] as the implementation language for use on massively parallel architectures has fuelled this interest.

Research using Prolog as the basis for a parallel logic evaluation scheme has developed along two fronts. On one hand, there are those concerned with designing new parallel logic languages to exploit *stream and-parallelism.* In particular, Parlog [Clark and Gregory 1986], Concurrent Prolog [Shapiro 1986] and GHC [Ueda 1986] are committed-choice languages which require the programmer to specify potential sources of parallelism explicitly. Commitment is delayed until the latest possible moment to ensure that a correct choice has been made. Once the evaluation has committed itself to a certain clause the decision cannot be undone, and as such these languages lose much of the *don't-care non-determinism* exploited in Prolog.

The other approach has concentrated on the design of parallel evaluation schemes for Prolog. Instead of using a sequential left-to-right top-down evaluation of the source program, a parallel strategy is used.

An *or-parallel evaluation* strategy takes advantage of or-parallelism in the source by executing alternative clauses for a procedure concurrently. This effectively replaces the backtracking mechanism used in a sequential evaluation strategy by searching for all solutions at the same time. By virtue of the nature of or-parallelism, this parallel search can be accomplished relatively simply without concern for conflicting bindings or overlapping search areas. Independent branches of work may be allocated to individual processing elements, which are chiefly modified Warren abstract machines (WAM's) [Warren 1983], with all the optimizations available in a sequential Prolog system. Several models have been proposed which make use of or-parallel evaluation strategies; these include the models designed by Ali [1987, 1986] and Shapiro [1987], the ORBIT model of Yasuhara and Nitadori [1984], Warren's SRI model [1987(b)], the DelPhi principle [Clocksin and Alshawi 1986], the Kabu-Wake inference model [Kumon *et al.* 1986], the Argonne model [Disz, Lusk and Overbeek 1987] and Aurora [Lusk *et al.* 1988].

The benefits accruing from or-parallel execution of Prolog are greatest in non-deterministic programs where several choicepoints exist, *e.g.* search-like problems. In deterministic programs where the search space is deep and narrow and where at most one solution can be found, no speed-up can be expected.

*Restricted and-parallel evaluation* strategies have also been researched in depth [Chang *et al.* 1985(a); DeGroot 1984, 1987; Hermenegildo 1986]. Using restricted and-parallel evaluation, the

literals in the body of a clause may be executed concurrently, provided that they do not share any variables bound to non-ground terms. The strategies developed in this area are more complex than those which exploit or-parallelism, since a partial ordering of the clauses is required to ensure that all shared variables are bound to ground terms before being allocated for parallel execution. In Prolog a literal may not bind its variables to ground terms, with the result that this ordering needs to be an ongoing process to ensure maximal concurrency.

*And-or parallel evaluation* schemes attempt to combine the benefits offered by each of the previous strategies. The restrictions on the and-parallel evaluation of literals sharing variables, however, still apply. Conery's And-Or Process Model (AOPM) [1987(a)] is probably the father of most parallel evaluation schemes, while the Reduce-Or Process Model (ROPM) [Kalé 1987(a)] provides an and-or parallel evaluation scheme which exploits both *consumer-instance parallelism* as well as the traditional and-or parallelism.

Finally, some researchers have touched on the problems of removing the restrictions and implementing *dependent and-parallelism.* This area is the least developed; partly owing to the number of problems to be overcome, and partly because of the lack of promising results. Most of the models designed to incorporate dependent and-parallelism have been based on *data-flow principles, e.g.* Epilog [Wise 1984, 1986]. Communication within the basic model is fairly excessive, causing even greater communication overhead when dependent and-parallel execution is incorporated. To make this approach viable, a basic model is required that limits communication from the outset.

This is the area of research introduced in this paper. Section 2 describes some of the other models in the field of and-or parallel evaluation strategies for Prolog. Section 3 introduces the DelPhi principle, since it forms the basis for the design of the new model. The next five sections are devoted to a full description of the APPN*et* (an And-Or Parallel Prolog Network) and its implementation. Section 9 gives an evaluation of the performance of the APPNet. The paper concludes with a comparison of the merits of the APPNet with respect to the ROPM, PEPSys and Epilog.

# 2 Other Research in the Parallel Evaluation of Prolog

This section gives an overview of other research in parallel evaluation schemes for Prolog. The survey of or-parallel evaluation schemes is by no means exhaustive. The models have been chosen however, for comparison with the DelPhi proposal.

## 2.1 Or-Parallel Execution Strategies

The or-parallel execution strategy described by Ali [1987] has much in common with the DelPhi model. Given a top level query, all PEs in the system start processing the goal simultaneously.

When a choicepoint is reached, each PE chooses a branch according to the splitting algorithm used. Splitting is done in much the same way as the automatic partitioning in the DelPhi machine. PEs continue executing until a leaf node is reached. After reporting their status to the *Manager Process*, they become idle. If several PEs arrive at the same solution, provision is made by the model to ensure that only one of these is reported.

The major difference between this model and the DelPhi principle is apparent in the method used to reassign jobs to idle PEs. In Ali's model, job reallocation takes place when the number of idle PEs becomes larger than a certain value. When this happens, one of the busy PEs is interrupted and its job is *copied* to all the idle PEs. This entails transferring the entire environment created by the busy PE, including variable bindings, choicepoint frames, the heap and the trail stack. If the interrupted PE has unexpanded choicepoints, the "idle" processors backtrack to the earliest choicepoint with untried branches and from there splitting occurs as before. If no unexplored choicepoints exist, the "idle" PEs continue processing the same subtree as the busy PE until splitting is possible.

The model is prone to two potential drawbacks in the reallocation of jobs. Firstly, the overhead in copying the environment may be quite substantial; and secondly, a busy processor may be interrupted frequently. The DelPhi machine avoids high overhead in job reallocation by communicating only context-free oracles to idle PEs, and is also able to control the frequency of interrupting busy PEs by altering the *checkin interval* accordingly.

Ali [1986] defines a second, enhanced model that uses global shared control information in an attempt to reduce the overhead associated with job reallocation. Whereas the earlier model did not rely on the presence of shared memory in the proposed architecture, this is now necessary. The shared memory contains control information about the status of each PE in the system. This is used to allow processors to reconstruct their own binding environment when changing jobs, thus removing the need for copying environments. Each PE still uses its local memory to record variable bindings.

The control information specifies, for each PE, the shortest route from the root node to the processor's present position in the search tree, the choicepoints evaluated by the PE, and also which processors share choicepoints. When a processor becomes idle, it selects a busy PE from which to obtain a new job. Before the processor can begin execution on the new portion of the search tree, it first reconstructs the required environment by backtracking to the latest choicepoint common to both processors.

Once the "idle" processor has reached the relevant choicepoint, execution continues as in the earlier model. The processor enters one of two modes: either *recomputation mode* or *computation mode*. If there are no unexpanded branches at the choicepoint, the processor enters recomputation mode and follows the busy processor until a choicepoint is processed that allows splitting to occur. Once the processor becomes solely responsible for the execution of a portion of the subtree, it

enters computation mode.

Although the second model has been described with the need for shared memory, Ali also suggests a way to implement the model without the use of shared memory. Both versions of the enhanced model offer significant improvements over the first model by reducing the overhead in changing from one job to another.

An or-parallel model has been developed independently by Shapiro [1987]. This model has similarities to both the DelPhi model and that of Ali, in that it supports independent processes with no sharing of environment or logical variables. All bindings are therefore local to each process.

The activation of new processes is, however, done at a different stage to that in the other two models. A single process, in *explore mode*, is started at the root node. When a choicepoint is reached with $n$ branches, $n$ new processes are created with a copy of the goal and a *prefix* giving the route from the root node through the tree to the relevant choicepoint. This prefix is analogous to the oracles used in the DelPhi machine.

Each newly created process starts at the root node and *traces* the computation according to the path set out in the prefix, until it reaches the choicepoint. At this stage the process enters *explore mode* and starts executing its own subtree.

Assuming an unlimited supply of processors, the time complexity of the DelPhi model is of the order $O(height(T))$, where $height(T)$ is the deepest path in the tree. The time taken to transmit the oracles and start new processes is ignored in this estimation. In Shapiro's model, all solutions may be found in time proportional to $O(height(T)^2)$. Process setup costs are once again omitted.

ORBIT, the parallel execution model defined by Yasuhara and Nitadori [1984], introduces the concept of a *process bundle* as the basic component for parallel execution. The model is designed for an architecture without shared memory, and attempts to avoid explosive communication overhead. This is done by implementing demand-driven or-parallelism, and by ensuring that the process bundles are large enough to allow a processor to execute a given bundle without interacting with other processors. The implementation is based on a sequential Prolog system, with additional capabilities for dividing the search space at a stored backtrack point whenever a request is received from an idle processor.

The Kabu-Wake inference method [Kumon *et al.* 1986] has much in common with ORBIT. Whenever an idle processor requests a job, the search tree is divided at the oldest unprocessed alternative on the stack. The sending processor backtracks temporarily to this point to create the correct binding environment for the new task and deletes the split alternative from its own stack. Overheads within each processing element are kept low since each processor executes sequentially, except when splitting subtasks.

The models described above have been designed for single-processor systems, without the need for

shared memory. The or-parallel models that follow adopt a different approach, and are targeted at shared-memory multiprocessors. This brings to light a new problem, namely that of dealing with multiple bindings for shared variables.

Warren's SRI model [1987(a), 1987(b)] describes a parallel execution algorithm for implementation on a shared memory architecture. In this model, one process is created initially to solve the top level query. Then at every non-deterministic or-node in the and-or proof tree, $n$-1 new processes are started, where $n$ is the outward branching factor at the node. Once created a processor (or *worker*) is allowed to execute until a leaf node is reached. At a choicepoint the current worker accepts the left-most branch, while the $n$-1 unassigned branches are allocated to the new processes started.

Bindings for shared variables, defined before a parallel split occurred, cannot be entered into the appropriate memory cell in shared memory, since several different bindings may be in existence at any one time. Instead, each worker has its own private *binding array* in local memory in which these bindings are stored. Any bindings generated for variables referenced since the last parallel branch, are recorded in the relevant variable cell in the shared stack, since the binding will be the same for all workers with access to it, and those which are created subsequently. Access to variable bindings may be done in constant time irrespective of whether they are shared or not.

When a worker runs out of work, it switches tasks to some new node with untried alternatives, updating its binding array to reflect the state of the binding array at the new node. Two possible methods have been considered for updating the binding array. The first involves reinitialising the array from scratch, by copying all bindings from the binding array current at the node. Since the new task may normally have a lot in common with the earlier job, the second, and more efficient, method is to backtrack to a common ancestral node, undoing bindings in the binding array in the normal way. From this common node the worker *skims* forward to the destination node, recording bindings as it goes along. Thus, any bindings generated before the common ancestral node are left untouched. The latter method is especially efficient if the distance between the two nodes, in terms of bindings generated, is small.

A potential disadvantage of this model is the cost of switching tasks; in the extreme case the distance between the two jobs, in terms of bindings to be undone and remade, may be very great. The model is thus unsuited to fine-grained parallelism.

The Argonne model [Disz, Lusk and Overbeek 1987] is an extension of the SRI model and introduces the concept of *favoured* and *unfavoured* shared variables to distinguish between variables said to belong to a particular processor and the so-called alien variables. A simple optimization in the use of the binding arrays defined by Warren [1987(a)] is also described.

One hash table per arc of the proof tree is used to record the bindings of shared variables made on

that arc. The hash tables on each branch in the proof tree are kept in a separate chain. Furthermore, each processor keeps a chain of pointers to hash tables containing relevant bindings, that is, bindings accessible to the processor.

Using the distinction between favoured and unfavoured bindings, variable instantiations are recorded as follows. Favoured bindings are entered directly into the value cell in shared memory, with a special tag recording to which processor the binding is relevant. The binding is also recorded in the hash table for the current arc. Unfavoured bindings are only entered into the current hash table.

Access to unfavoured bindings is done by searching through the chain of relevant hash tables. This access time is not bounded, but is proportional to the number of unfavoured bindings made so far on the current branch. To access a favoured variable, or one that is not shared, the value cell may be consulted directly.

Switching tasks is more efficient in the Argonne model than in the SRI model. The processor simply initialises its relevant chain of hash tables to be that which is current at the new node; this involves only the manipulation of certain pointers. The main disadvantage of the model is however, the unbounded access time for unfavoured variables.

A further attempt to solve the problem of storage management in an or-parallel execution model, has been proposed by Ciepielewski, Haridi and Hausman [Ciepielewski *et al.* 1989; Hausman *et al.* 1987]. The model is based on earlier work done by Ciepielewski and Haridi [1983], and has time characteristics similar to that of the SRI model.

The use of shared memory is however approached differently. While the SRI model allows each processor to maintain a binding list of bindings instantiated by the processor, creating a potentially large linear table, the idea pursued by Ciepielewski *et al.* associates (*versions*) vectors with shared variables, with one element of the vector for each processor in the system. Each element of a vector is used by only one processor to store and access bindings belonging to the processor. This can result in some wasted space, since an element in a vector is allocated for all processors in the system, even those that do not reference the shared variable. Access to variables is however, a constant-time operation.

Scheduling is done in much the same way as in the SRI model, with new processes being created whenever a non-deterministic choicepoint is reached. Task creation involves copying each *conditional binding* belonging to the parent process into the versions vector component corresponding to the new process.

To switch tasks, a processor backtracks to the *closest* node to the root with unexplored alternatives (an *open* node). Relevant conditional bindings are removed in the process. Information is maintained by processors to assist in determining which open node is the closest to the root. The measure of "closeness" may be either the depth of the node, or the number of bindings made be-

tween the root and the node, with the latter measure the more accurate. The cost of task switching is therefore dependent on the distance in the search tree to be covered during the installation of bindings.

The main difference between this model and the SRI model is the time at which space is allocated for variables. In the latter model, this happens as soon as an unbound variable is referenced, while in Ciepielewski's scheme, a versions vector is created when the variable gets its first binding. A drawback of this model, is the high space overhead, especially with a large number of processors in the system. Locking is also needed for certain variable accesses; this is not required in the SRI model.

Aurora [Lusk *et al.* 1988], a refined version of the SRI model, has been developed from the experience obtained in designing the SRI and Argonne models and from work done by Ciepielewski *et al.* The major improvement over the SRI model is the more efficient switching of tasks, which is accomplished by the use of a scheduler.

Two different schedulers, the Manchester Scheduler [Calderwood and Szeredi 1989] and the Argonne Scheduler, have been designed for Aurora. The first of these uses global information about the status of each worker (for example, the availability of work for sharing and its migration cost) to allocate work to idle workers. The status information allows the best idle worker, based on the lowest migration cost, to be chosen to receive available work.

The second, the Argonne scheduler, uses very little global data to effect scheduling. Instead, when a worker becomes idle, it actively seeks new work by examining the surrounding nodes for unexplored branches. All decisions are made locally as to whether to choose a branch from the current node or to move along an arc to a nearby portion of the tree. After each step the decision process is repeated.

The main contribution of the Aurora system is the use of the scheduler to coordinate the reallocation of jobs. By ensuring efficient switching from one task to another, one of the disadvantages of the SRI model is abolished.

## 2.2 And-Or Parallel Models

One of the first proposals to support both and- and or-parallelism was defined by Conery [Conery 1987(a), 1987(b)]. To implement *limited and-parallelism*, Conery uses the *generator-consumer* approach, where every shared variable must first be bound by the generating clauses before it can be consumed in parallel by remaining clauses. Parallelism is exploited automatically in that no programmer controls need be supplied.

The And-Or Process Model (AOPM) uses the concept of a *dynamic and-or tree* to define the exe-

cution strategy. Processes are represented by nodes in the tree and communication between nodes is via messages.

Two types of process exist: the *and-process* and *or-process*. An and-process is created to solve a goal. It in turn creates a descendant or-process for each subgoal. And-parallelism exists if more than one descendant is active at a time.

The sub-problem for each or-process is exactly one subgoal which comes from the parent's goal statement. The process either solves the subgoal immediately if it encounters a unit clause, or else spawns descendant and-processes to solve the body. Or-parallelism exists when more than one and-descendant is active.

A process can only communicate with its parent and immediate descendant (child) processes. Each message causes the recipient process to undergo a state transformation;

- a *success* message contains the bindings obtained in solving the problem and signifies to the parent process that the subgoal has been solved;
- *fail* is sent from a child process to its parent and signifies that the sub-problem could not be solved;
- *redo*, sent by the parent to an or-process which has already computed a successful answer, requests a further solution;
- *cancel* kills a descendant process when the parent knows it does not require further solutions.

The size of messages and the or-process setup time is reduced by allowing each or-process to solve just a single goal instead of the entire context of the problem state.

The AOPM has three components; an *ordering* algorithm, a *forward execution* algorithm and a *backward execution* algorithm. The *ordering algorithm* establishes the order in which literals are to be executed based on the existence of shared variables and without violating mode constraints. For each shared variable, one subgoal is designated the generator of bindings. It is quite feasible that a single subgoal could be the generator of some variables and the consumer of others. Based on rules and heuristics (such as the *left-most rule*, which designates the left-most literal containing the variable as the generator of the variable) it is possible to assign a generator to each variable. Because it is possible for a generator to bind a variable to a non-ground term, the ordering algorithm must be applied at run-time, after every successful unification.

From the ordering analysis it is possible to construct a *data-flow graph* which contains one node for each literal and a set of directed arcs indicating the generator-consumer relationships between literals. The *immediate predecessor* of a subgoal, $g$, is one which is the producer for a variable in $g$. Predecessors are in general either immediate predecessors or the predecessor of an immediate

predecessor. A special case arises with the head literal, which is the generator of all variables bound to ground terms on entry to the procedure and the consumer of all unbound variables.

The *forward execution algorithm* is essentially a *graph reduction* process which uses the data-flow graph to decide which goals to execute sequentially and which in parallel. The successful execution of the forward execution algorithm depends on an acyclic graph.

When an and-process ($A$) receives a success message (possibly containing some variable bindings) from a descendant or-process ($B$), the node in the data-flow graph representing $B$, and all outgoing arcs from this node may be removed. The and-process, succeeds when success messages have been received from all descendant or-processes; the graph for that node has then been reduced completely.

An or-process may be started by an and-process when all the predecessor literals to the and-subgoal have been solved. In terms of the data-flow graph this occurs when the corresponding node has no incoming arcs.

The ordering algorithm and forward execution strategy are sufficient for solving clauses defining deterministic functions, such as matrix multiplication, where only one output value exists for each output variable.

*Backward execution* is necessary in non-deterministic programs to prevent the generation of useless tuples of values. A *failure context* of all subgoals that have failed and a *redo list* are used by the semi-intelligent backtracking scheme. The failure context for each subgoal is initially empty, and as fail messages are received, numbers representing the failed subgoals are added to the list. The redo list for a subgoal contains the numbers of all predecessors of the subgoal (*i.e.*, those literals which are generators of some of the variables used in the subgoal) in the linear order determined by the ordering algorithm.

When an and-process receives a failure message from a descendant process, it must resolve some previously solved literal. The backward execution algorithm traces a path from the failed subgoal back to the root using the redo list in reverse order. Once the generator, to which a redo message must be sent, has been identified, all generators appearing after the failed subgoal must also be reset. Later consumer literals may have to be cancelled since they may use a variable that is being changed or reset. If at any stage the backward execution proceeds past a clause with no predecessors, or the head of the clause is reached, the and-process fails.

If the backward execution succeeds, the forward execution algorithm starts a new set of processes for the successor subgoals. These subgoals are then removed from the current failure context.

Conery's model suffers from high run-time computation overheads, with respect to the ordering algorithm. As the model does not include any compile-time analysis or user annotations, a complex ordering algorithm is required at run-time to confirm the instantiation status of variables after every

successful unification. A further weakness of the model is that the backward execution scheme has a rather high overhead in data structures and can cancel some processes needlessly.

Both deterministic and non-deterministic programs may be executed in parallel and optimal parallelism is achieved in most examples. The AOPM does not however support consumer-instance parallelism as defined by Kalé [1987(a)].

A model similar in many ways to the AOPM has been developed independently by Biswas, Su and Yun [1988]. Their proposal is defined for a multiprocessor architecture but does not rely on shared memory. Communication is restricted between parent and child processes and takes place over dedicated communication paths.

The main difference between the *limited-or (LOR) evaluation scheme* of Biswas *et al.* and the AOPM lies in the way that or-parallelism has been implemented. The LOR model does not collect solutions and, as such, or-parallel execution can be viewed as *demand-driven.* Although all the child processes corresponding to the candidate clauses of a subgoal are activated in parallel, the solutions from these processes are not accepted unless necessary. Therefore the demand for another solution corresponds to the need for backtracking in a sequential model. Whereas a "redo" instruction is automatically sent to a child and-process in the AOPM on receipt of a solution, this only happens when a further solution is required by the LOR scheme.

Woo and Sharma [1987] have developed a model based on the AOPM, but which introduces a third type of process, and which supports a form of *integrity constraint* imposed on a literal. The *fact process* is created by an or-process and performs the unification of the unary clause with the goal literal. The results of this unification are returned to the or-process. Answers received by the or-process are only stored in the answer list if they do not violate any of the constraints imposed on the appropriate literal.

Process cancellation in this model is handled in a different way to that in the AOPM in an attempt to minimise the unnecessary cancellation of processes when backtracking. Conery's backward execution algorithm assumes *no cause analysis*; that is, a failed literal reports only that it has failed and not the reasons for failure. Consequently, in some instances an incorrect literal is chosen for backtracking. Woo and Choe [1986] define a more efficient backtracking algorithm in which each literal keeps a *failure history*, which in turn may be used to identify the literal responsible for failure and consequently allow the correct literal to be selected for backtracking.

Lin and Kumar [1986(a)] attempt to reduce the run-time overhead associated with the AOPM by applying the ordering algorithm once only at compile-time. Using variable annotations and suggestions from the programmer, a linear ordering of the literals in each clause can be specified. The forward execution algorithm uses this linear ordering and run-time binding conditions to choose a generator for each shared variable. The algorithm makes use of tokens which are attached to each

variable and are passed between literals which reference the corresponding variable. The token exists until the variable is instantiated to a ground term. Depending on which tokens a literal has, it can determine whether it is suitable for execution or not.

The backtracking algorithm used in this model [Lin, Kumar and Leung 1986(b)] makes use of dependencies between literals in much the same way as described by Chang and Despain [1985(b)]. As such the backtracking scheme is more efficient than the naive backtracking implemented in Prolog.

The Reduce-Or Process Model (ROPM) [Kalé 1987(a), 1987(b)] has been designed with the aim of achieving an and-or parallel evaluation scheme for logic programming which is both complete and supports full or-parallelism. In place of the and-or proof tree for representing logic programs, Kalé uses a *reduce-or* proof tree to allow the distinct representation of actual sub-problems, taking into account the sub-problems arising from multiple solutions to a previous sub-problem.

The ROPM supports only *independent and-parallelism* and thus relies on some form of partial ordering of the literals in a reduce-node. Kalé recommends the use of a *data join graph (DJG)* to effect an ordering of literals whereby one literal per shared variable is designated the *producer* of values for that variable. Each arc in the DJG denotes a literal, while a node represents the joining of data received on the incoming arcs.

Assuming the availability of suitable ordering information, the ROPM starts executing by creating a single reduce-process, *P*, for the given query. Now, for each eligible literal in the DJG (that is, for those literals which have no unsolved predecessors in the graph), process *P* starts an or-process, with a single query. Each or-process in turn creates a reduce-process for each alternative clause whose head matches the given query. Having spawned the reduce-nodes and transferred information regarding the previous generation of reduce-nodes, the or-process terminates and any results obtained by the descendent reduce-processes are returned directly to the grandfather process *P*.

The creation of processes continues until a leaf node is reached, at which time either failure or a valid partial solution may be reported. Solutions filter back up the tree via ancestor reduce-nodes. When a reduce-node receives a partial solution, it must determine whether a complete solution to the query exists and if so it returns this to the grandparent reduce-process. If a complete solution has not yet been found, the reduce-process spawns an or-process for any literals which have now become eligible for execution. A clause is eligible for activation as soon as all shared variables, for which the clause is not the generator, are bound to ground terms. The new or-processes inherit the partial solutions from the or-children of all the predecessor literals.

Back-unification is used to translate solutions from descendant reduce-processes to the parent processes. Although the ROPM supports only independent and-parallelism, a relational join opera-

tion is still needed in back-unification. In contrast to the join operation used in Epilog (where it is required to solve binding conflicts caused by the parallel execution of and-clauses which share unbound variables), the use of the full join in this model ensures that any further instantiation of variable tuples is consistent over the whole tuple and not just with respect to the previously unbound variables being instantiated.

The ROPM as described is prone to potentially prohibitive overhead. This is largely due to the number of processes created and the interaction between processes both in copying existing bindings and in relaying results. Storage requirements for variable bindings can also be excessive, since in supporting the or-parallel search strategy, multiple bindings for some variables need to be maintained. Kalé suggests further research to develop a model which offers a compromise between maximal parallelism and smaller communication overhead.

The PEPSys model [Westphal *et al.* 1987; Ratcliffe and Syre 1987; Baron *et al.* 1988] supports the notion of retroactive process creation thus enabling the system to convert a sequential computation to a parallel one at low cost when additional resources become available. This ensures that resource usage is maximised without placing unnecessary overhead on a program that will eventually be executed sequentially.

Where idle processors are available, a busy processor may split its job at a branch point with unprocessed alternatives. The new process will share all variable bindings that occurred before the split with its ancestor process. A novel mechanism of time stamping a binding enforces consistency in the sharing of variable bindings. Variable bindings are not copied at the time of process creation, but instead whenever the new process first references the variable. If the new process persists, several inter-process communications may be necessary in order to gain access to all the common variables.

Where no idle resources are available, a process will backtrack to the latest unprocessed choicepoint and itself execute the remaining alternatives. Backtracking beyond a choicepoint where an or-branch has been ceded to a separate processor can only be done once all processes active at that point have terminated.

And-parallel process creation is done on demand. If the right-hand subgoal has not yet been distributed by the time a processor has completed the left-hand subgoal, the same processor continues executing the right-hand subgoal. Alternatively, where idle processors are available, the execution of the right-hand subgoal may be ceded to another processor. The first of the processors that completes its section of the and-branch, waits for the other processors to finish and then a join of the partial solutions is performed. Where multiple solutions are returned for either side of the and-branch, the cross product of the left and right-hand solution sets is performed incrementally, that is, as a new solution is generated on one side it is immediately joined with all the existing solutions on the other side and *vice versa.* Since only independent subgoals may be executed in

parallel no conflicting bindings for shared variables can exist. However, the successful operation of the incremental join is dependent on all partial solutions being available to all processors at all times. This may not always be so as certain solutions may be discarded on backtracking. As a result some recomputation may be necessary to re-generate certain bindings.

Disadvantages of the PEPSys model lie in the recomputation of and-subgoals necessary to ensure that a full cross product of partial solutions is generated, as well as the indirect access to non-local variables through hash-tables. The language designed to support the PEPSys model includes a right associative operator to indicate the independence of and-clauses and pragmas to denote or-clauses suitable for concurrent execution. The exploitable parallelism is expressed explicitly and left exclusively in the hands of the programmer.

Chengzheng and Yungui [1986, 1987] introduce the *or-forest* (instead of the traditional or-tree), to describe an and-or parallel execution strategy, which takes advantage of the independence of and-subgoals to avoid redundant execution caused by duplicating or-branches. The or-forest is constructed by splitting the subgoals for any goal $G$, into the following groups:

- $g_1, g_2, \ldots g_n$, where each subgoal in $g_i$ must be *independent* of each subgoal in $g_j$, with $i < j$, and $1 \leq i$ and $j \leq n$. This implies that none of these groups may share variables, although the subgoals within a group may do so.
- $g_0$, in which each subgoal is *interdependent* of at least one subgoal in every $g_i$, where $1 \leq i \leq n$. This group may be empty.

If the goal $G$ can be split into $n$ groups, then $n$ *independent son trees* can be created, with the root nodes labelled $g_1, g_2, \ldots g_n$. If $n > 0$, $G$ is termed a *seed node*. The number of successor nodes to $G$ is dependent on the number of solutions to each of the son trees. Let $k_i$ be the number of solutions for the tree $g_i$. The number of successors for $G = k_1 \times k_2 \times \ldots \times k_n$.

By partitioning subgoals in this way, it is possible to avoid redundant parallel computation of multiple alternative clauses, since any group of subgoals with multiple alternatives can only appear in a single son tree.

The parallel execution strategy starts at the root node of the or-forest. If the current node is a seed node, $n$ *slave processes* are created to evaluate the $n$ son trees. The results from all these processes are combined and processes may then be created for the successor nodes of the seed node. If the current node is not a seed node, processes may be created immediately to evaluate the successor nodes.

The execution strategy achieves the intended parallelism. Or-parallelism exists as each successor node has its own process, allowing multiple alternative clauses to be searched in parallel. Backtracking is not necessary as each alternative is searched by a separate process. Independent

and-parallelism is achieved by executing the son trees in parallel by slave processes.

Processes are logically independent, communicating only between parent and child to allow the transfer of environments (*i.e.*, the instantiation of the arguments in the head of the clause) and the return of results to take place. Communication overhead is comparatively low. The model does, however, rely on the creation of a comparatively large number of processes.

The Sync Model [Li and Martin 1986] defines a parallel execution strategy for the extended logic programming language, CLP. The model assumes a message-passing multi-processor architecture, with the processors interconnected into an augmented binary tree, known as a *Sneptree.*

CLP allows *variable annotations* in the clause body to specify the producers of bindings for shared variables. A *conditional operator* is also included in the language to serialise the execution of the clause body. Neither of these features are compulsory; thus any Horn clause program may be usefully executed using this model.

The model may be classified as a *multiple-solution data-driven model.* Two types of processes are created, *and-processes* and *or-processes.* An and-process corresponds to a goal, while an or-process represents a clause used to reduce the goal. Initially an and-process is created for the query. For each candidate clause unifiable with the goal, an or-process is created; or-parallel execution is thus initiated. If the clause in an or-process has a non-empty guard, an and-process is started for each goal in the guard. When all of these and-processes terminate successfully, an and-process is created for each goal in the body of the clause.

Conflicts in binding shared variables are prevented by designating one and-process as the producer of each variable, thus supporting limited and-parallelism. All other and-processes that use the variable are consumer processes, and will suspend computation until the values of the shared variables are received from the producer processes. A data-flow graph is constructed to represent the *producer-consumer* relationship between and-processes.

A non-leaf and-process succeeds when at least one of its descendent or-nodes succeeds. Bindings from the descendent nodes are transmitted by the and-node to both the parent node and all consumer sibling nodes. The arrival of these bindings allow the consumer processes, which have been suspended, to resume execution.

The creation of several or-processes by an and-process may result in multiple solutions being generated for a single query. All bindings computed are immediately transmitted to parent and sibling nodes, without requests being issued; hence the use of the term *data-driven* in describing the model.

With the generation of multiple alternative solutions and while supporting limited and-parallel execution, it is necessary to synchronize the use of bindings from different sources. If an and-process receives inputs from two different sources, the *cartesian product* of the streams is calculated to

form all input combinations. Where the two streams are generated by processes with the same ancestor, the *sets of cartesian products* over certain portions of the input streams provides the correct input combinations. The Sync model copes with this synchronization by allowing different generators to output different *Sync signals* together with the bindings.

The data-flow graph may be dynamically altered if a variable is not bound to ground terms by a producer process. In this happens a new generator process is chosen and *dynamic links* between the new generator and consumer processes are inserted into the graph. A simple test on the instantiation state of variables is sufficient to determine whether dynamic links are needed.

Although the model handles both deterministic and non-deterministic problems, it is particularly suited to non-deterministic problems with multiple solutions. Backtracking is entirely eliminated through support for or-parallel execution.

## 2.3 Restricted And-Parallelism

*Restricted and-parallelism* (also known as *limited and-parallelism*), allows only independent subgoals to be executed concurrently. This ensures that only variables bound to ground terms may be shared and thus circumvents the problems associated with maintaining consistency in shared bindings.

The Restricted And-Parallel (RAP) model of DeGroot [1984, 1987] relies on compile-time generation of *execution graph expressions* and a system of allocating a *type* to each variable in the program, to reduce the high run-time overhead associated with Conery's AOPM.

The type of each variable is classified as being one of *ground, variable* (*i.e.*, uninstantiated) or *non-ground, non-variable.* The types of most variables may be preset at compile-time. Variables appearing in the heads of clauses, however, are dependent on run-time values. The run-time typing algorithm uses an approximation technique to avoid traversing entire structures at high cost. Since only the type codes of the top level components of a structure are taken into consideration in determining the type of a variable after unification, the typing algorithm sometimes fails to detect independence amongst variables. This means that some parallelism may be undetected.

One execution graph expression is generated for each clause. All program graphs are constructed from the six graph expression types allowed. The limited number of graph expressions may result in a further loss of parallelism; this loss is, however, anticipated to be insignificant.

Run-time algorithms to support the graph expressions are inexpensive and comprise a test to determine whether two or more variables are ground (GPAR), a test to determine whether variables are independent (IPAR) and the *typing algorithm* applied at unification.

The RAP model is intended to execute on a tightly-coupled parallel architecture with each proces-

sor having a large local memory. Two expression stacks, *i.e.*, a parallel and a sequential stack, are used by a processor to keep track of subgoals to be executed. When a query is read, it is converted into an execution graph expression and placed on the sequential stack. Each processor checks the top of the sequential stack for the next piece of work. If the task is a goal, it is executed normally. If it contains a test (either GPAR or IPAR), the test is evaluated and the subgoals are placed either on the sequential stack or the parallel stack depending on the result of the test. When the sequential stack is empty, the processor obtains work from the parallel stack. If both stacks are empty, the idle processor may request work from an active processor. Division of work is permitted provided that the subtask to be allocated to the idle processor is non-trivial.

The backtracking algorithm in the RAP model is based on that proposed by Chang and Despain [1985(b)] and uses backtrack points, calculated at compile-time, for each literal. At most two backtrack points are established for each literal, for use when the literal is executed sequentially and in parallel respectively.

A model with more extensive compile-time analysis is described by Chang *et al.* [1985(a)]. This model achieves more parallelism than the AOPM of Conery by enabling parallelism in both forward and backward execution.

The static analysis depends on the user to supply the *activation modes* for each variable in the top level query; possible classifications are *ground, coupled* (where two variables share at least one unbound variable) and *independent* (*i.e.*, neither ground nor coupled). With this information it is possible to generate *activation modes* and *exit modes* for all procedures, based on worst case analysis. The output of this analysis is a dependency graph for each clause, which is used during forward execution. No run-time tests are used in this model.

Backtracking within a clause proceeds in a *semi-intelligent* way, using the information contained in the data dependency graph. Furthermore, where two backtrack paths do not interfere with one another, backtracking can proceed in parallel. This is not possible with Conery's model, since the dependency graphs are not static.

*Co-operative failure detection* is also possible in Chang's model. As soon as one of the literals operating in parallel detects a failure, backtracking can begin. The backtracking algorithm is, however, not able to cross clause boundaries, since the dependency graphs are based on static analysis. Where this is necessary, naive backtracking is used to select the last choicepoint with untried alternatives for re-evaluation. The selective backtracking of Pereira and Porto [1980] uses data available at run-time, with the result that backtracking can cross clause boundaries. This is, however, achieved with a considerably higher run-time overhead. Although the semi-intelligent backtracking scheme may not be as precise, the run-time costs are low.

Like the RAP model of DeGroot, this model is not guaranteed to explore and-parallelism to the

fullest, since the activation modes are based on worst case analysis and no run-time checks are performed.

Hermenegildo's RAP-WAM model [1986] combines a compile-time analysis with simple run-time checks, making it possible to choose at run-time between parallel and sequential evaluation. The model seems to offer a compromise between the system designed by Conery [1987(a)] and that by Chang *et al.* [1985(a)], since the compile-time analysis is not as extensive as that of the latter model, yet the run-time overhead is significantly smaller than that associated with the AOPM.

At compile-time, *conditional graph expressions* (CGE) are embedded within the original Prolog code as a result of the analysis done to determine which literals may contain shared variables. The syntax of the CGE's and the definition of the RAP technique used by Hermenegildo differ slightly from the corresponding execution graph expressions and the technique defined by DeGroot, since in the former model the clauses are actually "rewritten" to include the CGE's.

A CGE consists of a series of conditions (or checks), followed by a conjunction of goals. The checks include simple tests for independence among variables and whether a particular variable is bound to ground terms. If the checks evaluate to true at run-time, the goals inside the CGE may be evaluated in parallel. Otherwise the goals are executed sequentially.

Intelligent backtracking is performed using run-time information about goal independence to avoid unnecessary backtracking. During sequential forward execution, a choicepoint is created. Every time a CGE succeeds and a series of goals is executed in parallel, a *parallel call marker* (PCM) is pushed onto the stack. Also defined is a boolean variable (*inside*) which is set true when a series of parallel goals is evaluated for the first time.

On failure, the backward execution algorithm is employed [Hermenegildo and Nasr 1986]:

- If the latest recorded marker is a choicepoint, backtracking takes place as in a sequential model.
- If the marker is a PCM and *inside* is true, all parallel goals are killed, and the algorithm is applied recursively. This shows the intelligence of the backtracking scheme. Since parallel evaluation is underway, the CGE must have evaluated to true, and consequently the parallel goals do not share variables. Thus it is safe to kill all the parallel goals because redoing any of them will not result in any alternative solutions which will prevent the failure from occurring a second time.
- If the marker is a PCM and *inside* is false, naive backtracking occurs within the CGE. The first clause with untried alternatives, found while searching from right to left, is redone and, on successful completion, all clauses to the right of this clause are also redone. If no such clause is found, the backtracking algorithm is applied recursively.

Hermenegildo's model is fully compatible with the traditional WAM implementation of Prolog. Optimizations current in sequential systems are thus applicable to the model and the efficiency of sequential execution is comparable to that attained in a sequential implementation.

## 2.4 Other And-Parallel Scheduling Methods

Wise [1982, 1984, 1986] describes a data-flow model, which exploits both full or-parallelism and dependent and-parallelism, for *Epilog*, a parallel extension of Prolog. The logic component of the language is similar to that of Prolog, but the control mechanism of Prolog has been altered to create a non-deterministic decentralized parallel framework within which some residual sequential control is available to the programmer.

The sequential control mechanism in Prolog is dependent on the existence of both a stack containing control information and a heap for storing variable bindings. Neither of these can be used in Epilog, since they require some centralized manipulation and administration. Instead, a *dynamic frame* (or *dframe*) is created for each instance of a clause in Epilog. Each dframe can be thought of as a logical processor, containing all the information necessary to activate the clause at the appropriate time, as well as a number of binding arrays in which the bindings to the clause variables are recorded. Message passing in the form of packets or tokens establishes communication between dframes.

The decentralized control is enforced both by the arrival of a query packet resulting in the *activation* of a dframe, and by the return of result packets to a parent dframe. On receipt of a query packet, the dframe unifies the contents of the incoming packet with the clause head. As soon as all the conditions imposed by data dependent sequencers have been met, the literals in the body of the clause are activated in parallel. This ensures support for and-parallel execution.

Once a leaf node has been reached a result packet is returned to the parent dframe. This packet contains any bindings to parent dframe variables made by the child dframe. On receipt of the result packet, the parent dframe instantiates the relevant variables in accordance with the bindings returned. This process is known as *back-unification.*

It is possible that multiple result packets may be returned to a single parent dframe. This can be explained by the fact that each activated literal will simultaneously send packets to all clauses with a matching clause head (an *acquaintance clause*), thereby supporting or-parallel execution. On receipt of multiple result packets a set of *variable tuples* is established in the parent dframe. Any further arrivals of result packets from other and-literals are then back-unified with the complete set. This operation is in fact an *equijoin* across variables common to both sets of variable tuples.

To prevent the combinatorial explosion of new nodes and the associated high communication overheads normally associated with data-flow models, Epilog allows specific sequencing constructs.

Fixed sequencers cause the same execution ordering whenever a clause containing the constructs is evaluated, while data dependent sequencers provide the pre-conditions for clause activation. This latter group includes the *threshold values* tagged to a clause, whereby the minimum number of variables instantiated to ground terms in the head of the clause must be at least equal to the given threshold before the clause may be activated. The effect of introducing threshold values is to delay the activation of certain clauses until their execution will result in useful computation. Variable annotations are also allowed, which provide more specific control by requiring that the designated variables (either in the clause head or in the body) be bound or unbound, when activating a clause or during back-unification.

Tebra [1989] defines an *optimistic and-parallel* scheduling method for Prolog programs that allows literals to be executed in parallel irrespective of whether they contain uninstantiated shared variables. The model is designed for a shared memory multiprocessor architecture, with transputers used as the key components.

The approach adopted by Tebra assumes that binding conflicts are rare. Instead of using complex schemes to avoid such conflicts, the model implements *locking on shared variables, dynamic conflict detection* and a *"repair"* operation to correct bindings that are found to be inconsistent. The techniques used resemble those found in database systems which employ optimistic concurrency control [Kung and Robinson 1981].

*Solver processes* are used to solve and-parallel subgoals. The parent process maintains an array in which is recorded the status of all descendent solver processes. The status of a process is one of *busy, success* or *failure.*

Parallel resolution consists of three stages:

- the unification of the query with the head of a matching clause,
- creation of solver processes to solve the subgoals in the body of the clause, and
- monitoring the status array to ascertain when all descendent processes have succeeded.

The unification algorithm used in the model includes *validation of bindings*; if instantiation of a variable leads to failure, the validation routine decides whether the failure is genuine, or whether the current instantiation of the variable ought to be rejected since it was made by a process that *ran ahead* of the current one. A process is said to *run ahead* if the subgoal it represents appears to the right of another subgoal for which a process has not yet completed.

To implement the validation routine, unique priorities are assigned to all processes involved in parallel resolution. These priorities reflect the sequence of processes in a sequential interpretation. This priority is attached to all bindings made. If an inconsistent binding is found which was made by a process with a lower priority than the current process, the binding may be replaced and the

lower priority process is *pre-empted.* All processes which access the variable in question are also restarted. This means that all accesses to variables must be recorded.

Backtracking depends on the existence of processes with higher priority and access to the variables which are also accessible to the failed process. The backtracking algorithm scans all active processes to find candidates to be redone. If no candidates can be found, the failure is definite; otherwise the failed process repeatedly attempts to re-solve its goal. At the same time the process found by the backtracking algorithm is restarted to find alternative solutions.

The model offers a novel solution for exploiting full and-parallelism. It is, however, prone to high overhead in the setting of locks and monitoring the status array. Where many conflicts arise, parallel execution deteriorates and recomputation of subgoals increases.

# 3 The DelPhi Principle

The use of *oracles* to define a unique path through a proof tree is described by Clocksin and Alshawi [1986, 1988; Clocksin 1987]. The DelPhi principle and the corresponding DelPhi machine [Klein 1989], rely on splitting the proof tree into independent subtrees, with each subtree being allocated to a separate *processing element (PE).* The key principle involved in the search of proof trees is summarized hence:

> "The computation of a given single pre-determined path of the proof tree (from the topmost goal node to a terminal node) is isolated to a single processor."
> [Clocksin and Alshawi 1988]

The fundamental issues embodied in this proposal are:

1. that *no copying or sharing* of Prolog environments should be necessary when distributing the computation. (A Prolog environment contains both the data structures with the current variable bindings as well as control information such as that pertaining to choicepoints.)
2. that *no shared memory* be used,
3. that *no specialized hardware* be employed in the implementation of the model, and
4. that the parallelism inherent in Prolog programs should be *exploited implicitly* without the need for programmer annotations.

To preclude the necessity of transferring Prolog environments between processing elements (PEs), all PEs start their computation at the root of the proof tree. Then by following a path through the search space as defined by the oracle allocated to the particular PE, the computation progresses until one of three states is reached:

1. either a terminal node is reached and the PE can report the solution,
2. or the path leads to failure, in which case this is reported and the PE halts temporarily,
3. or the oracle string is exhausted before either of the previous conditions apply.

Exactly how the PE continues when reaching one of the latter two states is dependent on the control strategy used and will be covered superficially in a later section. The reader is referred to [Klein 1989; Wrench 1990] for a full enumeration of the control strategies investigated.

## 3.1 The Or-Proof Tree

The original motivation behind the DelPhi principle was the need to provide an efficient method of exploiting or-parallelism in Prolog. Thus to ensure that the search tree is suitably divided, it is necessary to convert the traditional and-or proof tree into an or-only tree, where any and-branches are appended to the appropriate leaf nodes. Consider the and-or tree shown in Figure 1, and the equivalent or-tree given in Figure 2. (Since the proof trees given represent a hypothetical Prolog program, the arcs emanating from and-nodes cannot be labeled with the respective named subgoals. Instead an alphanumeric label is assigned to each arc, representing a particular literal in the body of one of the clauses in the program (*e.g.* the label 3b represents the second literal in the body of the third clause in the program). Likewise, the arcs emanating from or-nodes are labeled with a clause number, *i*, representing the *i*-th clause in the program.)

Using the transformed or-proof tree, it is possible to enumerate the independent paths from the root node to the terminal nodes by composing the following ten oracles:

[1,1a,3,1b,1c,5], [1,1a,3,1b,1c,6], [1,1a,3,1b,1c,7],

[1,1a,4,4a,4b,1b,1c,5], [1,1a,4,4a,4b,1b,1c,6], [1,1a,4,4a,4b,1b,1c,7],

[2,2a,8,8a,8b,11,8c], [2,2a,8,8a,8b,12,8c], [2,2a,9], [2,2a,10]

From this is it evident that ten independent paths exist between the root node and the leaf nodes. Since the and-nodes have been decomposed into deterministic nodes, it is not vital to include the and-path in the oracle strings, and a more concise set of oracles can be defined thus:

[1,3,5], [1,3,6], [1,3,7],

[1,4,5], [1,4,6], [1,4,7],

[2,8,11], [2,8,12], [2,9], [2,10]

From the above it is clear that ten PEs can each be assigned an oracle to enable them to process a unique portion of the proof tree independently and without regard for, or communication with

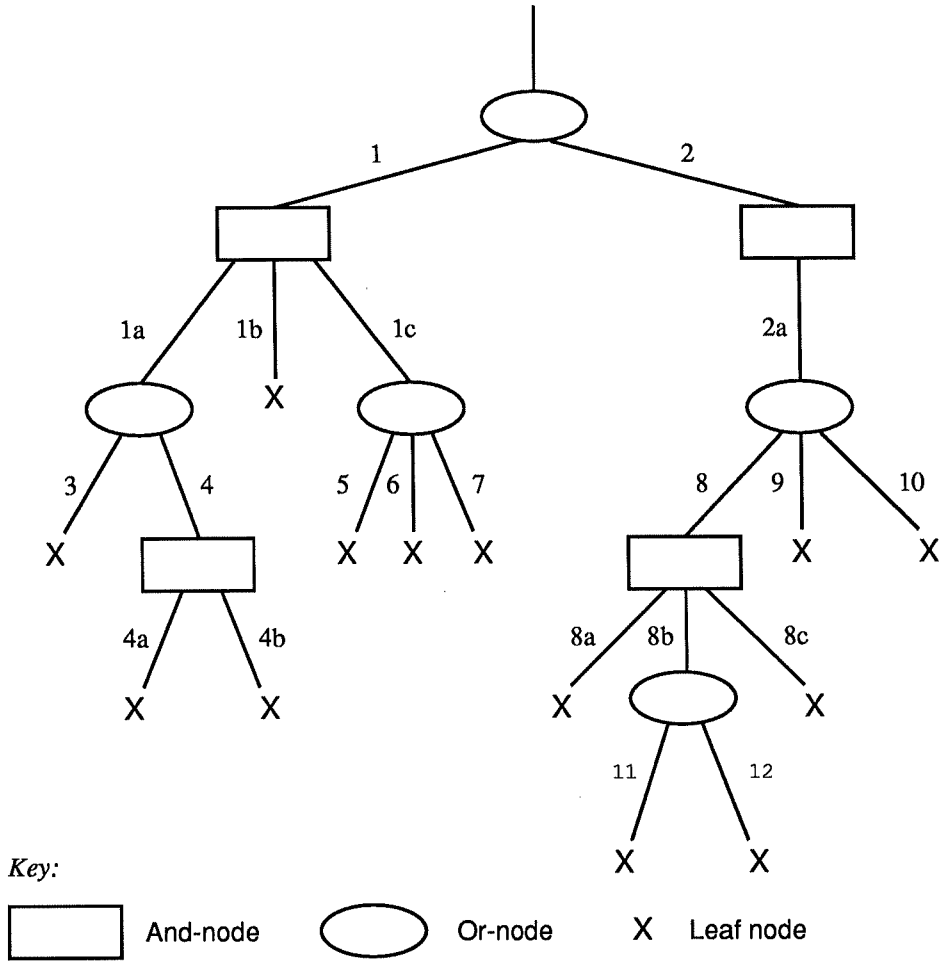


Figure 1: An and-or proof tree

the other PEs in the network. The variable bindings and state information needed by each PE, are generated locally, thus avoiding the issue of communication of environments. All solutions produced are *complete.*

By comparing the oracles that delineate the or-tree in Figure 2, it is apparent that a degree of *recomputation* is performed by certain PEs. This happens where an and-branch appears on more than one unique path, or where PEs duplicate earlier parts of the proof tree before reaching their unique subtree. The recomputation associated with duplicating and-branches should have no negative effect on efficiency when compared to a sequential system, since this is exactly what happens when backtracking takes place across an and-branch. The recomputation of earlier parts of the proof tree may be ignored if sufficient PEs are available. (The interpretation of sufficient is dependent on the control strategy used and will be expanded in a later section). Based on this assumption, the recomputation does not contribute to the total parallel computation time and is preferable (and mostly more efficient) than communicating environments between PEs (*cf.* the models for or-

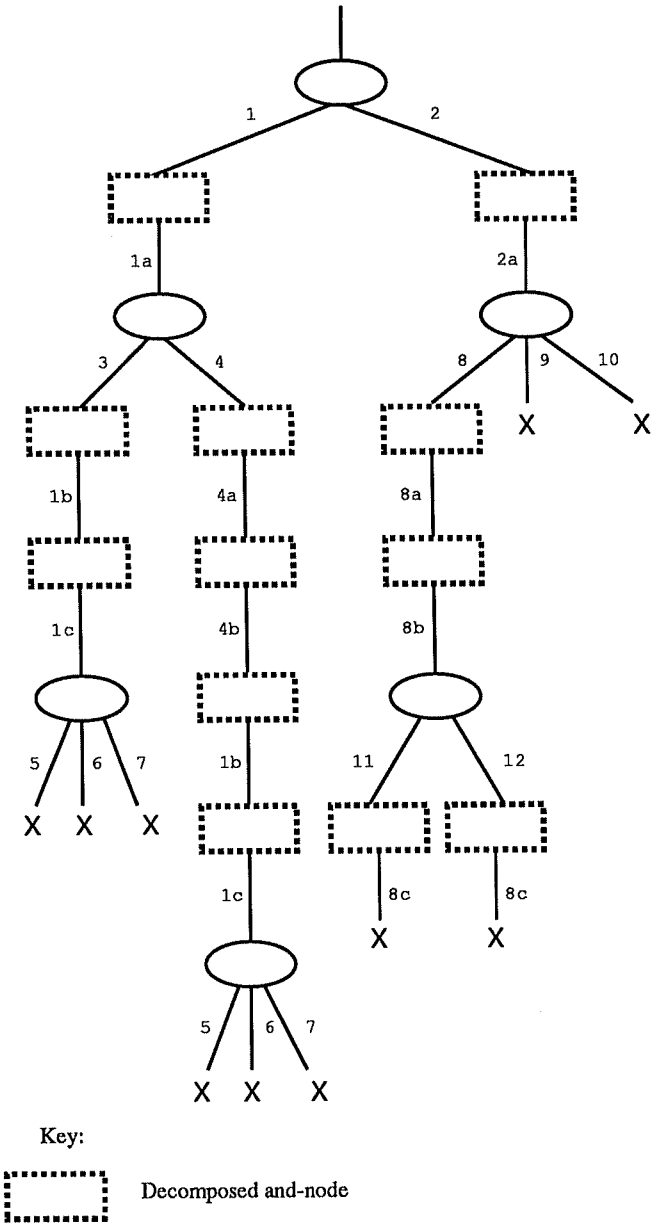


Figure 2: An or-only proof tree

parallel Prolog defined by Shapiro [1987] and Ali [1987] which share a similar recomputation cost, and that by Warren [1987(b)] which relies on copying environments).

## 3.2 Evaluation of the DelPhi Model

Two of the greatest overheads, with which researchers have had to contend in designing or-parallel execution strategies, are associated with the creation and access of variable bindings and with task creation and task switching. The DelPhi model does not suffer from either of these maladies.

Variable bindings are created and accessed as in a sequential Prolog implementation, where all bindings are directly accessible locally. Process creation is relatively inexpensive as it involves only the communication of an oracle, describing the path to be followed by the new process.

A number of non-deterministic benchmarks have been evaluated using one or more of the DelPhi control strategies. As the number of available processors increases, execution time decreases until no further or-parallelism is exploitable in the Prolog program. No degradation in performance has been detected when using all available processors, irrespective of the amount of parallelism present in the source program.

Understandably, since only or-parallelism is exploited, no speed-up is shown when executing deterministic problems. Efficiency is, however, comparable to that of a sequential evaluation scheme. The inability of the DelPhi machine to extract parallelism in deterministic problems is the factor that motivated the development of the APPNet.

# 4 A Novel And-Or Parallel Network

The APPNet has been developed adhering to the following design goals, many of which correspond to those given for the DelPhi machine.

1. The model should not rely on any shared memory, as the targeted architecture for the implementation is a distributed network of processing elements (PEs). The hardware for these processors should not require any specialized features, but should comprise whatever is readily available at the time.
2. The system should support unmodified Prolog programs. Support for parallelism is thus implicit and not dependent on user annotations, such as mode declarations. These declarations are used in most of the committed-choice languages, to allow some degree of control of the parallel execution to be left in the hands of the programmer, and in the restricted and-parallel execution models of DeGroot [1984] and Hermenegildo [1986], to enable a partial ordering to be created.
3. Communication overhead should be kept to a minimum. This is the major obstacle to be overcome when designing an and-parallel evaluation scheme. One way of addressing the problem is to partition the search space as far as possible into independent subtasks (*e.g.* the restricted and-parallel models). An alternative approach requires that a base model with a low overhead in communication, such as the DelPhi model [Clocksin and Alshawi 1986], is used.
4. The model should be complete with respect to or-parallelism. This implies that both the consumer-instance parallelism [Kalé 1987(a)], as well as the traditional or-parallelism, should

be exploited.

5. Maximal concurrency should be extracted within the bounds of the available resources. Where no parallelism is possible due to limited resources, programs should be executed with efficiency comparable to that possible in a sequential system. The overhead should therefore be restricted to actual parallel execution of programs, and not merely potential parallel execution. The APPNet has been developed to take advantage of concurrency using the available resources.
6. Finally, the cost of supporting or-parallelism should not be greater than the overhead inherent in backtracking through alternative nodes in a sequential Prolog system.

## 4.1 Oracles and the APPNet

The traditional and-or proof tree is used to represent the partitioned search space. In the same way as oracles are used by the DelPhi machine to allocate unique or-paths to independent PEs, the APPNet uses oracles that consist of two path enumerations, an *and-path* and an *or-path*, to assign individual subtrees to PEs. To illustrate this partitioning, consider the and-or tree given in Figure 1. It has been shown (in Section 3.1) that a maximum of ten PEs can acquire a unique portion of the given proof tree where partitioning is limited to or-branches. Where an and-or parallel evaluation strategy is employed, it is possible to allocate unique paths through the search space to thirteen PEs. The thirteen unique paths can be described by the thirteen oracles given below:

[1a][1,3], [1a,4a][1,4], [1a,4b][1,4], [1b][1], [1c][1,5], [1c][1,6], [1c][1,7],

[2a,8a][2,8], [2a,8b][2,8,11], [2a,8b][2,8,12], [2a,8c][2,8], [2a][2,9], [2a][2,10].

Since it is not practical to enumerate the clauses and literals in a Prolog program as has been done in Figure 1, a more suitable format for the oracles focuses on the branching factor at each node. Thus, the *i*th element (or bit) in the oracle (where bits are separated by commas) represents the branch to take at the *i*th node on the current execution path. Using this notation, the oracles presented above may be rewritten as:

[1][1,1], [1,1][1,2], [1,2][1,2], [2][1], [3][1,1], [3][1,2], [3][1,3],

[1,1][2,1], [1,2][2,1,1], [1,2][2,1,2], [1,3][2,1], [1][2,2], [1][2,3].

The notation used in the remainder of this paper to describe oracles is as shown above, *i.e.*, as a *pair of path enumerations*, representing the and-path and or-path respectively. Each path consists of a list of small integers, which define the choice to make whenever an appropriate and- or or-choicepoint is reached by the PE.

If each unique traversal of the tree is allocated to a separate PE, thirteen *partial solutions* (or partials) are generated. These will eventually be combined to produce the ten *complete* solutions, which were originally generated by the DelPhi machine using only an or-parallel evaluation strategy.

When a process is created, it starts executing at the root of the proof tree, and traverses a given subtree until a leaf node is reached. If the and-path is empty, then no and-parallelism has been extracted on the path and the solution found is complete. As such it can be recorded as one of the solutions to the query. If on reaching the leaf node, the PE has traversed only part of an and-branch (recorded by the non-empty and-path oracle), the solution found is then only a *partial solution* to the query and needs to be joined with the other partial solutions found by the PEs traversing the remaining branches of the common and-node.

## 4.2 The APPNet Configuration

Figure 3 depicts the and-or proof tree for the simple *map colouring* problem used in this section. Each leaf node is labeled with the respective fact as well as the corresponding oracle. Only non-deterministic nodes are shown in this and subsequent figures depicting Prolog proof trees.

```
map(A, B, C)   :- mainreg(A, B), adjacent(B, C).
mainreg(Reg1, Reg2) :- colour(Reg1, Reg2), primary(Reg1).
mainreg(Reg1, Reg2) :- colour(Reg2, Reg1), primary(Reg1).
adjacent(Reg1, Reg2) :- colour(Reg1, Reg2).
adjacent(Reg1, Reg2) :- colour(Reg2, Reg1).
colour(blue, yellow).
colour(blue, purple).
primary(blue).
```

With an appropriate partitioning strategy and sufficient PEs, ten distinct vertical slices can be distinguished. Thus one could expect a maximum of ten partial solutions to be generated by *vertical processors* scanning the tree in a top down (depth-first) sequence.

Once the vertical partitioning has been done, a process is required at each *horizontal level* of the tree (*i.e.*, a horizontal processor) to collect solutions found at that level and to reconcile any partial solutions resulting from a *distributed and-node* (DAN). A DAN arises when more than one PE is responsible for executing the branches leading from the and-node.

To contain communication overhead, it is feasible to allocate horizontal PEs only to those levels that contain distributed and-nodes. A PE is only capable of generating partial solutions when processing a path which consists of an incomplete and-node. Consequently, no processing would be done by horizontal PEs allocated to any levels other than those containing a DAN. Two horizontal

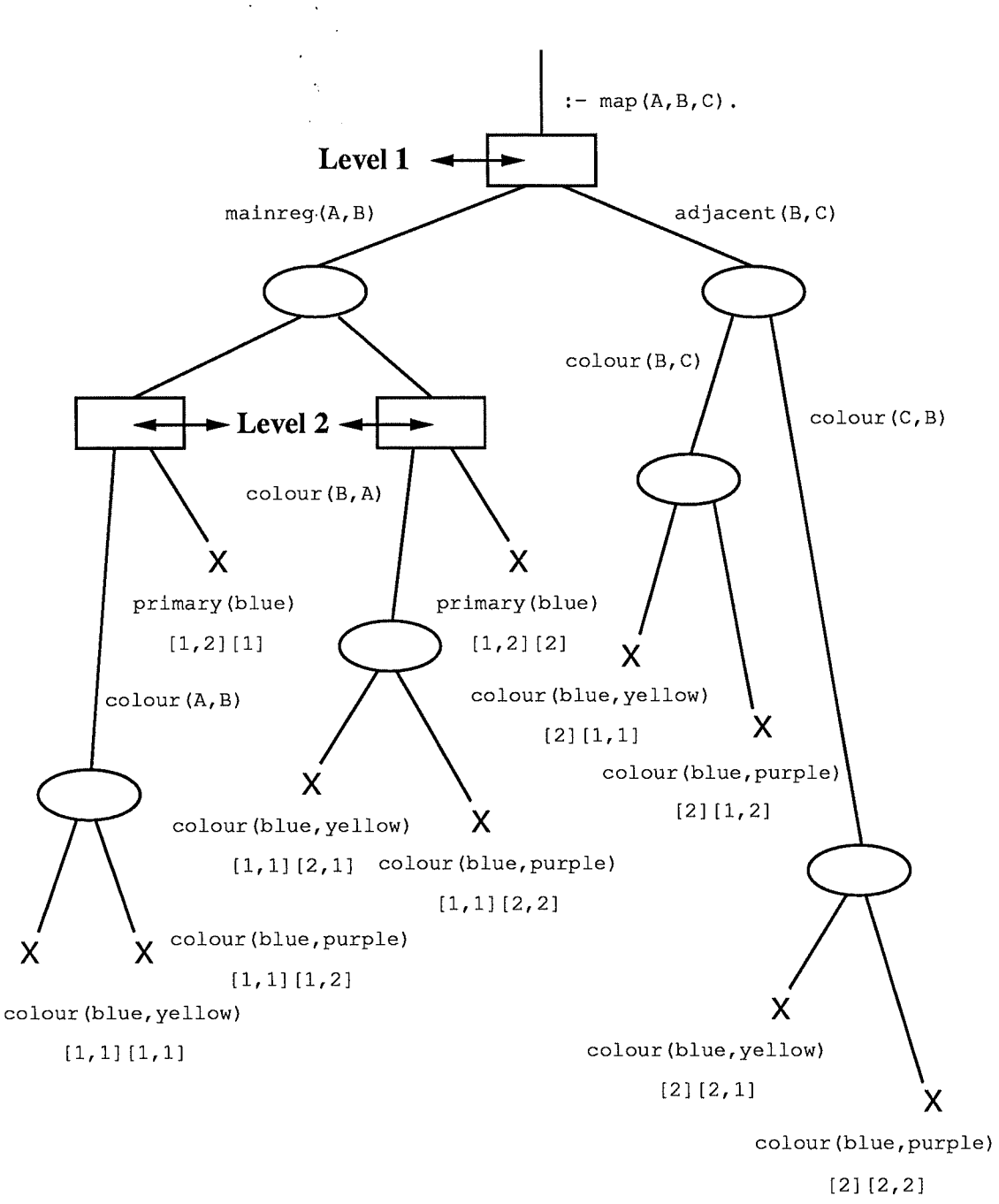


Figure 3: Proof tree for the *Map Colouring* problem

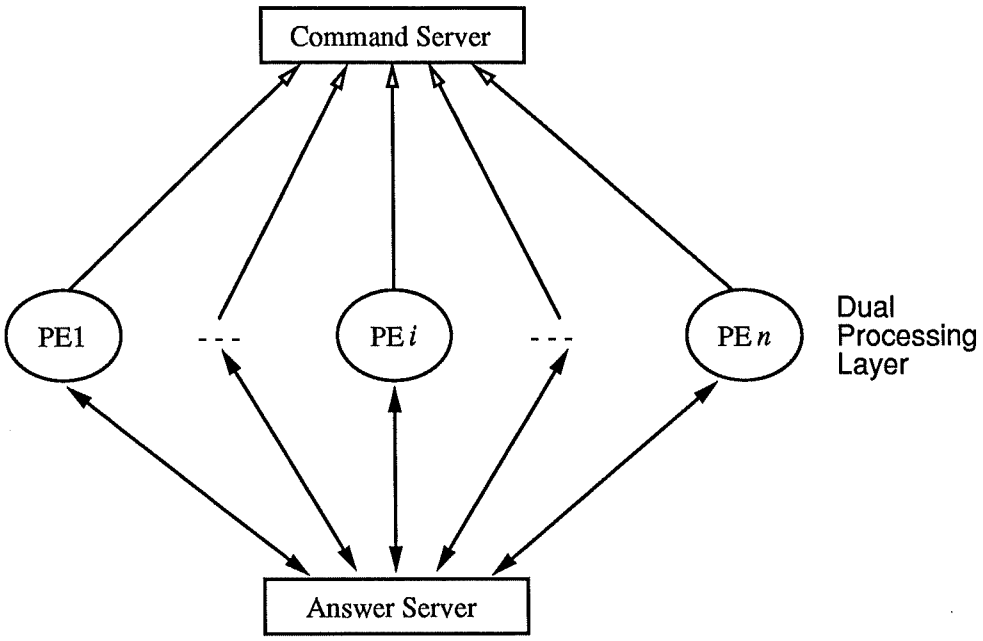


Figure 4: The APPNet configuration

PEs would suffice to complete the back-unification for the given map colouring problem at levels 1 and 2 respectively.

In practice this naive configuration turns out to be very inefficient, as the horizontal PEs are idle for a disproportionately large part of the whole computation. Furthermore the configuration necessitates multiple channels of communication between vertical PEs and horizontal PEs, as well as a routing algorithm so that partial solutions at different levels may be allocated to the appropriate horizontal PE.

To obviate the need for the individual communication channels between PEs, some centralized control in the collection and distribution of partial solutions has been introduced in the form of a new process, the *Answer Server*. All partial solutions generated by vertical PEs are sent to the Answer Server where they are stored until requests from idle horizontal PEs are received. This means that horizontal PEs are not dedicated to a single level of back-unification, but may be used to perform back-unification at several levels of the proof tree depending on the availability of partial solutions.

The inefficient resource usage can be solved by discarding the horizontal PE as a separate processing element and combining its role with that of the vertical PE. Back-unification may now be done by any PE, after completing the execution of its allotted vertical path. The resulting APPNet configuration is illustrated diagrammatically in Figure 4.

Each PE is an extended sequential Prolog processor, capable of interpreting oracles, and therefore able to direct its own execution to a distinct portion of the search space. All PEs are identical except for a unique identification number allocated by the Command Server when the PE is activated. This number serves as the only means of distinguishing between the processors.

## 4.3 A Two-Phase Parallel Computation Rule

Using the configuration given in the previous section, it is possible to describe a parallel computation rule for the APPNet. The model of execution is described in two phases, corresponding to the dual responsibility of the processing elements; firstly, in generating partial solutions and secondly, in combining them to form complete solutions.

At system startup, the Command Server configures the network by loading the entire user program plus the top level query into each PE. This creates an initial configuration suited to independent execution in each PE and also means that no additional loading of any of the source code needs to be done if a PE were to execute a different subtree at a later stage.

As soon as a PE receives the go-ahead, in the form of its unique identification number, it embarks on the execution of an allocated portion of the search space in an attempt to find a solution. The next section describes in detail the control strategies used for partitioning the search space, and the means available for communicating the individual paths to each PE. For the moment it suffices to say that the PE "knows" its allocated path and blindly executes this without any communication with the other processing elements. Parallelization of this phase of the computation relies on an intelligent division of the search space, and the fact that multiple PEs may be active simultaneously.

The second phase of the computation rule defines the back-unification of partial solutions; a process which is also executed in parallel by several PEs. If the solution generated by a PE is only a partial solution, the PE requests further partial solutions from the Answer Server. If none are available it logs its answer with the Answer Server and becomes idle. On the other hand if a valid partial is found, a complete copy of this partial is returned to the PE, where a join takes place. The resulting partial (or complete) solution is returned to the Answer Server. This can result in further back-unification if more valid partial solutions are available; if none are found, or if the solution generated is complete, the PE becomes idle.

Both the and- and or-paths are used in determining whether two partial solutions are eligible for back-unification or not. The process of back-unification is described in detail in Section 6.

The order of generation of partial solutions by the PEs and their subsequent arrival at the Answer Server is non-deterministic. At no time is the system required to conform to the order of generation of solutions in a sequential system.

# 5 Vertical Partitioning of the Search Space

The simplest partitioning strategy, *automatic partitioning*, causes the proof tree to be split vertically so that each PE is ideally allocated a unique vertical slice of the search space. Once the PE reaches that part of the tree for which it is solely responsible, normal backtracking (including the creation of local choicepoint frames) takes place.

A potential disadvantage of automatic partitioning arises if the proof tree is not symmetric. If so, it can happen that one PE does a disproportionate amount of work while the other PEs are idle for the most part. A second strategy, *job reallocation*, allows reallocation of parts of the search space. Using this strategy, an idle PE may acquire further work after completing its initial job.

## 5.1 Automatic Partitioning

The underlying partitioning strategy used in the APPNet is an extension of the automatic partitioning strategy devised for DelPhi [Klein 1989]. The extension to this is concerned with adding support for and-parallelism, which is not supported by the DelPhi model.

Automatic partitioning requires only a single communication between a PE and the Command Server to initialise two parameters in each PE. On creation every PE is assigned a unique *identification number (PE-id)* which is a small integer. It also ascertains the *total number of PEs (All-PEs)* in the network. Using these two parameters the PE is capable of guiding its execution towards a unique portion of the search space. The algorithm for automatic partitioning, as described below, is similar to the algorithm given by El-Dessouki and Huen [1980] for a variation of a branch and bound search on distributed processors. Ali [1987] uses a similar partitioning technique to allocate parts of a proof tree to the available processing elements in his implementation of an or-parallel execution strategy.

### 5.1.1 And-parallelism with sequential backtracking

Consider the series of snapshots in the execution of a Prolog query illustrated by the proof trees given in Figures 5 through 7. Assume that the network is configured with four PEs and an and-parallel evaluation strategy is used.

Figure 5 shows the proof tree in its undivided state. As the four PEs log in, their PE-id's are set to 1,2,3 and 4 respectively. The value of All-PEs for each PE is 4. In each PE these two parameters are static and remain unchanged throughout the computation. They reflect the position of the PE with respect to the whole search tree.

All four PEs start working from the top of the tree. With the knowledge that four PEs are available

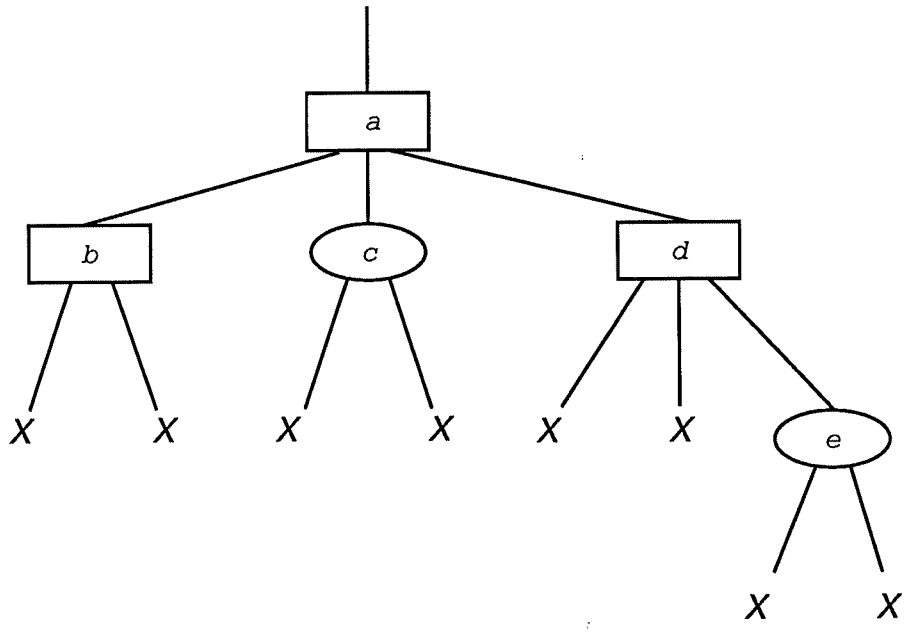


Figure 5: Proof tree used to illustrate automatic partitioning

in the network, PE1 on reaching the first and-branch choicepoint acquires only the first and-clause as part of its subtree. PE2 accepts the second and-branch and both PE3 and PE4 proceed by executing the third and-branch (Figure 6). How this division is accomplished is dependent on the values of the *dynamic identification (DYN-id)* and the *dynamic total (DYN-total)* in each PE. These variables are initialised to the values of PE-id and All-PEs respectively and are used to reflect the PEs position relative to a localised portion of the search tree. At any choicepoint, each PE is able to decide on its further path taking account of the values of DYN-id and DYN-total and with the knowledge of how many branches are available.

The *right-biased partitioning algorithm*, used in this example, has the consequence of shifting processing power to the rightmost nodes. Where the number of PEs is greater than the number of available branches, superfluous PEs are concentrated on the rightmost branch. Similarly where the number of available branches is greater than the number of PEs, any branches unaccounted for are allocated to the leftmost processor, thus cutting down the size of subtrees allocated to PEs with DYN-id's > 1. In the same vein, a left-biased algorithm can be described which favours the left of the search tree, while a no-bias algorithm attempts to assign the available processing power to the number of branches as fairly as possible.

Consider the position of PE1 (where the numbering convention for PEs is based on the PE-id value which remains unchanged for the life of the PE) at the first choicepoint. Three separate and-branches exist, and four processors (DYN-total = 4) are available. (This means that the fourth PE is in fact "spare" when division occurs at the first choicepoint). Using the right-biased allocation

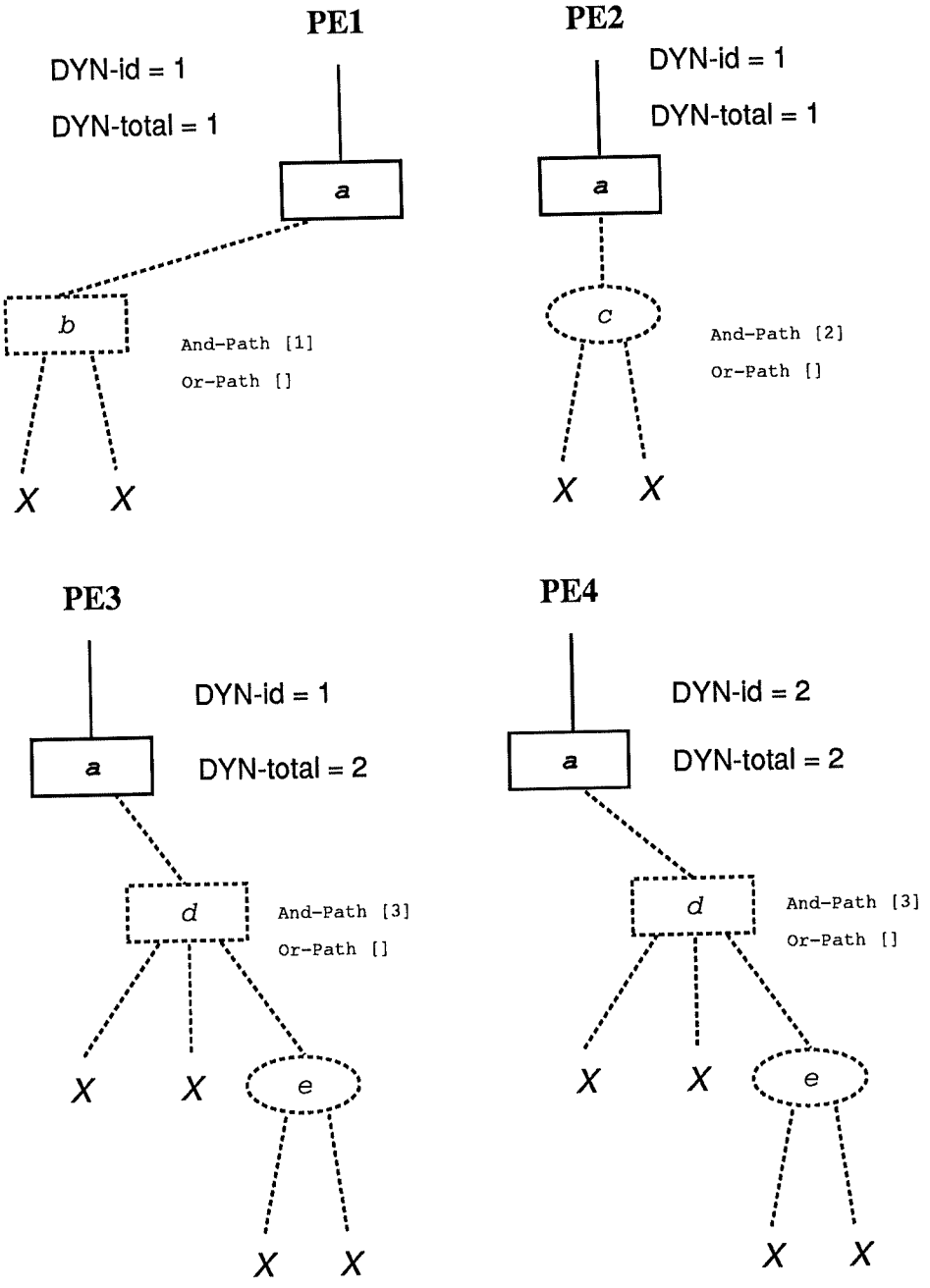


Figure 6: Right-bias partitioning with 4 PEs — stage 1

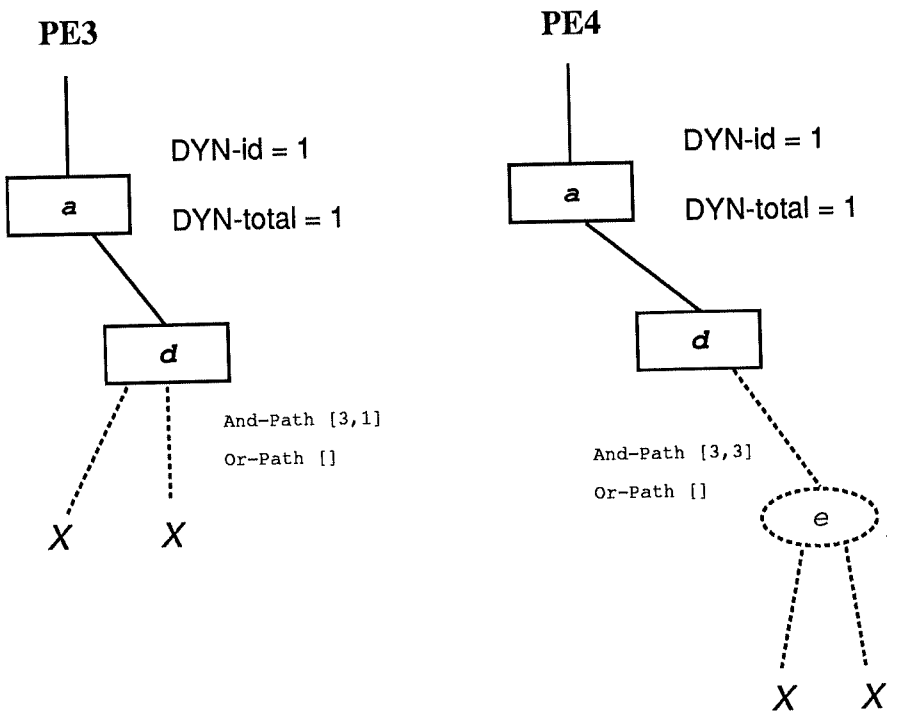


Figure 7: Right-bias partitioning with 4 PEs — stage 2

algorithm, PE1 acquires branch 1, corresponding to its DYN-id of 1. Similarly PE2 accepts the second and-branch. In each PE the DYN-id is reset to 1 and DYN-total becomes 1, since each PE is alone in executing its respective subtree. No further subdivision can result on either of these paths under the automatic partitioning scheme since DYN-total = 1 for both PEs. Hence, when PE1 reaches the second and-choicepoint (node *b*), both and-branches are retained and execution proceeds as in a sequential system. Likewise since no or-parallel strategy has yet been introduced, PE2 uses sequential backtracking to process the alternative clauses at node *c*.

When PE3 reaches the choicepoint at node *a*, the partitioning algorithm is applied and the PE acquires the third and-branch for execution. PE4 however, is also in contention for the same branch. Thus for PE3, the value of DYN-id = 1 and DYN-total = 2. Similarly for PE4, DYN-id = 2 and DYN-total = 2. Since both PE3 and PE4 have procured the same branch, further subdivision is possible at the next and-choicepoint.

This happens at node *d*, which has three and-branches. Now the number of PEs is smaller than the number of branches. Using the right-biased algorithm, PE3 (with DYN-id = 1) accepts the first two of these while PE4 (with DYN-id = 2) proceeds by executing the third and-branch. Both PEs are now solely responsible for a distinct portion of the search tree and consequently DYN-id = 1 and DYN-total = 1 for both. Further subdivision is not possible and the final subtrees traversed by PE3 and PE4 are shown in Figure 7.

The above illustrates how it is possible for each PE to be guided to its unique subtree, using a simple partitioning algorithm and two local variables, DYN-id and DYN-total. No further communication is necessary with the Command Server besides the initial "logging in" convention. Duplicate answers do not cause any problem as only those partial solutions found by PEs with DYN-id = 1 are recorded.

The left-biased and no-bias algorithms may also be used for partitioning the search tree. The *no-bias allocation scheme* is perhaps a more intuitive algorithm for partitioning the search space at and-choicepoints, since it assumes that all and-clauses require roughly the same amount of computation. Some slight bias is however, inevitable where the number of PEs is neither a multiple nor factor of the branching factor at the choicepoint.

Before the discussion turns to support for or-parallelism it should be mentioned that each PE keeps an unambiguous record of the path it has followed by generating a complete oracle during the computation. Thus at every and-choicepoint where some partitioning takes place, an extra bit is added to the and-path oracle reflecting the branch taken by the PE. The oracle for PE1 on completion of all computation and assuming the no-bias partitioning strategy is used would be [1,1] and that for PE2 [1,2], where the numbering of branches is based on a left-to-right decimal ordering. No or-path is included in the oracle since or-parallelism has not yet been considered in the partitioning strategy. The implementation details pertaining to the generation and use of oracles are explained in Section 8.

### 5.1.2 Combining and-parallelism, or-parallelism and sequential backtracking

In the examples presented in the previous section, alternative solutions were found by sequential backtracking. Support for full or-parallelism can, however, be instrumented in much the same way as that described for and-parallelism. Consider the proof tree given in Figure 5 and the first stage of partitioning that results when using the no-bias partitioning algorithm with five PEs (Figure 8).

Automatic partitioning may also be carried out at or-choicepoints, whenever there is an excess of PEs assigned to the same branch (*i.e.*, where DYN-total > 1). Thus in Figure 8, where a non-deterministic or-choicepoint exists at node *c* and DYN-total = 2, further partitioning can be done. Figure 9 shows the final subtrees allocated to PE3 and PE4 respectively when applying the partitioning algorithm at both non-deterministic and- and or-nodes.

In addition to keeping a record of the and-path in the oracle, it is also necessary to record the or-path when an and-or parallel execution strategy is used. In the example given in Figure 9, PE3 records an and-path of [2] and an or-path of [1], while the corresponding oracle for PE4 is [2][2].

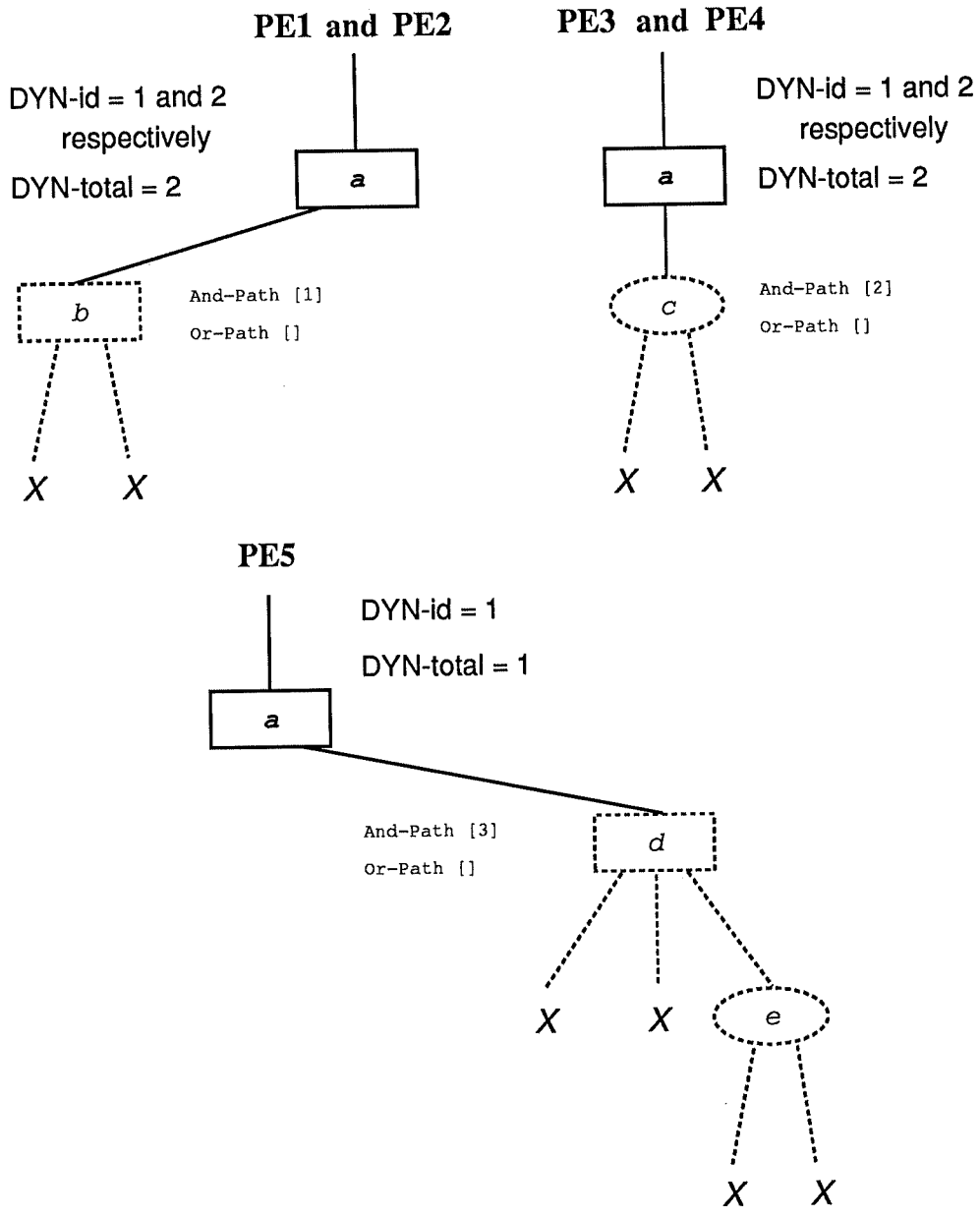


Figure 8: No-bias partitioning with 5 PEs — stage 1

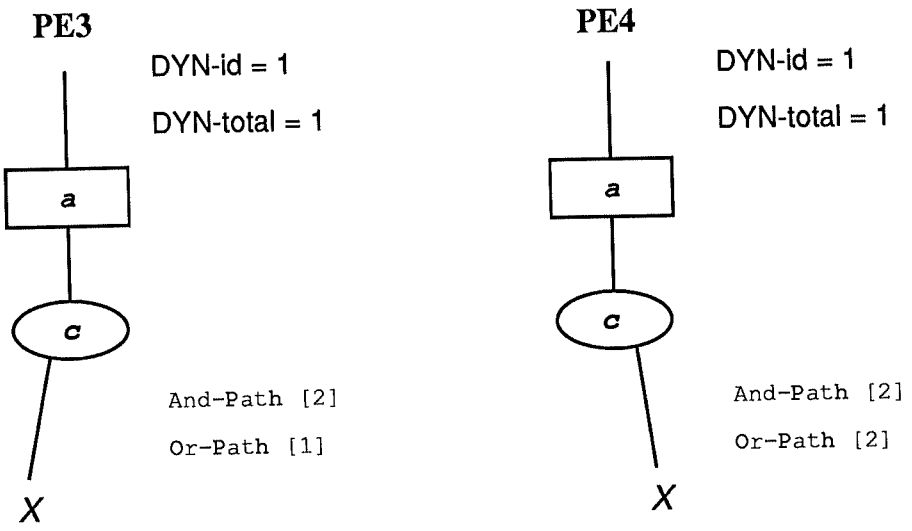


Figure 9: No-bias partitioning at and-nodes and or-nodes

Partial answers submitted by PE3 and PE4 cannot be mistaken for anything other than alternative solutions to the same and-branch since they share an identical and-path, while the oracles differ with respect to the or-path. The only overhead in supporting or-parallelism (over and above that for implementing and-parallelism) is thus in maintaining a local record of the current or-path.

No further partitioning can be carried out in the subtree allocated to PE5 (in Figure 8), and consequently the alternative clauses at node *e* are traversed by sequential backtracking. Ideally with sufficient processors a complete partitioning of the search tree would be possible, so that each PE processes only a single path from the root node to a terminal node.

The partitioning algorithms described for use in exploiting and-parallelism, namely right-bias, left-bias and no-bias, are equally applicable for partitioning at or-nodes. However the influence of each algorithm differs for different proof trees. The right-biased algorithm for or-parallel control gives a fairer division of the search space for tail-recursive programs. This is intuitive as the shape of the corresponding proof tree tends to be distended towards the right. With a right-biased allocation, processing power is concentrated on those clauses which are potentially recursive.

## 5.2 Job Reallocation

Automatic partitioning on its own causes a useful partitioning of the search tree. However the ability to reallocate a portion of the search space to an idle PE after it has evaluated an initial subtree is a worthwhile extension to the control strategy. Ideally this reallocation may be used to correct any unbalanced allocation of the workload.

Job reallocation uses automatic partitioning as before, but requires substantially more communi-

cation with the Command Server. At a given interval, each busy PE checks with the Command Server whether there are any other PEs which are idle. If so, the PE divides its subtree and cedes the newly created job (or jobs) to the idle processors.

The interval at which these *checkins* take place, is based on the evaluation of a pre-determined number of choicepoints. (The user can specify this number when selecting the reallocation strategy). In the extreme case where both the and- and or-checkin interval = 1, the PE communicates with the Command Server at every choicepoint. This strategy produces unwieldy communication overhead, without any improvement in efficiency. A checkin interval of at least 500 is more realistic, but still shows little or no improvement over the straight forward automatic partitioning strategy.

### 5.2.1 Supporting and-parallelism

Consider the proof tree given in Figure 5. With three PEs in the network and using the *no-bias automatic partitioning* algorithm, the initial division will create the subtrees shown in Figure 10, for PE1, PE2 and PE3 respectively. The oracles after processing node *a* for each PE are [1][], [2][] and [3][] respectively.

Given that PE1 completes the execution of its subtree before PE3 reaches node *d*, reallocation of part of the latter's job is possible. Assume that the *and-checkin interval* = 1 (the or-checkin interval has no meaning in the absence of support for or-parallelism). On reaching node *d*, PE3 ascertains from the Command Server that an idle PE exists and consequently division of its subtree takes place using the same partitioning algorithm as that used in the automatic partitioning. Figure 11 shows the new division of the subtree between PE3 and PE1, after altering the internal parameters.

The only communication needed to effect the reallocation involves sending the oracle describing the new job to the idle PE. This is done via the Command Server as no physical communication channels exist between the PEs.

The oracle of PE3 just before executing node *d* is [3][]. This is sent as the new job description to PE1. At the same time, the DYN-total of PE3 is reset to 2. Thus, on reaching node *d*, PE3 accepts the first two and-branches thereby extending its and-path to [3,1].

PE1 is reawakened with its DYN-total reset to 2 and its DYN-id = 2. The PE restarts execution by following the given oracle from the root of the proof tree. While it is tracing the path defined by the oracle, the PE executes in "*following mode*". This means that no subdivision can take place (even though DYN-total = 2) and the PE does not *checkin* with the Command Server (*i.e.*, the checkin facility is disabled). It is only once the given oracle has been exhausted that the PE reverts to *execution mode* and proceeds normally.

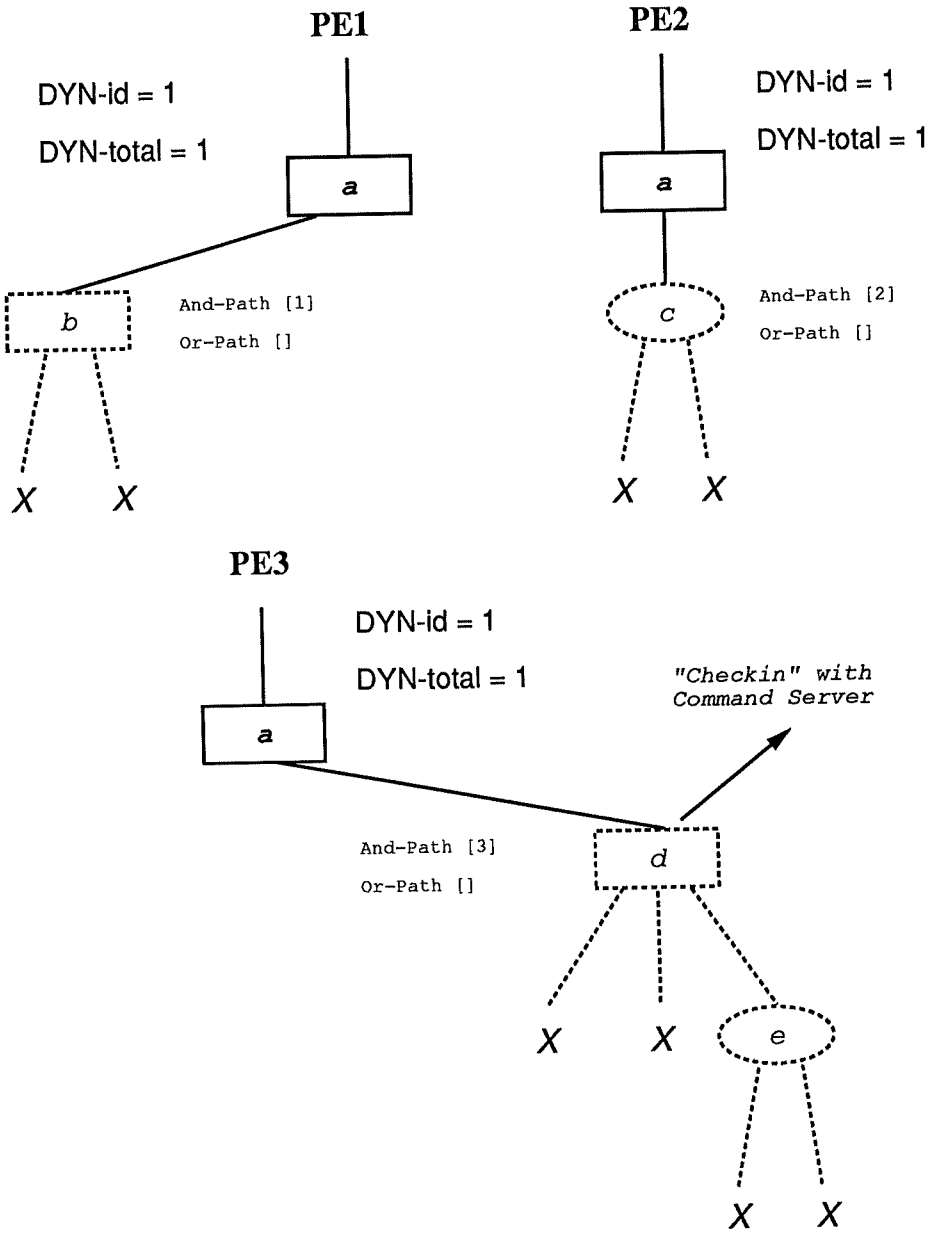


Figure 10: No-bias partitioning with job reallocation — stage 1

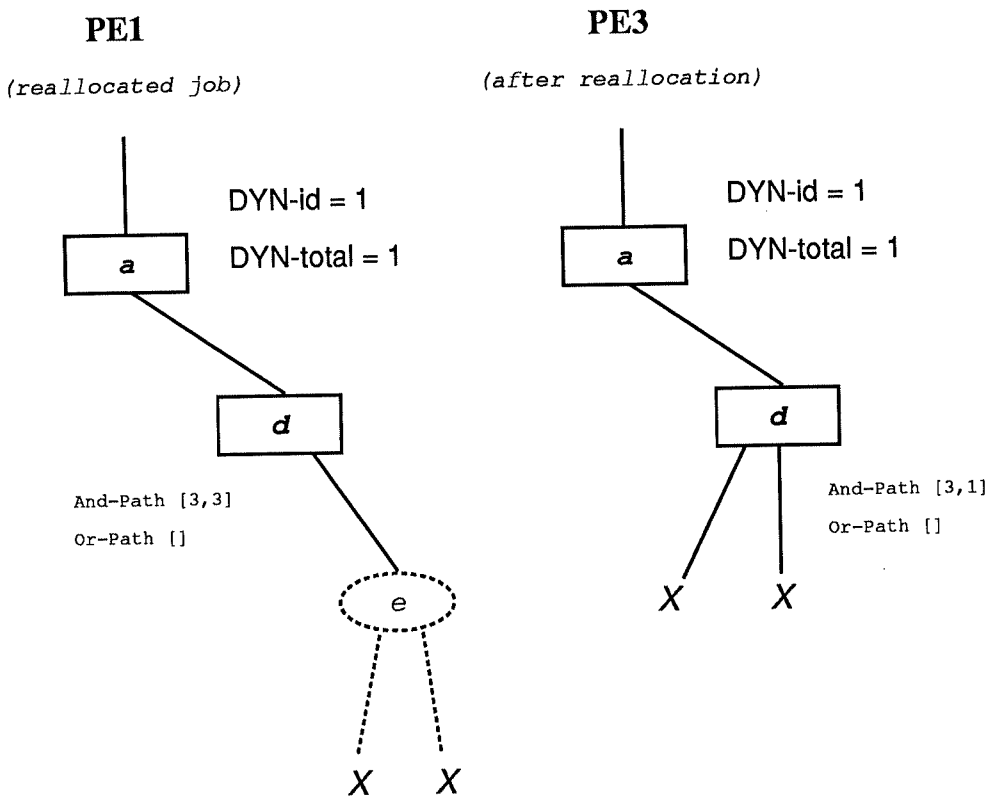


Figure 11: No-bias partitioning with job reallocation — stage 2

In this example, when PE1 reaches node *d*, no further oracle bits are available and thus partitioning takes place as before based on the DYN-id and DYN-total parameters. PE1 extends its oracle to be [3,3][] and the DYN-id = DYN-total = 1. The *checkin facility* is re-enabled and further job reallocation is possible given the existence of idle processors.

It is important to note that when a job is sent to the Command Server for reallocation, the complete oracle, including both the current and-path and the or-path, make up the job description. It is easy to see the downfall of this system if this were not so. Figure 12 shows the subtree allocated to PE2. On processing node *e*, with and-checkin interval = 1 and with PE1 idle, reallocation can take place. The and-path at this point is [2] and the or-path [1]. Both these oracles are sent to PE1 (via the Command Server) which ultimately executes the subtree given in Figure 12. No backtracking beyond the or-node *c* is attempted as a choicepoint was not established when PE1 was in *following mode*. PE2 on the other hand on completion of the first and-clause at node *e*, continues by backtracking through node *c*. The new execution path of PE2 is given in the second diagram in Figure 12.

If PE1 were not given the or-path oracle current at node *e*, it would not be in *following mode* with respect to or-branches by the time it reaches node *c*. (This is consistent with the definition

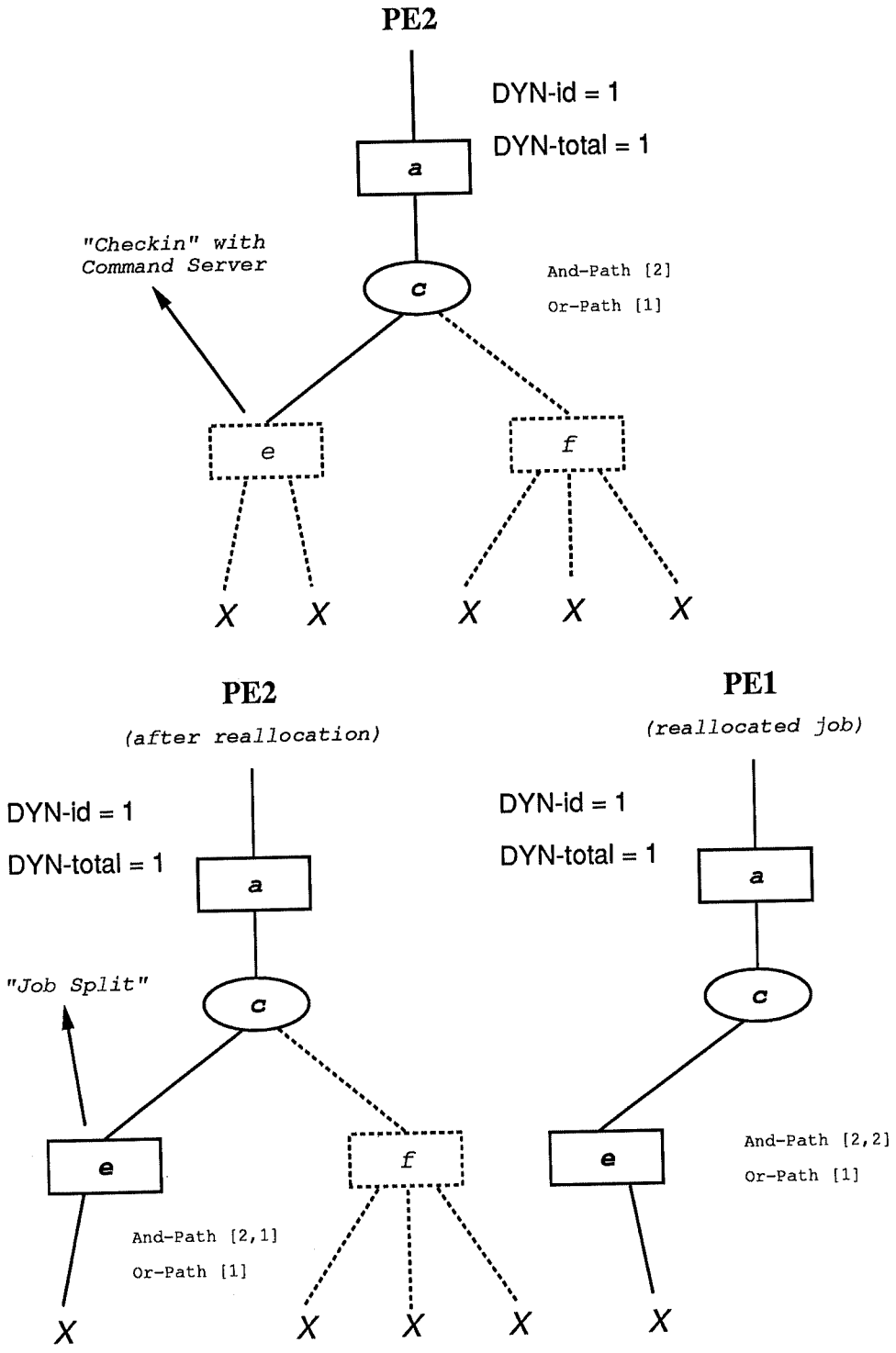


Figure 12: No-bias partitioning with job reallocation — all stages

of *following mode* which persists so long as there are still bits in the given oracle to specify the choice of branches at future choicepoints). As a result, a choicepoint would be established causing eventual backtracking to occur at node $c$, and consequently duplicating effort done by PE2 in executing the second or-branch without the possibility of reaching a portion of the subtree for which it would be solely responsible. In addition, duplicate answers would be reported for this part of the proof tree since both PEs have a DYN-id equal to one. Where the or-path is included in the job description, PE1 cannot backtrack through node $c$ and processes only the delineated subtree.

### 5.2.2 Checkin with and-or parallelism

In supporting an and-or parallel control strategy, job reallocation can also take place at an or-choicepoint if there are idle processors. This happens in much the same way as reallocating a job at an and-node. The *or-checkin interval* is used to determine how frequently communications are sent to the Command Server to ascertain the existence of idle processors. As when splitting a job at an and-node, both the or-path and and-path oracles current at the or-node where the split occurs are sent to the idle processors.

The reassign strategy enforces more overhead than the automatic allocation strategy not only with respect to the additional communication overhead, but also in that a complete record of both the and- and or-path needs to be maintained. In the automatic partitioning strategy, the and-or path is only recorded until DYN-total = 1, since after this no further subdivision of tasks can take place. This is not so when using the checkin strategy because subdivision of a job can occur anywhere in the subtree and hence the need for maintaining the complete and-or path history.

## 6 Back-Unification of Partial Solutions

The discussion so far has centered on the control necessary to divide the proof tree into vertical slices and on how these subtrees are subsequently allocated to available PEs. The fact that the partial solutions generated during this computation phase need to be combined into final solutions has been introduced, but without heeding the problems associated with multiple variable bindings and consistency. This section focuses on solutions to these problems and explores alternative strategies for the process of back-unification.

### 6.1 The Format of Partial Solutions

The form a partial solution takes resembles that used in a sequential Prolog system, where answers are reported as a collection of bindings to arguments in the goal clause. Given a top level query,

`map(A,B,C)`, and the Prolog program for `map/3` given in Section 4.2, a possible solution would be

```
A=blue, B=yellow, C=blue
```

Solutions generated in the APPNet are similarly represented in the form

```
(A=blue, B=yellow, C=blue).
```

For conciseness and to reduce the volume of terms to be back-unified, the representation is simplified to

```
(blue, yellow, blue)
```

and the position of each argument within the list becomes relevant.

Variables appearing as arguments in the goal clause, but absent from the and-branch on which the partial solution is generated may be uninstantiated when recording the partial solution. Here an underscore signifies an uninstantiated argument. The partial solution

```
(blue, yellow, _)
```

is the consequence of executing an and-branch in which the third argument to `map/3` is absent.

If partial solutions contain only bindings for those arguments appearing in the top level query, local variables used intra-procedurally cannot be taken into consideration in back-unification. Consider a Prolog version of the *quicksort* algorithm. (The code for this benchmark is given in full in the appendix).

```
quicksort([], Sol, Sol).
quicksort([First|Rest], Sol, Accum) :-
        split(Tail, First, Low, High),
        quicksort(Low, Sol, [First|Other]),      /*(2)*/
        quicksort(High, Other, Accum).           /*(3)*/
```

The subgoals (2) and (3) could realistically be executed in parallel once `Low` and `High` have been instantiated by `split/4`. However, if the convention of generating partial solutions consisting only of possible bindings for arguments in the head of the goal is applied, incomplete information is retained. Since the local variable `Other` is not represented in the head of the clause no account is taken of its binding during back-unification. It may therefore be necessary to record additional variable bindings in the partial solutions in order to produce correct back-unification.

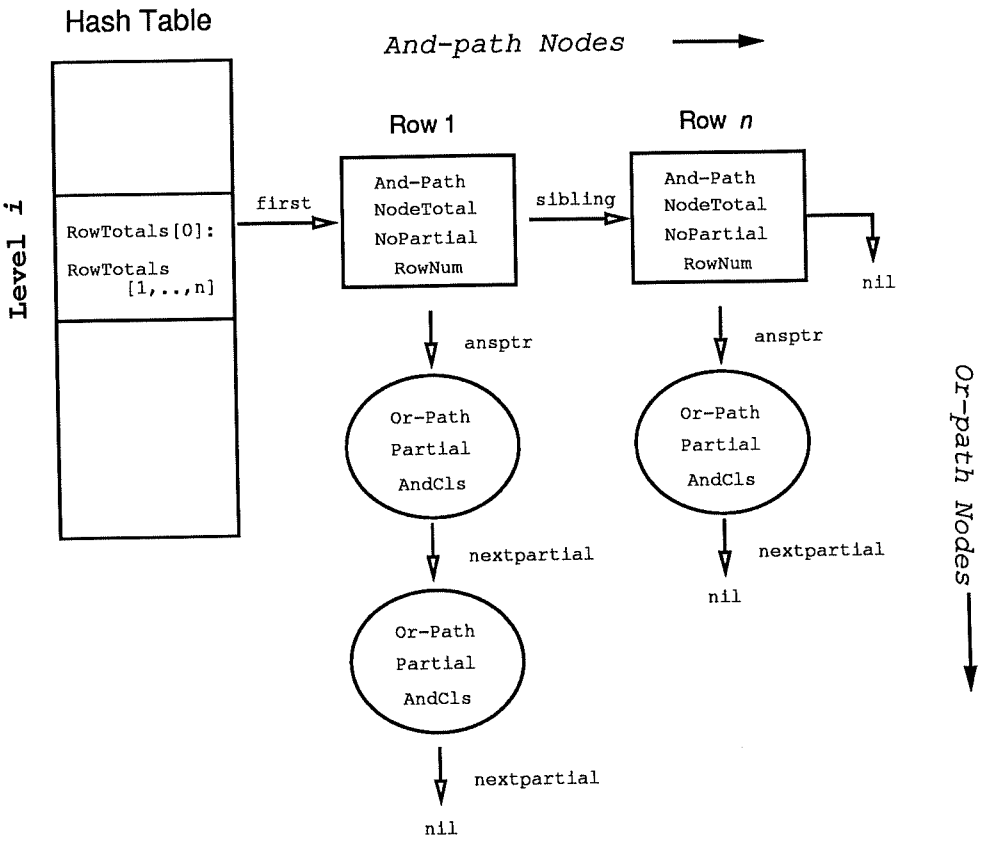


Figure 13: Definition of partials buffer

For the correct evaluation of `quicksort/3`, partial solutions should consist of bindings for `Sol`, `Accum`, `First` and `Other`. The order of the variables within the template must be the same whenever a partial solution is generated. Partial solution templates are derived during the preprocessing phase and are discussed further in Section 7.

## 6.2 The Answer Server and Partials Buffer

The Answer Server receives all solutions, whether partial or final solutions, from the PEs. Final solutions require no further processing other than being logged in the Answer Server's global log file for perusal by the user.

Partial solutions are accompanied by an oracle, denoting the exact and-or path executed by the PE in generating each partial solution. The Answer Server uses a hashed, dynamically linked data structure (the *partials buffer*, PB) to store the partial solutions. Hashing takes place on the length of the and-path oracle, which represents the depth of the search tree with respect to and-nodes (*i.e.*, the and-level) at which the partial solution was found. Figure 13 depicts the definition of the PB data structure diagrammatically.

Based on the and-level at which the partial solution was generated, an and-path node is inserted in the PB and the necessary changes are made to the `RowTotals` vector in the corresponding element of the hash table. `RowTotals[0]` records the number of and-path nodes inserted at that level, thereby reflecting the number of different and-paths of a given length for which partial solutions have been received. The remaining entries, `RowTotals[1] to RowTotals[`$n$`]` (with $n < 500$), contain a record of the number of or-path nodes inserted at each and-path node in rows 1 to $n$ respectively. At each level in the structure, all partial solutions are inserted in a strictly FIFO sequence and no re-ordering of nodes takes place. Once inserted a partial solution is never physically removed, and consequently the storage structure never shrinks.

Information contained in each and-path node includes the current and-path oracle (`And-Path`), the row number for this node (`RowNum`), a pointer (`ansptr`) to the list of or-path nodes depicting the alternative partial solutions with the particular and-path, and a count of the number of these or-path nodes (`NodeTotal`). And-path nodes are dynamically linked via a pointer (`sibling`).

The `NoPartial` field is true if a `[NO]` partial is generated with the current and-path. This happens if one of the distributed and-subgoals fails. The or-path attached to a `[NO]` partial is also recorded. The `NoPartial` field allows back-unification to be short-circuited if a failed subgoal has been detected. In the absence of alternative "real" partial solutions, no back-unification needs to be carried out. The `[NO]` is simply propagated to the next level in the partials buffer with the appropriate adjustments to the attached oracle.

Each or-path node contains the actual partial solution (`Partial`), stored as an array of bytes in the format described earlier, the corresponding or-path oracle (`Or-Path`) and a record of the number of and-subgoals executed to obtain the partial solutions represented by this node (`AndCls`). A pointer (`nextpartial`) links the or-path nodes.

## 6.3 Valid Path Oracles in Back-Unification

The *raw* partial solutions generated during the execution of the *map colouring* program (given in Section 4.2) are shown as appropriate nodes in the PB illustrated in Figure 14. No back-unification has taken place. It is assumed that ten PEs are used to evaluate the problem.

On back-unification at level 2 in the PB, the cross product of the solution sets for each and-path node is computed, taking into account valid path oracles. In calculating the cross product, all possible pairings of the partial solutions are considered. Two partial solutions may be back-unified if their and-path oracles differ only with respect to the final element (or bit). Without considering the validity of the or-paths of the partial solutions, the possible pairings in this example are given below. Partial solutions conform to the template `(A,B,C)` created in response to the query `map(A,B,C)`. Each partial is followed by the relevant oracle, consisting of the and-path and or-

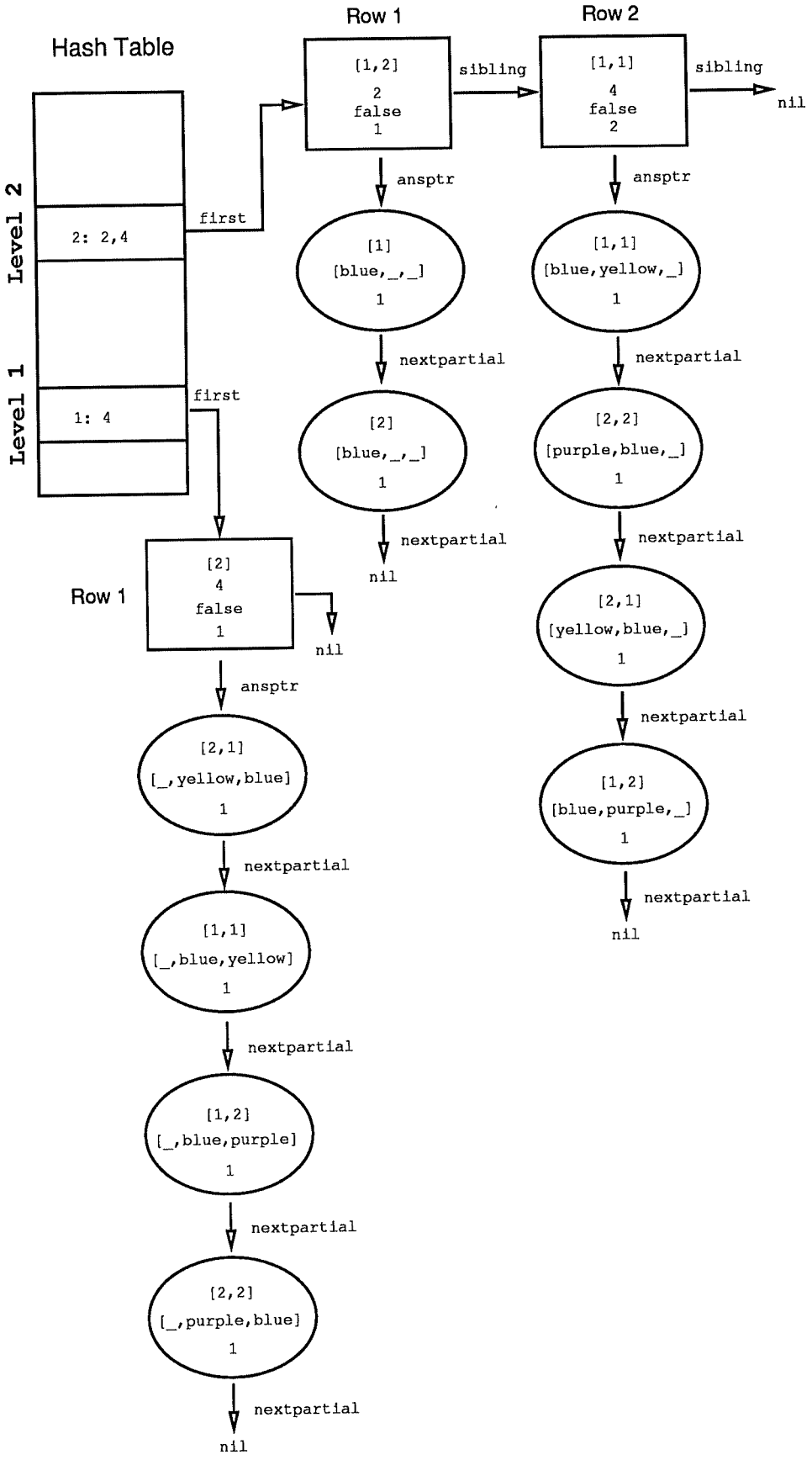


Figure 14: Partials buffer for *Map Colouring* — stage 1

path respectively. This convention is used in all future references to partial solutions.

1. `(blue, yellow, _)` [1,1] [1,1] and `(blue, _, _)` [1,2] [1]
2. `(blue, yellow, _)` [1,1] [1,1] and `(blue, _, _)` [1,2] [2]
3. `(blue, purple, _)` [1,1] [1,2] and `(blue, _, _)` [1,2] [1]
4. `(blue, purple, _)` [1,1] [1,2] and `(blue, _, _)` [1,2] [2]
5. `(yellow, blue, _)` [1,1] [2,1] and `(blue, _, _)` [1,2] [1]
6. `(yellow, blue, _)` [1,1] [2,1] and `(blue, _, _)` [1,2] [2]
7. `(purple, blue, _)` [1,1] [2,2] and `(blue, _, _)` [1,2] [1]
8. `(purple, blue, _)` [1,1] [2,2] and `(blue, _, _)` [1,2] [2]

From Figure 3 however, it is apparent that or-path conflicts arise in attempting to join the partial solutions in each of the pairs 2, 4, 5 and 7 above. For example in pair 2 above, the first solution has an or-path [1,1] (*i.e.*, it has been generated on the leftmost or-branch). The second solution in the pair has an or-path of [2], since it is a solution to the second branch of the first or-node. When the split occurs at the and-nodes at level 2 in Figure 3, one or-node has already been executed. Thus all partial solutions to be back-unified at level 2 must agree on the first element (or bit) of their respective or-paths. The *ormatch* variable associated with each and-node is used to record the number of or-nodes executed before the and-node is reached (and as such reflects the length of the or-path current at that and-node). At level 2 *ormatch* is thus equal to one. When resolving partial solutions, the value of *ormatch* is used as the measure of validity of an or-path oracle.

The or-paths in the participating partial solutions in each of pairs 2,4,5 and 7 do not agree on the minimum subset demanded by the *ormatch* variable at and-level 2. In resolving the or-path conflicts, it is apparent that only four legal join operations should be carried out at level 2. Back-unification of the partial solutions in pairs 1 and 3 results in the creation of two new partial solutions

`(blue, yellow, _)` [1] [1]
`(blue, purple, _)` [1] [1]

The and-path for the partial solutions formed during the join operation is one bit shorter than the original and-paths for the participating partial solutions. This is analogous to moving one and-node closer towards the root of the search tree. Consequently, the new partial solutions may be inserted at a lower level of the partials buffer. This results in a new and-level 1 as depicted in Figure 15. The or-path for the new partial solutions is taken to be the first $n$ bits of the original or-paths, where $n$ = *ormatch*.

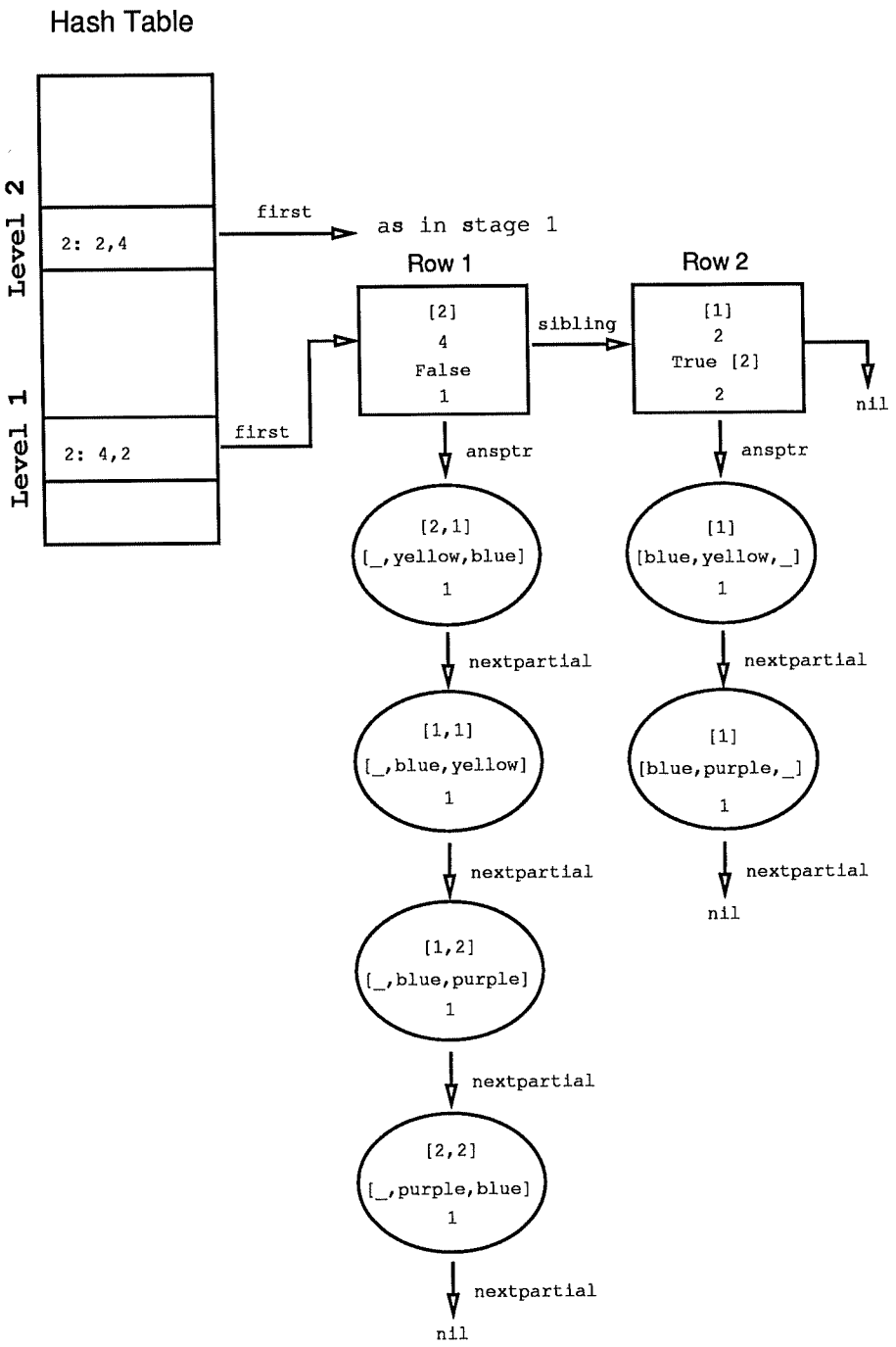


Figure 15: Partials buffer for *Map Colouring* — stage 2

Back-unification is now possible at level 1. The *ormatch* parameter is zero, since at the time the split occurred (*i.e.*, at the and-node at level 1 in Figure 3), the or-path was empty. Consequently the or-path can be disregarded in determining valid partial solutions, and a full cross product is performed on the solution sets for and-paths [1] and [2] giving the two final solutions

```
(blue, yellow, blue)
(blue, purple, blue).
```

The importance of the or-path oracle in conjunction with the *ormatch* parameter is further highlighted by the following example for which a proof tree is given in Figure 16.

```
doit(Num1, Num2)    :- one(Num1), two(Num2).
doit(Alph1, Alph2) :- a(Alph1), b(Alph2).
one(1).
two(2).
a(a).
b(b).
```

The goal, `doit(NumAlph1, NumAlph2)`, when executed in a sequential system would produce the results

```
NumAlph1 = 1, NumAlph2 = 2;
NumAlph1 = a, NumAlph2 = b.
```

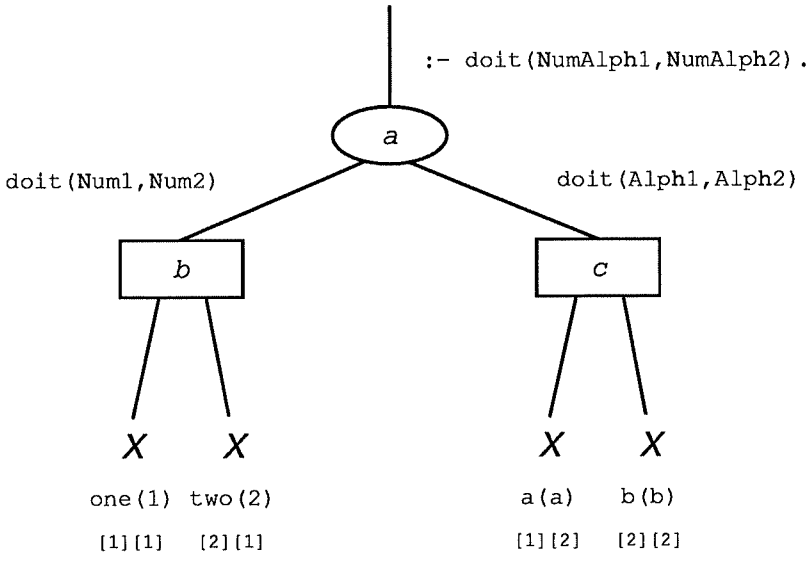


Figure 16: Proof tree for `doit/2`

In the APPNet, four partial solutions are produced:

1. `(1, _)` [1] [1]
2. `(a, _)` [1] [2]
3. `(_, 2)` [2] [1]
4. `(_, b)` [2] [2]

With reference to the and-or tree in Figure 16, it is clear that by the time either of the and-nodes (node *b* or node *c*) is executed, the or-path oracle has a length of 1 and therefore *ormatch* = 1. On back-unification, the first bit of the or-oracle must match in all partial solutions participating in the back-unification process. This clearly rejects the possibility of incorrectly joining partial solutions 1 and 4 (or 2 and 3) even though their and-paths are compatible.

## 6.4 Back-Unification Strategies

Two alternative strategies have been considered for the back-unification of partial solutions. The first method employs total back-unification, while the second allows the back-unification process to be done incrementally.

### 6.4.1 Total level-based back-unification

*Total back-unification* means that no partial solutions from a particular level of the partials buffer may be allocated for back-unification unless the resulting join will be *complete* at that level. For example, if an and-node contains four and-subgoals, each of which is processed by a separate PE, thus generating four partial solutions, no back-unification takes place until all four partial solutions have been submitted to the Answer Server. Only one PE is then involved in the back-unification process at a single level.

Consider the and-or proof tree given in Figure 17. The convention established in Section 5 is upheld, so that only non-deterministic nodes are depicted in the proof tree. Assume that seven PEs are used in the evaluation and hence each of the possible partial solutions is generated by a different PE. A possible order of generation of partial solutions is given by the enumeration of the leaf nodes, where partial_1 is the first partial solution to be generated and partial_7 the last.

Since partial_1 is the first to be recorded by the Answer Server, no back-unification is possible at this stage. The resulting partials buffer is represented in Figure 18.

No back-unification is possible with the generation of partial_2 either, since the or-paths conflict and there are not enough partial answers to effect a complete join. The same holds true with the arrival of partial_3. The revised PB data structure is given in Figure 19.

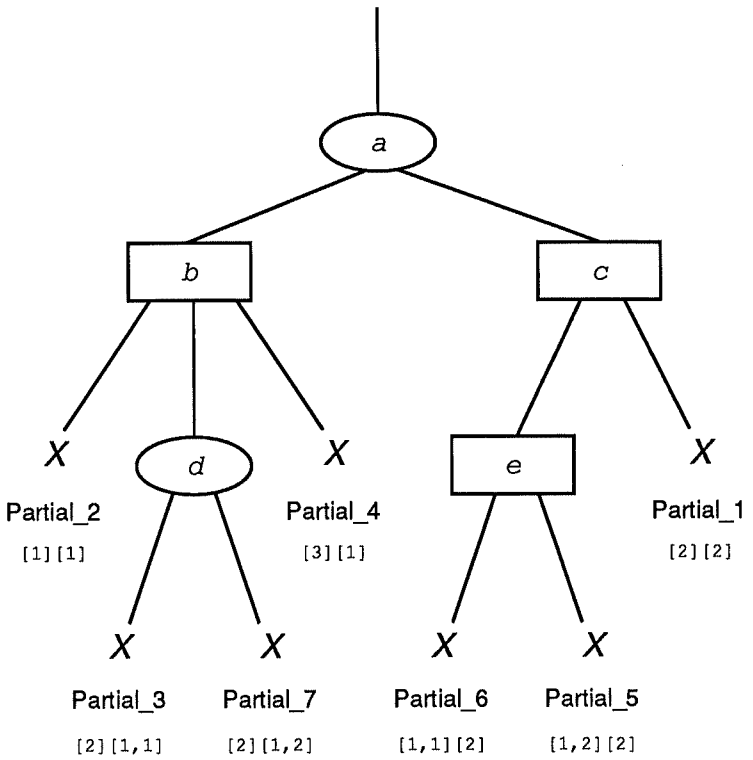


Figure 17: Proof tree used to illustrate total back-unification

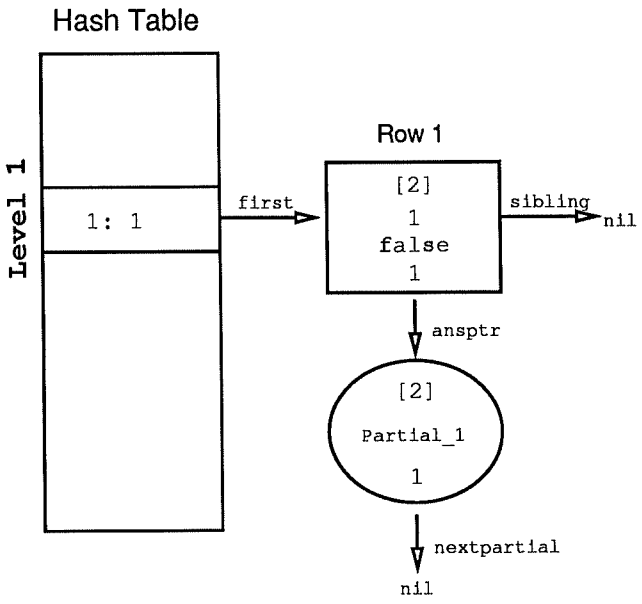


Figure 18: Partials buffer — stage 1 in total back-unification

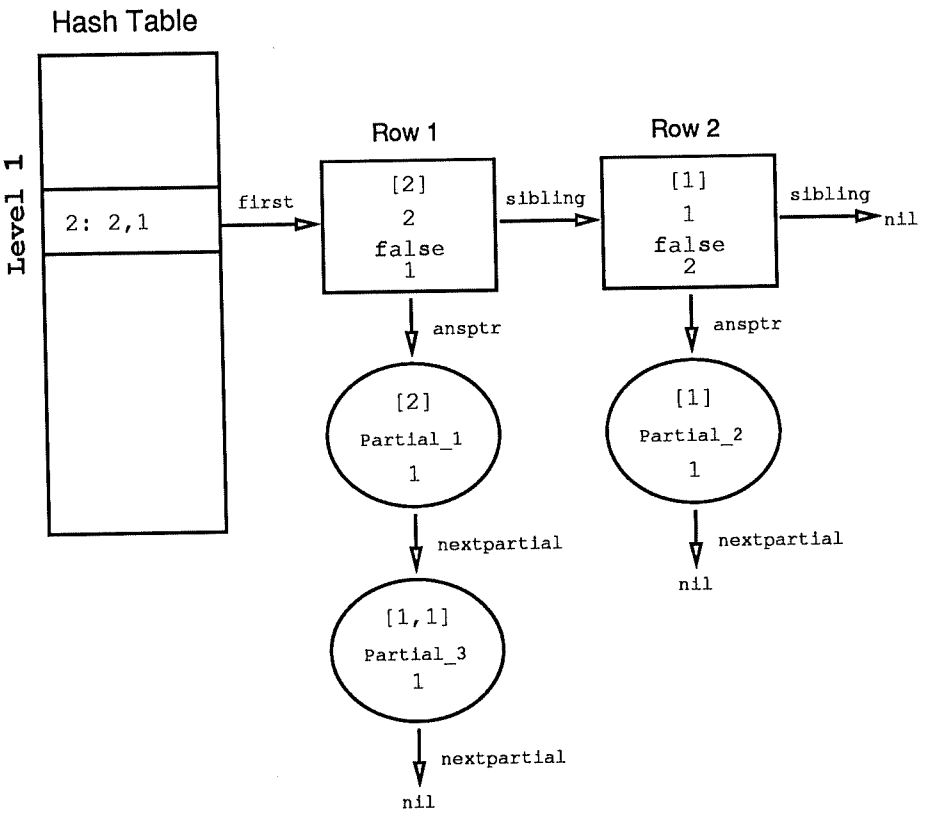


Figure 19: Partials buffer — stage 2 in total back-unification

The generation of partial_4 initiates the back-unification process. A total join of partial_2, partial_3 and partial_4 is performed, creating a complete final answer. The Prolog implementation retains information about the and-branching factor at every distributed and-node. Thus to determine whether a total join is possible, this information is merely compared to the corresponding value of `AndCls`.

The structure of the PB ensures that the potential candidates for back-unification can be readily identified. The partial solutions to be joined with partial_*i* are chosen as follows:

> Let *r* be the row and *j* be the and-level in the PB at which partial_*i* is inserted.
>
> Select all valid or-path nodes from each of the and-path nodes (except the node at row *r*) at level *j*. An or-path node is valid if its or-path oracle corresponds to the or-path of partial_*i* with respect to the first *n* bits, where *n* is equal to the value of *ormatch* at the relevant and-node.

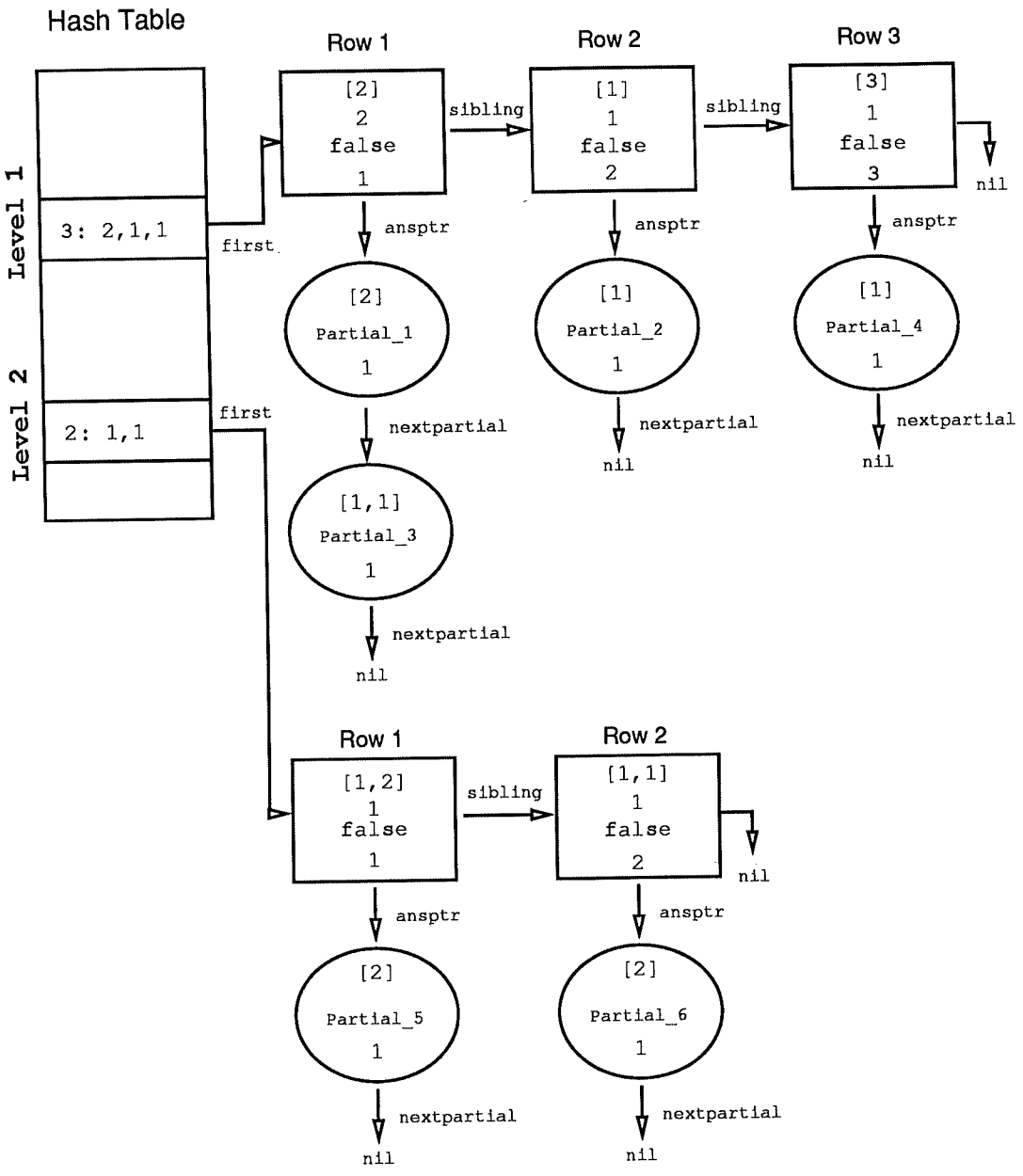


Figure 20: Partials buffer — stage 3 in total back-unification

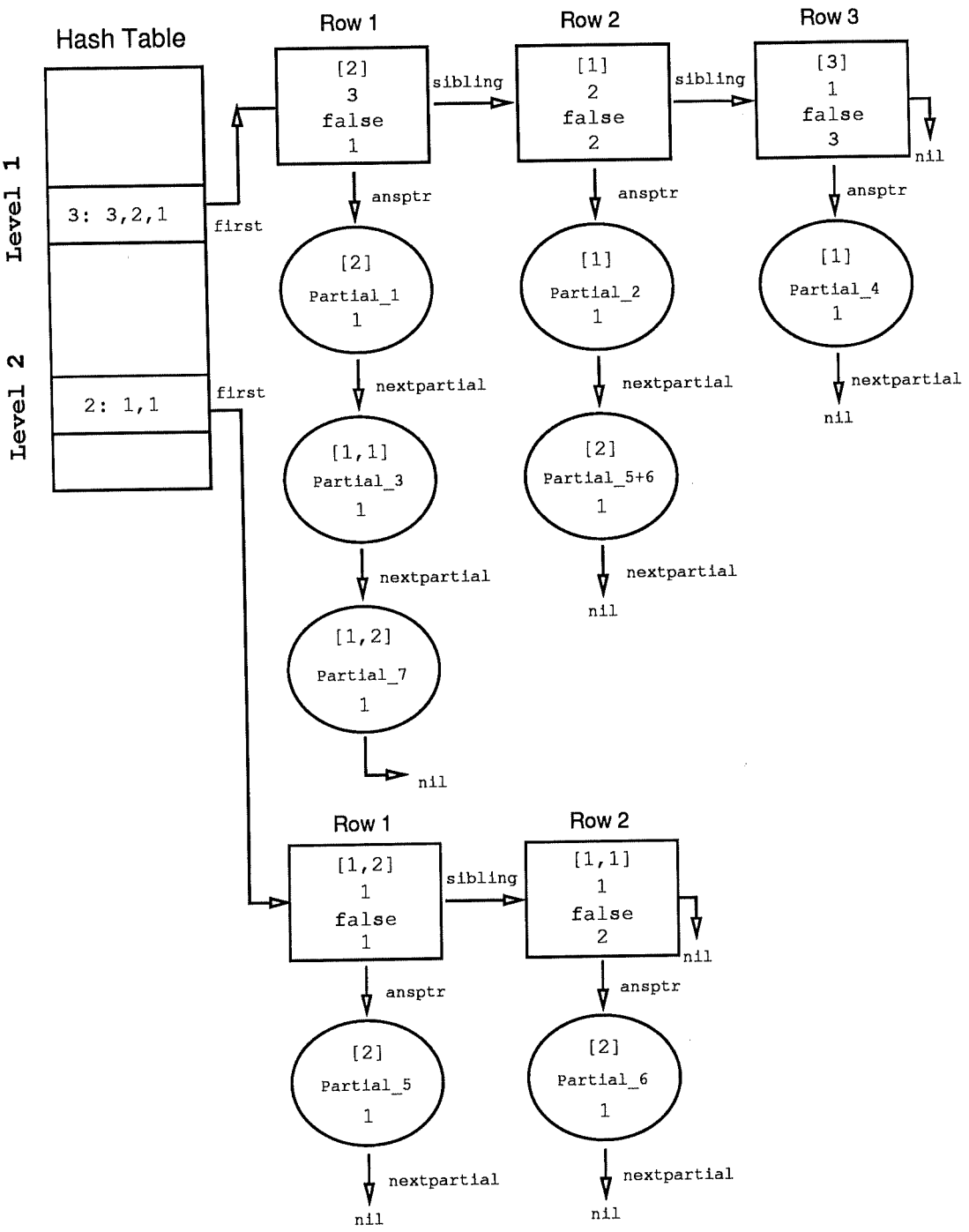


Figure 21: Partials buffer — stage 4 in total back-unification

Partial_5 with and-path [2,2], is inserted at and-level 2 in the PB and fails to bring about back-unification. With the arrival of partial_6 however, a total join at level 2 is possible. This results in the generation of a new partial, partial_5+6 with and-path [1] and or-path [2]. Before this partial solution is received by the Answer Server, the partials buffer contains the nodes shown in Figure 20.

The addition of partial_5+6 at level 1 initiates further back-unification. Partial_5+6 and partial_1 are joined to produce a second final answer. Partial_7 is the last to be generated leaving the partials buffer in a state as shown in Figure 21. A third final solution is created by joining partial_2, partial_4 and partial_7. Due to conflicting or-paths, partial_5 and partial_6 cannot be considered in this join. Partial_1 and partial_3 are ignored as they have the same and-path as partial_7 and are therefore alternative solutions to the same and-branch. The consistent use of the oracles coupled to the partial solutions is thus crucial to the success of the back-unification process.

Advantages of total back-unification include executing only a single join operation for every parallel and-branch, and a simpler storage structure for partial answers since partially combined results are never stored. An inherent disadvantage is that the final join may involve a large number of partial solutions (depending on the number of subgoals in the top level procedure), causing the overhead to be fairly substantial. Considering that the final answer to any query can only be produced once all partial solutions have been generated, the final join takes place only after all the actual computation has been done. This can increase the overall parallel time of the system.

### 6.4.2 Incremental back-unification

Although this strategy is similar to the incremental join used in the ROPM, the approach taken in the implementation thereof is notably different. When a PE finds a partial answer it submits a request to the Answer Server for any corresponding partial solutions. If any exist, irrespective of whether they are sufficient to effect a complete join, they are passed back to the PE for back-unification. Thus the criteria for allocating partial solutions to be back-unified is solely dependent on finding partial solutions with matching oracles.

Allocation of partial solutions is done from a single level of the partials buffer at a time. The length of the and-path of the newly computed partial is used as an index into the buffer to find the level containing potential partial solutions for back-unification. Any and-path node (represented by rectangular boxes in the PB diagrams) at this level, except the one with an and-path identical to that of the partial, may be considered for back-unification.

The incremental process of back-unification is more easily illustrated by means of an example. Consider the and-or proof tree shown in Figure 22, with the order of generation of the partial solutions specified by the labels at the leaf nodes. The APPNet system does not rely on any partic-

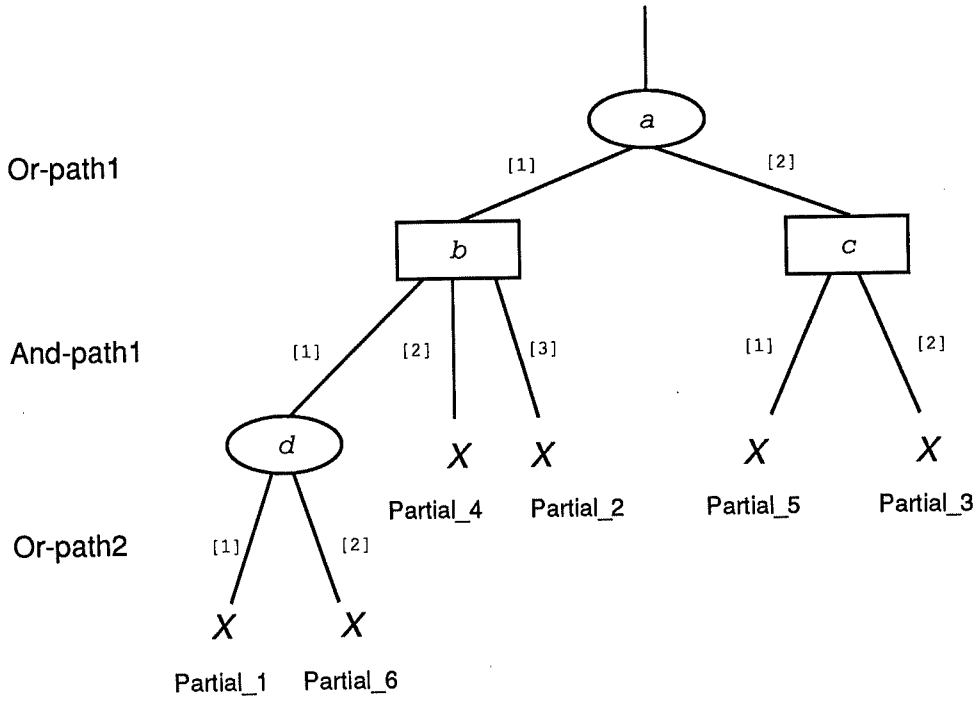


Figure 22: Proof tree used to illustrate incremental back-unification

ular ordering of the partial solutions; for ease of explanation an arbitrary order has, however, been specified.

With the arrival of partial_1, no back-unification is possible and the partial solution is stored as before by the Answer Server in the partials buffer at the and-level 1 (Figure 23). A more complex data structure is however used to represent a partial solution, since it is now possible for each or-path node to contain several *partly-joined* partial solutions. A new field, *rowsjoined*, is introduced to record which partial solutions have contributed to a partly-joined solution by noting the row numbers of all partials involved in the join. This field is represented by a long integer with each binary digit corresponding to a row in the PB. If a bit is set, it means that a partial solution from the corresponding row in the PB has been joined to form the new partly-joined solution. In all subsequent figures, the *rowsjoined* variable is shown as a binary number, with the least significant bit on the right. The remaining fields in the and-path and or-path nodes are used in the same way as in total back-unification.

With the arrival of partial_2, with and-path [3] and or-path [1], the back-unification process is started. The Answer Server stores partial_2 at and-level 1, and allocates partial_1 for back-unification. Included in the message packet sent to the PE is an *allocation mask* which describes how many partial solutions from each row in the PB have been allocated for back-unification. The mask for the current allocation is {1,0}. This signifies that one partial solution from the and-path node at row 1 and zero solutions from row 2 have been demarcated for back-unification. No solutions are

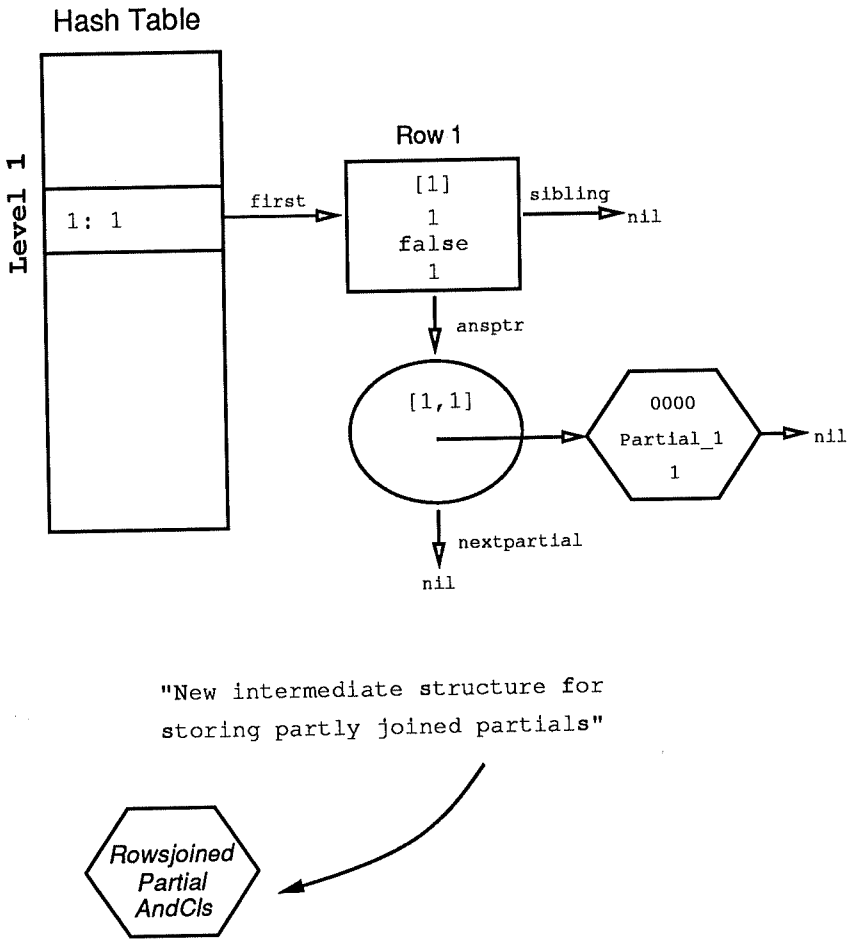


Figure 23: Partials buffer — stage 1 in incremental back-unification

allocated from row 2, since the only solution recorded in this row is partial_2, which is already known to the PE. The allocation mask is used in the join operation to ensure that a full join is performed on the allocated partial solutions. No two partial solutions from the same row in the partials buffer may be joined, since they are alternative solutions to the same and-branch. The information in the allocation mask is also used to set the appropriate bits in the *rowsjoined* field of the newly created partly-joined solution.

The back-unification process is not complete since the branching factor at the relevant and-node is 3, and only two branches have been accounted for. A new partly-joined answer, partial_1+2, is created and this is reinserted into the PB once more in row 2 at level 1. Row 2 is chosen since this is where the original partial which initiated the back-unification process (that is, partial_2) was inserted. The resulting PB is depicted in Figure 24. The least significant bit of the *rowsjoined* field is set, signifying the back-unification with a partial solution from row 1.

No back-unification is initiated when partial_3 is received at the Answer Server as there are no

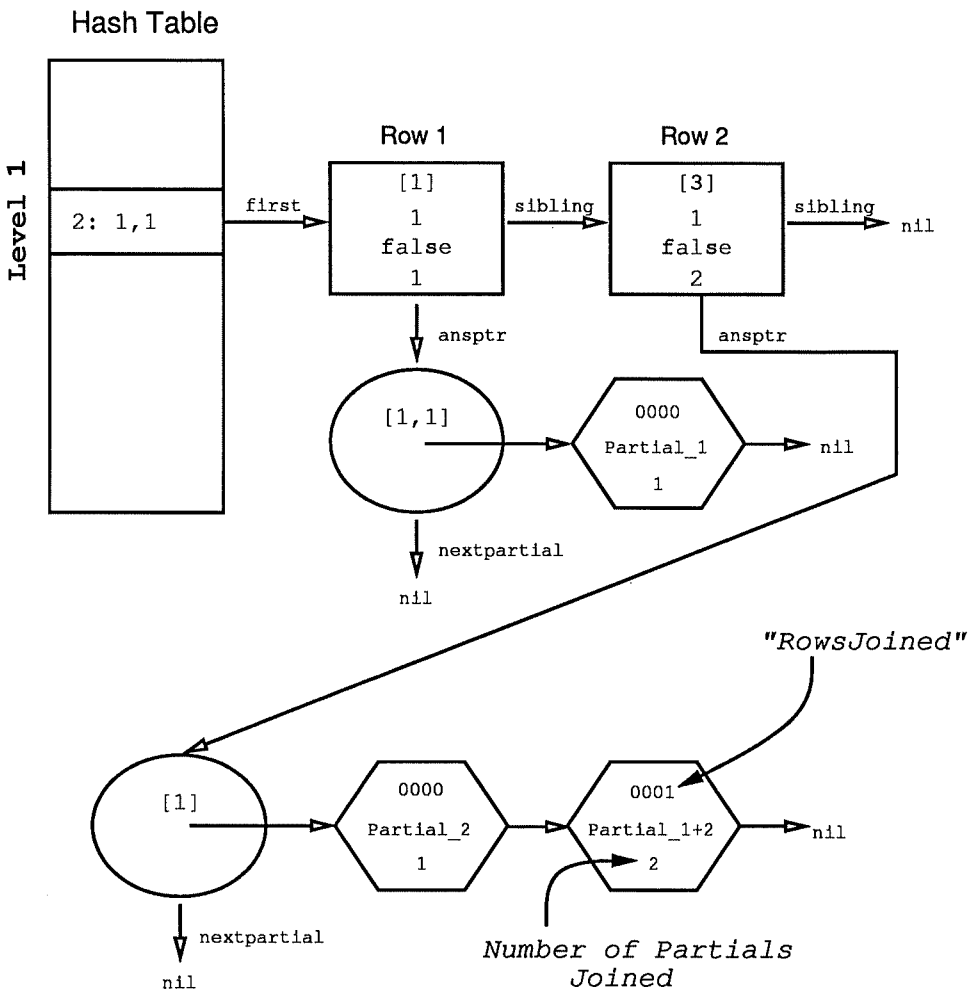


Figure 24: Partials buffer — stage 2 in incremental back-unification

partial solutions with valid or-path oracles. A new and-path node to contain partial_3 is, however, created.

On finding partial_4, with and-path [2] and or-path [1], back-unification is possible between partial_4, partial_1 and partial_2. Figure 25 shows the partials buffer after inserting partial_4 in row 3 at and-level 1.

The algorithm used for choosing partial solutions for incremental back-unification follows. In this algorithm, partly-joined solutions take preference over *raw* partial solutions, *i.e.*, those partial solutions which have not undergone any back-unification.

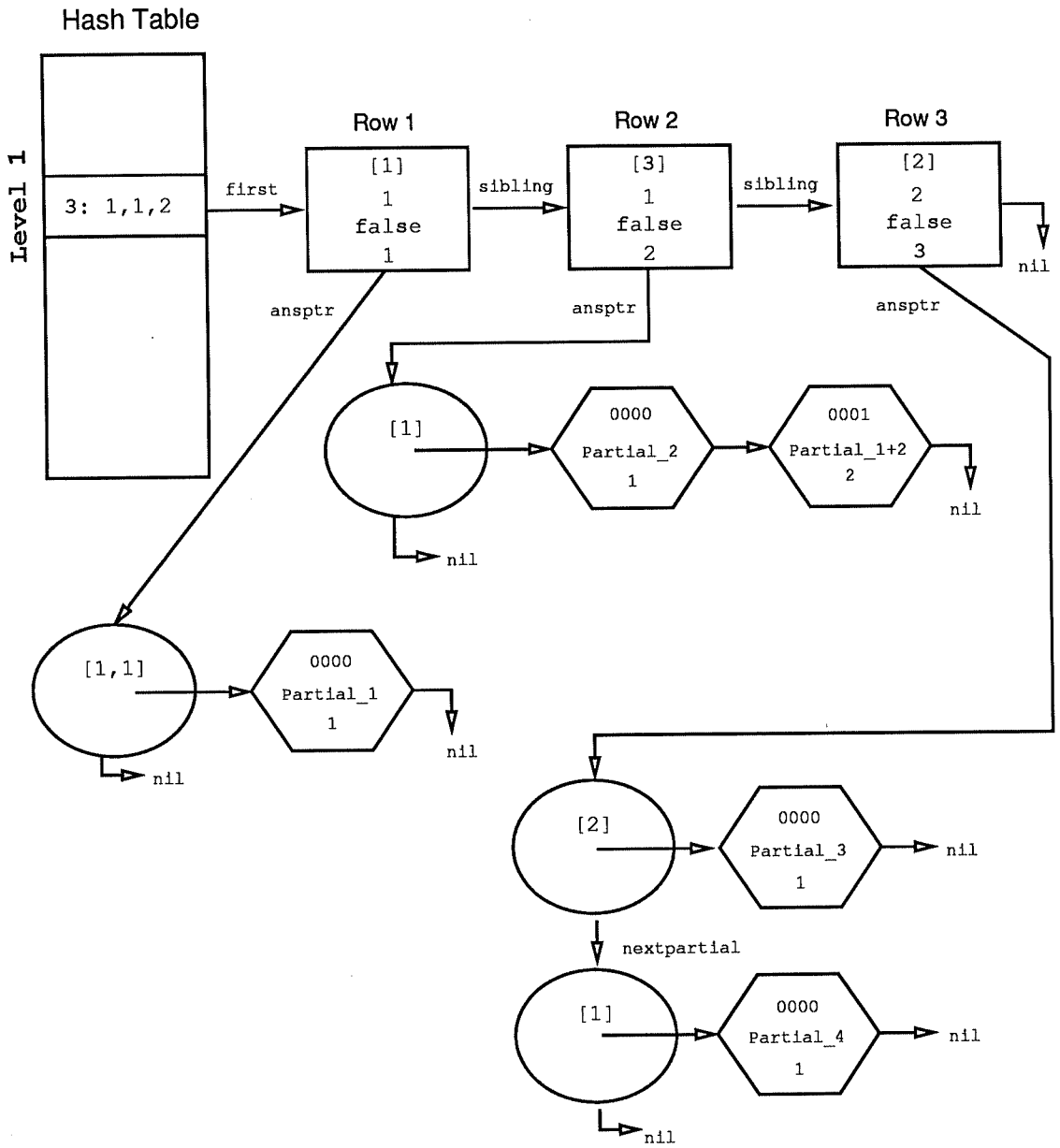


Figure 25: Partials buffer — stage 3 in incremental back-unification

Let $j$ be the level and $i$ be the row in which the newly found partial (partial_4 in the current example) has been inserted.

For each row $r$ at level $j$, where $r \neq i$, iterate

*total[r]* = 0

For each or-node $p$ in row $r$, with a valid or-path, iterate

Select all partly-joined solutions which have not been combined with any partial solutions from row $i$. This can be detected by checking the $i$th bit in the *rowsjoined* field of each partly-joined solution. If the bit is set, the partial is ignored; otherwise *total[r]* = *total[r]* + 1 and the partial is allocated.

If no valid partly-joined solutions are found, allocate the raw partial from or-node $p$ and increment *total[r]*.

The $r$th element of the allocation mask is set to the value of *total[r]*.

According to this algorithm, partial_1+2 and partial_1 are allocated for back-unification with partial_4. The allocation mask is {1,1}. One partly-joined partial solution is formed during this back-unification, partial_1+4, formed by combining partial_1 and partial_4. The back-unification of partial_4 and partial_1+2 is complete and partial_1+2+4 may be reinserted into the PB with a modified and-path, [], at and-level 0. As the and-path is empty, no further back-unification can take place and partial_1+2+4 is in fact a final solution to the query. As such it is not re-inserted into the PB.

With the arrival of partial_5 (and-path [1] and or-path [2]) the PB structure resembles that given in Figure 26. Scrutiny of the buffer reveals that the only valid partial solution for back-unification with partial_5 is partial_3. The allocation mask is {0,0,1}. The resulting join is once again complete, since there are no further levels at which back-unification may take place. Partial_3+5 is therefore another final solution to the query.

Partial_6 may be joined with partial_2 and partial_4, giving a third complete solution and two partly-joined solutions. Partial_1+2 cannot be considered for back-unification since it contains partial_1, which comes from the same row as partial_6. Therefore the raw solution, partial_2, is chosen instead. The same applies for the allocation from row 3, where partial_4 is chosen instead of partial_1+4, which also contains a partial from row 1. The partly-joined solutions, partial_2+6 and partial_4+6, formed during this back-unification are inserted into the PB in row 1.

The back-unification process is now complete. No further *raw* partial solutions are generated and thus all final solutions have been computed. Figure 27 shows the final state of the PB.

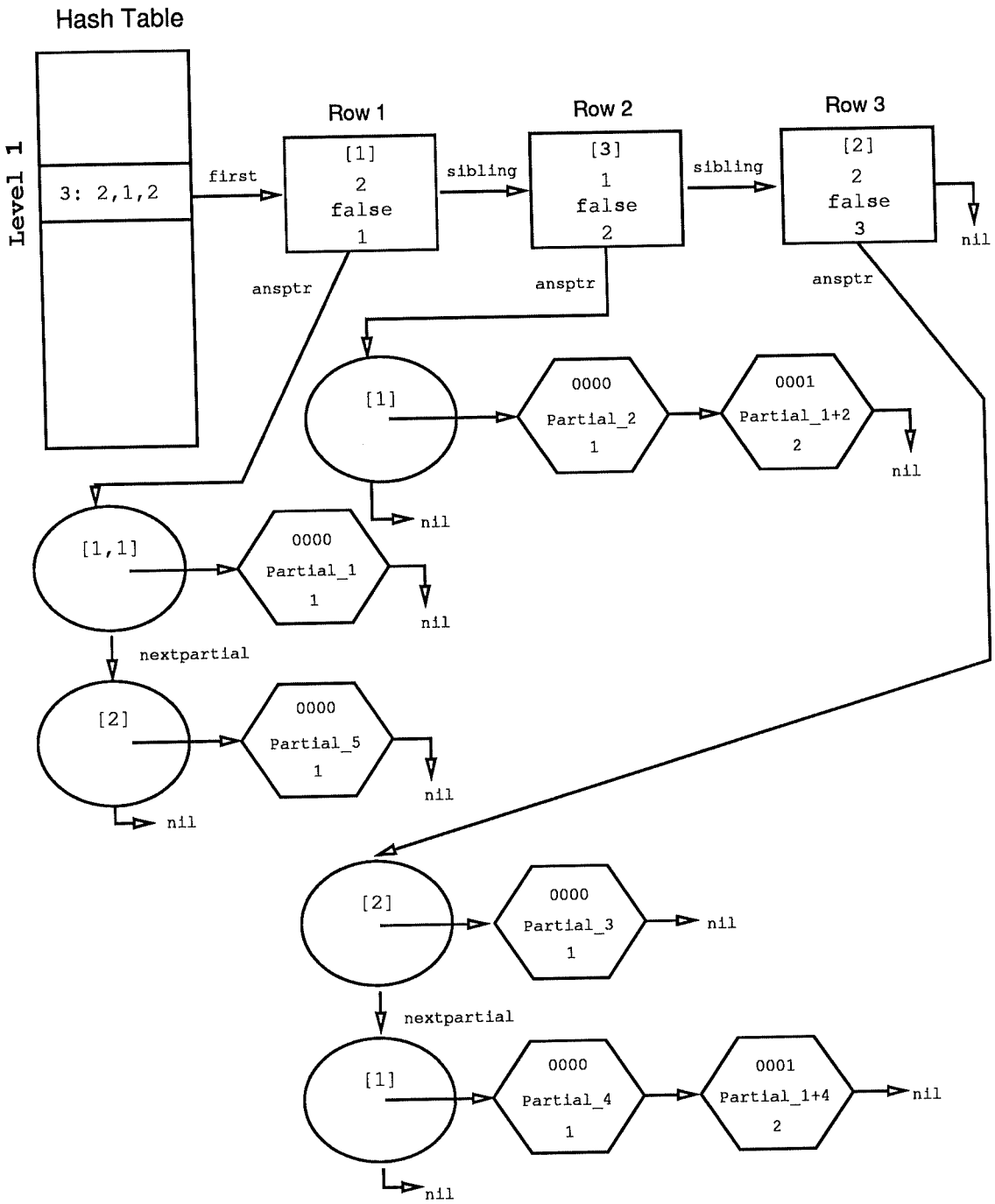


Figure 26: Partials buffer — stage 4 in incremental back-unification

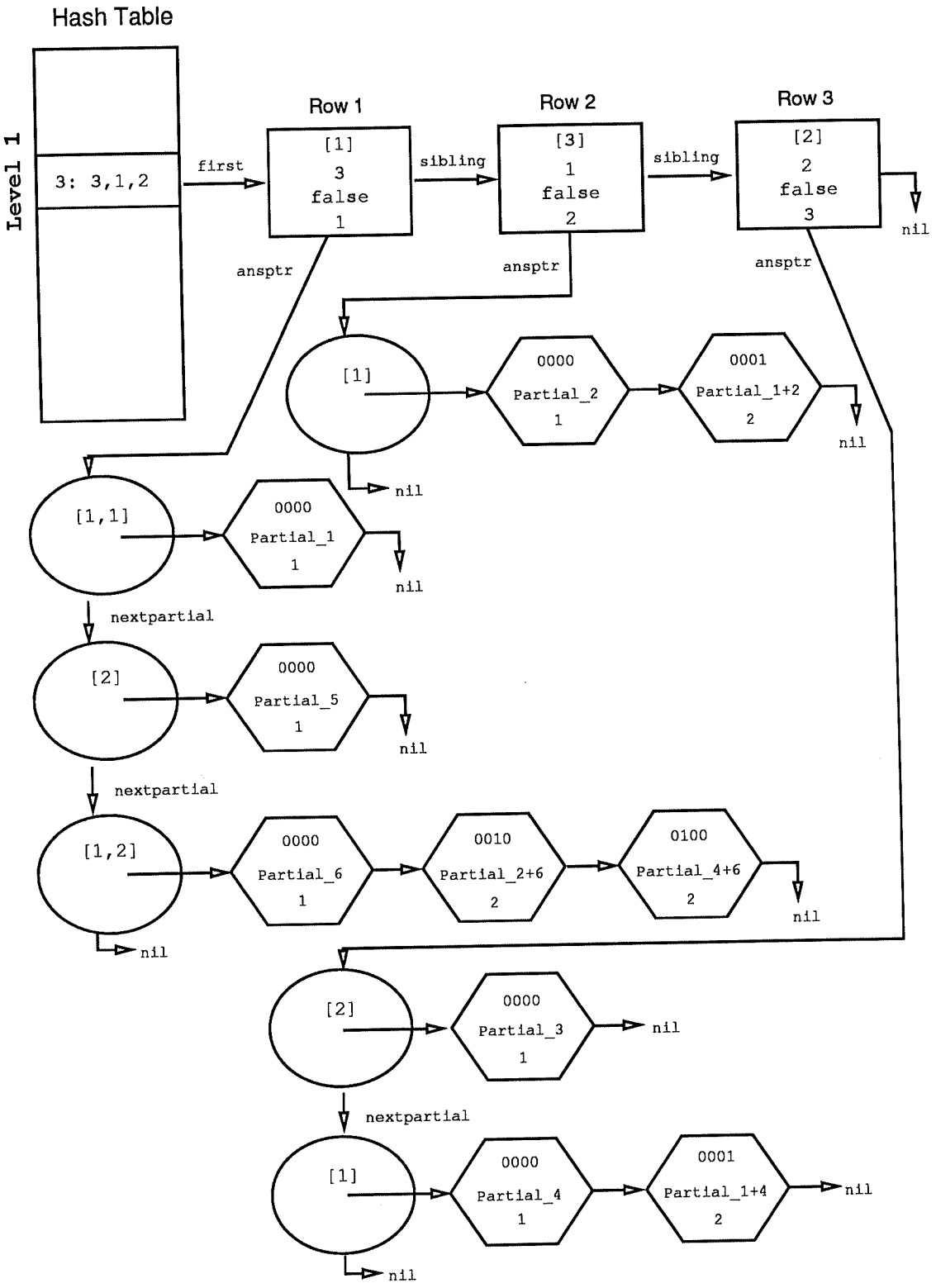


Figure 27: Partials buffer — final stage in incremental back-unification

The main advantage of using the incremental strategy is that processes which would otherwise remain idle can contribute to the generation of final solutions even before all partial solutions have been computed. However, support for this strategy implies a more complicated allocation strategy, as well as the availability of a significantly increased storage capacity in the Answer Server. This is caused by allowing partial solutions to be recorded in several stages of back-unification. More communications occur between the Answer Server and the PEs, which can increase the possibility of contention even with a small number of PEs in the network.

Most of the disadvantages of this strategy, however, fall away when computing deterministic problems, in which all partial solutions usually contribute to the complete final answer. As soon as a partial solution is received at the Answer Server, it is back-unified with all existing partials. Since there are no alternative solutions for any branch, the allocation is done by choosing either the partly-joined partial solution if one exists, or the raw partial solution for each or-node. The check to ensure that the partly-joined solutions do not contain any partial solutions from the row at which the newly generated solution has been inserted is then redundant. The allocation strategy is thus substantially simplified and the majority of join operations will involve only a small number of partial solutions.

# 7 Modifications to Prolog

By virtue of the nature of Prolog source code, it is possible to exploit or-parallelism implicitly, without any additional annotations supplied by the user. In supporting and-parallelism, however, some explicit code is necessary to designate those literals in a clause body eligible for execution in parallel.

Eligibility in some and-parallel systems requires that the literals do not share unbound variables [DeGroot 1984; Hermenegildo 1986]. In the APPNet the eligibility restriction merely ensures that literals are not executed before it is feasible to do so, taking into account the instantiation state of input variables. In a sequential Prolog evaluation, the onus rests with the programmer to ensure that input variables are suitably instantiated. The use of a parallel control strategy in the APPNet, however, is liable to cause the execution of subgoals in an unspecified order. Therefore, the APPNet itself is responsible for evaluating predicates only when it is safe to do so.

To support and-parallel execution in the APPNet, additional information is included in the Prolog program to specify those parts of the program which may be usefully executed concurrently. This information is analogous to the thresholds introduced in the Epilog language and is only necessary as a means of curtailing the and-parallel execution where this is obviously ineffectual and where no solution will result. The information is supplied automatically by a preprocessor after analysis of the program.

The analysis carried out is static and hence relies on the fact that any program intended for parallel execution is static. For this reason it is not possible, at this stage, to allow programs containing `assert`, `retract` and `call` predicates to be executed in the APPNet.

The static analysis used is based on that devised by Mellish [1986] and extended by Debray and Warren [1986], and focuses on *automatic mode inference* for programs. This entails the classification of variables occurring in each clause in the program with respect to how they are instantiated at a given point in the clause. From this classification, the instantiation state of variables on entry to a procedure (the *calling pattern* of the procedure) and on exit from a procedure (the *success pattern* of the procedure) may be determined.

Thus given a set of procedures and an initial goal without mode information for the goal variables, the analysis attempts to find the greatest lower bound on the calling patterns for each procedure. The greatest lower bound on the calling pattern for a procedure is defined as the minimum level of instantiation for the arguments in the head of the procedure. The analysis makes use of known calling patterns and success patterns, for example for system and user predicates and functions. The output of this phase is an annotated program, showing those literals suitable for and-parallel execution.

A further use for the static analysis carried out in the APPNet is in determining the template for partial solutions. The template always contains the unbound arguments, and those bound to non-ground terms, appearing in the query. However, where local variables are used in literals which are potential candidates for and-parallel evaluation, the template is expanded to include these temporary variables.

A third reason for the introduction of the preprocessing phase is that the APPNet cannot cope with the extra-logical constructs, *cut*, `not`, `var` and `nonvar`, without appropriate modification to the parallel computation rule. The successful outcome of the use of these constructs is tied in with the depth-first, left-to-right computation rule normally used in a sequential Prolog execution environment. With a parallel execution strategy certain compromises are therefore essential to ensure the correct interpretation of programs in which these constructs are present.

By passing the user program through a preprocessing phase before execution in the APPNet, it is possible to tolerate the presence of the extra-logical constructs, albeit with some reduction in the parallelism. For the execution of those predicates containing extra-logical constructs, the parallel computation rule is replaced temporarily by a sequential one. Research is currently underway to ascertain the extent to which the parallel execution needs to be inhibited when executing programs with extra-logical features. As far as the APPNet is concerned at this stage, a conservative approach is adopted and the execution strategy reverts to being a sequential one for the extent of the influence of the extra-logical construct.

# 8 Implementation

The APPNet runs on a network of up to nineteen MicroVAX processors connected by an Ethernet. There is no reason in principle to prevent the addition of more processors to the network. In the absence of shared memory, the APPNet relies strictly on message passing for communication. An RPC scheme has been built on top of the *socket* primitives provided by the 4.2 BSD/ULTRIX operating systems [Leffler *et al.* 1989]. The APPNet has been developed along the lines of a traditional client/server communication model, with the PEs configured as clients to both the Command Server and Answer Server.

## 8.1 The Extended WAM

Each PE is a modified sequential WAM with a communications interface. SB-Prolog, developed at SUNY, Stony Brook, has been used as the kernel upon which the APPNet has been developed. SB-Prolog adheres to standard Clocksin and Mellish [1981] syntax. The Prolog system consists of a compiler and a run-time evaluation system.

To support oracle creation and manipulation, the SB-compiler has been modified by Clocksin [Klein 1989] to generate indexing information for all predicates. This information includes the number of alternative clauses for each predicate, as well as a breakdown of the candidate clauses based on the first argument. Indices on the first argument are classified as being of type integer, list, null list, constant, structure or variable.

The indexing information is used by the oracle_loader [Klein 1989] to generate additional WAM codes used in support of or-parallel evaluation. The traditional WAM instructions, **try**, **retry** and **trust**, have been supplemented by **apptry**, **appretry** and **apptrust** respectively. Each of these instructions is followed by the number of arguments in the head of the clause, and the entry point (ep) of the clause. The new set of instructions, used in conjunction with an or-parallel control strategy, ensures that unlike a sequential system where a choicepoint stack frame is created at every or-choicepoint, the choicepoint frame is only established where or-parallel processing is inhibited. This occurs for example, where resources have been expended and no job reallocation is permitted. Sequential backtracking then takes place at any or-choicepoints executed after partitioning of the search space has ceased. In addition an **ormax** instruction is generated for each group of alternative clauses giving the number of clauses in the group. This information is necessary for the automatic partitioning algorithm to enable PEs to pick the correct clause according to their individual id's and the given branching factor. The SB-simulator has been modified appropriately to recognise and interpret the extended WAM instruction set.

In the same way as the ormax instruction is used to guide or-parallel partitioning, the `$andmax/2`

and `$andmax/3` meta-level predicates are used to isolate those subgoals able to be executed in parallel. These predicates also supply the information needed for automatic partitioning, namely the number of subgoals at the and-choicepoint. The `$andmax` predicates are inserted into the user program during the preprocessing phase.

`$andmax/2` is used when only the uninstantiated arguments in the head of the called predicate are to be represented in the partial solutions, while `$andmax/3` includes a third parameter to specify the list of relevant variables of which bindings will be retained in the partial solutions. The first argument to both `$andmax/2` and `$andmax/3` gives the branching factor at the choicepoint, while the second gives the list of the and-clauses to be executed.

## 8.2 Extended Choicepoint Frames

To support and-or parallel execution with sequential backtracking, it is necessary to extend the information stored in an or-choicepoint stack frame. In a sequential Prolog system [Gabriel *el al.* 1984] a choicepoint stack frame is created at every non-deterministic or-node and contains the following information current at the time of creation:

arguments to procedure call

| | |
|---|---|
| *le_reg* | current activation record (or current environment) |
| *cp_reg* | procedure return address |
| *tr_reg* | top of trail stack |
| *h_reg* | top of heap |
| *b_reg* | previous choicepoint frame |

entry point of next alternative clause *i.e.*, retry or trust instruction

In a purely sequential system this information is sufficient to reset the state of computation whenever backtracking occurs. With parallel execution, additional information is required to reset the relevant and-path and or-path on backtracking. In the ensuing discussion reference is made to both a *current* and a *given* oracle. The *current* oracle reflects the exact path (in terms of and-path and or-path) up to the current point in execution. The *given* oracle is either that which has been sent by the Command Server (for example, if job reallocation has been done), or the path up to the point where no further division of the proof tree can take place, whichever is the greater.

Three additional registers are saved in the choicepoint frame, namely

| | |
|---|---|
| *and_reg* | current and-path |
| *gen_areg* | given and-path |
| *or_reg* | current or-path |

The *and_reg* and *or_reg* point to the last bits in the *current* and-path and or-path respectively. The *gen_areg* records the last bit in the *given* and-path. Thus on backtracking if the given and-path is greater than the current and-path, it is necessary to reset the and-path only as far as *gen_areg*. This ensures that the PE remains in a *following mode* as far as the and-path is concerned until the given and-path is exhausted, irrespective of the amount of backtracking that takes place in the meantime.

The need for a similar register to record the *given* or-path is precluded by the mechanism of sequential backtracking itself and the creation of choicepoint frames. As has been mentioned in Section 5, a choicepoint frame is not created if the PE is in a *following mode*. Hence, by virtue of the definition of "following mode" (which exists as long as there is an oracle to follow) there is no possibility of ever backtracking beyond a *given* or-path.

# 9 Performance Evaluation

The performance of the APPNet has been gauged over a period of time and with several Prolog benchmarks. The code for three of these benchmarks is given in the appendix in the preprocessed form suitable for execution in the APPNet.

All measurements have been obtained without exclusive use of the Ethernet or the host machines. The results are therefore susceptible to fluctuations caused by additional traffic and usage of the host machines by third parties. For this reason, the performance results listed below have been obtained from several executions of each benchmark. The lowest execution time for each benchmark for a given number of PEs has been retained.

The measurement of overall *actual time* for the system is taken from when the first PE connects with the Command Server until the last PE terminates. This measurement includes:

- loading the Prolog system in each PE,
- loading the code for the benchmark,
- distributing the goal,
- computing all solutions to the problem,
- logging initialisation and termination messages from the PEs locally and with the Command Server,
- logging all partial solutions locally and with the Answer Server,
- logging all complete solutions locally and with the Answer Server,

Besides the actual time measurement, further parameters obtained during each trial are also taken into consideration in the evaluation of the APPNet. These are:

1. the "ideal" parallel time of the system (*Maximum time*), which is equivalent to the *total cputime* of the PE with the longest life. (System time is defined as the time the CPU has spent in the UNIX kernel, while user time is the time spent executing in user mode. The sum of these gives the *total cputime*, which represents the amount of time the CPU has been active on the particular task.) The *maximum time* measurement gives a more realistic measure of the benefits accruing from the distributed approach. This depends on the assumptions that within the system all PEs can be started at roughly the same time and that a computation that involves one unit of execution time can be done in equal real time on all PEs.
2. the *overhead* of the system calculated as the sum of the user time of the Answer Server and the Command Server. System time is ignored in this instance, since it consists largely of the time spent waiting for connections, and as such is most vulnerable to third-party traffic. This overhead parameter represents the additional cost of maintaining a distributed parallel environment over a sequential system.
3. the *average* execution time per PE in the distributed network calculated using the *total cputime* of each process. This would be an indication of the performance of the system if it were possible to distribute the workload equally amongst the PEs in the system. This is, however, rarely achieved in practice owing to the nature of the search space, where paths leading to leaf nodes need not be of the same length or necessarily involve the same amount of computation. In addition, it is not possible to obtain the final solution until the last partial solution has been generated, with the result that a disparity will always occur in the execution times of the $n$–1 PEs and the $n$th PE.

In the evaluation of each benchmark, a graph is presented showing four curves, which correspond to the *actual time*, *maximum time*, *average time* and *overhead* parameters explained above.

## 9.1 Exploring Or-Parallelism

The traditional *8-Queens* problem, without any extra-logical features in the code, shows measurable speed-up when executed on the APPNet using an or-parallel execution strategy. The performance results obtained on the APPNet exhibit similar behavioural characteristics to those given for the DelPhi machine [Klein 1989]. This proves that the addition of the potential for and-parallel execution does not degrade the performance obtained using an or-parallel execution strategy.

The graph in Figure 28 presents the performance of the benchmark in the APPNet when using a right-biased, or-parallel partitioning strategy with no job re-allocation, and without and-parallelism.

Using up to a maximum of nine processing elements, a speed-up in the execution of the benchmark

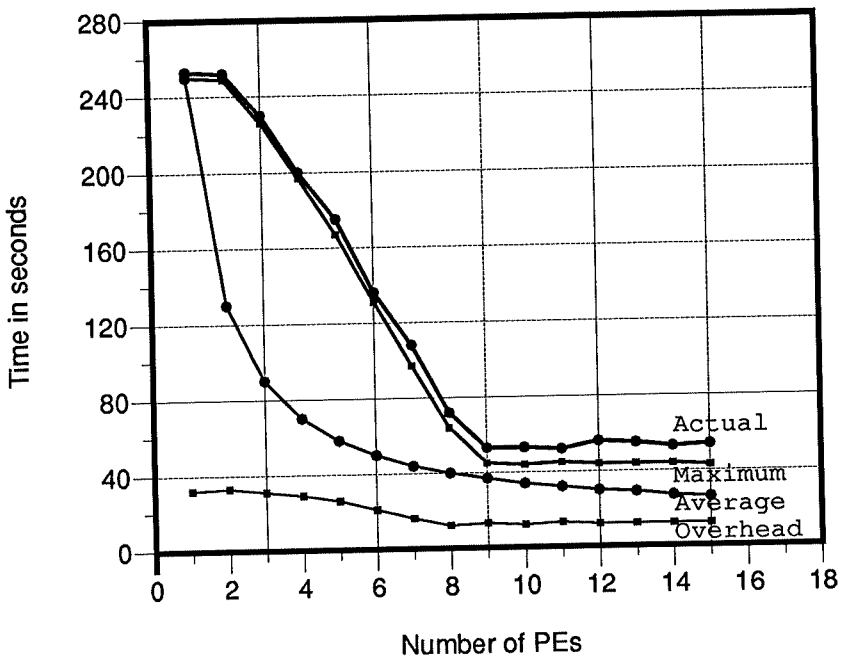


Figure 28: *8-Queens* performance and overhead graphs

is evident. In particular, the ratio of the actual execution time when using nine PEs to that of a system with a single PE is 1:5. No further speed-up is obtained when increasing the number of PEs beyond nine. This behaviour may be explained by considering that the number of PEs generating complete solutions remains at eight, even with a network of up to 40 PEs. The available or-parallelism in the benchmark can therefore be fully exploited with only nine of the maximum of 40 PEs in the network.

With a network of more than nine PEs, the execution time remains roughly constant. No degradation of the system due to increased network traffic is evident as the number of PEs increases.

Intuitively, one would expect server overhead to increase with the addition of PEs. For the 8-Queens benchmark, however, this is not so. A reason for this is that two counteracting factors make up the overhead measurement. The time that the Command Server and Answer Server are active decreases as the number of PEs increases (since overall execution time decreases), while the actual computation (in terms of messages processed) increases for the Command Server. This gives the net result as shown by the overhead curve in Figure 28.

## 9.2 And-Parallel Execution

It is not as easy to establish a behavioural pattern for the different biases in the and-parallel strategies. The behaviour is more dependent on individual programs and not on a generic type of

program as is evident in the or-parallel strategies with respect to recursive and non-recursive programs. The former class of programs lends itself to the right-biased, or-parallel strategies, since the excess computing power is shifted to the right of the search tree.

The available biased and-parallel strategies may have different effects even on the same problem depending on the order in which subgoals are called and the computational weight of individual procedures in the program. For the benchmarks tested, the differences in performance for the alternative and-bias strategies are slight. One reason for this is the nature of the and-parallel subgoals, with the computational weight of these fairly equal within each program.

### 9.2.1 Discrete Fourier Transform

Figure 29 illustrates the performance of the 64-point *discrete Fourier transform* [Clocksin 1988]. No or-parallelism is exploited in these trials; a no-bias, and-parallel strategy with incremental back-unification is used without job reallocation.

The performance of this benchmark conforms to the expected ideal for the APPNet system, in that speed-up occurs with the addition of PEs when using the incremental join operation for back-unification. With this strategy and the available PEs, the point has not yet been reached where the performance of the system starts to deteriorate due to congestion. Using a total back-unification strategy, degradation starts to occur with approximately thirteen PEs in use. This is to be expected,

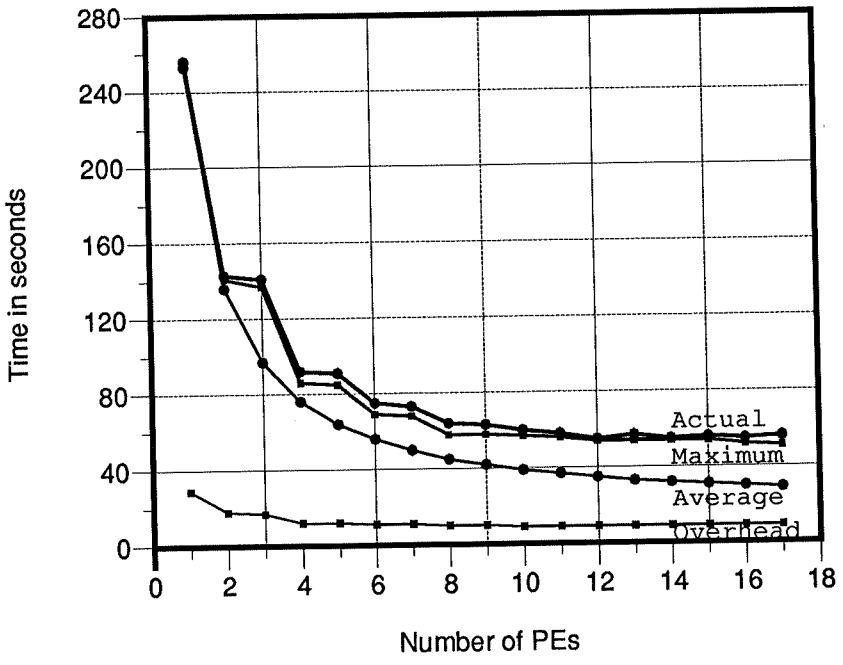


Figure 29: *Discrete Fourier Transform*: incremental back-unification

since with the addition of a PE a further partial is generated, which in turn can only be back-unified once all the partial solutions at the relevant level have been found. As more partial solutions are generated, the time needed for back-unification grows. When using the total back-unification strategy a great deal of this overhead occurs after most of the computation has been completed.

The overhead curve in Figure 29 resembles the curve plotted for the 8-Queens benchmark (*cf.* Figure 28). The same explanation is valid for this benchmark, that is, that the increased computation in the Answer Server is offset by the decrease in overall execution time. The net result is a decrease in overhead as the number of PEs increases.

### 9.2.2 Quicksort

The performance measurements for *Quicksort* on 1000 elements using a no-bias, and-parallel strategy with incremental back-unification are shown in Figure 30. No significant difference in performance is experienced when using the two back-unification strategies. This is to be expected as the Quicksort program is highly recursive with an and-branching factor of only two. At each level of recursion a binary join takes place, with the result that even when using the total back-unification strategy, the number of partial solutions to be joined in the final join at the top level of recursion remains constant irrespective of the number of PEs in the network.

No speed-up is evident as the number of PEs increases. A reason for this behaviour may lie in the

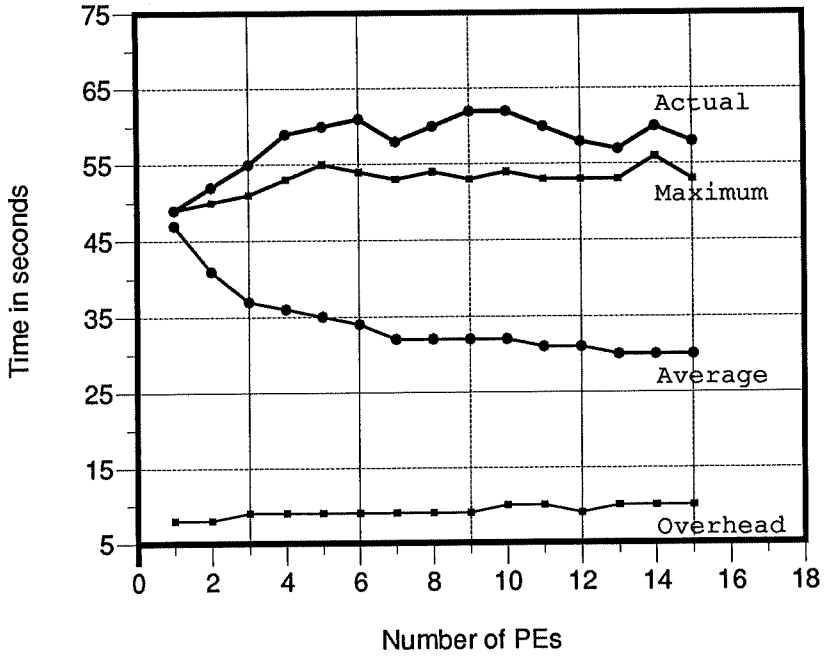


Figure 30: *Quicksort*: incremental back-unification

size of the partial solutions generated. In comparison to the partial solutions generated in the discrete Fourier transform, each partial is significantly larger, which means that the back-unification process for Quicksort will contribute significantly to the overall execution time. In addition the granularity of the distributed computation in the Quicksort is smaller than that in the Fourier transform. This means that the point at which it becomes cheaper to do the computation locally rather than to distribute it, is reached earlier. Overhead increases gently with increased number of PEs in the network. This confirms the idea that the computation inherent in the Quicksort (which ultimately involves only integer comparisons) is too fine-grained to be expected to cover the cost of distribution.

## 9.3 And-Or Parallel Execution

A *map colouring* program has been used to test the performance of the APPNet when using an and-or parallel execution strategy. The results plotted in Figure 31 show that a negative speed-up is observed with the addition of processors. The granularity of the distributed computation is relatively small, which may account for the negative results. Overhead is largely constant with a network of four or more PEs and there appears to be no bottle-neck at the Answer Server in receiving the solutions.

A relationship exists between the number of successful PEs, the distribution of final answers and

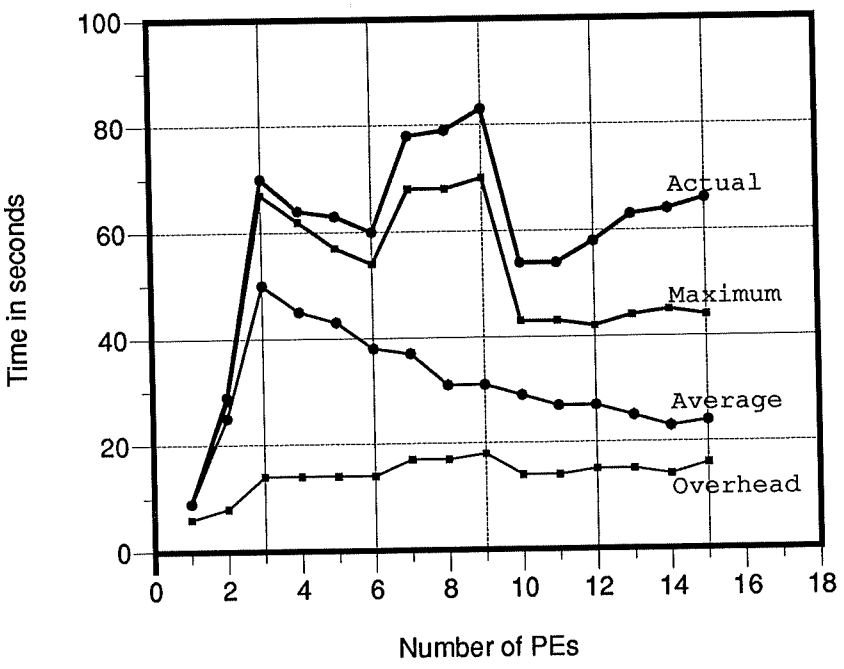


Figure 31: *Map Colouring*: incremental back-unification with and-or parallelism

the execution times for this benchmark. Peaks in the actual time curves are accompanied by a small number of successful PEs (*i.e.* those PEs which generate final solutions) and an uneven distribution among these successful PEs of final answers. Similarly, local minima in the graphs occur where the distribution of final answers is balanced and involves a larger number of successful PEs.

### 9.4 Choice of Benchmarks

It seems appropriate at this stage to comment on the choice of benchmarks used in the evaluation of the APPNet. No application type benchmarks have been used. Programs such as parsers, expert systems and the like, contain bulky code which obscures the actual area of investigation. These programs are also mostly shallow in terms of search depth. Consequently their execution does not demand much computation.

Instead, benchmarks have been chosen so that it is known in advance what demands are being placed on the system, and hence the performance results may be interpreted accordingly. The behaviour of the Prolog programs used is well-defined, allowing the system to be analysed instead of agonising over the programs.

## 10 The APPNet versus the ROPM, Epilog and PEPSys

The design of the APPNet was influenced by three and-or parallel models, namely the ROPM [Kalé 1987(a)], Epilog [Wise 1984] and PEPSys [Westphal *et al.* 1987], and the or-parallel DelPhi model [Clocksin and Alshawi 1986]. In this section the similarities and differences between the APPNet and the and-or parallel models are highlighted. With the exception of PEPSys, all these models have been designed for use on distributed systems without shared memory. PEPSys however relies on a shared memory architecture.

The APPNet supports dependent and-parallelism without the need for any partial ordering of clauses nor variable annotations. Some static analysis is done to produce the threshold information necessary to curtail ineffectual parallelism. The ROPM also relies on static analysis to curtail parallelism, however, the analysis is also required to produce a partial ordering of independent clauses.

The evaluation scheme used in PEPSys relies on pragmas to designate both those or-branches, which the programmer considers to be candidates for parallel execution, and the independent and-branches with potential for parallel execution. The PEPSys model therefore entrusts all parallel control to the discretion of the programmer. Inherent in the Epilog data-flow model is a partial ordering of clauses brought about by the use of thresholds so that a clause may only be fired if

a minimum number of input variables are bound. Epilog however, also allows the programmer some additional control over the computation with the use of variable annotations and the CAND and COR constructs.

The treatment of partial solutions in the APPNet is similar to that employed in the ROPM. In the latter model the computed partial solutions are returned to ancestor reduce-nodes, where they await the arrival of further partial solutions before back-unification takes place. This is similar to the behaviour of the naive configuration introduced in Section 4.2 where two kinds of virtual process are used to enable back-unification to take place.

The two models differ in the number of processes created during the computation. In order to find one partial solution, a single process is created in the APPNet and no communication with other PEs is necessary. Contrast this to the execution model of the ROPM which creates a new process at every choicepoint, and transmits a complete copy of the environment to the newly created process. The PEPSys model also relies on passing down common bindings to processes created at run-time, although this is only done on demand and thus avoids unnecessary communication if the created process terminates quickly.

# 11 Conclusion

This paper has described in some detail, a model for exploiting inherent parallelism in the evaluation of Prolog programs. The algorithms used for dividing the search space amongst the available processing elements in the first stage, as well as the back-unification strategies used in the second stage of evaluation, have been presented.

The APPNet has been implemented on MicroVAX processors using an Ethernet for message passing. The performance results show that the APPNet is a viable model for the evaluation of deterministic problems. The evidence drawn from the evaluation of non-deterministic problems is not conclusive, but we suspect that communication overhead is a significant contributor to the overall execution time. It may be argued that with an alternative communication model, the APPNet would be as useful in evaluating non-deterministic problems as it is currently in the evaluation of deterministic programs.

The APPNet may be further improved by implementing a combination of the *producer* mechanism and back-unification. The potential benefits of such a scheme, which would undoubtedly involve additional run-time analysis, remain to be investigated.

## 12 Acknowledgements

Thanks are due to William Clocksin for his constructive criticism and for reviewing an earlier draft of this report; to Luke Tunmer for the use of his PostScript previewer; and to John Bradshaw for his continued support.

Financial support for this research was provided by the Royal Commission for the Exhibition of 1851, London; the UK Overseas Research Student Awards Scheme; the Foundation for Research and Development, Pretoria; the Cicely Haworth Wahl Scholarship, South Africa; Rhodes University, Grahamstown. I acknowledge the support of Digital Equipment Corporation, grant ERG UK-017 to W.F. Clocksin.

## References

[1] Ali, K.A.M., "Or-Parallel Execution of Horn Clause Programs based on WAM and Shared Control Information", *Research Report SICS R88010*, Swedish Institute of Computer Science, 1986.

[2] Ali, K.A.M., "Or-Parallel Execution of Prolog on a Multi-Sequential Machine", *International Journal of Parallel Programming* **15**(3), 189–214, 1987.

[3] Baron, U., J. Chassin de Kergommeaux, H. Hailperin, M. Ratcliffe, P. Robert, J-C Syre and H. Westphal, "The Parallel ECRC Prolog System PEPSys: an Overview and Evaluation Results", *Proceedings of the International Conference on Fifth Generation Computer Systems*, 841–849, ICOT, Tokyo, 1988.

[4] Biswas, P., S-C. Su and D.Y.Y. Yun, "A Scalable Abstract Machine Model to Support Limited-Or (LOR) / Restricted-And Parallelism (RAP) in Logic Programs", *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, 1160–1179, 1988.

[5] Calderwood, A. and P. Szeredi, "Scheduling Or-Parallelism in Aurora: The Manchester Scheduler", *Proceedings of the Sixth International Conference on Logic Programming*, 419–435, 1989.

[6] Chang, J-H., A.M. Despain and D. DeGroot, "And-Parallelism of Logic Programs based on a Static Data Dependency Analysis", *COMPCON*, 218–225, IEEE, Spring 1985(a).

[7] Chang, J-H. and A.M. Despain, "Semi-Intelligent Backtracking of Prolog based on Static Dependency Analysis", *Proceedings of the 1985 International Symposium on Logic Programming*, 10–21, 1985(b).

[8] Chassin de Kergommeaux, J. and P. Robert, "An Abstract Machine to Implement Or-And Parallel Prolog Efficiently", *Journal of Logic Programming* **8**(3), 249–264, 1990.

[9] Chengzheng, S. and T. Yungui, "The Or-Forest Description for the Execution of Logic Programs", *Proceedings of the Third International Conference on Logic Programming*, in *Lecture Notes in Computer Science Vol. 225*, 710–717, Springer-Verlag, 1986.

[10] Chengzheng, S. and T. Yungui, "The Sharing of Environment in And-Or Parallel Execution of Logic Programs", *Proceedings of the Fourteenth Annual International Symposium on Computer Architecture*, 137–144, IEEE Computer Society Press, Washington, 1987.

[11] Ciepielewski, A. and S. Haridi, "A Formal Model for Or-Parallel Execution of Logic Programs", in *Information Processing 83* (R.E.A. Mason, ed.), 299–306, Elsevier Science Publishers, North-Holland, Amsterdam, 1983.

[12] Ciepielewski, A., S. Haridi and B. Hausman, "Or-Parallel Prolog on Shared Memory Multiprocessors", *Journal of Logic Programming* **7**, 125–147, 1989.

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

[14] Clocksin, W.F., "Principles of the DelPhi Parallel Inference Machine", *Computer Journal* **30**(5), 386–392, 1987.

[15] Clocksin, W.F., A Technique for Translating Clausal Specifications of Numerical Methods into Efficient Programs, *Journal of Logic Programming* **5**(3), 231–242, 1988.

[16] Clocksin, W.F. and H. Alshawi, "A Method for Efficiently Executing Horn Clause Programs using Multiple Processors", *Technical Note CCSRC-3*, Cambridge Computer Science Research Centre, Cambridge, 1986.

[17] Clocksin, W.F. and H. Alshawi, "A Method for Efficiently Executing Horn Clause Programs using Multiple Processors", *New Generation Computing* **5**, 361–376, 1988.

[18] Clocksin, W.F. and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, 1981.

[19] Conery, J.S., *Parallel Execution of Logic Programs*, Kluwer Academic Publishers, Boston, 1987(a).

[20] Conery, J.S., "Implementing Backward Execution in Nondeterministic And-Parallel Systems", *Proceedings of the Fourth International Conference on Logic Programming*, 633–653, 1987(b).

[21] Debray, S.K. and D.S. Warren, "Automatic Mode Inference for Prolog Programs", *Proceedings of the 1986 International Symposium on Logic Programming*, 78–88, 1986.

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

[23] DeGroot, D., "Restricted And-Parallelism and Side-Effects", *Proceedings of the 1987 International Symposium on Logic Programming*, 80–91, IEEE, 1987.

[24] Disz, T., E. Lusk and R. Overbeek, "Experiments with Or-Parallel Logic Programs", *Proceedings of the Fourth International Conference on Logic Programming*, 576–600, 1987.

[25] El-Dessouki, O.I. and W.H. Huen, "Distributed Enumeration on Between Computers", *IEEE Transactions on Computers* **C-29**(9), 818–825, 1980.

[26] Fuchi, K., "The Direction the Fifth Generation Computer Project will take", *New Generation Computing* **1**, 3–9, 1983.

[27] Gabriel, J., T. Lindholm, E.L. Lusk and R.A. Overbeek, "A Tutorial on the Warren Abstract Machine for Computational Logic", *Technical Report ANL-84-84*, Argonne National Laboratory, Argonne, Illinois, 1984.

[28] Hausman, B., A. Ciepielewski and S. Haridi, "Or-Parallel Prolog made Efficient on Shared Memory Multiprocessors", *Proceedings of the 1987 International Symposium on Logic Programming*, 69–79, 1987.

[29] Hermenegildo, M.V., "An Abstract Machine for Restricted And-Parallel Execution of Logic Programs", *Proceedings of the Third International Conference on Logic Programming, Lecture Notes in Computer Science Vol. 225*, 25–39, Springer-Verlag, 1986.

[30] Hermenegildo, M.V. and R.I. Nasr, "Efficient Management of Backtracking in And-Parallelism", *Proceedings of the Third International Conference on Logic Programming, Lecture Notes in Computer Science Vol. 225*, 40–54, Springer-Verlag, 1986.

[31] Kalé, L.V., "The Reduce-OR Process Model for Parallel Execution of Logic Programs", *Proceedings of the Fourth International Conference on Logic Programming*, 616–632, 1987(a).

[32] Kalé, L.V., " 'Completeness' and 'Full Parallelism' of Parallel Logic Programming Schemes", *Proceedings of the 1987 International Symposium on Logic Programming*, 125–133, 1987(b).

[33] Klein, C.S., *Exploiting Or-Parallelism in Prolog using Multiple Sequential Machines, Ph.D.* dissertation, University of Cambridge, 1989.

[34] Kumon, K., H. Masuzawa, A. Itashiki and Y. Sohma, "Kabu-Wake: a New Parallel Inference Method and its Evaluation", in *Digest of Papers COMPCON 86* (A.G. Bell, ed.), 168–172, IEEE Computer Society Press, 1986.

[35] Leffler, S.J., M.K. McKusick, M.J. Karels and J.S. Quarterman, *The Design and Implementation of the 4.3BSD Operating System*, Addison-Wesley Publishing Company, Massachusetts, 1989.

[36] Li, P.P. and A.J. Martin, "The Sync Model: A Parallel Execution Method for Logic Programming", *Proceedings of the 1986 International Symposium on Logic Programming*, 223–234, 1986.

[37] Lin, Y-J. and V. Kumar, "A Parallel Execution Scheme for Exploiting And-Parallelism of Logic Programs", *Proceedings of the 1986 International Conference on Parallel Processing*, 972–975, IEEE, 1986(a).

[38] Lin, Y-J., V. Kumar, and C. Leung "An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs", *Proceedings of the Third International Conference on Logic Programming*, in *Lecture Notes in Computer Science Vol. 225*, 55–68, Springer-Verlag, 1986(b).

[39] Lusk, E. *et al.*, D.H.D. Warren *et al.* and S. Haridi *et al.*, "The Aurora Or-Parallel Prolog System", *Proceedings of the International Conference on Fifth Generation Computer Systems*, 819–830, ICOT, 1988.

[40] Mellish, C.S., "Abstract Interpretation of Prolog Programs", *Proceedings of the Third International Conference on Logic Programming*, 463–474, 1986.

[41] Pereira, L.M. and A. Porto, "An Interpreter of Logic Programs using Selective Backtracking", *Report 3/80*, Departamento de Informatica, Universidade Nova de Lisboa, 1980.

[42] Ratcliffe, M. and J-C. Syre, "The PEPSys Parallel Logic Programming Language", *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, 48–55, 1987.

[43] Shapiro, E., "Concurrent Prolog: a Progress Report", *IEEE Computer* **19**(8), 44–58, 1986.

[44] Shapiro, E., "An Or-Parallel Execution Algorithm for Prolog and its FCP Implementation", *Proceedings of the Fourth International Conference on Logic Programming*, 311–337, 1987.

[45] Tebra, H, *Optimistic And-Parallelism in Prolog*, *Ph.D.* dissertation, Vrije Universiteit, Amsterdam, ICE Printing, 1989.

[46] Ueda, K., "Introduction to Guarded Horn Clauses", *ICOT Journal* **13**, 9–18, 1986.

[47] Warren, D.H.D., "An Abstract Prolog Instruction Set", *Technical Note 309*, SRI International, 1983.

[48] Warren, D.H.D., "Or-Parallel Execution Models of Prolog", *Technical Report*, Department of Computer Science, University of Manchester, 1987(a).

[49] Warren, D.H.D., "The SRI Model for Or-parallel Execution of Prolog — Abstract Design and Implementation", *Proceedings of the 1987 International Symposium on Logic Programming*, 92–102, 1987(b).

[50] Westphal, H., P. Robert, J. Chassin and J.C. Syre, "The PEPSys Model — Combining Backtracking, And-Parallelism and Or-Parallelism", *Proceedings of the 1987 Symposium on Logic Programming*, 436–448, 1987.

[51] Wise, M.J., "EPILOG: Re-interpreting and Extending Prolog for a Multiprocessor Environment", in *Implementations of Prolog* (J.A. Campbell, ed.), 341–351, Ellis Horwood Limited, Chichester, 1984.

[52] Wise, M.J., *Prolog Multiprocessors*, Prentice-Hall, Englewood Cliffs, 1986.

[53] Woo, N.S. and K-M. Choe, "Selecting the Backtrack Literal in the AND/OR Process Model", *Proceedings of the 1986 Symposium on Logic Programming*, 200–210, 1986.

[54] Woo, N.S. and R. Sharma, " An And-Or Parallel Execution System for Logic Program Evaluation", *Proceedings of the 1987 International Conference on Parallel Processing*, 162–165, 1987.

[55] Wrench, K.L., *A Distributed And-Or Parallel Prolog Network, Ph.D.* dissertation submitted, University of Cambridge, 1990.

[56] Yasuhara, H. and K. Nitadori, "ORBIT: A Parallel Computing Model of Prolog", *New Generation Computing* **2**, 277–288, 1984.

# Code for Benchmark Programs

## Example 1

/* Prolog program for *Quicksort* */
/* Two versions are presented, using append/3 and tail-recursion respectively */
/* Top-level query *e.g.*

```
:- quisortx([8,14,9,12,3,1,50,5,7,11,2,15,44,22,12,4,6,18,19,33],
            Ans, []). */

      qsplit([], _, [], []).
      qsplit([A|X], H, [A|Y], Z) :- A < H, qsplit(X, H, Y, Z).
      qsplit([A|X], H, Y, [A|Z]) :- A >= H, qsplit(X, H, Y, Z).

       /* Using append/3 */

      quisort([], []).
      quisort([H|T], S) :-
             qsplit(T, H, A, B),
             $andmax(2, [A1, B1],
                     [quisort(A, A1), quisort(B, B1)]),
             append(A1, [H|B1], S).

      append([], B, B).
      append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).

       /* Tail Recursive Version */

      quisortx([], X, X).
      quisortx([H|T], S, X) :-
             qsplit(T, H, A, B),
             $andmax(2, [S, H, Y, X],
                     [quisortx(A, S, [H|Y]), quisortx(B, Y, X)]).
```

## Example 2

/* Prolog program for the *n-Queens Problem* */
/* No cuts or extra-logical features are used in this version */
/* Top-level query `:- get_solutions(8, Answers).` */

```
get_solutions(Boardsize, Soln) :-
        solve(Boardsize, [], Soln).

solve(Bs, [square(Bs, Y) | L], [square(Bs, Y) | L]).
solve(Boardsize, Initial, Final) :-
        newsquare(Initial, Next, Boardsize),
        solve(Boardsize, [Next | Initial], Final).

newsquare([square(I, J) | Rest], square(X, Y), Boardsize) :-
        I < Boardsize, X is I + 1,
        snint(Y, Boardsize),
        notthreatened(I, J, X, Y),
        safe(X, Y, Rest).
newsquare([], square(1,X), Boardsize) :-
        snint(X, Boardsize).

snint(X, X).
snint(N, NPlusOneOrMore) :-
        M is NPlusOneOrMore - 1,
        M > 0, snint(N,M).

notthreatened(I, J, X, Y) :-
        I \== X, J \== Y,
        U1 is I - J, V1 is X - Y, U1 \== V1,
        U2 is I + J, V2 is X + Y, U2 \== V2.

safe(X, Y, []).
safe(X, Y, [square(I,J) | L]) :-
        notthreatened(I, J, X, Y), safe(X, Y, L).
```

# Example 3

/* Prolog program for *Discrete Fourier Transform* */

/* Top-level query *e.g.*

```
:- dft4([c(2,0), c(3,0), c(4,0), c(1,0)], Sol). */

      alternate([], [], []).
      alternate([A,B|T], [A|T1], [B|T2]) :- alternate(T, T1, T2).

      gval([H|T], 0, H).
      gval([H|T], Index, Val)  :-
             Index > 0, NewI is Index - 1,
             gval(T, NewI, Val).

      dftsplit(p([I], V), Val, _, Inp) :- gval(Inp, I, Val).
      dftsplit(p([A,B|T], V^P), Val, N, Inp) :-
             alternate([A,B|T], L1, L2),
             T1 is P * 2,
             P1 is T1 mod N,
             power(2, P, VP, 1),
             $andmax(2,
                     [ dftsplit(p(L1, V^P1), A1, N, Inp),
                       dftsplit(p(L2, V^P1), A2, N, Inp)] ),
             cmult(c(VP, 0), A2, Tem1),
             cadd(Tem1, A1, Val).

      convert([], []).
      convert([H|List], [c(H, 0)|ComList]) :-
             convert(List, ComList).

      cadd(c(A, B), c(C, D), c(X, Y)) :-
             X is A + C, Y is B + D.

      cmult(c(A, B), c(C, D), c(X, Y)) :-
             T1 is A * C, T2 is B * D * (-1),
             X is T1 + T2,
             T3 is A * D, T4 is B * C,
             Y is T3 + T4.

      power(Base, 0, 1, _).
      power(Base, 1, Ans, Acc) :-
             Ans is Acc * Base.
```

```
power(Base, P, Ans, Acc) :-
       P > 1, NewAcc is Acc * Base,
       P1 is P - 1, power(Base, P1, Ans, NewAcc).

ex(4, Inp, P, X) :-
   dftsplit(p([0,1,2,3],w^P), X, 4, Inp).

dft4(Inp, [X0,X1,X2,X3]) :-
  $andmax(4, [ ex(4,Inp,0,X0), ex(4,Inp,1,X1),
               ex(4,Inp,2,X2), ex(4,Inp,3,X3)]).
```

## Example 4

/* Prolog program for the *Map Colouring Problem* */

/* Top-level query `:- map(A,B,C,D).` */

```
map(A, B, C, D) :-
       $andmax(5, [ next(A, B), next(B, C), next(A, C),
                    next(C, D), next(B, D) ]).

next(X, Y) :- colour(X, Y).
next(X, Y) :- colour(Y, X).

colour(green, red).
colour(blue, red).
colour(blue, green).
```