The Aurora Abstract Machine and its Emulator

Mats Carlsson
Swedish Institute of Computer Science
PO Box 1263, S-16428 KISTA, Sweden

Péter Szeredi
Department of Computer Science
University of Bristol
Bristol BS8 1TR, U.K.

Draft version: 19 October 1995

The Aurora Abstract Machine and its Emulator

Mats Carlsson
Swedish Institute of Computer Science
P.O. Box 1263, S-16428 KISTA, Sweden

Péter Szeredi
Department of Computer Science
University of Bristol
Bristol BS8 1TR, U.K.

Abstract

Aurora is a prototype or-parallel implementation of the full Prolog language for shared-memory multiprocessors, developed as part of an informal research collaboration known as the "Gigalips Project". This report describes the abstract machine of the implementation, expressed in terms of a Prolog engine adapted for or-parallel execution by means of the SRI model. An algorithmic interface defining the communication between the engine and scheduler components of each Aurora process is described, and enables different engine and scheduler components to be used interchangeably. Both the interface and the engine are fundamentally revised versions of those used in previous generations of the implementation.

1. Introduction

This report documents the Prolog engine used in the second generation of the Aurora Or-Parallel Prolog implementation by SICS, University of Bristol, and Argonne National Laboratory. The working name of this implementation has been "Foxtrot", to distinguish it from its precursor implementations "Charlie", "Delta" and "Echo".

Aurora is an implementation of the SRI model for or-parallel execution of Prolog for shared-memory multiprocessors [Warren 87b]. From the outset, the goal has been to make Aurora a practical Prolog system which can run significant, existing applications, with little or no change in the application programs. Consequently, Aurora must support the *full* Prolog language, including side-effect operations, with the same semantics as when executed sequentially.

Our approach to or-parallelism is to let the implementation exploit any available parallelism in the application program, without requiring special annotations by the user. In other words, the or-parallel implementation is simply seen as a "Prolog accelerator" which should speed up programs with some degree of parallelism in them, and should certainly not cause any programs to run much slower.

The Aurora implementation comprises two main components: the *engine* and the *scheduler*. The two components form a *worker*, where the engine performs the actual Prolog work and the scheduler divides the available work between engines. The communication between the engine and the scheduler is defined by a semi-formal algorithmic *interface*, in terms of operations provided by the scheduler for the engine and vice versa, as well as administrative issues such as file name conventions for combining the scheduler code and the engine code. The interface was produced by Péter Szeredi and appears herein as a separate chapter. See chapter 3, page 8.

The Aurora/Foxtrot engine is based on SICStus Prolog 0.6 [SICS 88], a portable implementation of the Warren Abstract Machine (WAM) for Prolog, with extensions for the SRI model of or-parallel execution. This report focuses on these extensions and provides details about the implementation of the abstract instruction set and of the interface services provided to the scheduler. The earlier Aurora implementations were based on SICStus Prolog 0.3, which executes about 20% slower than version 0.6.

Other Aurora components include a compiler, implemented in Prolog and documented elsewhere [Carlsson 90], and a large runtime system which implements the built-in predicates of Aurora, implemented in C and Prolog. Several schedulers have been produced [Butler et al. 88], [Calderwood and Szeredi 89], [Szeredi 89], [Brand 88].

The rest of this report is organised as follows. Chapter 2 contains a synopsis of the SRI Model. Chapter 3 defines the interface between the scheduler and the engine. Chapter 4 describes the

storage model, on three levels of abstraction: how a number of stacks are organised into a search tree which can be explored in parallel, how the various objects of the abstract machine are represented, and finally how each stack is built up from a linked list of physical memory blocks. Chapter 5 describes the emulator, with emphasis on how operations such as backtracking, pruning and certain instructions are extended from the underlying sequential implementation. This chapter also deals with the machinery implementing straightening, contraction and suspension, new concepts introduced by the SRI model. Chapter 6 gives semantics for the abstract instruction set i.e. for an extended WAM instruction set. Chapter 7 treats certain implementation details which have been omitted from previous chapters, to prevent the various descriptions from becoming too complex. The report ends with a short concluding chapter.

The semantics of the abstract machine instructions and basic operations will be given in pseudocode, similar to the C programming language. See the Appendix for a synopsis of the pseudocode language. The discussion about text macros and their semantics is of special importance, since many operations are defined as such. Throughout the report, code is written using typewriter font, whereas syntactic variables are written in a slanted font. We sometimes use the notation *Prefix*...*Suffix* as an abbreviation for a set of function names.

The descriptions of the instructions will closely follow the actual emulator code. However, a number of optimisations have been omitted from this exposition in order not to obscure the presentation.

Section 2.1 is materially a synopsis of [Warren 87b]. Szeredi wrote Chapter 3; the remaining parts were written by Carlsson.

2. The SRI Model

In [Warren 87a] a progression of models for or-parallel execution of Prolog is developed from first principles. A later paper, [Warren 87b], selects one of these models, the SRI model, as the most promising one for an actual implementation and discusses the model in some detail. In this chapter, we summarise the SRI model and describe how the actually implemented model, which we call the Aurora model, differs in some details from the original SRI model.

2.1 The Pure SRI Model

This section is quoted from Section 2 through 2.1 of [Warren 87b]. We call the described model the *pure* SRI model, as it omits certain extensions, discussed in later sections of that paper.

"In the SRI model, ..., a state of computation is viewed abstractly as a *tree*, comprising *nodes* and *arcs*. Each node is labelled with a *task* consisting of a triple *(C,G,B)* where C is a list of *clauses*, G is a list of *goals*, and B is a list of *bindings*. The meaning of the task is 'try to solve the goals G in the context of bindings B using clauses C as candidates for matching the first goal of G'. The initial *root* node is labelled with *(C0,G0,[])*, where *G0* is the initial query, *C0* is the list of clauses that are candidates for matching its initial goal, and `[]` is the empty list.

"The principal operation which grows the tree is *extension*, which is essentially just classic resolution. Extension operates on a node reducing its task *(C,G,B)* to the task *(C0,G,B)*, where *C0* is simply C minus its first clause. At the same time, extension adds a new node labelled with the new subtask *(C1,G1,B1)* as the last child of the reduced node. The head of the first clause of C is *unified* with the first goal of G in the context of bindings B. *G1* is G with its first goal replaced by the new goals from the body of the matching clause instance. Any bindings to variables in G are added to the front of B to give *B1*. *C1* is the list of clauses that are candidates for matching the first goal of *G1*. If the match fails, no new node is added, but C is still reduced to *C0*.

"The object of growing the tree is to compute a set of *solution* nodes. A solution node is one where the goal list has been reduced to an answer, which one can think of formally as a singleton goal for a special predicate, ..., with no matching clauses. When a solution node is found, the answer is notionally added to a set of answers held at the root node. ...

"We assume computation is performed by *workers*. ... In the SRI model, each worker has a *binding array* to give immediate access to the bindings it is currently working with. The binding array serves to preserve the important property of sequential systems that all basic unification steps are fast, constant-time operations. The binding array mirrors the contents of the binding list of the particular node that the worker is working at. As workers move up and down the tree, they modify

their binding arrays incrementally, using the difference between the binding lists of the nodes they are moving from and to.

"The extension operation is all that is necessary to grow the tree and complete the computation. However, since all we are interested in ultimately are the solutions generated, it is possible to prune and simplify the tree while it is being grown. This is important to minimise the space the tree occupies. It also helps to streamline the extension operation and make it easier for the workers to move about the tree.

"The central idea is to retain only those parts of the tree and associated structures that are relevant for generating further solution nodes. A key concept is that of a dead node. We call a node *dead* if its clause list is empty; otherwise it is *live*. A live node is one at which there is work available. A dead node is one where there is no work.

"As soon as a node becomes dead, its goal list is of no further interest and can be logically discarded. Physically this means one can discard any part of the goal list ... that is not being physically shared. This is the main concept underlying *tail recursion optimisation* and *garbage collection* in sequential Prolog systems. In addition, the dead node itself may be removed provided there is at most one arc below it, i.e. provided it is not a *fork* node. When the dead node is removed, its binding list can be logically discarded. In practice, this means part of the binding list can be physically discarded if there is nothing below the dead node. Dead node removal is the main concept underlying *backtracking* and *optional choicepoints* (cf. the WAM *trust* operation) in sequential Prolog systems.

"Dead node removal is related to the concepts of branchpoint and of conditional and unconditional bindings. A *branchpoint* is a node that is either live or a fork. A binding is *conditional* if the variable in question is shared or potentially shared with another branch of the tree, i.e. there is a branchpoint at or below the point where the variable was created and above the point of binding. Otherwise the binding is *unconditional*. An unconditional binding does not need to be recorded in the binding list. Instead it can be recorded by physically replacing the variable with its bound value (implemented in practice by assignment to a variable value cell). A binding may be initially conditional and therefore recorded in the binding list, but may later become unconditional. Such bindings can in principle be *promoted* by removing them from the binding list and substituting the variable. In the SRI model, promotion is complicated by the need to remove promoted bindings from the binding array as well.

"We encapsulate the opportunities for simplifying the tree in three operations: dieback, contraction and straightening, which are performed in addition to the extensions which constitute the real work.

"The *dieback* operation removes a dead node from the tip of a branch. It corresponds to backtracking in sequential Prolog. It discards those parts of the goal list and binding list that are not inherited from the node above.

"The *contraction* operation removes a dead node from the interior of the tree, and takes place when there is an extension on the last clause for a nonfork node. It corresponds to the WAM 'trust' operation. It allows the first goal of the dead node to be discarded, and permits certain bindings to be made unconditional.

"The *straightening* operation also removes a dead node from the interior of the tree, and occurs when there is a dieback to a dead fork leaving just one other branch. It is in many ways similar to a 'cut' in a sequential Prolog system. Besides discarding the dead node, it allows some number of goals from the dying node to be logically discarded, namely those that are not inherited by the node below. It also allows some bindings on the node below to be promoted, namely those that are not inherited from the dead node and which refer to variables created after the node above the dead node."

2.2 The Aurora Model

The actually implemented model closely follows the above specification, but some discrepancies do exist. As usual we think of the search tree as growing downwards.

In the pure SRI model, the tree is homogeneous, any node is potentially sharable, and workers cooperate in exploring the search tree in a completely distributed and symmetric fashion by the principal tree operations. However, a totally unleashed parallel model would be inefficient due to the small grain size that would result. In order to have a reasonably efficient execution strategy, it becomes necessary to divide the tree into two parts, a *public part* above i.e. starting at the root of the tree, and a *private part* below. The public part consists of fork and nonfork nodes connected into a tree. The private part consists of linear chains of nodes, each chain being rooted in the public part. A chain is either *suspended*, as described below, or *active*, i.e. being explored by a worker operating as a sequential Prolog engine. The number of workers is an upper bound on the number of active branches.

As a device for manually restricting the parallelism, predicates can be annotated as *sequential* or *parallel*. Nodes corresponding to choicepoints for such predicates inherit the annotation. A sequential node will never become a fork, i.e. its children will be explored sequentially. By default, predicates are parallel.

Private nodes can be either *local*, if they belong to a contiguous sequence of adjacent nodes on the top of the stack of the worker currently executing the given branch, or *remote* otherwise. Each branch is said to have a *public region*, a *remote region*, and a *local region*, corresponding to the

nodes they contain. The distinction between local and remote nodes arises from the mapping of the search tree onto a number of *stack groups*, each with a unique *owning worker*. A stack group consists of the *environment*, *node*, *global* and *trail* stacks, which are extensions of the corresponding WAM stacks.

The oldest private node in a branch is called the *sentry* node of the branch. The youngest local node of an active branch is an *embryonic* node. This node corresponds to the choicepoint to be created next, i.e. the choicepoint fields are undefined in this node. When the next choicepoint is created, and the embryonic node is fleshed out (the choicepoint fields filled in), a new embryonic node is created as the child of the previous one. In the rest of this report, there is no distinction between a node and a choicepoint; the two words merely stress different aspects of the same concept.

The exploration by a worker of an active branch constitutes that worker's *assignment*, which normally terminates if the branch dies back into the public part. It is worth noting that the boundary between the public and private parts can be dynamically adjusted downwards after the assignment was started. The assignment terminates prematurely if the branch is suspended, or if it is pruned by some other worker.

In all three cases, the worker must find a new assignment, either an alternative clause of a public node to explore, or a suspended branch to *resume*, if the cause for suspension has ceased to exist. Before the worker can proceed with the new assignment, it must *position* itself at the new assignment, incrementally removing bindings from the binding array as it moves up from its current position to a common ancestral node N, and incrementally adding bindings to the binding array as it moves down from N to the new position. The number of bindings made on the given path of movement is called the *migration cost*, since it is proportional to the updating overhead of binding arrays.

There are three *pruning operators* currently supported by Aurora: the conventional Prolog *cut*, which prunes all branches to its right and a symmetric version of cut called *commit*, which prunes branches both to its left and right. Both the cut and the commit operations are sensitive to being in the scope of a smaller cut, i.e. they will not go ahead if there is a chance of them being pruned by a cut with a smaller scope. The third type of pruning operator is the *cavalier commit* which is executed immediately, even if endangered by a smaller cut. The cavalier commit is provided for experimental purposes only, it is expected to be used in exceptional circumstances, for operations similar to `abort` in Prolog.

These operators may try to prune shared parts of the tree, in which case the scheduler is requested to prune those parts. Such requests can cause the current branch to suspend (if the current branch could be pruned by another operator with smaller scope [Hausman et al. 88]), or can be refused if the current branch is being pruned by another worker.

Related to pruning operators is the concept of *speculative* work. In the Aurora model, a branch which is in the scope of a cut is speculative, i.e. it is able to produce a solution only if the pruning operator is actually never executed. Thus speculative work is likely to be less useful than nonspeculative work.

There are two distinct reasons for suspension. First, suspension is used to preserve the *observable semantics* of Prolog programs executed by Aurora: when a built-in predicate with some side-effect is reached on a non-leftmost branch of the search tree, and when a pruning operator is reached on a branch which could be pruned by another operator with smaller scope, the execution needs to be suspended. Furthermore the scheduler can decide to suspend the current branch when it believes less speculative work can be done somewhere else in the search tree. The suspension mechanism is the same for the two cases: the suspending worker stores the state of the execution at the tip of the branch and moves away to find another assignment. The suspended branch will be resumed when the cause for suspension ceases to exist, unless it is pruned before that.

3. Interface between Engine and Scheduler

This chapter defines the algorithmic interface (version 2.13) between the engine and the scheduler components of Aurora. The definition does not fully cover some aspects of the engine/scheduler interface within Aurora, e.g. those related to debugging and performance analysis. This definition is a fundamentally revised version of the previous interface (version 1.04) [Calderwood 88].

3.1 An Overview of the Interface

The principal duty of the scheduler is to provide the engine with work, i.e. Prolog code to be executed. The thread of control thus alternates between the two components: the engine executes a piece of Prolog code, then the scheduler finds the next piece of code to be run, passes control back to the engine, etc. A natural way of implementing this interaction is to put the scheduler *above* the engine: the scheduler *calls* the engine when it finds a suitable piece of work to be executed and the engine *returns* when such an assignment has been finished. In fact this scheme was the basis of earlier interfaces in Aurora [Calderwood 88].

In the current version of Aurora a different approach is used. The execution is governed by the engine: whenever it runs out of work (finishes an assignment), it calls an appropriate scheduler function to provide a new piece of work. The advantage of this scheme is that the environment for Prolog execution (e.g. the set of WAM-registers) is not destroyed when an assignment is terminated and need not be rebuilt on returning to work. This is of special importance for Prolog programs with fine granularity (i.e. small assignment size), where switching between engine and scheduler code is very frequent.

Figure 3-1 shows the top view of the current interface. This is centered around the engine doing work. All the other boxes in the picture represent scheduler functions called by the engine. Note that the names of all scheduler functions are prefixed with `Sched_`, as a convention.

The functions shown in Figure 3-1 are arranged in three groups:

- finding work (left side of Figure 3-1);
- communication with other workers during work (lower part of Figure 3-1), e.g. when cuts or side effect predicates are to be executed;
- certain events during work that may be of interest to the scheduler (right side of Figure 3-1), e.g. creation of nodes.

Other scheduler functions, not listed above, are needed for some special events (e.g. user interrupts, initialisation and closing down of the whole Aurora system).

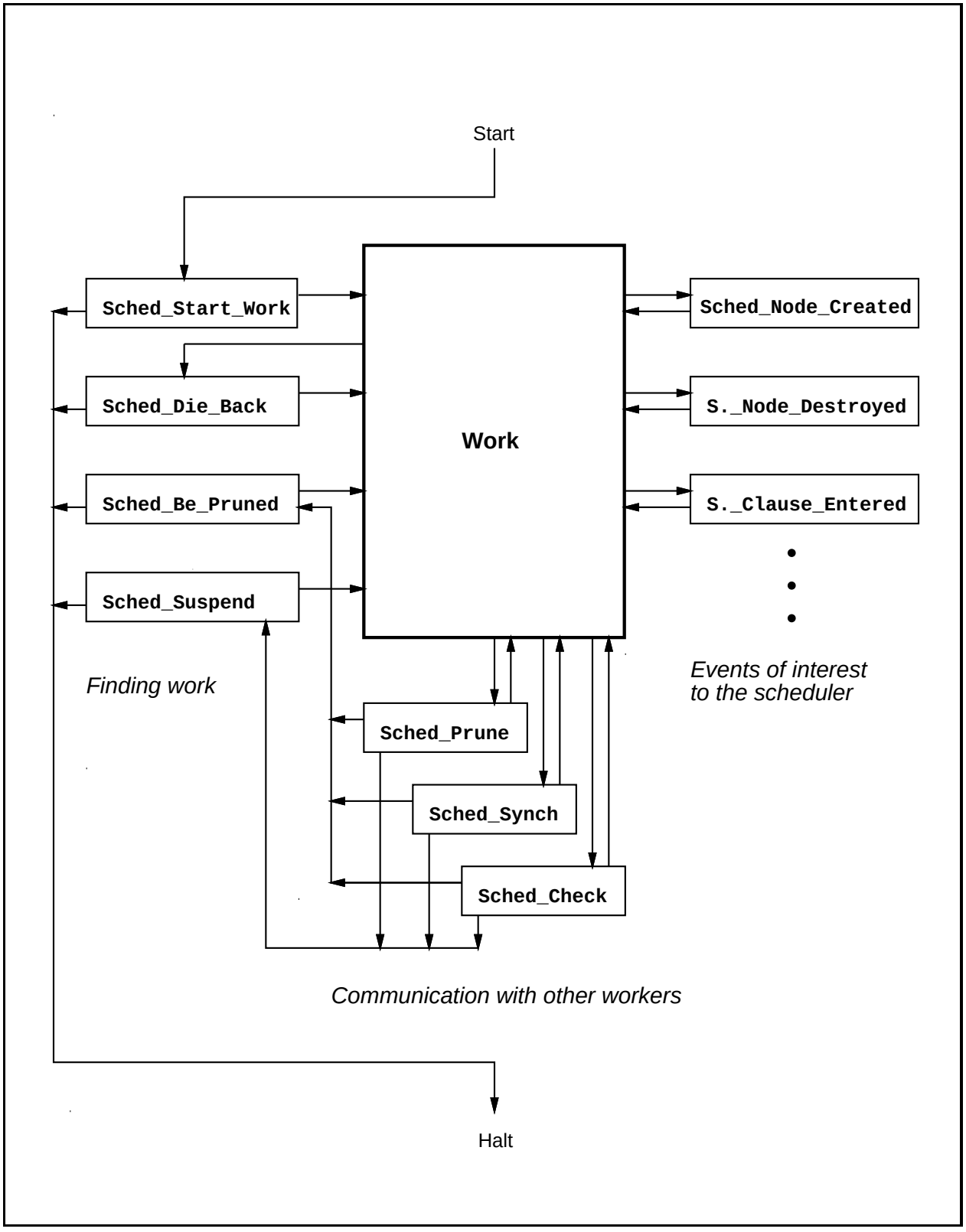

Figure 3-1: Top level view of the interface

The four boxes on the left of Figure 3-1 represent the so called *functions for finding work*:

`Sched_Start_Work`
: is used to acquire work for the first time, immediately after the initialisation of the worker;

`Sched_Die_Back`
: is called when the engine dies back to a public node;

`Sched_Be_Pruned`
: is invoked when the worker is killed by another one;

`Sched_Suspend`
: is called when the worker has to suspend its current branch.

These functions will differ in their initial activities, but normally will continue with a common algorithm for "looking for work". This algorithm has two possible outcomes: either work is found, or the whole system is halted. Correspondingly each of the functions for finding work has two exits: the normal one (shown on the right side of the function boxes in Figure 3-1) leads back to work, while the other exit (left hand side) leads to the termination of the whole Aurora invocation.

See section 5.1, page 63 and section 5.4, page 74 for a description of how the functions for finding work fit into the engine's backtracking logic.

The next group of interface functions provided by the scheduler is depicted at the bottom of Figure 3-1. These functions are called during work, when the engine may require some assistance from the scheduler:

`Sched_Prune`
: —when a cut or commit is executed (see section 5.3, page 71);

`Sched_Synch`
: —when a predicate with a side-effect is encountered;

`Sched_Check`
: —at every Prolog procedure call (see section 6.6, page 99).

The frequent invocation of `Sched_Check` is necessary so that the scheduler can answer requests (e.g. interrupts) from other workers without too much delay. Note however that a scheduler may choose to do the checks only after a certain number of `Sched_Check` invocations.

The above functions have three exits. The normal exit (depicted by upwards arrows in Figure 3-1) leads back to work. The other two exits correspond to premature termination of the current assignment, when the current branch has been pruned or has to suspend (left and downwards arrows). In both cases the engine will do the housekeeping operations necessary for the given type

of assignment termination, and proceed to call the scheduler via the appropriate function for finding work.

The third group of functions shown in Figure 3-1 (right hand side) corresponds to some engine activities during work that may be of interest to the scheduler. As an example, `Sched_Node_Created` (and the corresponding `Sched_Node_Destroyed`) can be used to keep track of the presence of parallel nodes in the private region—as a prospective source of work for other workers. Similarly `Sched_Clause_Entered` can be utilised for maintaining information about the presence of pruning operators in the current branch (See section 3.5, page 28).

Now let us turn to the other side of the interface: the functions supplied by the engine for use in the scheduler.

The engine is responsible for maintaining the node stack, a principal data area of major importance to the scheduler. The engine defines the node data type, but the scheduler is expected to supply a number of fields to be included in this structure for its own purposes.

Among the node fields defined by the engine some are of interest to the scheduler. Access functions for these fields (`Node_Level`, `Node_Parent`, `Node_Alternatives`) are provided in the interface.

The scheduler specific fields of the node data structure normally include pointers describing the topology of the tree. For example, most schedulers will have fields storing a pointer to the first child and the next sibling of a node.

The engine keeps track of the current sentry node during work. The scheduler is required to supply a new sentry node to the engine, when returning from functions for finding work (see below).

The engine provides functions for the memory management of the node stack. The engine needs assistance in the reclamation of public and sentry nodes: the scheduler has to call the `Mark_Node_Reclaimable` function when a particular node is known to be not used any more. A consecutive sequence of reclaimable nodes on the top of a worker's stack gets physically reclaimed when this worker starts a new assignment and executes an `Allocate_Node` function (see below).

The engine supplies a number of functions to be used by the scheduler in specific circumstances. Figure 3-2 shows the engine's involvement in the functions for finding work.

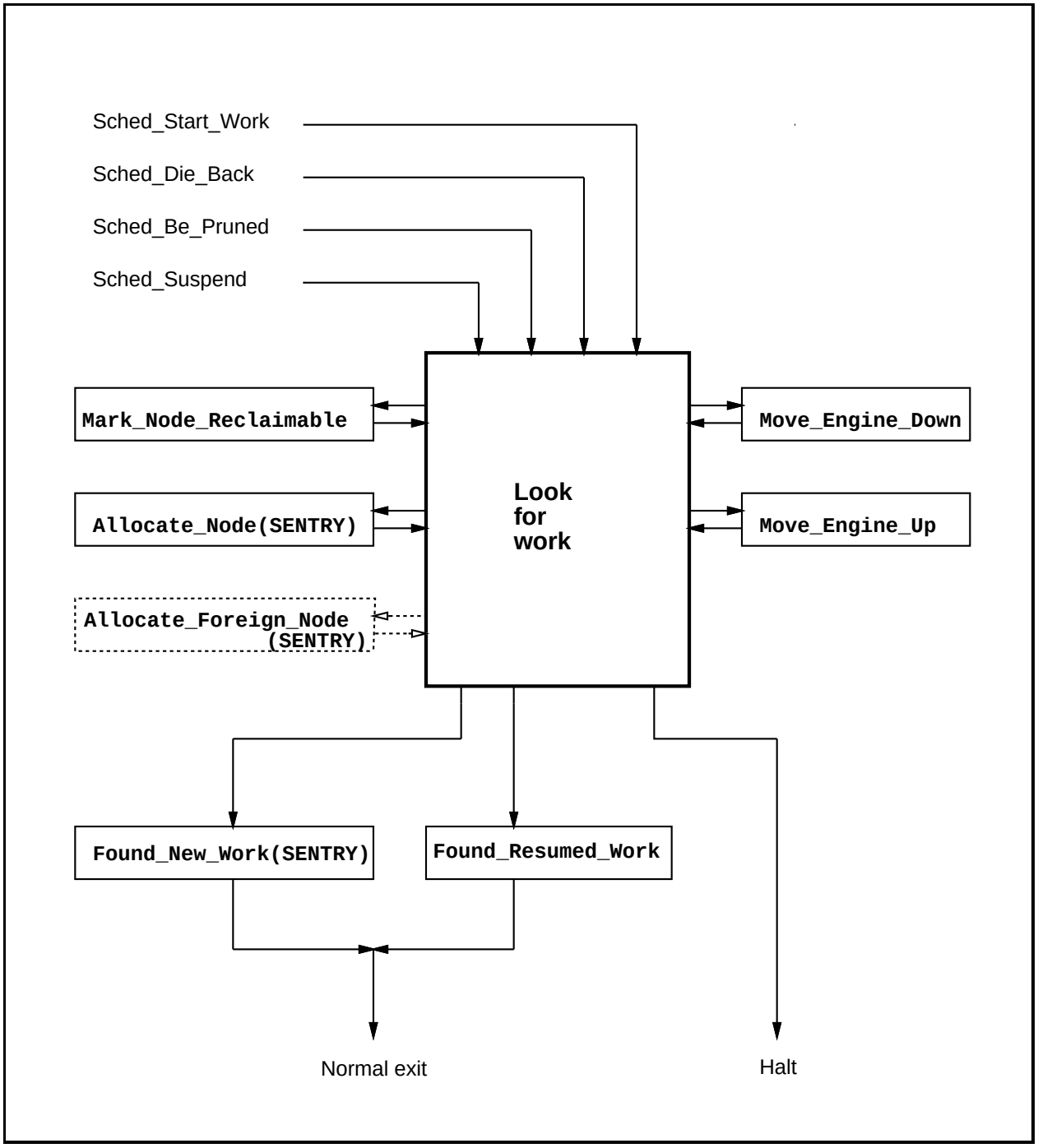

Figure 3-2: Engine functions used in looking for work

On entry to these functions, the engine (i.e. the binding array) is positioned at or below the youngest public node on the branch. While looking for work, the scheduler can ask the engine to move to a new position. Functions `Move_Engine_Up` and `Move_Engine_Down`, shown on the right hand side of Figure 3-2, update the binding array to correspond to a new position up or down the current branch. Before returning to the engine, the scheduler has to position the binding arrays above the new sentry node.

The left hand side of Figure 3-2 shows the engine functions for memory management of the node stack. While looking for work, the scheduler may use the `Mark_Node_Reclaimable` function. When the scheduler decides to reserve a new piece of work from a live public node (work node), it has to create a sentry node for the new branch. This is done using the `Allocate_Node` function, which first removes all the nodes that have been marked as reclaimable from the top of the worker's stack and then allocates a new sentry node.

The new sentry node serves as a placeholder for the new assignment. The scheduler reserves an alternative from the work node (by reading and advancing the `Node_Alternatives` field of the node) and at the same time inserts this node into the search tree (into the sibling chain below the work node).

In some of the schedulers a worker may wish to hand a piece of work to another worker (which is actually idle, i.e. looking for work). The `Allocate_Foreign_Node` function makes it possible to allocate a node on another worker's stack. This function is shown in a dotted box in Figure 3-2 to draw attention to the fact that it is actually invoked by another worker, i.e. the worker who wants the piece of work reserved. The reserving worker will, of course, notify the idle worker that work has been reserved and that it has to return to the engine (but this is an internal matter for the scheduler, with no interface implications).

The bottom part of Figure 3-2 shows the possible exit paths from the functions for finding work. The actual work found can correspond either to a new branch or to a branch which was hitherto suspended and can be resumed now. Functions `Found_New_Work` and `Found_Resumed_Work` are used to notify the engine about the type of the work found, and to supply the new sentry node. The box for `Found_New_Work` in Figure 3-2 shows the `SENTRY` argument to highlight the fact that this argument should be the same as the one returned in `Allocate_...Node`.

Finally, let us discuss a specific aspect of the interface related to the contraction and straightening operations. When such an operation, involving the removal of a dead node from the tree, is applied to a non-local (i.e. remote or public) node, it needs to be implemented by *bypassing* the dead node, i.e. deleting the dead node from the parent-child chain. Note that the segments in the other stacks that correspond to a bypassed node are still needed until the given branch of the tree finally dies back. This means that the original value of the `Node_Parent` field has to be preserved by the engine for use during backtracking, and a separate *bypassed parent* field is required in the node data structure if bypassing is to be implemented.

As bypassing in the public region of the tree requires synchronisation, it has been decided that the scheduler will be responsible for the implementation of this operation. The engine notifies the scheduler when a (remote) private node becomes dead (`Sched_Node_Destroyed`), so that the scheduler can perform the bypassing operation in the remote region as well. On the other hand, the scheduler will call the `Mark_Node_Bypassed` function, when a node ceases to be accessed by the scheduler due to a bypassing operation being performed.

An issue related to bypassing is that of the implementation of the `Node_Parent` field of nodes. The engine offers two alternative implementations of `Node_Parent`, depending on a static switch `SCHED_WANTS_COMPACT_PARENT`: a faster one (the default) and a more compact one. If a scheduler does perform bypassing, it will use the engine's `Node_Parent` field only to initialise its own bypassed parent field, so it will normally select the compact representation. On the other hand, non-bypassing schedulers will normally use the default, fast implementation.

Contraction proper is supported in the interface by the function `Sched_Get_Work_At_Parent`. This function contracts the public region by one node if it takes the last alternative from a public node and certain other conditions are met.

See section 5.5, page 78 for details about the engine's compact-parent configuration.

The rest of this chapter gives a detailed description of the interface. In section 3.2, page 14, the data types common to the engine and scheduler are described. In section 3.3, page 16 and section 3.4, page 25, the set of functions defined by the scheduler and the engine, respectively, are presented. Note that all functions are implemented as C macros, hence we will use the term "macros" rather than "functions" in the sequel. In section 3.5, page 28, a description is given of the data related to static properties of clauses that is supplied to the scheduler by the engine.

3.2 Common Data Types

3.2.1 Nodes

The principal data structure common to the scheduler and the engine is the `node` structure:

`struct node`

> This is an extension of the WAM choice point. The fields of this data structure are notionally split into two parts: a scheduler part and an engine part (see section 4.2.3, page 42). There are no common fields in the nodes, access by the scheduler to engine fields being entirely via engine-provided macros and vice versa.

The actual merging of the engine specific and scheduler specific fields is implemented as follows. The engine is responsible for declaring the `struct node` data type and the scheduler supplies a file, '`sch.node.h`', which contains the scheduler specific fields to be included in nodes.

The engine provides the following principal macros for node access:

`int Node_Level(struct node *NODE)`

This macro returns the *level* of `NODE`. The level numbers form a sequence which strictly increases with the distance from the root. This field is filled in by the engine in all nodes, including embryonic nodes (cf. `Allocate_Node`).

`struct node *Node_Parent(struct node *NODE)`

This macro evaluates to the parent field of `NODE`, holding a pointer to the parent of `NODE`. This field is filled in by the engine in all nodes, including embryonic nodes (cf. `Allocate_Node`).

`struct alternative *Node_Alternatives(struct node *NODE)`

This macro evaluates to the *next alternative* field of `NODE`. This field is maintained by the engine in non-embryonic private nodes to hold a pointer to the next unexplored alternative (or `NULL`, if the node is dead). The `Node_Alternatives` field is not meaningful in embryonic nodes.

The scheduler should preverve the meaning of this field in public nodes, updating `Node_Alternatives` when an alternative is reserved, or when a branch is pruned. Correspondingly the macro `Node_Alternatives(NODE)` can appear on the left hand side of an assignment, i.e. it is an *lvalue*.

3.2.2 Alternatives

An additional common static data structure, the `alternative` is introduced to allow the schedulers to keep static data related to clauses.

`struct alternative`

This data structure replaces the '`try`', '`retry`' and '`trust`' WAM instructions. Each clause of the user program is represented by an alternative structure, which stores a pointer to the code of that clause and a pointer to a successor alternative, if any. If a predicate is subject to indexing, there may be several chains of alternatives corresponding to different instantiations of the indexing argument position, and so several alternatives can refer to the same clause. See section 4.2.7, page 52.

The scheduler-related part of `struct alternative` can accommodate any (static) information the scheduler wishes to associate with clauses. The scheduler can derive this data from the information supplied by the engine in `Sched_Alternative_Created` (See section 3.5, page 28). Currently there are two types of static data supplied by the engine:

- information about sequential predicates—a predicate can be declared sequential by the user, to prohibit parallel exploration of branches corresponding to the predicate; this information is normally stored in each alternative of the predicate.

- pruning information—data on the number of pruning operators (cuts, commits and conditional expressions) contained in the clause or the predicate.

The engine is responsible for declaring the `struct alternative` data type and the scheduler supplies a (possibly empty) file, '`sch.alternative.h`', which contains the scheduler specific fields to be included in alternatives.

The only engine field in the alternative structure that is of interest to the scheduler is the following:

`struct alternative *Alternative_Next(struct alternative *ALT)`
: This macro returns a pointer to the alternative following `ALT` in the chain of alternatives.

3.2.3 Boolean Type

The engine defines the following macros to aid readability:

`BOOL` a type defined as `unsigned int`

`FALSE` a macro evaluating to 0

`TRUE` a macro evaluating to 1

3.3 Macros Provided by the Scheduling Code

The macros in this section have the current sentry node as their first argument, whenever this is meaningful. This means that the engine will always supply the current sentry node to the scheduler, so that the latter does not have to keep a variable pointing to the sentry node.

Macros marked with a dagger (†) need not necessarily be supplied by all schedulers. If such a macro is not defined by the scheduler, it is assumed to be empty (i.e a no-operation), unless noted otherwise.

3.3.1 Finding Work

This section presents the macros for finding work. As described in section 3.1, page 8, these macros have two exits: a normal one, when work is successfully acquired, and one for the termination of the whole Aurora run. Rather than to use return codes, the additional exit is implemented by having an extra argument, `IF_HALTED`, in each of the macros. The `IF_HALTED` argument contains a piece of code to be executed when the scheduler detects that the system has been halted—this code will normally be a branch instruction to transfer control to the appropriate place in the engine code.

The engine (i.e. the binding array) is positioned at or below the parent of the sentry node before calling the macros for finding work. The scheduler should position the engine above the new sentry, using the `Move_Engine_`... macros. The engine should also be notified about the type of work found and the new sentry node. This is done by calling either `Found_New_Work` or `Found_Resumed_Work` (see section 3.4.1, page 26) before returning from the macros for finding work.

`void Sched_Die_Back(struct node *SENTRY, IF_HALTED)`

This macro is called when the engine reaches a public node in the course of backtracking. The scheduler performs the dieback operation to the parent of `SENTRY`. This should involve marking the `SENTRY` node as reclaimable, following some scheduler-specific activities for the sentry node (e.g. deleting it from the sibling chain). Finally the scheduler proceeds to acquire a new assignment.

`void Sched_Suspend(struct node *SENTRY, IF_HALTED)`

This macro is invoked when a suspension request has been received from the scheduler and the engine has performed its housekeeping activities for suspension. The scheduler performs any outstanding activities for suspension and then proceeds to acquire a new assignment.

`void Sched_Be_Pruned(struct node *SENTRY, IF_HALTED)`

This macro is invoked after the worker noticed that it has been pruned and the engine died back to `SENTRY`. At this moment all the nodes up to and including `SENTRY` should be regarded as destroyed (the `Sched_Node...Destroyed` macros will *not* be called for these nodes). The scheduler continues dying back in the cut region (including marking the `SENTRY` node as reclaimable) and then proceeds to acquire a new assignment.

`void Sched_Start_Work(IF_HALTED)`

This macro is called in each engine immediately after starting the system. The scheduler will return as soon as the worker can acquire an assignment. For one of the workers this will happen immediately, the new work being created by reserving the only alternative from the root node. The other workers will have to wait in an idle loop until work is made available by this worker.

3.3.2 Communication with other workers

The following macros are invoked by the engine during normal Prolog work in situations when the scheduler needs to be consulted. During the execution of these macros the scheduler may decide that the current branch of execution should be abandoned, either because it has been cut, or because it must suspend. To allow the scheduler to communicate its decision, each of the macros has the following arguments:

`IF_PRUNED`
: the code to be executed when being pruned,

`IF_SUSPENDED`
: the code to be executed when suspended.

Both these arguments are normally branch instructions. On the `IF_PRUNED` branch the engine dies back to the sentry node and invokes the `Sched_Be_Pruned` macro. On the `IF_SUSPENDED` branch the engine does the housekeeping operations for the suspension and invokes `Sched_Suspend`.

`void Sched_Check(struct node *SENTRY, IF_PRUNED, IF_SUSPENDED)`
: This macro is called by the engine on every Prolog procedure invocation to allow the scheduling code to take care of any outstanding parallel business—e.g. releasing work to idle workers, processing interrupts, etc.

`void Sched_Prune(struct node *SENTRY, struct node *CUT_NODE,`
`int PRUNING_OP, IF_PRUNED, IF_SUSPENDED)`
: This macro is called when the engine has to execute a pruning operation up to (and including) `CUT_NODE`. The `PRUNING_OP` argument determines the type of the pruning operator:

$$\texttt{PRUNING\_OP} > 0 \rightarrow \textit{cut};$$
$$\texttt{PRUNING\_OP} = 0 \rightarrow \textit{cavalier commit};$$
$$\texttt{PRUNING\_OP} < 0 \rightarrow \textit{commit}$$

The engine has already executed the private part of the pruning operation prior to invoking this macro. The scheduler now executes the remaining, public part of pruning. If the scheduler decides that the pruning operation cannot be fully executed, `IF_PRUNED` or `IF_SUSPENDED` is invoked.

This macro is called even if only private nodes are to be pruned. This is to allow the scheduler to maintain data related to pruning operations.

Note that the above encoding of `PRUNING_OP` is actually done by the Aurora compiler, and that the absolute value of this argument is used by the engine (it indicates the number of live temporary WAM registers to be saved on suspension (see section 6.8, page 104)).

`void Sched_Synch(struct node *SENTRY, struct node *ROOT_OF_SUBTREE, BOOL LEFTMOST, IF_PRUNED, IF_SUSPENDED)`

This macro is used to support the execution of built-in predicates with side-effects. Normally such predicates are executed only when their branch becomes leftmost in the whole tree. There are, however, some predicates (e.g. those used to implement `bagof`), for which a weaker condition is enough: they can be executed if not endangered by a cut within a given subtree.

The `LEFTMOST` argument encodes the type of the check needed before the execution of the branch can continue. If `LEFTMOST` is `TRUE`, the branch should be leftmost in the subtree rooted at the `ROOT_OF_SUBTREE` node. If `LEFTMOST` is `FALSE`, the scheduler needs to ensure that the current branch is not endangered by a cut within the subtree rooted at the `ROOT_OF_SUBTREE` node. Note that a scheduler which does not keep track of pruning operators can just ignore the value of the `LEFTMOST` argument and always check that the branch is leftmost in the given subtree.

If the appropriate condition is satisfied, and the branch itself has not been pruned, then the scheduler returns normally (and the execution of the branch can continue). Otherwise either `IF_PRUNED` (if the branch has been pruned) or `IF_SUSPENDED` is invoked.

The value of the `LEFTMOST` argument is a compile time constant, i.e. it can be used in C directives for conditional compilation (`#if`), if required.

3.3.3 Events of Interest to the Scheduler

Schedulers may require to be informed of certain events occurring during the private execution phase, such as

- predicates and clauses being entered—this can be utilised for maintaining information about the presence of cuts endangering the branch being executed;
- nodes being created, reused and destroyed—this is used by the scheduler to keep track of any work (i.e. live parallel node) being present in the private part of the branch being executed, as well as for maintaining information about cuts;
- the parent field of a node being filled in or updated—this allows the engine to maintain a child pointer in the private region, if it wishes to do so;
- alternatives being created—to calculate and store any static information the scheduler may wish to use (See section 3.5, page 28);
- the whole system being halted.

3.3.3.1 Entering Predicates and Clauses

†`void Sched_Nondet_Pred_Entered(struct node *SENTRY)`

This macro is called at the very beginning of a nondeterministic predicate. The need for this macro arises from the fact that the shallow backtracking optimisation used in the Aurora engine (see section 4.2.3.2, page 45) involves delaying, and sometimes completely avoiding the creation of nodes. `Sched_Nondet_Pred_Entered` is provided to allow for any scheduler activities that need to be done prior to other macros (e.g. `Sched_Clause_Entered`) being invoked.

†`void Sched_Clause_Entered(struct node *SENTRY, struct alternative *ALT, BOOL FIRST)`

This macro is invoked when the engine enters a clause. The `ALT` argument can be used to access any static data (e.g. pruning information) associated with the clause being entered. The `FIRST` argument is `TRUE` if the clause entered is the the first element of an alternative chain, and is `FALSE` otherwise.

The value of the `FIRST` argument is a compile time constant, i.e. it can be used in C directives for conditional compilation (`#if`), if required.

3.3.3.2 Creating and Destroying Nodes

†`void Sched_Node_Created(struct node *SENTRY, struct node *NODE)`

This macro is called whenever a proper (non-embryonic) `NODE` is created by the worker ('`try`' WAM instruction). Immediately prior to the invocation of this macro, an embryonic node will have been created as a child of `NODE` (and `My_Embryonic_Node()` points to that node). This means that the scheduler can initialise fields both in `NODE` and its embryonic child, if it wishes to do so.

†`void Sched_Node_Reused(struct node *SENTRY, struct node *NODE)`

This macro is called when the engine backtracks to `NODE` and is going to take a non-last alternative from it ('`retry`' WAM instruction). The macro is called before the next alternative pointer of the node (`Node_Alternatives`) is advanced.

†`void Sched_Node_Destroyed(struct node *SENTRY, struct node *NODE,`
`BOOL TO_BE_REMOVED)`

This macro is called whenever a private `NODE` is to become dead due to the last alternative being taken ('`trust`' WAM instruction). When the macro is called, the next alternative field of the node is still pointing to the last alternative. This may be important if the scheduler wishes to access information stored in the alternatives.

The `TO_BE_REMOVED` argument is `TRUE` if the engine will remove the `NODE` in question from the tree, and is `FALSE` if the dead node will have to remain in the tree (the latter is normally the case for remote nodes). The scheduler may wish to bypass the `NODE` if `TO_BE_REMOVED` is `FALSE`.

The value of the `TO_BE_REMOVED` argument is a compile time constant.

†`void Sched_Nodes_Destroyed(struct node *SENTRY, struct node *FROM, struct node *UP_TO, BOOL TO_BE_REMOVED)`

This macro is called whenever a sequence of private nodes (from `FROM` up to and including node `UP_TO`) is to become dead because the worker itself executed a pruning operator. If a worker is cut by another worker, this macro will not be invoked.

The `TO_BE_REMOVED` argument is `TRUE` if the engine will be able to remove the node `UP_TO` from the tree, and is `FALSE` if the dead node will have to remain in the tree.

The value of the `TO_BE_REMOVED` argument is a compile time constant.

The invocation order of the node creation and clause entry macros is affected by the shallow backtracking optimisation. As creation of nodes is delayed, `Sched_Node_Created` will be called *after* the first `Sched_Clause_Entered`, while `Sched_Node_Reused` and `Sched_Node_Destroyed` will be called *before* the corresponding macro for clause entry.

The graph in figure 3-3 gives the invocation order of the relevant macros for a specific (nondeterministic) predicate, assuming the whole predicate is executed (no pruning takes place). Initial and final states are marked by in- and outgoing arrows:

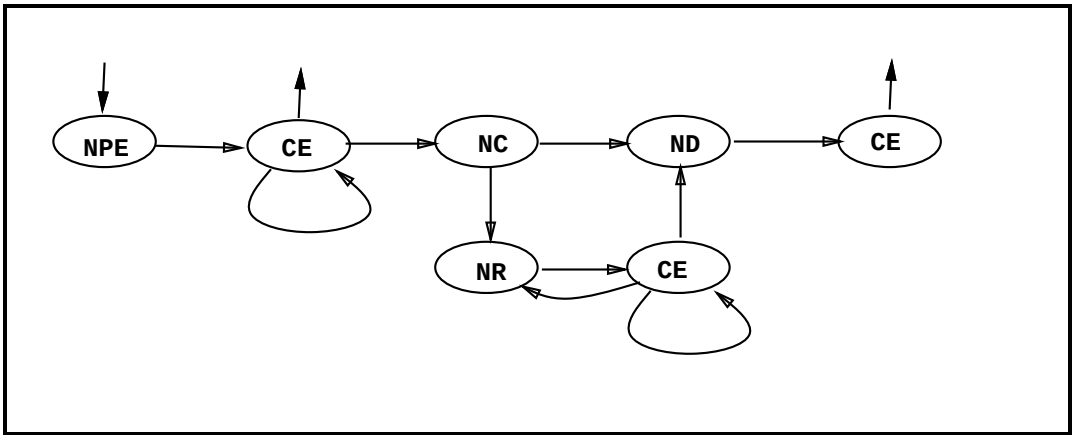

Figure 3-3: Order of events related to node creation and destruction

where

```
NPE = Sched_Nondet_Pred_Entered
CE  = Sched_Clause_Entered
NC  = Sched_Node_Created
NR  = Sched_Node_Reused
ND  = Sched_Node_Destroyed
```

3.3.3.3 Other Events

The following macros are to notify the scheduler of miscellaneous events:

†`void Sched_Halt(struct node *SENTRY)`
: This macro is called when the `halt` built-in is processed.

†`void Sched_Alternative_Created(struct alternative *ALT,`
`struct alternative *FIRST_ALT,`
`BOOL IS_PARALLEL,`
`int MAX_CUTS, int MAX_COMMITS,`
`int ASSUMED_CUTS, int ASSUMED_COMMITS,`
`BOOL IS_INTERNAL, BOOL CONTAINS_IF, BOOL ENDS_IN_A_FAIL)`
: This macro is called whenever a new alternative `ALT` is created by the Aurora compiler back-end. `FIRST_ALT` is the first alternative of the chain to which `ALT` belongs. The scheduler may use the `Alternative_Next` engine macro to scan and update the whole chain, if it wants to keep some global piece of information (relating to the whole chain) up to date. The last eight macro arguments carry the static information for the clause corresponding to this alternative. See section 3.5, page 28 for the description of this data.

†`void Sched_Private_Parent_Stored(struct node *SENTRY,`
`struct node *NODE, struct node *PARENT)`
: This macro is called whenever the engine stores or updates the parent field of a private `NODE` to point to `PARENT`, provided this `PARENT` node is itself private. This is useful if a scheduler requires a child field to be kept up to date in private nodes: each time `Sched_Private_Parent_Stored` is called, the scheduler should update the child field of `PARENT` to point to `NODE`.

3.3.4 Optimisations

This section describes two macros related to specific optimisations.

3.3.4.1 Backtracking to a live public node

When a worker backtracks to a live public node and is able to take a new branch from there, several administrative activities can be avoided. The sentry node can be re-used, instead of being marked as reclaimable and re-allocated as the new embryonic sentry node. There is scope for a related optimisation in the scheduler administration: rather than deleting the old sentry from the sibling chain and then installing it as the last sibling, one can move the sentry node to the end of the sibling chain (or do nothing if the old sentry was the last child). To allow this optimisation

to be applied, the engine calls the following macro prior to `Sched_Die_Back`, if the parent of the sentry node is live (see section 5.2.2.3, page 69).

†`void Sched_Get_Work_At_Parent(struct node *SENTRY, struct node *PARENT, struct alternative *ALT, BOOL MAY_CONTRACT, IF_CONTRACTED)`

This macro is called when the engine backtracks to a live public node `PARENT`. If the scheduler, following the necessary synchronisation operations, still finds `PARENT` to be live, it can reserve an alternative from that node. The alternative reserved should be returned in the `ALT` output argument. `SENTRY` is the current sentry node when this macro is called, and it is assumed to become the sentry for the new branch (unless contraction is applied, see below). If the scheduler cannot (or does not wish to) take work from `PARENT`, it should set `ALT` to `NULL` and return. In this case `Sched_Die_Back` will be called to acquire a new piece of work.

The `MAY_CONTRACT` argument shows whether the engine allows for the contraction operation to be applied (i.e. whether `PARENT` and `SENTRY` are on the top of the stack of the worker executing this macro). If this is `TRUE`, and the last alternative is being taken from `PARENT`, then the scheduler may decide to use `PARENT` rather than `SENTRY` as the new embryonic node. In this case `IF_CONTRACTED` should be executed just before exiting the macro.

If this macro is not supplied by the scheduler, the `ALT` = `NULL` result is assumed.

If contraction is applied, the engine treats this as if the public node had been made private before the last alternative was taken. This results in the macro `Sched_Node_Destroyed` being invoked (see section 3.3.3.2, page 20).

3.3.4.2 Bypassing

If a scheduler implements bypassing, the engine may make use of the "bypassed parent" field maintained by the scheduler.

†`struct node *Sched_Node_Bypassed_Parent(NODE)`

This macro returns a pointer to the bypassed parent as maintained by the scheduler. It should be defined only if the scheduler performs bypassing both in the remote and the public region.

3.3.5 Initialisation

The initialisation of the system proceeds as follows. The engine evaluates the command line arguments, sets up the data structures corresponding to the master worker, creates the root node and calls the `Sched_Init` macro. `Sched_Set_Up_Worker` is now called for the master. Subsequently *n-1* slave processes are created, where *n* is the required number of workers. Each slave process will invoke `Sched_Set_Up_Worker` as its first scheduler macro. Following this, all processes will start up the normal engine routine, the next scheduler macro invoked being `Sched_Start_Work`.

`void Sched_Init(struct node *ROOT, int NUMBER_OF_WORKERS, int ARGC, char *ARGV[])`

Initialises all the scheduler specific data in the `ROOT` node and sets up the global data structures for the scheduler. The `ARGC` and `ARGV` parameters contain the conventional description of the command line arguments, enabling the scheduler to check the presence of some options on the command line. Note that the engine actually supplies the whole original command line (including the command name and the arguments affecting the engine).

`void Sched_Set_Up_Worker(struct node *STACK, int SELF_ID)`

Performs worker specific initialisations for the workers. `STACK` is an input argument which refers to the worker's node stack (to be used in `Allocate_Foreign_Node`). `SELF_ID` is an output argument: the scheduler returns a unique number between 0 and `NUMBER_OF_WORKERS-1` identifying the worker in question. The identification number for the master worker is 0. This number has no further relevance to the interface, it may be used by the engine as an index if it requires to set up an array of some data structures for workers.

`void Sched_Deinit()`

Restores everything to the state that existed before `Sched_Init` was called (e.g. releasing allocated memory). It is used in the implementation of the built-in predicates `save` and `restore`.

3.3.6 Interrupt Handling

The user of the Aurora system may interrupt the execution and request various actions to be taken, e.g. abort, exit, continue etc. This service is implemented mostly on the engine side of the interface, but there is some need for involvement by the scheduler. The interrupts are processed by an additional process created by the master worker, and the following three scheduler macros are invoked in that interrupt-handling process.

`void Sched_Block()`

This macro is called upon detection of an interrupt. The scheduler is requested to stop all workers as soon as they reach their next `Sched_Check` macro invocation.

`void Sched_Abort(struct node *CUT_NODE)`

This macro always follows a `Sched_Block` macro invocation. It is called when the user requests the execution to be aborted, following an interrupt. The scheduler should cause all workers to die back as if a pruning operation has been executed up to and including `CUT_NODE`. This pruning operation should be a completely cavalier one, i.e. a worker should perform it without regard to its position in the tree.

If the `CUT_NODE` is private when this macro is called, the scheduler should ensure that it is made public before causing the workers to die.

`void Sched_Unblock()`

This macro is called, following a `Sched_Block`, when the user has given a non-abort answer at an interrupt. The execution of all workers should continue.

3.3.7 Static Switches

The following macros provide static information to the engine about the scheduler. They should be defined to be either `TRUE` or `FALSE`.

`BOOL SCHED_WILL_COMPLETE_PRUNING`

should be set to `TRUE` if the scheduler will perform the public pruning operation upon resuming a branch suspended because of a cut or a commit (the engine should not perform the pruning operation after receiving control from the scheduler).

`BOOL SCHED_WANTS_COMPACT_PARENT`

should be set to `TRUE` if the scheduler requires the compact representation of the `Node_Parent` field. This is normally the case if the scheduler implements bypassing (see section 3.1, page 8).

3.4 Macros Provided by the Engine

Some macros described in this section are not in the *minimal* set of macros needed for the implementation of the scheduler (e.g. those supporting a particular optimisation). These macros are marked with a double dagger (‡). See See section 5.6, page 81 for descriptions of the implementation of these macros.

3.4.1 Notification of Work Found

The following two macros should be called by the scheduler before returning from a macro for finding work. Their task is to supply the address of the `NEW_SENTRY` node to the engine and to allow for any preparations specific to the type of work found to be performed. The engine (binding array) should be positioned above the new sentry node when these macros are invoked.

`void Found_Resumed_Work(struct node *NEW_SENTRY)`

This macro notifies the engine that a suspended branch is to be resumed, as the next assignment for the worker. `NEW_SENTRY` should be a sentry node corresponding to a suspended branch, i.e. `Node_Suspended(NEW_SENTRY)` must be `TRUE`.

`void Found_New_Work(struct node *NEW_SENTRY, struct alternative *ALT)`

This macro informs the engine that the next assignment corresponds to a newly reserved alternative. `NEW_SENTRY` must be an embryonic sentry node allocated using `Allocate_...Node`. `ALT` should be the alternative reserved from the parent of `NEW_SENTRY`. Note that this macro should not be called when work is acquired in the `Sched_Get_Work_At_Parent` macro.

3.4.2 Moving in the search tree

The following macros are applicable to public nodes. They should only be called from within macros for finding work.

`void Move_Engine_Down(struct node *DOWN_TO)`

This macro updates any engine-specific data structures (normally just the binding array) for the movement from the current position down to node `DOWN_TO` (and sets the current position to `DOWN_TO`).

`void Move_Engine_Up(struct node *UP_TO)`

This macro updates engine-specific data structures for the movement from the current position up to node `UP_TO` (and sets the current position to `UP_TO`).

‡`void Migration_Cost(struct node *FROM, struct node *DOWN_TO, unsigned int COST)`

The returned value `COST` is proportional to the number of bindings in the trail from node `FROM` down to node `DOWN_TO`.

3.4.3 Allocation of Nodes

`void Allocate_Node(struct node *PARENT, struct node *EMBRYONIC)`

This macro performs a reclaiming operation on the worker's own stack and allocates an embryonic node on the top of the stack. The level and parent fields of the node are initialised and a pointer to the node is returned in the `EMBRYONIC` output argument. This macro can be invoked only from within macros for finding work.

‡`void Allocate_Foreign_Node(struct node *STACK, struct node *PARENT, struct node *EMBRYONIC)`

This does the same operation as `Allocate_Node`, but on the supplied `STACK`: it performs a reclaiming operation and allocates an embryonic node on the `STACK`, initialising the parent and level fields of the node, and returning a pointer to the node in the `EMBRYONIC` output argument. This macro should only be invoked when the worker which owns the `STACK` is executing a macro for finding work.

3.4.4 Reclaiming Nodes

`void Mark_Node_Reclaimable(struct node *NODE)`

This macro allows the the scheduler to inform the engine that `NODE` is no longer required in the computation. It should be called during backtracking for all public and sentry nodes.

`void Mark_Suspended_Branch_Reclaimable(struct node *SENTRY)`

This macro should be used for cleaning the branch that has been suspended and later cut. `SENTRY` must be a sentry node of a suspended branch. All nodes below (but excluding) `SENTRY` are marked as reclaimable.

‡`void Mark_Node_Bypassed(struct node *NODE)`

This macro should be called by a scheduler when `NODE` is bypassed and it is certain that no further reference will be made to `NODE`. This makes it possible for the engine to recover the space occupied by `NODE`. Note that in the current Aurora engine this feature is not implemented.

‡`BOOL Node_Reclaimable(struct node *NODE)`

This macro returns `TRUE` if `NODE` has been marked as reclaimable.

3.4.5 Extending the Public Region

`void Make_Public(struct node *NEW_SENTRY)`

This macro should be invoked when the scheduler is preparing to extend the public region down to the parent of `NEW_SENTRY`. The engine is thus given the opportunity to perform any initialisation of the nodes to be made public. `NEW_SENTRY` becomes the new sentry node.

3.4.6 Further Node Handling

`struct node *My_Embryonic_Node()`

This macro returns the current embryonic node of the worker. `My_Embryonic_Node` is to be used only during work, i.e. it should not be used between entry to a macro for finding work and the corresponding `Found_..._Work`.

The value of `My_Embryonic_Node()` changes when nodes are created and destroyed. More exactly, it is set to point to the child of `NODE` immediately before `Sched_Node_Created(..., NODE)` is invoked. `My_Embryonic_Node()` will be reset to point to `NODE` after the call of `Sched_Node_Destroyed(..., NODE, TRUE)` returns to the engine.

‡`BOOL Valid_Node(struct node *NODE)`

This macro checks if `NODE` is a valid node address in the sense that accessing its fields will not cause memory access violation.

‡`BOOL Node_Suspended(struct node *NODE)`

This macro returns `TRUE` if `NODE` is a sentry node corresponding to a suspended branch, otherwise it returns `FALSE`. After the `Found_Resumed_Work(NODE)` macro has been called, the value of `Node_Suspended(NODE)` will become `FALSE`.

‡`char *Node_Pred_Name(struct node *NODE)`

This macro returns a pointer to a string containing the predicate name corresponding to `NODE`. It is used exclusively for debugging and graphic tracing.

‡`struct node *Node_Successor(struct node *NODE)`

This macro returns the successor of node `NODE`, provided `NODE` is live. For live private nodes the successor is equivalent to the child of the given node. This macro can be useful for traversing the private branch downwards (towards younger nodes), if the scheduler decides not to maintain child pointers in the private region (it will have to fill in child pointers in the dead private nodes, though).

3.5 Static Information

This section describes the static data supplied by the engine to the scheduler when a new clause (alternative) is compiled or loaded. This information is transmitted by the following function call (see section 3.3.3.3, page 22):

```
Sched_Alternative_Created(struct alternative *ALT,
  struct alternative *FIRST_ALT,
  BOOL IS_PARALLEL,
  int MAX_CUTS, int MAX_COMMITS,
  int ASSUMED_CUTS, int ASSUMED_COMMITS,
  BOOL IS_INTERNAL, BOOL CONTAINS_IF, BOOL ENDS_IN_A_FAIL)
```

There are two types of static data supplied: data relating to predicates being sequential or parallel, and information about the presence of pruning operators in the clause or predicate.

3.5.1 Sequential Predicates

Each predicate is either sequential or parallel, as requested by the user. Information about this property of a predicate is supplied to the scheduler via the `IS_PARALLEL` argument of the `Sched_Alternative_Created` macro, for each clause of the given predicate:

`BOOL IS_PARALLEL`
: `TRUE` if the alternative belongs to a parallel predicate, `FALSE` otherwise.

3.5.2 Pruning Information

Information about the presence of pruning operators in a clause may be needed by the scheduler to perform a cut more efficiently or to distinguish between speculative and non-speculative work, as in [Hausman 90]. We tried to define a general format for data about pruning properties, so that various scheduling algorithms can derive specific data, according to their needs.

Note that we decided to exclude data on cavalier commits from the pruning information, as this operation is expected to be used only for handling exceptional circumstances (similar to `abort` in Prolog).

If one disregards disjunctions, the information needed about pruning is quite simple. A scheduler may wish to know whether a clause contains cuts or commits. For more exact pruning algorithms the number of occurences of each pruning operator may be needed. The fact that a clause ends in a call to `fail`, may also be of interest: when such a clause is entered, all the pending pruning operations become inaccessible to the current branch. The simple set of pruning data would thus consist of three items for each clause: the number of cuts, the number of commits and the Boolean value indicating whether the clause ends in a failing call (i.e. `fail`, but in future global compile time analysis might discover this property for other calls).

The presence of disjunctions makes the situation more complicated. The original Prolog program undergoes a transformation when being compiled: all disjunctions, conditional and negated goals are replaced by calls to *internal* predicates. Note that for the purpose of the present discussion conditional and negated goals are treated as disjunctions. This is achieved by replacing `(`*IF* `->` *THEN*`)` by `(`*IF* `->` *THEN* `; fail)` and `(\+` *NOT*`)` by `(`*NOT* `-> fail ; true)`.

An internal predicate is created for each disjunction in the program, with a clause formed from each disjunct. This transformation has special significance from the point of view of pruning, as

any conditional expression (... *IF* `->` *THEN* ...) is transformed to an expression containing a cut to the internal predicate. Note that the *THEN* part may contain cut or commit operators which prune the whole original user predicate. Thus pruning operators with different scopes can be present in a single (internal) clause.

Another complication stems from the fact that there are several possible execution paths through a clause containing a disjuction, making it impossible to predict the exact number of pruning operators to be encountered in a clause. As the schedulers will normally be interested in keeping track of the maximal number of outstanding pruning operators, data on the maximal number of cuts and commits in a clause seems to be a suitable substitute. A correction to the maximal number of outstanding pruning operators can be made when an internal clause is actually entered. This correction should be the difference between the number of operators assumed for the whole internal predicate and the actual data for the given clause. Our final set of pruning data will thus include the following items for each clause:

`int MAX_CUTS`
`int MAX_COMMITS`
: —the maximal number of cuts and commits in the clause,

`int ASSUMED_CUTS`
`int ASSUMED_COMMITS`
: —the maximal number of cuts and commits in the whole internal predicate, of which the clause in question is a member (relevant to internal clauses only),

`BOOL IS_INTERNAL`
: —to distinguish internal clauses from ones defined by the user,

`BOOL CONTAINS_IF`
: —to account for the cut operator generated from the conditional expression (relevant to internal clauses only),

`BOOL ENDS_IN_A_FAIL`
: —to indicate if the clause ends in a call to `fail`.

To give a more formal definition of the above data, let us introduce a few auxiliary notions concerning clauses and disjunctions. For each alternative in the compiled code we must distinguish between two cases (denoting the clause corresponding to the alternative in question by C):

1. C is a clause in the original user program.
2. C is a clause of an internal predicate, i.e. corresponds to a disjunct.

Let the Boolean expression *user_def(C)* be `TRUE` for case 1 and `FALSE` for case 2 and let *orig_body(C)* denote the body of the original user clause to which C corresponds (case 1), or the disjunct in the original user disjunction to which C corresponds (case 2). Furthermore let *orig_disj(C)* denote the original form of the whole disjunction of which C is a member (case 2 only).

Given the above definitions, the pruning information for a clause C is defined as follows:

```
int MAX_CUTS         = max_cuts(orig_body(C));

int MAX_COMMITS      = max_commits(orig_body(C));

int ASSUMED_CUTS     =
        (
        user_def(C) → 0;
        otherwise → max_cuts(orig_disj(C))
        )

int ASSUMED_COMMITS =
        (
        user_def(C) → 0;
        otherwise → max_commits(orig_disj(C))
        )

BOOL IS_INTERNAL     =
        (
        user_def(C) → FALSE;
        otherwise → TRUE
        )

BOOL CONTAINS_IF     =
        (
        internal(C) and orig_body(C) ≡ 'IF -> THEN' → TRUE;
        otherwise → FALSE
        )

BOOL ENDS_IN_A_FAIL =
        (
        orig_body(C) ≡ '..., fail' → TRUE;
        otherwise → FALSE
        )
```

where

```
max_cuts(B) =
    (
    cut_operator(B) → 1;
    B ≡ '(A -> C)' → max_cuts(C);
    B ≡ 'G1, ..., GN' → max_cuts(G1) + ... + max_cuts(GN);
    B ≡ 'A1; ...; AN' → max(max_cuts(A1), ..., max_cuts(AN));
    otherwise → 0
    )

max_commits(B) =
```

```
(
commit_operator(B) → 1;
B ≡ '(A -> C)' → max_commits(C);
B ≡ 'G1, ..., GN' → max_commits(G1) + ... + max_commits(GN);
B ≡ 'A1; ...; AN' → max(max_commits(A1), ..., max_commits(AN));
otherwise → 0
)
```

4. Storage Model

The abstract machine described herein is a version of the Warren Abstract Machine (WAM) [Warren 83], augmented for the SRI model for Or-Parallel Execution. An overview of a precursor implementation of the SRI model as an extension of the WAM is given in [Lusk et al. 88]. This chapter will describe the storage model of the new implementation.

4.1 Terms and their Representation

The main types of terms are *variable references*, *constants*, and *compound terms*.

Each clause is divided into a number of *chunks*, where a chunk is a sequence of goals that compile to inline instructions followed by a procedure call or by the end of clause. The head counts as part of the first chunk. The exact definition is given in [Carlsson 90].

Each clause is classified as either *recursive*, if it has more than one chunk, *iterative*, if it has one chunk that ends with a procedure call, or *halting*, if it doesn't contain any procedure calls.

The variables in a given Prolog clause are statically classified as *permanent* or *temporary*. Intuitively, a variable is permanent if it has to survive a procedure call, otherwise it is temporary. The exact definition is given in [Carlsson 90]. In WAM notation, temporary variables are denoted by X(n), and permanent variables by Y(n).

The (immediate) binding of a variable is stored in its *value cell*. At any given time, a variable can be *unbound*, *conditionally bound* or *unconditionally bound* to another term. A procedure known as *dereferencing* follows a chain of bound variables until an unbound variable or a non-variable is encountered. A binding is conditional if there can be another execution path which may bind the variable to something else, and must therefore be undoable. Conditional bindings are recorded on the binding list, also called the *trail stack* (see section 4.2.4, page 46), and are undone as that stack is unwound.

Unbound variables can be *global*, in which case they reside on the global stack (see section 4.2.5, page 50), or *local*, in which case they reside on the environment stack (see section 4.2.2, page 38). Permanent variables are stored in the environment stack. Temporary variables are stored in a bank of *argument registers* and acquire a global stack cell if explicitly initialised as unbound.

A global variable may be bound to any term *except* to a local variable. A procedure known as *globalising* creates a new global variable and binds a local variable to it, ensuring that the stack variable henceforth dereferences to the global stack.

The value cell of a variable that is unconditionally bound simply contains the binding. For a variable V which is not unconditionally bound, the value cell contains V's unique *variable number* identifying V's location in the binding array. Global variables are assigned negative numbers, decreasing with age; local variables get numbers ≥ 0, increasing with age. The corresponding binding array location contains zero if V is unbound, otherwise it contains the binding. The mapping from variable numbers to binding array locations is described later (see section 4.2.6, page 51).

In the sequel we shall use the Boolean expression `CondVAR(`V`)` to denote whether the binding of a variable V is (or should be) conditional. The extression is true iff the number of V is in the range [`GU`, `LU`[, where `GU` and `LU` are abstract machine registers to be described later.

A constant is an *atom* or a *number*. A compound term is composed of a *functor* (an atom) and some *arguments* (arbitrary terms). *Lists* are a special case of compound terms and are optimised both in the instruction set and in memory, since the list functor need not be stored explicitly. All other compound terms are called *structures*.

Terms and variable numbers are represented as *tagged words* (data type `TAGGED`). A tagged word is a bit string with two parts:

tag This indicates what kind of term the tagged pointer denotes, and serves to distinguish variable numbers from terms. The symbolic tag names `VNO`, `REF`, `CON`, `STR` and `LST` denote variable number, variable reference, constant, structure and list, respectively. Notice that for this description there is a single `REF` tag for all variable references, but the actual implementation uses different tags for local and global variable references.

value For compound terms, this is a pointer to the representation of the term. For variable references, this is a pointer to the value cell. For constants, this could be a symbol table index to uniquely identify the constant. For variable numbers, this is the actual number, positive or negative.

We shall use the following access functions for tagged words:

`int TagOf(TAGGED T)`
: evaluates to the tag part of the tagged word `T`.

`TAGGED *ValueOf(TAGGED T)`
: evaluates to the value part of the tagged word `T`, i.e. to a pointer to the representation of `T` if `T` is a variable reference or a compound term.

`BOOL IsVar(TAGGED T)`
: is true iff `T` is a variable reference (bound or unbound).

`int ArityOf(TAGGED T)`
: where `T` is a compound term evaluates to the arity of `T`.

`struct node *ChoicepointOf(TAGGED T)`
: evaluates to a choicepoint reference encoded as the Prolog integer `T`.

`TAGGED TagChoicepoint(struct node *N)`
: converts a choicepoint reference `N` to a Prolog integer.

`TAGGED Tag(int C, TAGGED *P)`
: This combines the tag value `C` and the pointer `P` into a tagged word.

4.2 Data Areas

The data segment of the virtual memory space of each Aurora process is divided into a nonshared and a shared section. The nonshared section contains the *abstract machine registers* and the *binding array*.

The shared section contains a *static area*, for information which is saved from one query to another, and four *stacks*, for information which is not needed upon backtracking.

The static area contains a variety of objects used by the Aurora runtime system (e.g. the Prolog program itself). The only objects relevant to this report are structures that represent the alternative clauses of a predicate (see section 4.2.7, page 52).

The static area is operated as a memory pool in which objects of arbitrary size can be allocated. The relative address order of the various areas is not critical. The memory management of the static area is based on the dynamic shared memory allocation routines provided by the C library (`shmalloc()`, `shrealloc()`, and `shfree()` for the Sequent Symmetry).

Each stack consists of a number of *blocks*, doubly linked in a list. Each block grows toward increasing addresses. The blocks are allocated with the same routine as the static area, but they are never freed nor expanded. The *stack pointer* of a stack points at the first free address, i.e. at its top.

In contrast to the basic WAM model, we have split the local stack into an *environment stack* and a *node stack*. There is also a *global stack*, and a *trail stack*. There is no explicit PDL, just the implicit C PDL.

We shall describe the implementation of stacks from three different points of view. In this section we shall consider the relevant objects stored in each stack. See , page 123 for full type definitions of such objects. In the following section, we shall describe how the search tree is mapped onto the stacks of the Aurora processes. Finally, we shall describe how each stack is built up from blocks.

4.2.1 Abstract Machine Registers

The current computational state is held in the memory areas and in the abstract machine registers, pointing into the areas. The registers are collected into a `struct wam` record, referred to by the C variable `w`, with the following fields. The WAM register set corresponds to fields `insn` through `local_uncond`. The `previous_node` field supports the pruning operators. The `next_alt` and `frame2` fields support shallow backtracking, an issue not treated in the basic WAM. The `global_var` and `local_var` fields are sufficient extensions for supporting the SRI memory model on a single process. In addition to these, the remaining fields are necessary extensions for supporting multiple workers exploring a search tree:

`INSN *w->insn`
: Program counter (the WAM `P` register).

`INSN *w->next_insn`
: Continuation pointer (the WAM `CP` register). The current continuation is always immediately preceded by a `call` instruction.

`struct frame *w->frame`
: Current environment (the WAM `E` register).

`struct node *w->node`
: Current choicepoint (the WAM `B` register).

`struct frame *w->local_top`
: Environment stack pointer (the WAM `A` register).

`TAGGED *w->trail_top`
: Trail pointer (the WAM `TR` register).

`TAGGED *w->global_top`
: Global stack pointer (the WAM `H` register).

`TAGGED *w->structure`
: Structure pointer (the WAM `S` register).

`TAGGED w->term[]`
: Argument register vector (the WAM `Ai/Xi` registers).

`TAGGED w->global_uncond`
: Lowest conditional global variable number (called `GU` above; the `HB` register plays this role in the WAM).

`TAGGED w->local_uncond`
: Lowest unconditional local variable number (called `LU` above; the `B` register plays this role in the WAM).

`struct node *w->previous_node`
: Embryonic node at predicate entry. This supports the pruning operators.

`struct frame *w->frame2`
: Permanent variable base register (called `E2` in [Carlsson 89]).

`struct alternative *w->next_alt`
: Alternative clause during shallow backtracking; claimed alternative clause after finding work.

`TAGGED w->global_var`
: Lowest global variable number currently in use.

`TAGGED w->local_var`
: Lowest local variable number currently not in use (called `LV` in [Lusk et al. 88]).

`struct node *w->choice_top`
: Node stack pointer, points at the embryonic node.

`struct node *w->own_node`
: Topmost choicepoint in the worker's stack, i.e. the physical predecessor of the embryonic node. Called `Bp` in [Lusk et al. 88].

`struct node *w->own_sentry_node`
: Boundary between the remote and local regions of the search tree. See section 4.3, page 53.

`struct node *w->sentry_node`
: Boundary between the public and remote regions of the search tree, i.e. the sentry node. See section 4.3, page 53.

For convenience, in the sequel we will use the abbreviation:

`TAGGED X(int i)`
: Same as `w->term[i]`.

In addition to the above register set, each stack is associated with certain registers, represented as global (but nonshared) C variables, for memory management purposes. See section 4.4, page 57.

The register set is not minimal. The following invariants hold. The situation is complicated by the *shallow backtracking* optimisation [Carlsson 89]. See section 4.2.3, page 42 for a description of choicepoint fields:

```
w->node ≡ (the current choicepoint is complete ?
            w->choice_top->parent :
            w->choice_top);

w->own_node ≡ w->choice_top->own_node;

w->global_uncond ≡ w->node->global_var;

w->local_uncond ≡ w->node->local_var;
```

Although redundant, these four *shadow registers* are maintained so as to avoid frequent recomputation of their values. They are recomputed whenever `w->choice_top` changes and at nondeterminate procedure calls.

4.2.2 Environments

Recursive clauses need an environment to store permanent variables and continuation information in. Environments are created in the environment stack and are represented as `struct frame` records. The fields of an environment e are:

`struct frame *`e`->frame`
: Continuation environment pointer. Called `CE(E)` in WAM.

`INSN *`e`->next_insn`
: Continuation program counter. Called `CP(E)` in WAM.

`TAGGED `e`->local_var`
: Lowest available local variable number for the permanent variables of this environment. Called `CL(E)` in [Lusk et al. 88].

`TAGGED `e`->term[]`
: Permanent variables. Called `Yi` in WAM.

For convenience, in the sequel we will use the abbreviation:

`TAGGED Y(int i)`
: Same as `w->frame2->term[i]`.

Thus the `local_var` field is the only Aurora extension. It was introduced mainly for the purpose of compile-time allocation of local variable numbers and generalised *environment trimming*. The `local_var` field is also essential for seniority tests, as address comparisons cannot be used. Such tests occur when the ages of a choicepoint and an environment are compared (when the environment stack pointer and the next available variable number are computed), and when the ages of a local variable and an environment are compared (for the `put_unsafe_value` instruction). The compiler ensures that the chain of `local_var` environment fields form a strictly increasing sequence. Moreover, for any environment e and choicepoint n for a branch of the search tree, e`->local_var` $\geq$ n`->local_var` iff e is younger than n.

4.2.2.1 Allocating Environments

In the original WAM, environments are allocated at the first occurrence of a permanent variable in the compiled code of a recursive clause by the `allocate` instruction. The definition given in [Warren 83], translated to C syntax and using our register and field names, is:

```
allocate:
{
  struct frame *curenv = w->frame;

  w->frame = (curenv < w->node ? w->node : curenv+EnvSize(w->next_insn));
  w->frame->next_insn = w->next_insn;
  w->frame->frame = curenv;
}
```

In Aurora, this instruction has been split into `allocate1` and `allocate2`, where the latter immediately precedes the first procedure call of a clause. This measure, analogous to that described in [Carlsson 89], avoids some fruitless work in case of unification failure before the first procedure call.

Furthermore, Aurora introduces a permanent variable base register, `w->frame2`, which is used as the base register for accessing permanent variables. This means that `w->frame` is left unchanged up to the `allocate2` instruction, and the shallow backtracking optimisation can take advantage of this fact. It also means that the `proceed` instruction, which terminates halting clauses and transfers control to the continuation of a recursive clause, must refresh the `w->frame2` register, since it is going to be used in the continuation.

```
allocate1:
        w->frame2 = w->local_top;

allocate2:
        w->frame2->frame = w->frame;
        w->frame2->next_insn = w->next_insn;
        w->frame2->local_var = w->local_var;
        w->frame = w->frame2;

proceed:
        w->frame2 = w->frame;
        w->insn = w->next_insn;
```

4.2.2.2 Creating Local Variables

Local variables are allocated by a combination of compile-time and run-time techniques. The environment stack location of a local variable `Y(`*i*`)` is computed by adding an offset to the current value of `w->frame2`. Analogously, the variable number of `Y(`*i*`)` is computed by adding an offset to the current value of `w->local_var`. The offsets are computed at compile time as operands of the following instructions:

`put_variable(y(`*n*`),`*i*`,`*j*`)`
: Set `X(`*i*`)` to reference the new unbound variable `Y(`*n*`)` whose stack location and variable number are computed as *n* `+ w->frame2` and *j* `+ w->local_var` respectively.

`put_void(y(`*n*`),`*j*`)`
: Create the new unbound variable `Y(`*n*`)` whose stack location and variable number are computed as *n* `+ w->frame2` and *j* `+ w->local_var` respectively.

where a sequence of the latter instruction immediately precedes the `allocate2` instruction, to ensure that the environment is complete at the first procedure call. This measure is necessary for garbage collection considerations.

4.2.2.3 Trimming and Deallocating Environments

Environment trimming was introduced in the WAM as a generalisation of last call optimisation, and allows a portion of the environment to be *logically* deallocated by every procedure call in recursive clauses. The deallocated portion corresponds to those variables that are no longer in use in the rest of the clause. However, the deallocated variables might be needed again if an alternative execution path is taken by backtracking. Therefore, the memory occupied by the deallocated portion is not necessarily made available by environment trimming. Instead, the environment stack pointer is computed as a function of certain WAM registers and of the `EnvSize` operand of the `call` instruction that immediately precedes the current continuation. This computation may occur long after the actual procedure call.

Environment trimming is generalised in Aurora to apply to the binding array as well. Each time the environment is trimmed, a portion of the binding array is trimmed too, corresponding to those deallocated permanent variables that were created as local variables. Analogously, the next available local variable number is computed as a function involving the `VarCount` operand of the `call` instruction, whose Aurora format becomes:

`call(`*P*`,`*EnvSize*`,`*VarCount*`)`
: Call procedure *P* where *EnvSize* permanent variables and *VarCount* slots for permanent variables in the binding array are still to be used in the clause body. The compiler sets

VarCount to 1 even if no binding array locations are in use, to ensure that the chain of `local_var` environment fields form a strictly increasing sequence.

The last procedure call of a recursive clause is encoded by a `deallocate` instruction which logically deallocates the remaining part of the environment, followed by an `execute(`*P*`)` instruction.

The environment stack pointer and next available variable number are computed by the following function:

```
trim_environment_stack()
{
  w->local_top = (w->frame->local_var < w->local_uncond ?
                  w->own_node->local_top :
                  w->frame + EnvSize(w->next_insn));
  w->local_var = (w->frame->local_var < w->local_uncond ?
                  w->local_uncond :
                  w->frame->local_var + VarCount(w->next_insn));
}
```

although they are not actually recomputed by this formula each time one of the registers they depend on changes its value. Instead, they are kept up to date by the following policy, details of which are given in the description of the various instructions (see chapter 6, page 86).

- Upon failure the registers are restored from the relevant choicepoints.
- At the first procedure call of a recursive clause, the registers are simply incremented by the respective operands of the instruction.
- At a subsequent procedure call of the same clause, instruction, the environment is trimmed (the registers are decremented) unless it is protected by a younger choicepoint. If trimming is possible, the stack pointer is recomputed using the environment pointer and *EnvSize*, and the variable number is recomputed using the base variable number of the current environment and *VarCount*.
- At the last procedure call of the clause, the registers are restored as the current environment and its base variable number, unless the current environment is protected by a younger choicepoint.
- At a pruning operator, the registers are recomputed using the macro, unless the pruning operator occurs between the `allocate1` and the first `call` instruction of a clause, in which case a new environment has already been started, which freezes the environment stack top, and so no trimming takes place.

4.2.3 Choicepoints

Choicepoints are created in the node stack and are represented as `struct node` records. Choicepoints are not of a fixed size since they contain a varying number of argument registers. This caused complications in the Echo design [Lusk et al. 88], where the node stack grew towards decreasing addresses, because the address of the next choicepoint to be allocated (the *embryonic node*) could not be determined ahead of time. As a work-around, nodes were split into a fixed size part (the "scheduler node"), and a varying size part (the "choicepoint"), with an explicit pointer from the former to the latter. For a given node N, the predecessor address was computed as a function of the arity of N. The successor address was computed as the value of the explicit pointer stored in the "scheduler node" adjacent to N.

Figure 4-1 depicts the Echo layout, with the "scheduler node" parts in grey. Two complete nodes and the embryonic node are shown. When the embryonic node is fleshed out, an explicit pointer is stored, linking the "scheduler node" to the "choicepoint", and a new embryonic node is created:

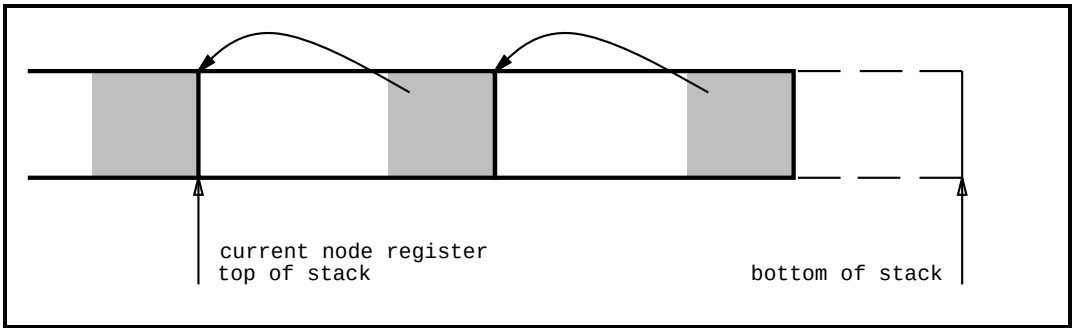

Figure 4-1: Node stack layout, Echo version

In the Foxtrot design, this complication is avoided by organising the node stack so that it grows towards increasing addresses. An explicit pointer links each node with its predecessor. The successor address can be computed as a function of the arity. We feel that this layout fits the SRI model better, as the current node pointer becomes decoupled from the top of stack pointer, and the latter becomes the natural representation of the embryonic node. Figure 4-2 shows three complete nodes and the embryonic node. When the embryonic node is fleshed out, a new embryonic node is created and an explicit pointer is stored, linking it with the completed node:

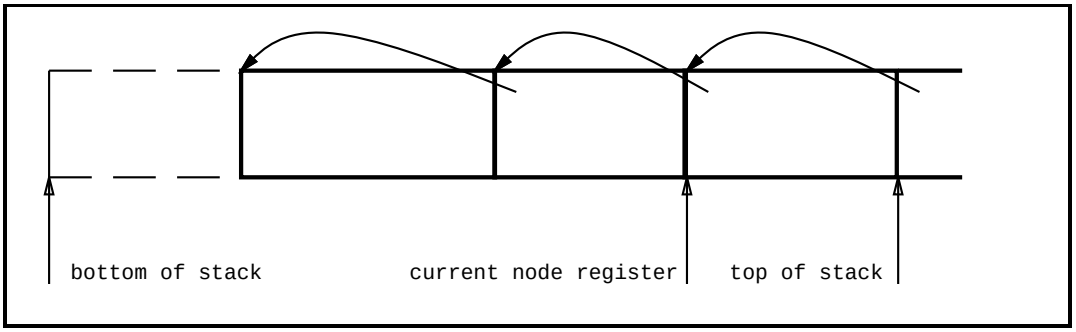

Figure 4-2: Node stack layout, Foxtrot version

Choicepoint records have the following fields. The choicepoints of the basic WAM have fields corresponding to `own_node` through `term[]`. The `local_top` field was introduced to enable storing choicepoints and environments in two separate stacks. The `global_var` and `local_var` fields are sufficient extensions for supporting the SRI memory model on a single process.

The remaining fields are necessary extensions for supporting multiple workers exploring a search tree. The `level` field serves as a unique age indicator for nodes, and is used when such ages need to be compared, for instance when a pruning operator needs to detect whether its scope extends beyond the worker's stack segment, and in certain services that the engine provides to the scheduler. The `parent` field links a node to its (logical) parent node, which is usually different from its physical predecessor if a worker has taken an alternative from a remote node or if it has been assigned one, or if it has resumed a suspended branch. In the engine's alternative, more compact, node layout, the logical parent does not need to be stored as an explicit field. Instead, `parent` should be thought of as an abbreviated expression involving other node fields. See section 5.5, page 78.

The fields of a node *n* are:

`struct node *`*n*`->own_node`
: Physical predecessor node, called `B'(B)` in WAM.

`TAGGED *`*n*`->trail_top`
: Saved trail stack pointer, called `TR'(B)` in WAM.

`TAGGED *`*n*`->global_top`
: Saved global stack pointer, called `H'(B)` in WAM.

`struct alternative *`*n*`->next_alt`
: Alternative clause, called `BP(B)` in WAM.

`struct frame *`*n*`->frame`
: Saved current environment pointer, called `BCE(B)` in WAM.

`INSN *`*n*`->next_insn`
: Saved continuation pointer, called `BCP(B)` in WAM.

`TAGGED `*n*`->term[]`
: Saved argument registers, called `A(`*i*`)` in WAM.

`struct frame *`*n*`->local_top`
: Saved environment stack pointer.

`TAGGED` *n*`->global_var`
: Saved lowest used global variable number.

`TAGGED` *n*`->local_var`
: Saved lowest unused local variable number.

`int` *n*`->level`
: Distance from the root node in nodes.

`struct node *`*n*`->parent`
: Logical parent node. Its value is well defined in the entire tree.

For convenience, in the sequel we will use the abbreviation:

`TAGGED A'(int i)`
: Same as `w->node->term[i]`.

In addition to the engine fields described above, the node is also likely to contain scheduler specific fields.

4.2.3.1 Basic Operations

During ordinary, local, work, the life cycle of a node consists of three states:

minimal
: Advancing the `w->choice_top` pointer is tantamount to creating a new embryonic node. At this time, the `level`, `own_node`, and `parent` fields are filled in. Embryonic nodes are also created when new assignments are started, at which time reclaimable nodes on the top of the stack are reclaimed (cf. `Allocate...Node`).

partial
: When a nondeterminate predicate call is made, a choicepoint has to be created. The current implementation uses the shallow backtracking optimisation. Its key idea is to postpone or avoid altogether creating choicepoints and restoring state from them. Thus at the time of the nondeterminate call, only three pieces of information (the `trail_top`, `global_top`, and `global_var` fields) are stored in the embryonic node, making it a partial node.

complete
: When creating a full choicepoint cannot be postponed any longer, all the remaining fields of a partial node are filled in, making it a complete node. This occurs when the `neck(_)` instruction is reached, when the engine decides to suspend the current branch, and in preparation to performing garbage collection.

The life cycle normally terminates when the last alternative of a nondeterminate call is taken, at which time the partial or complete node reverts to the minimal state. The life cycle may

also terminate prematurely as the result of a pruning operation or by scheduler interaction. The following operations are used for changing node states:

```
void make_node_partial()
{
  w->node = w->choice_top;
  w->node->trail_top = w->trail_top;
  w->node->global_top = w->global_top;
  w->node->global_var = w->global_var;
  w->global_uncond = w->global_var;
  w->local_uncond = w->local_var;
  Sched_Nondet_Pred_Entered(w->sentry_node);
}

void make_node_complete()
{
  int i = w->next_alt->arity;

  w->node->next_alt = w->next_alt, w->next_alt = NULL;
  w->node->frame = w->frame;
  w->node->next_insn = w->next_insn;
  w->node->local_top = w->local_top;
  w->node->local_var = w->local_var;
  w->own_node = w->node;
  w->choice_top = &A'(i);
  while (i>0)
    --i, A'(i) = X(i);
  w->node->child = w->choice_top;
  w->choice_top->level = w->node->level+1;
  w->choice_top->own_node = w->node;
  w->choice_top->parent = w->node;
  Sched_Private_Parent_Stored(w->sentry_node, w->choice_top, w->own_node);
  Sched_Node_Created(w->sentry_node, w->node);
}
```

4.2.3.2 Shallow Backtracking

The shallow backtracking optimisation is inherited with minor changes from the sequential implementation [Carlsson 89]. The following points summarise the Aurora version:

- Whenever a predicate call occurs, the engine is in its default *deep backtracking phase*. The number of live temporary registers is determined by the arity of the called predicate, the embryonic node is in the minimal state, the current node register (`w->node`) refers to the parent of the embryonic node, and the alternative clause register (`w->next_alt`) contains `NULL`.

- When a predicate is called and there is more than one possibly matching clause, the engine enters its *shallow backtracking phase*. The embryonic node is turned into a partial node (`make_node_partial()`), and `w->next_alt` is set pointing at the second candidate clause. In the shallow backtracking phase, the information stored in the partial node suffices for restoring the engine state if backtracking should occur. The shallow backtracking phase terminates if the `neck(_)` instruction is reached, in which case the embryonic node is made complete and a new embryonic node is created (`make_node_complete()`), and when backtracking into the last candidate clause. In both cases, `w->next_alt` is set to `NULL`.
- The shallow backtracking phase is also re-entered by (deep) backtracking to a choicepoint with more than one remaining candidate clause. At this time, `w->next_alt` is set pointing at the second remaining candidate clause.

Thus the value of `w->next_alt` determines which phase the engine is in. The values of `w->node` and `w->choice_top` determine whether the current choicepoint is partial or complete (the node is partial iff `w->node` $\equiv$ `w->choice_top`). The latter check is used when taking the last alternative in the shallow backtracking phase, to determine whether there is a complete choicepoint to destroy.

4.2.4 The Trail Stack

The main use of this area is to record conditional variable bindings. The binding of a variable V must be made conditional iff a choicepoint has been created after the unbound variable was created (`CondVAR(`V`)`). Upon backtracking, entries are simply popped off the trail stack and the corresponding binding array elements are reset to 0.

It is a property of the SRI model that a trailed item always refers to a variable whose value cell contains a binding array reference. This property is crucial when installing and deinstalling bindings, as the binding array location in which to install or deinstall a binding is found by indirection through the value cell.

After a pruning operation which trims the node stack, it becomes mandatory to tidy the portion of the trail which is younger than the oldest node that was deleted. Tidying means to reprocess all bindings which earlier were recorded as conditional and *promote* these to unconditional where appropriate. If the trail is not tidied, the above property might be violated due to environment trimming, with fatal effects when attempting to install or deinstall non-existent variable bindings.

Trail entries are organised as (*ref*,*value*) pairs, where the *ref* and *value* parts of a trail entry reference TR are accessed as TR`[0]` and TR`[1]`, respectively, and *ref* may be one of the following:

a variable reference

The entry denotes a trailed variable binding. The *value* part eventually contains the bound value, but filling in the value parts is delayed until the trail entry becomes part of a non-local stack segment (see section 4.3.2, page 54). This optimisation is due to Seif Haridi. In the sequel, filling in value parts is called *publishing bindings*.

forward_link

The entry denotes a forward link to another memory block in the same trail stack, where the *value* part points at the first trail entry in the next memory block, just after its leading backward link. Forward and backward links are introduced when the system detects that a new memory block must be allocated for the trail stack, and may occur anywhere inside a trail segment.

backward_link

Such entries have two uses. Every memory block, except the first one, of every trail stack begins with a backward link to the previous memory block in the same trail stack. Such a backward link has a matching forward link in the previous memory block.

Backward links are also used when the logical extension of the trail is in another worker's trail stack, or deeper inside the worker's own trail stack. No forward links are needed in this situation, which occurs whenever a new *cactus stack arm* (see section 4.3, page 53) is started.

In both situations, the *value* part points at a location immediately after the last trail entry of the logical extension of the trail.

Figure 4-3 illustrates a situation with two workers, w1 and w2. There are two blocks in w1's trail stack and one block in w2's trail stack. The white areas denote the trail component of the current branch; grey areas do not belong to the current branch. The (logical) trail starts at the top of w1's trail stack, passes a block boundary, winds trough part of w2's trail stack and back into w1's trail stack and continues elsewhere:

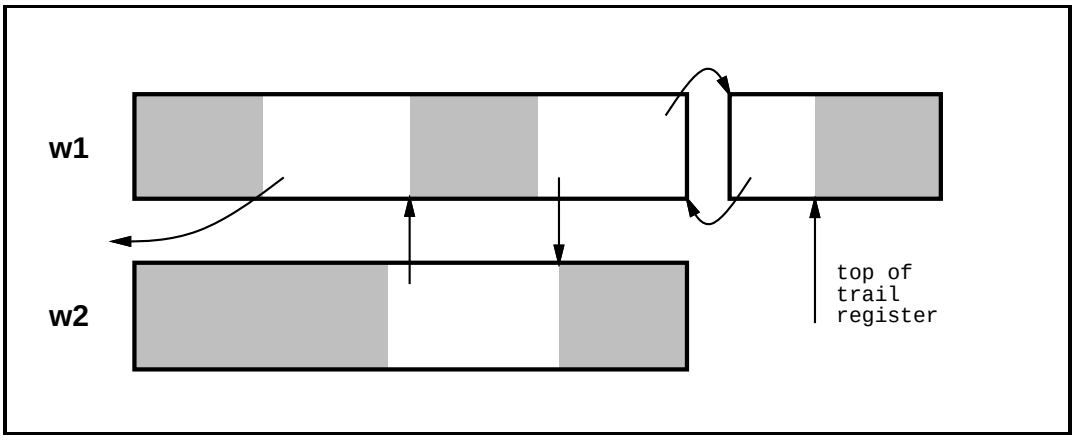

Figure 4-3: Trail stacks, trail blocks and trail links

4.2.4.1 Basic Operations

When a variable is conditionally bound, the binding is recorded by pushing a reference to the variable on the trail. If the public part of the tree is subsequently extended, or if the current branch is suspended, the value part of trail items corresponding to bindings are copied from the binding array, thus publishing the bindings to make them available for installation and deinstallation by other workers.

In preparation for starting a new assignment, the scheduler must position the engine by first moving it up from its current position to a common ancestral node, and then optionally moving it down to the node where there is work. Moving up is associated with *deinstalling bindings* in the binding array, and moving down is associated with *installing bindings* in the binding array.

The remaining basic operations include logically connecting two trail segments, and an incrementing operation which is transparent to forward pointers.

`void TrailBinding(TAGGED *TR, TAGGED Ref)`

This pushes a conditionally bound reference `Ref` onto the trail stack, incrementing the stack pointer `TR`.

```
{
  TR[0] = Ref;
  TR += 2;
}
```

`void PublishBinding(TAGGED *TR)`

The trail pointer `TR` is decremented, and the value part of the next binding recorded on the trail is filled in from the binding array. However, if the next trail item is a backward link, the link is followed instead.

```
{
  TR -= 2;
  if (IsVar(TR[0]))
    TR[1] = CondBinding(TR[0]);
  else if (TR[0] ≡ backward_link)
    TR = (TAGGED *)(TR[1]);
}
```

`void DeinstallBinding(TAGGED *TR)`

The trail pointer `TR` is decremented, and the next binding recorded on the trail is installed in the binding array. However, if the next trail item is a backward link, the link is followed instead.

```
{
  TR -= 2;
  if (IsVar(TR[0]))
    CondBinding(TR[0]) = 0;
  else if (TR[0] ≡ backward_link)
    TR = (TAGGED *)(TR[1]);
}
```

`void InstallBinding(TAGGED *TR)`

The trail pointer `TR` is decremented, and the next binding recorded on the trail is installed in the binding array. However, if the next trail item is a backward link, the link is followed instead.

```
{
  TR -= 2;
  if (IsVar(TR[0]))
    CondBinding(TR[0]) = TR[1];
  else if (TR[0] ≡ backward_link)
    TR = (TAGGED *)(TR[1]);
}
```

`void CountBinding(TAGGED *TR, int COST)`

The trail pointer `TR` is decremented, and the counter `COST` is incremented. However, if the next trail item is a backward link, the link is followed instead.

```
{
  TR -= 2;
  if (IsVar(TR[0]))
    COST++;
  else if (TR[0] ≡ backward_link)
    TR = (TAGGED *)(TR[1]);
}
```

`void TrailLink(TAGGED *TR, TAGGED *EXT)`

A backward link is stored on the trail stack at `TR`, to form a logical extension with the trail segment ending at `EXT`.

```
{
  TR[0] = backward_link;
  TR[1] = (TAGGED)EXT;
  TR += 2;
}
```

`void TrailIncrement(TAGGED *TR)`

The trail pointer `TR` is incremented in such a way that links between memory blocks in

the same trail stack are transparent. This operation is only used within the local part of the tree.

```
{
  TR += 2;
  if (TR[0] ≡ forward_link)
    TR = (TAGGED *)(TR[1]);
}
```

4.2.5 The Global Stack

This area is mainly used for constructing instances of compound terms during Prolog execution. New objects are allocated by incrementing the `w->global_top` register. This area also holds the value cells of global variables, either created explicitly when temporary variables are initialised, or implicitly by *globalising* local variables.

The details of the various objects follow:

list objects
: These consist of the two arguments of the list.

structure objects
: These consist of the functor followed by the arguments of the structure. The functor is represented as a tagged word with a value field containing both arity and atom table index.

variable objects
: These consist of just one word: the value cell.

4.2.5.1 Creating Global Variables

The variable number of a new unbound variable is allocated by decrementing the `w->global_var` register, and is stored in the new value cell.

`TAGGED NewVAR()`
: This creates a global variable and returns a reference to it.

```
{
  TAGGED t1 = Tag(REF, w->global_top);

  *w->global_top++ = --w->global_var;
  return t1;
```

```
}
```

4.2.6 The Binding Array

In the SRI model, conditional variable bindings are stored in the worker's private binding array, to allow constant-time access to variable binding, and in the trail, to allow workers to migrate from one branch to another in the search tree, maintaining a one-to-one correspondence between binding array contents and the variables that have been conditionally bound between the root node and the worker's current position.

The current Aurora design uses a single binding array for conditional variable bindings. The Echo design [Lusk et al. 88] used two separate arrays.

As mentioned earlier (see section 4.1, page 33), every variable in the SRI model is associated with a unique variable number, positive or negative. A base register `binding_base` points at the array element corresponding to variable number 0. An array element contains 0 iff the corresponding variable is unbound or if that element is outside the part of the array which is currently in use.

Figure 4-4 depicts the binding array. The range of variable numbers currently in use is defined by the registers `w->global_var` and `w->local_var`. The subrange of variable numbers for variables which need to be bound conditionally is defined by the registers `w->global_uncond` and `w->local_uncond`. Grey areas are currently not in use and are filled with zeroes.

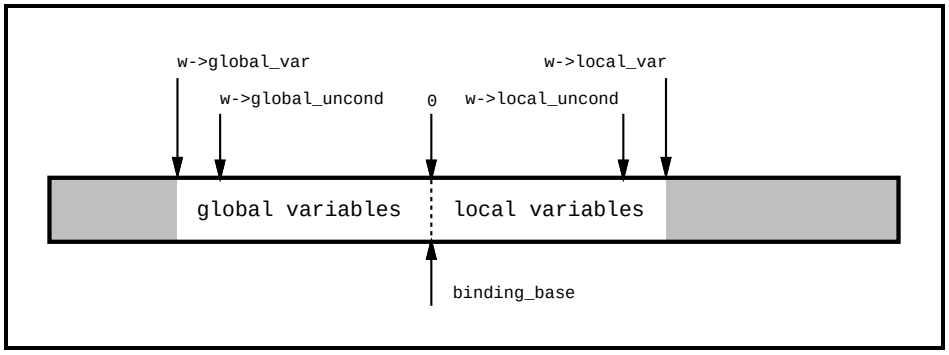

Figure 4-4: The binding array

An advantage of the current design is a better utilisation of the available memory space. If one part of the binding array overflows, more space can be provided by shifting the base register and array contents towards the part which is not yet full. If both parts fill up, a new area must be allocated, and the old binding array contents must be moved there. In the Echo design, the areas

allocated for the global and local parts of the binding array were defined at compile time, with no provision for overflow handling.

An advantage of the Echo design is that variable numbers could be actual pointers to array elements, but we view the capability for overflow handling well worth the price of a small constant overhead (accessing the base register) in addressing array elements.

The following macro is relevant for a dereferenced variable which is unbound or conditionally bound, i.e. whose value cell contains a variable number:

`TAGGED CondBinding(TAGGED V)`

is an abbreviated expression for accessing the binding array location of `V`, and may appear as the left hand side of an assignment statement as well as in expressions.

```
{
  return binding_base[*ValueOf(V) - Tag(VNO, 0)];
}
```

4.2.7 Alternatives

Compiled clauses are stored in the static area. When a predicate call occurs in the emulator, a procedure known as *indexing* filters out a subset of the clauses that can potentially match the call. The indexing mechanism considers the principal functor if the first argument register only. To support the indexing mechanism, chains of alternatives are stored as linked lists of `struct alternative` records in the static area.

Figure 4-5 depicts a situation with three clauses. Alternative records are displayed as circles. For this predicate, indexing distinguishes between four cases, corresponding to the first argument being `a`, `b`, a variable reference, and something else:

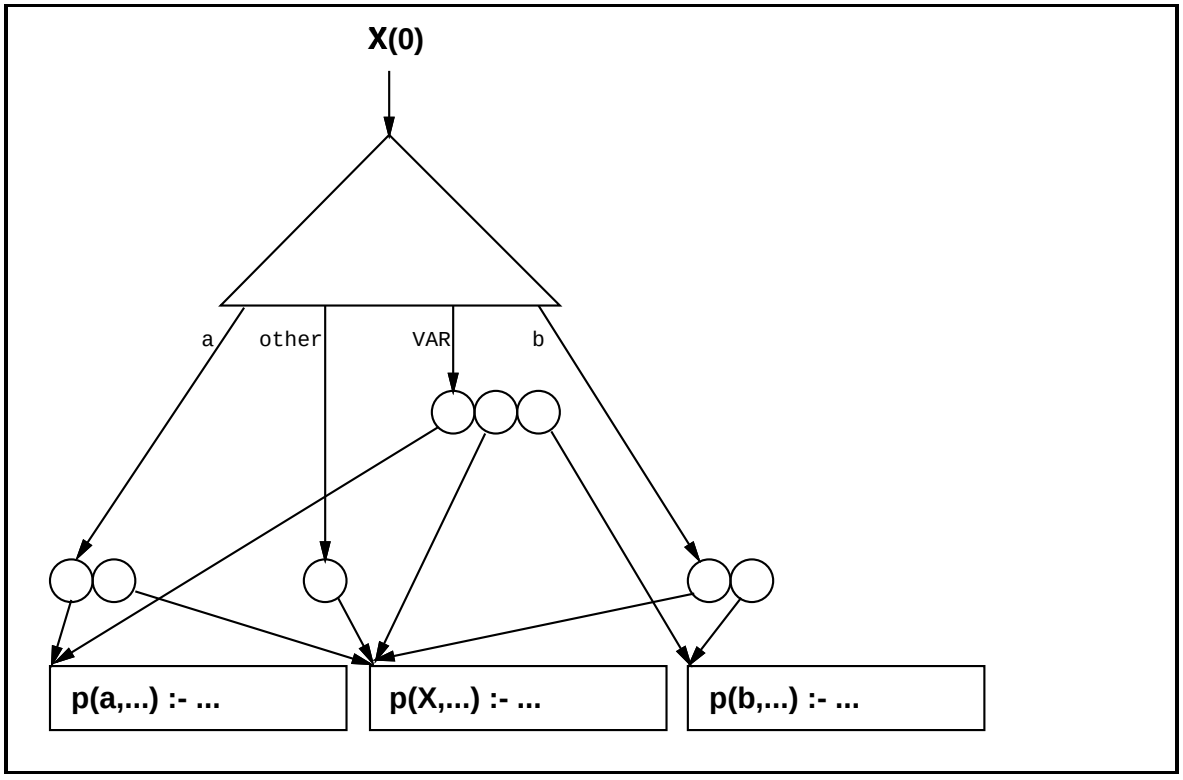

Figure 4-5: Indexing three clauses

For an alternative a, only the following fields of are relevant to this report:

`int` *a*`->arity`
: The arity of the predicate in question. This defines how many argument registers to restore on backtracking.

`struct alternative *`*a*`->next`
: A pointer to the next alternative for this chain of alternatives, or `NULL` if this is the last alternative in the chain.

`INSN *`*a*`->insn`
: A pointer to the starting address of the clause that this alternative denotes.

In addition to the engine fields described above, the alternative is also likely to contain scheduler specific fields encoding static information.

4.3 Cactus Stack Management

4.3.1 Stacks and Segments

Each stack can be viewed on two different levels of abstraction. The low-level view is as a linked list of memory blocks, and is the subject matter of a later section (see section 4.4, page 57). The high-level view is as a sequence of *segments* which make up the nodes and arcs of the abstract tree of the pure SRI model, and is the subject matter of this section. Recall that in the Aurora model, each worker is said to own his stack group. To support the or-parallel model, the WAM stacks are generalised to "cactus stacks" mirroring the shape of the search tree [Lusk et al. 88].

Each node (choicepoint) constitutes a node stack segment. The initial node of a worker's node stack is initialised with the relevant stack pointer fields but does not belong to any computation. Instead, the `own_node` field of this node contains the stack pointer while looking for work, and is used by the `Allocate_Node` and `Allocate_Foreign_Node` operations. We shall use the notation `MyRoot` to denote the initial node of the current worker in later descriptions. A pointer to that node is also passed to the scheduler in the `Sched_Set_Up_Worker` macro.

Each subsequent node N defines a *segment group* consisting of N itself and a corresponding segment, possibly empty, from each of the other three stacks, namely:

- The environment stack segment consisting of the data from N`->own_node->local_top` to N`->local_top`.
- The global stack segment consisting of the data from N`->own_node->global_top` to N`->global_top`.
- The trail stack segment consisting of the data from N`->own_node->trail_top` to N`->trail_top`.

There is also an *embryonic segment group* whose node component is the embryonic node and whose other components consist of any data on the environment, global, and trail stacks that is younger than the current node.

4.3.2 Node Aspects

Each non-embryonic node (and the segment group it determines) is classified as one of the following:

local A single worker has exclusive read-write access to the segment group. This is the situation in sequential Prolog: variables can be unconditionally bound, environments can be trimmed, nodes can be physically deleted by cuts and failure. All segment groups are initially local and are created on the stacks owned by the worker. Local segments are reclaimed by the normal stack mechanism. A remote node reverts to

being local if its successor is its child node C and C is a local node or the embryonic node.

remote A single worker has exclusive read and restricted write access to the segment group: variables can only be conditionally bound and environments cannot be trimmed, but alternatives may be taken from nodes and nodes can be marked as logically deleted by cuts and failure. Local nodes become remote when a branch is suspended. Remote segment groups contain the same information as local ones, but additionally the value part of trail entries are filled in.

public All workers have shared read access to the segment group: variables can only be conditionally bound, environments cannot be trimmed, taking and pruning alternatives requires assistance from the scheduler. Local and remote nodes become public on request by the scheduler, when it decides to extend the public region of the search tree. Public segment groups contain the same engine-specific information as remote ones.

ghost The segment group is marked as reclaimable (logically deleted), and no longer belongs to the search tree. It constitutes a hole in the stacks. The memory occupied by such holes is reclaimed by the worker that owns the stacks when it finds a ghost node at the top of its node stack. A remote node becomes a ghost when the sole accessing worker backtracks over it. A public node becomes a ghost when the scheduler determines that no worker needs to access it.

In the Echo design, remote nodes did not exist, and suspending a branch caused local nodes to become public. Remote nodes were introduced to reduce the suspension overheads.

4.3.3 Cactus Stacks, Arms and Branches

A node is a *sprout node* if its logical parent node is not the same node as its physical predecessor. Several nodes may share the same parent node, and at most one child node can be a non-sprout node. The search tree is formed by linking nodes by their parent pointers. The parent of a sprout node must be remote or public.

A *cactus stack arm* is a contiguous sequence of segment groups such that the first node component is a sprout node, subsequent node components are non-sprout nodes, and the successor of the last node component is either a sprout node, a ghost node, or an embryonic node. Thus a worker's stack group is a sequence of cactus stack arms and ghosts, but those cactus stack arms may belong to many different branches of the search tree.

A *branch* of the search tree consists of the stack segments determined by the chain of nodes that begins at a worker `w`'s embryonic node and ends at the root node. That chain may span several cactus stack arms and consists of the embryonic node, followed by some number of local nodes, followed by some number of remote nodes, followed by some number of public nodes. The boundary between the local and remote part corresponds to the `w->own_sentry_node` register, which points

at a designated local node. The boundary between the remote and public part corresponds to the `w->sentry_node` register, which points at a designated private (local or remote) node.

Figure 4-6 shows two possible situations with two workers. There is a single public node (1). In the first situation (A), worker w1 suspended the left branch at node 2, then worker w2 resumed it and suspended again at node 3, and then worker w1 resumed it and is working at the embryonic node below node 4. Worker w2 picked up a new alternative from node 1. At this point, the left branch consists of the embryonic node, one local node, two remote nodes, and one public node. In the second situation (B) the left branch has died back, and worker w1 has picked up a third alternative from node 1. Node 3 is now a ghost in worker w2's stack, and can only be reclaimed after node 5 has been reclaimed.

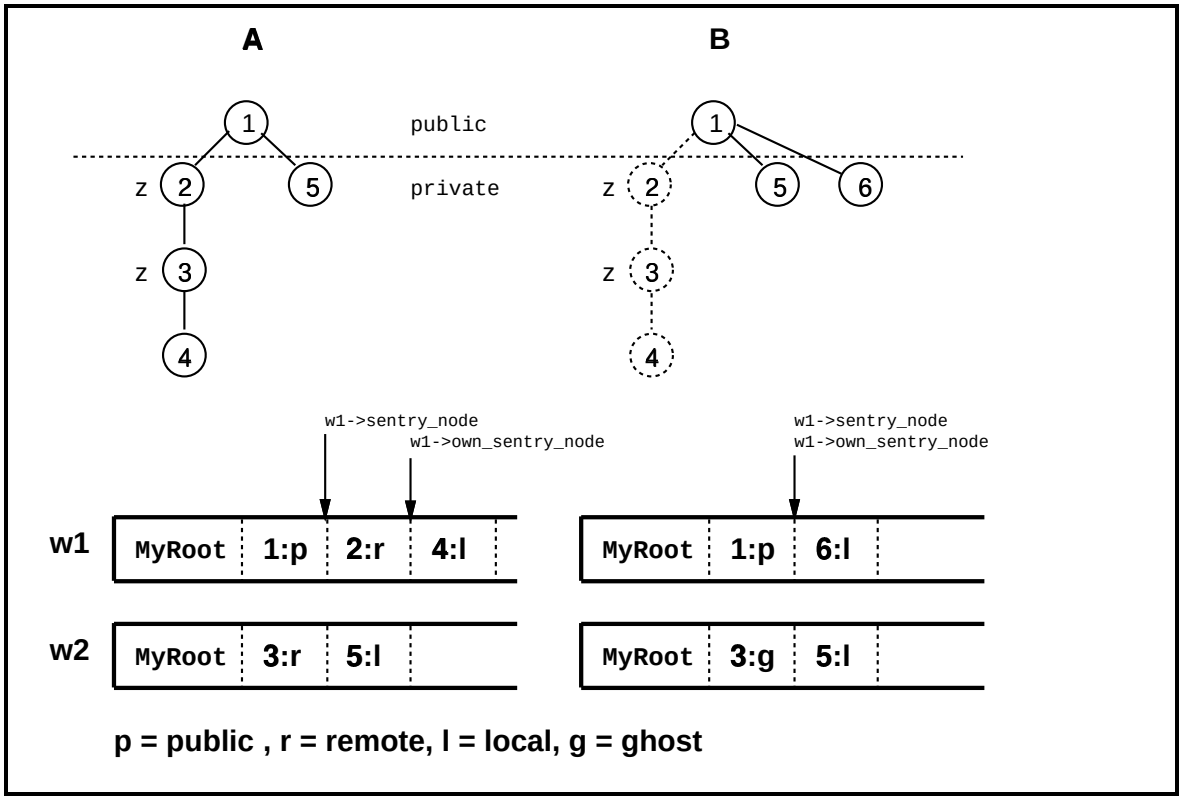

Figure 4-6: Storing nodes in two workers' stacks

4.3.4 Task Switching

When an alternative is taken from a parent node and a new assignment is started by a worker, or when a suspended branch is resumed, a new cactus stack arm is started if the parent node is not at the top of the node stack of that worker. Otherwise, the current cactus stack arm is simply extended. The new cactus stack arm or its extension will initially be local, but a worker may from time to time extend the public part of the current arm. The assignment terminates if backtracking

occurs and there are no private nodes with alternatives left, if the current branch is pruned by another worker, or if the worker decides to suspend the current branch. In the latter case, the embryonic stack segment is "fleshed out" into a full node, the local section is made remote, and the engine returns control to the scheduler.

The embryonic sprout node of a new cactus stack arm is placed at the top of the worker's node stack, after moving the stack pointer back over any ghost nodes. The stack pointers of the other stacks are adjusted by fetching them from the relevant fields of the predecessor of the embryonic node. The remaining engine registers are fetched from the parent node, in particular the `global_var` and `local_var` fields.

Thus memory reclamation in the node stack is done in unison with reclamation in the other stacks. This rather simple scheme does not exploit all opportunities for memory reclamation that exist in a Prolog system:

backtracking
: A whole segment is reclaimed.

trusting
: The tip stack segment is attached to the (empty) embryonic stack segment, and the tip node is reclaimed.

pruning
: A number of stack segments are attached to the embryonic stack segment, and a number of nodes are reclaimed. The environment stack is trimmed, and some bindings in the trail are promoted.

garbage collection
: The global stack can be garbage collected from time to time.

Garbage collection is currently not performed at all in Aurora, and no memory is currently reclaimed when non-local nodes are trusted or pruned. Instead, the non-local nodes are marked as reclaimable for later physical reclamation. The interface allows for an improved memory allocation scheme, however. By means of the `Mark_Node_Bypassed` macro, the scheduler may inform the engine that it ensures that no more references will be to a certain node, leaving it up to the engine to the engine to reclaim the memory occupied by the node itself (but not the memory occupied by the other corresponding stack segments). This optimisation has not been applied in the current implementation.

4.4 Low-Level Memory Management

4.4.1 Overview

In SICStus Prolog, each Prolog stack resides in a contiguous block of memory. Stack overflows are handled by a combination of garbage collection and stack shifting techniques. The latter operation amounts to moving the contents of a block to a bigger block somewhere else in memory, and relocating relevant pointers.

The Echo design essentially inherited the sequential design, but without provision for handling stack overflows due to the problems of performing garbage collection and stack shifting on memory areas shared by several processes. Thus a rather serious drawback of that design was that the user had to supply large enough stack sizes for his problem when running the system.

In the new Aurora design, each stack consists of a doubly linked list of memory blocks. When a stack overflow occurs, a new block is simply allocated and linked in at the end of the list, and the relevant top of stack pointer is set to point at the base of the new block. To avoid fragmentation problems, progressively larger blocks are created (currently, each new block is allocated twice as big as the previous block). Although the present design makes stack shifting unnecessary, a garbage collector is still needed in a production system, but is yet to be designed and implemented. As a first step, Ali has suggested an extension for the SRI model of the sequential garbage collection algorithm [Appleby et al. 88] to make it applicable to local parts of the search tree [Ali 89]. Patrick Weemeeuw [Weemeeuw 89] has addressed the problem of garbage collection of the public parts of the tree.

4.4.2 Implementation

Each memory block has a header part, represented as a `struct blockheader` record *b*, containing the fields:

`int` *b*`->size`
: Block size in bytes.

`int` *b*`->used`
: Bytes in use. This is maintained solely for statistics purposes, and is set when it is discovered that there is not enough space in the current block beyond the top of stack pointer, at which time the stack pointer is advanced to the base of the next block.

`struct blockheader *`*b*`->previous`
: `NULL` or previous memory block.

`struct blockheader *`*b*`->next`
: `NULL` or next memory block.

The header part is immediately followed by the actual data. For convenience, we shall use two macros for converting between the start of data and the header part:

```
struct blockheader *CharToBlock(char *C)
{
  return ((struct blockheader *)C)-1;
}

char *BlockToChar(struct blockheader *B)
{
  return (char *)(B+1);
}
```

For every worker, each of the four stacks is associated with two *bounds registers*:

`char *`*STACK*`_low`
: Pointer to the data part of the current memory block.

`char *`*STACK*`_warn`
: A limit register, calculated as the end of the current memory block, less an amount which depends on the stack in question. When the stack pointer passes this limit, an out-of-bounds condition is triggered.

where *STACK* is `global`, `local`, `trail`, or `node`.

From time to time, the engine checks the stack pointer of each stack against the corresponding bounds registers. The exact policy for doing such checks differs from stack to stack and is described below, but the algorithm for handling an out-of-bounds stack pointer T is the same for all stacks.

The goal of this algorithm is to adjust the bounds registers and T in such a way, that the out-of-bounds condition ceases to exist. This is achieved by allocating a new block and advancing both the bounds registers and the pointer T itself, or by updating the bounds registers only (when backtracking to a state in which T points to a previous block).

The algorithm can be stated as follows. A precondition is that T is in the current block (but possibly above the limit register) or in a previous block. If this were not the case it would mean that the bounds checking policy had failed, and T would have passed the end of block, causing a memory overrun.

```
Let B be the current block.
While T is not in B,
  let B be the predecessor of B.
While there is not enough space in B,
  allocate a successor block for B if it doesn't have one,
  let B be the successor of B,
  let T be the start of data of B.
Recompute the bounds registers.
```

The algorithm is actually coded as follows. We show here how the bounds check is performed for the trail stack. The other stacks are treated similarly. Since blocks can be placed anywhere in shared memory, the test whether T is in the current block requires two address comparisons:

```
void check_trail_overflow()
{
  if (w->trail_top < trail_low ∨ w->trail_top ≥ trail_warn)
    Bounds_Handler(w->trail_top, trail_low, trail_warn, 8, 8);
}

void Bounds_Handler(char *TOP, char *LOW, char *WARN,
                    int PAD, int MARGIN)
{
  struct blockheader *current = CharToBlock(LOW);

                                /* Find block containing stack pointer,
                                   at or before the current one. */
  while (TOP < LOW ∨ TOP ≥ LOW + current->size)
    current = current->previous,
    LOW = BlockToChar(current);

                                /* Find an appropriate block
                                     with enough free space in it,
                                     at or after the current one. */
  while (TOP+PAD ≥ LOW + current->size)
  {
    current->used = TOP - LOW;
    if (current->next ≡ NULL)
      current = allocate_block(current);
    else
      current = current->next;
    link_blocks(TOP, current);
    TOP = LOW = BlockToChar(current);
  }

  WARN = LOW + current->size - MARGIN;
}
```

where `allocate_block()` creates a new memory block and puts it at the end of the linked list, updating the relevant pointers, and where `link_blocks()` stores forwarding pointers at the old and new stack tops if the stack in question is the trail stack (see section 4.2.4, page 46). `link_blocks()` does nothing for the other stacks. The *MARGIN* parameter is a function of the stack in question, and *PAD* is usually equal to *margin*, but may be different in special situations, e.g. if the engine needs to ensure that some amount of free memory exists on the global stack below the limit register. It is worth noting that `Bounds_Handler` is defined to be a *macro* which updates its *TOP*, *LOW*, and *WARN* arguments.

4.4.3 Bounds Checks Policy

An optimal policy for bounds check performs such tests just often enough to ensure that the memory blocks and binding array are not overrun.

Our implemented policy ensures that the local and global parts of the binding array cannot run out of space before an out-of-bounds condition is detected for respectively the environment and global stacks, at the cost of wasting some space in the binding array. Thus during ordinary, private execution, no overflow checks are needed for the binding array in addition to the bounds checks performed for the stacks. An explicit array overflow check is needed only when a new assignment is started and bindings are installed from shared trail segments.

The policy for bounds checks of the various stacks during private execution is:

global stack

The global stack pointer is bounds checked at each resolution step. Certain WAM instruction and built-in predicates explicitly check that a sufficient margin exists in the current memory block. The compiler ensures that such instructions are emitted when the default margin is insufficient.

Whenever the boundary registers are updated, the binding array is also checked to ensure that the size of the unused, global part of the binding array is not less than the size of the current memory block of the global stack. This caters for the following worst case situation:

One branch of the search tree creates lots of ground terms but no unbound variables, thus no binding array space is needed. Then the global stack pointer is set by backtracking pointing at the beginning of a memory block. Then an alternative execution branch fills the memory block with unbound variables, thus binding array space corresponding to the whole memory block is needed.

If there is not enough space in the global part of the binding array, more space is provided by "stealing" space from the local part, as described earlier (see section 4.2.6, page 51).

environment stack

The environment stack pointer is bounds checked whenever it is computed as an offset from the current environment pointer.

The local and global parts of the binding array are treated analogously. Thus whenever the boundary registers are updated, the local part of the binding array is also checked to ensure that the size of its unused part is not less than the size of the current memory block of the environment stack.

node stack

The node stack pointer is bounds checked whenever a new node is completed.

trail stack The trail stack pointer is bounds checked whenever an item is pushed on the trail. This is probably suboptimal, but safe.

5. The Emulator

This chapter discusses the Aurora emulator. It plays a central role in the implementation as it executes the abstract Prolog instructions, sometimes with scheduler assistance, and sometimes provides assistance to the scheduler.

This chapter is organised as follows. We first treat the following topics, which are all affected by the SRI Model and may need scheduler interaction: initialisation, backtracking, pruning operators, assignment termination, and straightening and contraction. We then give semantics for the services provided by the engine for the scheduler.

All descriptions will initially assume the default engine configuration. Some descriptions have to be modified in the compact-parent configuration; see section 5.5, page 78 for details.

5.1 Initialisation

When the engine starts up, it acquires its first piece of work by means of the `Sched_Start_Work` macro. Like the other macros for finding work, this macro must be able to reposition the worker in the tree by means of `Move_Engine_Up` and `Move_Engine_Down`. The engine records its current position in the variable `my_position`. Similarly, the macros for finding work must be able to allocate embryonic nodes by means of the `Allocate...Node` macros, which use a field of the worker's initial node (`MyRoot`) to access the stack pointer.

```
{
  MyRoot->own_node = w->choice_top; /* save node stack pointer */
  my_position = the root of the search tree;
  Sched_Start_Work(goto exit_wam;);
  emulate();
exit_wam:
}
```

where the `emulate` function initialises the abstract machine registers and starts emulating the abstract machine code.

5.2 Backtracking

From the engine's point of view, the main complication of or-parallel execution is its impact on the backtracking routine. This routine has to take into consideration the various (local, remote, public) regions of the search tree as well as for shallow and deep backtracking. Shallow backtracking is used in the local region only, to maximise its efficiency. The scheduler assists backtracking in the public region.

A special complication arises in the remote region. When a remote environment is trimmed, there is a danger of the worker's own environment stack pointer being set to point to another worker's stack. This would result in environment records being created outside a worker's own stack. To prevent this from happening it suffices to enforce the following *invariant*. If the invariant holds, the formula for computing the environment stack pointer (`trim_environment_stack()`) will ensure that the stack pointer stays in the worker's own stack:

> No remote node younger than the node referred to by the `w->node` register belongs to the current computation.

This invariant holds initially for every assignment. It would normally not hold after a straightening or contraction operation, which normally removes the dead node. But in the remote region, the engine enforces the invariant by *not* removing any node until it has completed backtracking over its *last* alternative (dieback), at which time it is marked as *logically* deleted. Physical reclamation is handled by the worker in whose stack the node is located.

As a side-effect of this measure, all variable bindings in the remote region become classified as conditional, and all accesses to the stacks of other workers become read-only, with the exception of taking alternatives from remote nodes, bypassing dead remote nodes, and marking them as reclaimable.

The backtracking procedure is called `backtrack()` which recomputes, possibly aided by the scheduler, the program counter as the entry address of an alternative Prolog clause that execution should proceed from. The different cases are treated in the sections below. In all cases, the first action taken is to deinstall all conditional bindings that have been made since the current choicepoint was created:

```
void backtrack()
{
  deinstall_bindings();
  if (w->next_alt ≠ NULL)
    w->insn = shallow_backtracking()->insn;
  else
    w->insn = deep_backtracking()->insn;
}
```

```
void deinstall_bindings()
{
  while (w->trail_top ≠ w->node->trail_top)
    DeinstallBinding(w->trail_top);
}
```

5.2.1 Shallow Backtracking

This mode of backtracking is used if the value of the `w->next_alt` (next shallow alternative) register is not `NULL`. Conditional bindings younger than the current choicepoint have already been deinstalled. Certain registers are refreshed, the `w->next_alt` register is updated, and if the last alternative was taken (contraction), the current choicepoint `w->node` and shadow registers have to be adjusted and the scheduler is informed. Finally, the scheduler is notified that a clause taken from a private node is being backtracked to:

```
struct alternative *shallow_backtracking()
{
  struct alternative *alt = w->next_alt;

  w->global_top = w->node->global_top;
  w->global_var = w->node->global_var;
  w->next_alt = alt->next;
  if (w->next_alt ≡ NULL)
  {                              /* taking last alternative */
    if (w->node ≠ w->choice_top)
    {                            /* current choicepoint is complete */
       Sched_Node_Destroyed(w->sentry_node, w->node, TRUE);
       w->choice_top = w->node;
       w->own_node = w->node->own_node;
    }
    w->node = w->node->parent;
    w->local_uncond = w->node->local_var;
    w->global_uncond = w->node->global_var;
  }
  Sched_Clause_Entered(w->sentry_node, alt, FALSE);
  return alt;
}
```

5.2.2 Deep Backtracking

When shallow backtracking is impossible, the engine may first have to skip over a number of dead remote nodes until it finds a node that is live or public. Dead remote nodes must be marked as reclaimable (ghosts), so that the owning workers can handle their physical reclamation. A dead node must not be marked as reclaimable, however, while the stack segment it belongs to is still in use, in particular before all conditional bindings have been deinstalled. A dead sentry node must not be marked at all, since it will be reclaimed by the public backtracking mechanism. Ghost nodes must also be "unstitched" from the tree by updating the relevant `parent` and `level` fields.

When a suitable live or public node is found, there are three possible cases: local, remote, and public.

```
struct alternative *deep_backtracking()
{
  while (w->node ≠ w->sentry_node->parent ∧
         w->node->next_alt ≡ NULL)
    skip_dead_node(w->node);
  if (w->node ≡ w->sentry_node->parent)
    return public_backtracking();
  else if (w->node ≡ w->own_node)
    return local_backtracking(w->node->next_alt);
  else
    return remote_backtracking(w->node->next_alt);
}
```

While searching for a suitable node, it can happen that the remote section winds back to the top of the worker's own stack. According to the definitions given earlier (see section 4.3.2, page 54), a sequence of remote nodes must revert to being local in such situations. Thus the boundary between the remote and local regions must be recomputed. To find a suitable node, the following code is executed:

```
void skip_dead_node(struct node *n)
{
  w->node = n->parent;
  deinstall_bindings();
  if (n ≡ w->own_node)
  {                              /* n is local */
    w->choice_top = n;
    w->own_node = n->own_node;
  }
  else if (n ≠ w->sentry_node)
  {                              /* n is remote and non-sentry */
    unstitch_node();
```

```
    Mark_Node_Reclaimable(n);
  }

  w->local_uncond = w->node->local_var;
  w->global_uncond = w->node->global_var;
}

unstitch_node()
{
  if (w->node ≡ w->own_node)
    compute_own_sentry_node();  /* recompute boundary */
                                /* and update sentry node */
  w->choice_top->parent = w->node;
  Sched_Private_Parent_Stored(w->sentry_node, w->choice_top, w->node);
  w->choice_top->level = w->node->level+1;
}
```

where the local/remote boundary is computed by the following function. *Precondition:* `w->own_node` must be a private node.

```
void compute_own_sentry_node()
{
  w->own_sentry_node = w->own_node;
  while (w->own_sentry_node->parent ≡ w->own_sentry_node->own_node ∧
         w->own_sentry_node ≢ w->sentry_node)
    w->own_sentry_node = w->own_sentry_node->own_node;
}
```

The implemented backtracking routine is an optimised version of the above account. The gory details have been left out for clarity.

5.2.2.1 Local Backtracking

This mode of backtracking is used when the current node is a local node. Conditional bindings younger than the current choicepoint have already been deinstalled. Relevant engine registers are refreshed, and the `w->next_alt` register is loaded with the next alternative clause, if it exists, enabling shallow backtracking for that alternative. If it does not exist, the current choicepoint and shadow registers have to be recomputed. Finally, the scheduler is notified that the choicepoint is being destroyed or reused and that a clause taken from a private node is being backtracked to:

```
struct alternative *local_backtracking(struct alternative *alt)
{
  deep_failure(alt->arity);
  w->next_alt = alt->next;
  if (w->next_alt ≡ NULL)
  {
    Sched_Node_Destroyed(w->sentry_node, w->node, TRUE);
    w->choice_top = w->node;
    w->own_node = w->node->own_node;
    w->node = w->node->parent;
    w->local_uncond = w->node->local_var;
    w->global_uncond = w->node->global_var;
  }
  else
    Sched_Node_Reused(w->sentry_node, w->node);
  Sched_Clause_Entered(w->sentry_node, alt, FALSE);
  return alt;
}

void deep_failure(int N)
{
  int i=0;

  w->global_top = w->own_node->global_top;
  w->global_var = w->node->global_var;
  w->local_top = w->own_node->local_top;
  w->local_var = w->node->local_var;
  w->previous_node = w->node->parent;
  w->frame = w->node->frame;
  w->next_insn = w->node->next_insn;
  while (i<N)
    X(i) = A'(i), i++;
}
```

5.2.2.2 Remote Backtracking

This mode of backtracking is used when the current node is a remote node. Conditional bindings younger than the current choicepoint have already been deinstalled. A backward link must be stored on the worker's own trail stack, linking it with its logical extension, and most engine registers are refreshed. The `next_alt` choicepoint field is loaded with the next alternative clause. If the last alternative is taken, contraction would normally be applicable, but it is disabled in the remote section, and it is up to the scheduler to bypass the dying node (`Sched_Node_Destroyed`). Finally, the scheduler is notified that the choicepoint is being destroyed or reused and that a nonshared clause is being backtracked to:

```
struct alternative *remote_backtracking()
{
  w->trail_top = w->own_node->trail_top;
  TrailLink(w->trail_top, w->node->trail_top);
  deep_failure(alt->arity);
  if (alt->next ≡ NULL)
  {
    Sched_Node_Destroyed(w->sentry_node, w->node, FALSE);
    w->node->next_alt = NULL;
  }
  else
  {
    Sched_Node_Reused(w->sentry_node, w->node);
    w->node->next_alt = alt->next;
  }
  Sched_Clause_Entered(w->sentry_node, alt, FALSE);
  return alt;
}
```

5.2.2.3 Public Backtracking

This mode of backtracking is used when the current node is public, and splits into two cases:

1. The current node has alternatives left, and the worker manages to claim one of them, reusing the current sentry node and avoiding many of the administrative duties performed by the general public backtracking procedure.

   Case 1 has a precondition: the sentry node must be the embryonic node in order to maintain the topology of the tree. Case 1 splits into two subcases.

   Subcase 1a: The current node is at the top of the worker's stack, and the claimed alternative is the last alternative of the node. This means that the *contraction* optimisation is possible, which effectively transforms the public node into a new sentry node, and execution proceeds as in local backtracking.

   Subcase 1b: Otherwise, if the current node is not at the top of the worker's stack, a backward link is stored on the worker's own trail stack, linking it with its logical extension at the current node. Execution proceeds as in subcase 2b.

2. The scheduler is asked to do a general, non-local search for work after saving the node stack pointer for later access when a new embryonic is to be created. On exit from this search procedure, the found work (the claimed alternative) is stored in `w->next_alt`. There are again two subcases:

   Subcase 2a: The found work corresponds to a resumed branch. Execution proceeds as in private backtracking.

   Subcase 2b: The work was found at a live public node. Relevant engine registers are refreshed, and the scheduler is notified that a shared clause is being backtracked to.

```
struct alternative *public_backtracking()
{
  struct alternative *alt;

  if (w->sentry_node ≡ w->choice_top)
  {
    Sched_Get_Work_At_Parent(w->sentry_node, w->node, alt,
                             (w->node ≡ w->own_node),
                             { /* case 1a */
                               w->sentry_node = w->node;
                               w->own_sentry_node = w->node;
                               return local_backtracking(alt);
                             });
    if (alt ≢ NULL)
    {                        /* case 1b */
      if (w->node ≢ w->own_node)
      {
        w->trail_top = w->own_node->trail_top;
        TrailLink(w->trail_top, w->node->trail_top);
      }
      goto shared_clause;
    }
  }

  MyRoot->own_node = w->choice_top; /* save node stack pointer */
  my_position = w->node;
  Sched_Die_Back(w->sentry_node, goto exit_wam;);
public_fail:
  w->local_uncond = w->node->local_var;
  w->global_uncond = w->node->global_var;
  alt = w->next_alt;
  w->next_alt = NULL;
  if (w->choice_top ≢ w->sentry_node)
  {                          /* case 2a */
    if (w->node ≡ w->own_node)
      return local_backtracking(alt);
    else
      return remote_backtracking(alt);
  }
  else
  {                          /* case 2b */
shared_clause:
    deep_failure(alt->arity);
    Sched_Clause_Entered(w->sentry_node, alt, FALSE);
    return alt;
  }
}
```

where branching to the label `exit_wam` terminates the current worker process (see section 5.1, page 63).

5.3 Pruning Operations

Pruning operations require scheduler assistance if their scope extends into the public region of the tree. A host of algorithms for pruning public parts of the tree have been developed in Hausman's thesis [Hausman 90]. The Aurora engine is neutral with respect to these algorithms. It is responsible for pruning the private region, implemented by the `prune_tree` procedure below. As a final step, it calls the `Sched_Prune` scheduler macro to notify the scheduler that a pruning operation which may extend into the public region has taken place.

We shall treat two cases: pruning strictly within the local region, and pruning beyond the local region. These cases are handled by separate procedures. Finally, the environment stack top is recomputed, the trail stack is trimmed, and the scheduler is notified that a pruning operation has occurred, for its own bookkeeping:

```
void prune_tree(struct node *C, int OP)
{
  if (C->level ≥ w->own_sentry_node->level)
    prune_local_region(C);
  else
  {
    prune_local_region(w->own_sentry_node);
    if (C->level ≥ w->sentry_node->level)
      prune_remote_region(C);
    else
      prune_remote_region(w->sentry_node);
  }
  trim_environment_stack();
  tidy_trail();
  Sched_Prune(w->sentry_node, C, OP, goto kill_task;, goto suspend_task;);
}
```

where C is the youngest node to be cut; OP determines the type of the pruning operator; `tidy_trail()` denotes the action of tidying the trail, described below; branching to the label `kill_task` occurs if the current assignment has been pruned by another worker (see section 5.4.1, page 75); and branching to the label `suspend_task` occurs if the pruning operation has to suspend (see section 5.4.2, page 76).

5.3.1 Pruning the Local Region

The scheduler is notified that a number of local nodes have died. Part of the local region is physically reclaimed, and relevant registers are updated:

```
void prune_local_region(struct node *C)
{
  Sched_Nodes_Destroyed(w->sentry_node, w->node, C, TRUE);
  w->node = C->parent;
  w->own_node = C->own_node;
  w->choice_top = C;
  w->local_uncond = w->node->local_var;
  w->global_uncond = w->node->global_var;
}
```

where C is the youngest node to be cut.

5.3.2 Pruning the Remote Region

The scheduler is notified that a number of remote nodes have died. The remote nodes are explicitly killed, since they cannot be physically reclaimed yet (see section 5.5, page 78):

```
void prune_remote_region(struct node *C)
{
  struct node *n = w->own_sentry_node->parent;

  Sched_Nodes_Destroyed(w->sentry_node, n, C, FALSE);
  while (n ≠ C->parent)
    n->next_alt = NULL,
    n=n->parent;
}
```

where C is the youngest node to be cut.

5.3.3 Tidying the Trail

As explained earlier (see section 4.2.4, page 46), it is mandatory after a pruning operation to reprocess such entries in the trail stack which are younger than the oldest node that was deleted, since all such entries were made with the "wrong assumption" about when a variable binding must be conditional. Tidying the trail is easiest understood as reconsidering all such entries with the correct assumptions in force. Each reconsidered binding is either kept conditional and trailed, or is promoted to being unconditional, in which case the space for the corresponding trail entry can be reused.

The oldest deleted node prior to the pruning operation corresponds to the embryonic node afterwards. The values of `w->trail_top` and `w->choice_top->trail_top` define the boundaries of the trail stack region to be tidied. In this section, these boundaries will be called `TIDY_HIGH` and `TIDY_LOW`, respectively.

Notice that it is quite possible for a pruning operation to leave the `w->node` register unchanged, in particular when pruning into the remote section. In this case, the trail need not, and must not, be tidied, since no nodes have been deleted, and the value of `w->choice_top->trail_top` is undefined if no new nodes have been created during the current assignment.

The trail tidying algorithm does not have to preserve the order of trail items corresponding to bindings that must be kept as conditional, and such items do not have their value parts filled in. The algorithm is best described in terms of manipulation of two temporary pointers, `t1` and `t2`, as shown in figure 5-1. At each step, the portion of the trail to be tidied is partitioned as follows. The fact that the trail stack is divided into blocks is immaterial to the algorithm:

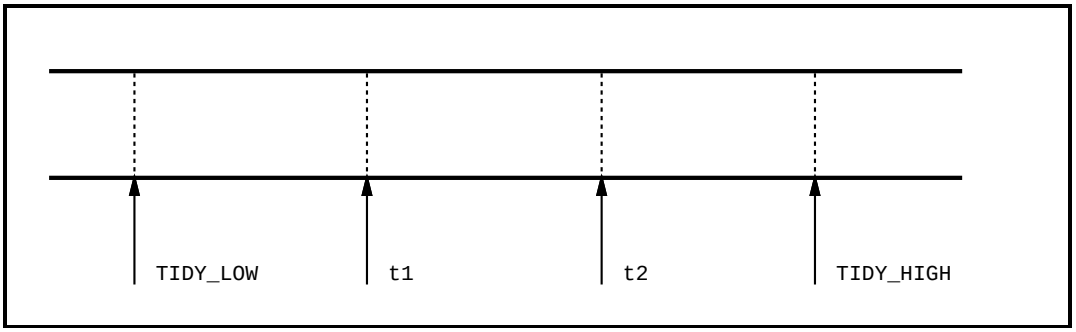

Figure 5-1: Pointers used while tidying the trail

where the part between `TIDY_LOW` and `t1` contains variables found to be still conditional, the part between `t1` and `t2` constitutes a hole, and the part between `t2` and `TIDY_HIGH` contains variables still to be considered.

The temporaries `t1` and `t2` are initialised as `TIDY_LOW`, and the final value of `t1` defines the new trail stack pointer:

Initially `t1` = `t2` = `TIDY_LOW`.
While (`t2` $\neq$ `TIDY_HIGH`)
  read the next trailed variable X at `t2`, incrementing `t2`
  if X should now become unconditionally bound
    promote X
  else
    write X at `t1`, incrementing `t1`.
Store `t1` as the updated trail stack pointer.

The algorithm is coded as follows:

```
void tidy_trail()
{
  TAGGED *t1 = w->choice_top->trail_top;
  TAGGED *t2 = t1;

  while (t2 ≠ w->trail_top)
  {
    if (¬CondVAR(t2[0]))
    {
      TAGGED value = CondBinding(t2[0]);

      CondBinding(t2[0]) = 0;
      *ValueOf(t2[0]) = value;
    }
    else
    {
      t1[0] = t2[0];
      TrailIncrement(t1);
    }
    TrailIncrement(t2);
  }
  w->trail_top = t1;
}
```

5.4 Assignment Termination

Recall that there are three reasons for an assignment to be terminated:

- the engine has died back to a public node (normal termination),
- the current branch has been pruned by another worker, or
- the current branch needs to be suspended.

In all three cases of assignment termination, the worker has to be assigned a new piece of work. The three situations are somewhat different, and each situation corresponds to a scheduler macro that the engine must call. Prerequisites for all three cases are that the local region must be empty and that the worker must be positioned at or below the sentry node.

`Sched_Die_Back`

The scheduler is notified that the engine has normally terminated its assignment and is ready for a new piece of work.

`Sched_Be_Pruned`

The scheduler is notified that the engine has completed cleaning up its private region in response to a previously received order to prune the current branch. Such orders can arise in connection with the following scheduler macros: `Sched_Check`, called on every procedure invocation; `Sched_Prune`, called when pruning operations are performed; and `Sched_Synch`, called when the engine insists that the current branch be leftmost in some subtree or outside the scope of any pruning operators in some subtree.

`Sched_Suspend`

The scheduler is notified that the engine has completed transforming its private region into a suspended branch in response to a previously received order to suspend the current branch. Such orders can be given by the scheduler in connection with the same macros as in the previous item. Note that suspension can arise on the scheduler's own initiative, if it decides that the current branch should be abandoned for a less speculative one.

The engine ensures that the expression `Node_Suspended(`N`)` is `TRUE` if N is the sentry node of a suspended branch. This is implemented by letting the `frame` field of the sentry node point at the tip of the suspended branch. Resuming the suspended branch (`Found_Resumed_Work`) restores the original contents of the `frame` field, and causes `Node_Suspended(`N`)` to be `FALSE`. The method of reusing the `frame` field in this way is due to Péter Szeredi.

We have already described (see section 5.2.2.3, page 69) how `Sched_Die_Back` fits into the normal backtracking mechanism. When the engine resumes control after the other two macros have completed it simply branches to the `public_fail` label which is part of the public backtracking logic.

We shall describe below the actions taken by the engine when pruning or suspending a branch, prior to calling the respective scheduler macro.

5.4.1 Pruning a Branch

The node stack pointer is set at the bottom of the local region and is saved for later access when a new embryonic node is to be created for a new assignment for the current worker. The engine leaves the shallow backtracking phase if necessary, deinstalls all conditional bindings of the current assignment, and marks all remote nodes as reclaimable, except the sentry node.

```
kill_task:
  {
    TAGGED *tr = w->trail_top;
    TAGGED *trlim = w->sentry_node->parent->trail_top;
    struct node *n = w->own_sentry_node;
```

```
    w->choice_top = n;
    MyRoot->own_node = n;      /* save node stack pointer */
    my_position = w->sentry_node->parent;
    w->next_alt = NULL;
    while (tr ≠ trlim)
      DeinstallBinding(tr);
    while (n ≠ w->sentry_node)
      Mark_Node_Reclaimable(n),
      n = n->parent;
  }
  Sched_Be_Pruned(w->sentry_node, goto exit_wam;);
  goto public_fail;
```

5.4.2 Suspending a Branch

A special complication arises from the fact that pruning operators, which compile to WAM instructions, may cause suspension. Thus the emulator must be capable of preserving the exact state at the point of the pruning operator, including all live argument registers, the embryonic node at predicate entry, and the program pointer (the `w->term[]`, `w->previous_node`, and `w->insn` registers, respectively). When the suspended branch is resumed, the state is restored again, and execution continues at the same point in the program.

Suspension can be thought of as a call to a predicate `synch(...)` being inserted into the code, where `synch` is defined as:

```
synch(...) :- fail.
synch(...).
```

The arguments of the call to `synch(...)` are the argument registers that need to be saved at the moment of suspension. The state of the node stack at suspension corresponds to the moment when the first clause of `synch` has just failed. At that moment of execution, all the information relating to the current execution is stored on the stacks. Resumption can be viewed as taking the second alternative of the `synch` predicate.

The above account is a somewhat simplified picture of the suspension mechanism. Saving the state is actually implemented by taking the following steps:

- An environment is created, and the `w->previous_node` register is stored as one of its permanent variables. Having done this, the continuation register `w->next_insn` becomes available, and the program counter `w->insn` is stored in it.

- The current branch is sealed by creating a node containing all live temporary variables, simulating a call to `synch`. The scheduler is notified about the simulated call too. The node will have one alternative (corresponding to the second `synch` clause) which points at the instruction `deallocate_and_jump` (q.v.). Upon resumption, that instruction is responsible for restoring the state just as it was when the branch was suspended.
- The local region of the trail stack is completed by filling in the values of all conditional bindings.
- The sentry node is converted to a suspended sentry node, saving the tip node's `frame` field in the newly created environment.
- The node stack pointer is saved for later access when a new embryonic node is to be created for a new assignment for the current worker.

```
suspend_task:
  {
    int arity = NRegs(w->insn);

    w->frame2 = w->local_top;
    Y(0) = TagChoicepoint(w->previous_node);
    Y(1) = (TAGGED)w->sentry_node->frame;
    w->frame2->local_var = w->local_var;
    w->frame2->frame = w->frame;
    w->frame2->next_insn = w->next_insn;
    w->frame = w->frame2;
    w->next_insn = w->insn;
    w->local_top = &Y(2);
    w->local_var++;

    w->previous_node = w->choice_top;
    make_node_partial();
    Sched_Clause_Entered(w->sentry_node, suspend_alt[arity], TRUE);
    w->next_alt = suspend_alt[arity]->next;
    make_node_complete();

    {
      TAGGED *tr = w->node->trail_top;
      TAGGED *trlim = w->own_sentry_node->parent->trail_top;

      while (tr ≠ trlim)
        PublishBinding(tr);
    }

    w->sentry_node->frame = (struct frame *)(w->node);
    MyRoot->own_node = w->choice_top; /* save node stack pointer */
    my_position = w->node;
    Sched_Suspend(w->sentry_node, goto exit_wam;);
    goto public_fail;
  }
```

where `suspend_alt` is an array of references to chains of two alternatives representing the two `synch` clauses above, indexed by arity. The second alternative contains a single `deallocate_and_jump` instruction. The expression `NRegs(...)` accesses an operand of the instruction that caused the suspension, indicating how many temporary variables that are live at that point.

5.5 Straightening and Contraction

5.5.1 Introduction

Straightening and contraction are significant optimisations of the SRI model, as they can considerably reduce the set of nodes that the scheduler has to consider when looking for work as well as the amount of movement in the tree. These optimisations are trivially implemented in sequential implementations, since dead nodes can immediately be reclaimed by the usual stack mechanism, even if they are still logically part of the computation. Aurora uses this simple mechanism to implement contraction (trusting a node) and straightening (pruning nodes) in the local region.

The situation is more complicated for the remote and public regions, however, since the engine cannot reclaim nodes by the simple stack mechanism unless they are private and adjacent to the top of the stack. Instead, nodes in these regions have to be kept in the tree until they have died back, at which time they are marked as reclaimable and can be reclaimed by their owner. In other words, contraction and straightening are effectively disabled in these regions, and if remote nodes are made public, the scheduler may encounter dead nodes. Such nodes are useless for the scheduler in most respects. Therefore, it may choose to *bypass* such nodes, i.e. to delete them from the scheduler's parent-child chains, whenever they appear. For this purpose, the scheduler is notified by the engine whenever a remote node dies (`Sched_Node_Destroyed`). A "bypassing" scheduler will have to exercise extra care when bypassing a dead remote sentry node, to maintain the topology of the tree.

A "bypassing" scheduler typically maintains its own parent-child chains, and may have little or no use for the `Node_Parent` function. If this is the case, it may advise the engine that the scheduler accepts a more compact, but slower, implementation of the `Node_Parent` function. For this purpose, the engine can run in its alternative *compact-parent configuration*, in which the parent field is not stored as an explicit choicepoint field. It is up to the engine to choose which configuration to use; the scheduler merely offers its opinion. The compact-parent configuration is described below.

Support for contraction proper is provided in both configurations by the `Sched_Get_Work_At_Parent` macro (q.v.). This macro gives the engine the opportunity to reuse a public node as the sentry node, effectively contracting the public region by one node. This is possible if the node is located at the top of the worker's node stack.

5.5.2 Implementation

The key idea in the implementation of the compact-parent configuration is due to Péter Szeredi who made the observation that the `parent` field is almost redundant to the engine, since it has the same value as the `own_node` field in all nodes except in sprout nodes (the initial nodes of new cactus stack arms). In the default configuration, sprout nodes are initially embryonic and are not treated specially in the implementation. When a new arm is started in the compact-parent configuration, a complete *auxiliary node* is allocated for the sole purpose of recording the parent node of the new arm, prior to creating the embryonic sprout node. The auxiliary node does not logically belong to the tree but provides some unused fields for storing the logical parent. The `frame` field is arbitrarily chosen for this purpose.

The descriptions given earlier in previous sections have assumed the default configuration. We discuss below the impact by the compact-parent configuration on the implementation.

implementing `parent` *fields*

In the compact-parent configuration, *NODE*`->parent` is not explicitly stored in embryonic nodes. Instead, it is a shorthand for the expression

```
(NODE->own_node->level ≡ -2 ?
    (struct node *)(NODE->own_node->frame) :
    NODE->own_node)
```

allocating embryonic nodes

When these extend an existing arm there is no change. But as described in the previous section, when they start a new arm, they first allocate an auxiliary node and place the embryonic sprout node adjacent to it. In addition to the `frame` and `own_node` fields, the `trail_top`, `local_top`, and `global_top` fields acquire values for memory allocation purposes. Finally, the `level` field is set to `-2` to identify the node as an auxiliary node. All other fields are left undefined.

unstitching nodes

When backtracking over a dead remote node, the `frame` field of the auxiliary node, which holds the logical parent, must be updated, since the (old) parent is no longer part of the computation. However, if the remote section winds back to the top of the worker's own stack, the memory occupied by the (now useless) auxiliary node must be reclaimed, and the boundary between the remote and local regions must be recomputed.

```
unstitch_node()
{
  if (w->node ≡ w->own_node->own_node)
  {                              /* reclaim auxiliary node,
                                    recompute boundary */
    w->choice_top = w->own_node;
    w->own_node = w->node;
```

```
    compute_own_sentry_node();
  }
  else                              /* update auxiliary node */
    w->own_node->frame = (struct node *)(w->node);
                                    /* and update sentry node */
  Sched_Private_Parent_Stored(w->sentry_node, w->choice_top, w->node);
  w->choice_top->level = w->node->level+1;
}
```

Figure 5-2 illustrates how a "bypassing" scheduler might arrange its parent-child pointers for three different cases. The logical parent of the local region is stored in the auxiliary node. In all three cases, the remote region consists of a (live) sentry node and adjacent node 'X', and the local region is empty (consists of the embryonic node only). The node 'X' is initially live (1). When 'X' is trusted or pruned, it becomes bypassed (2). Finally, when the worker dies back to the sentry node, 'X' is marked as reclaimable, and the `frame` field of the auxiliary node is set pointing at the sentry node, which is now the logical extension of the local region (3).

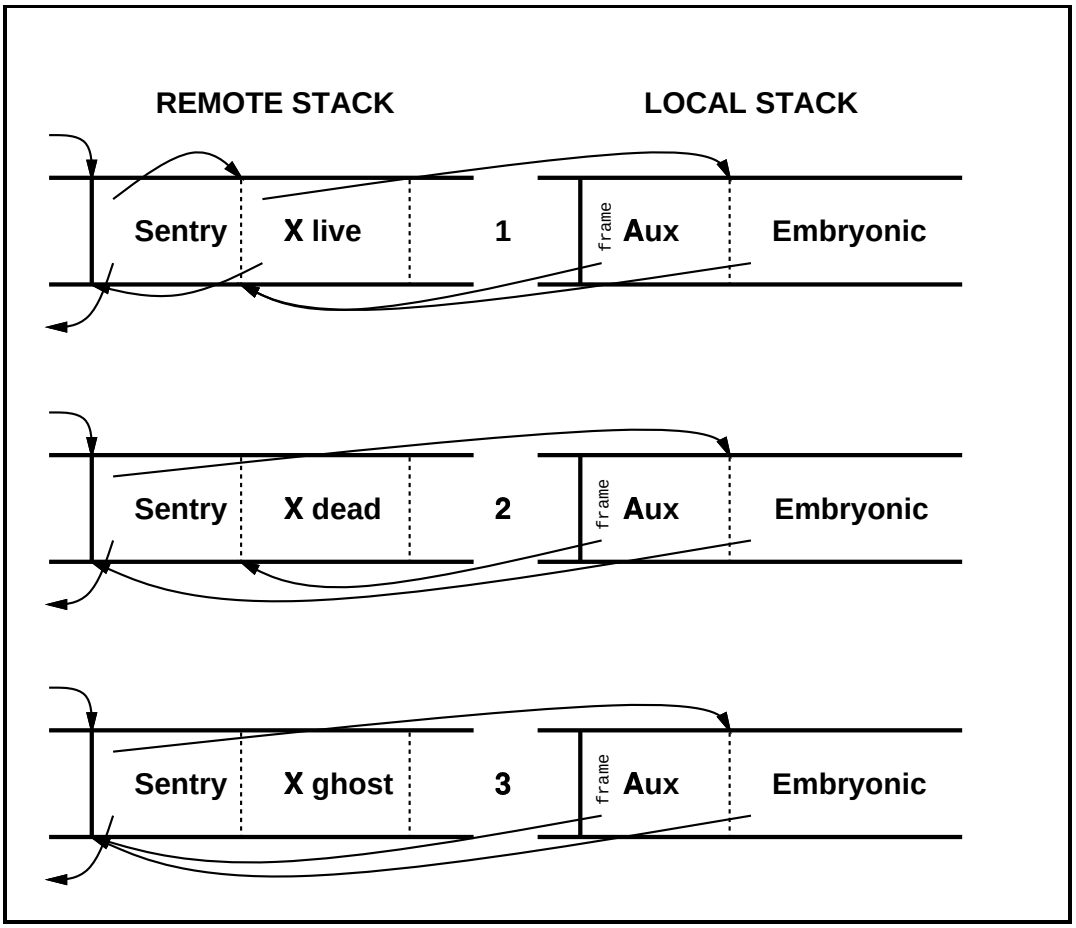

Figure 5-2: Bypassing and dying back over remote nodes

5.6 Macros Provided to the Scheduler

As part of the agreed protocol between the scheduler and the engine, the engine has to provide various services to the scheduler. The major services are described in the table below. Other services include support for shared memory allocation and locking. See section 3.4, page 25.

`int Node_Level(struct node *NODE)`
`struct node *Node_Parent(struct node *NODE)`
`struct alternative *Node_Alternatives(struct node *NODE)`

These are trivial access macros for the `level`, `parent`, and `next_alt` node fields, respectively.

`struct alternative *Alternative_Next(struct alternative *ALT)`

This is a trivial access macro for the `next` field of alternative records.

`void Found_Resumed_Work(struct node *NEW_SENTRY)`

This macro is responsible for bringing the engine back to a state at which a suspension occurred earlier. The argument `NEW_SENTRY` always refers to the sentry node of a suspended branch, and the worker is positioned at or above its parent. The relevant registers are set up so that the engine is ready to execute the normal deep backtracking code.

- The sentry node register is set, the tip node is retrieved from the sentry node, and the `w->previous_node` and the contents of the sentry node's `frame` pointer is restored.
- All conditional bindings made in the resumed branch are installed, thus positioning the engine at its tip.
- An embryonic node is allocated, and the single, formal alternative of the tip node is retrieved.
- The boundary between the remote and local parts is computed.
- The trail stack pointer is set and linked with its logical extension if the tip node is not on the worker's own stack.

```
{
  w->sentry_node = NEW_SENTRY;
  w->node = (struct node *)(w->sentry_node->frame);
  w->frame2 = w->node->frame;
  w->previous_node = ChoicepointOf(Y(0));
  w->sentry_node->frame = (struct frame *)(Y(1));

  Move_Engine_Down(w->node);
  Allocate_Node(w->node, w->choice_top);
  Sched_Private_Parent_Stored(w->sentry_node, w->choice_top, w->node);
  w->own_node = w->choice_top->own_node;
  w->next_alt = w->node->next_alt;
  if (w->node ≡ w->own_node)
  {                              /* extend the current arm */
```

```
    w->trail_top = w->node->trail_top;
    compute_own_sentry_node();
  }
  else
  {                              /* start a new arm */
    w->trail_top = w->own_node->trail_top;
    TrailLink(w->trail_top, w->node->trail_top);
    w->own_sentry_node = w->choice_top;
  }
}
```

`void Found_New_Work(struct node *SENTRY, struct alternative *ALT)`

This macro is responsible for preparing the engine for exploring a new alternative. The argument `SENTRY` refers to a previously allocated embryonic node, the worker is positioned at or above its parent, and `ALT` is an alternative taken from some public node by the scheduler. The relevant registers are set up so that the engine is ready to execute the normal deep backtracking code. The private region of the new (or extended) branch will initially be empty.

```
{
  w->next_alt = ALT;
  w->sentry_node = SENTRY;
  w->own_sentry_node = w->sentry_node;
  w->choice_top = w->sentry_node;
  w->own_node = w->sentry_node->own_node;
  w->node = w->sentry_node->parent;
  Move_Engine_Down(w->node);
  if (w->node ≡ w->own_node)
  {                              /* extend the current arm */
    w->trail_top = w->node->trail_top;
  }
  else
  {                              /* start a new arm
    w->trail_top = w->own_node->trail_top;
    TrailLink(w->trail_top, w->node->trail_top);
  }
}
```

`void Move_Engine_Down(struct node *DOWN_TO)`

The effect of this is to install all conditional bindings residing in the trail segments between nodes the current position and `DOWN_TO`, and to record the current position as `DOWN_TO`.

```
{
  TAGGED *tr = DOWN_TO->trail_top;
  TAGGED *trlim = my_position->trail_top;

  while (tr ≠ trlim)
    InstallBinding(tr);
```

```
  my_position = DOWN_TO;
}
```

`void Move_Engine_Up(struct node *UP_TO)`

The effect of this is to deinstall all conditional bindings residing in the trail segments between nodes the current position and `UP_TO`, and to record the current position as `UP_TO`.

```
{
  TAGGED *tr = UP_TO->trail_top;
  TAGGED *trlim = my_position->trail_top;

  while (tr ≠ trlim)
    DeinstallBinding(tr);
  my_position = UP_TO;
}
```

`void Migration_Cost(struct node *FROM, struct node *DOWN_TO, int COST)`

The effect of this is to compute as `COST` the migration cost of changing the engine's position from `FROM` to `DOWN_TO`. The estimate is implemented by simply counting the bindings recorded in the trail segments.

```
{
  TAGGED *tr = DOWN_TO->trail_top;
  TAGGED *trlim = FROM->trail_top;
  int cost = 0;

  while (tr ≠ trlim)
    CountBinding(tr, cost);

  COST = cost;
}
```

`void Allocate_Node(struct node *PARENT, struct node *EMBRYONIC)`
`void Allocate_Foreign_Node(struct node *STACK, struct node *PARENT,`
`  struct node *EMBRYONIC)`

where the `STACK` argument defaults to the worker's initial node (`MyRoot`), computes as `EMBRYONIC` an embryonic node by moving the stack pointer as far as possible, until the tip node is no longer a ghost. The `parent` and `level` fields are initialised in the embryonic node; the `own_node` field has a valid value already.

```
{
  struct node *top = STACK->own_node;
  struct node *above = top->own_node;

  while (Node_Reclaimable(above))
    STACK->own_node = above,
```

```
    top=above,
    above=above->own_node;

  top->parent = PARENT;
  top->level = PARENT->level+1;

  EMBRYONIC = top;
}
```

`void Mark_Node_Reclaimable(struct node *NODE)`

A single node is marked as reclaimable.

```
{
  NODE->level = -1;
}
```

`void Mark_Suspended_Branch_Reclaimable(struct node *SENTRY)`

This macro marks an entire suspended branch as reclaimable, up to but excluding the sentry node.

```
{
  struct node *n = (struct node *)(SENTRY->frame);

  while (n ≠ SENTRY)
    Mark_Node_Reclaimable(n),
    n = n->parent;
}
```

`BOOL Node_Reclaimable(struct node *NODE)`

This is `TRUE` if `NODE` has been marked as reclaimable.

```
{
  return NODE->level < 0;
}
```

`void Make_Public(struct node *NEW_SENTRY)`

This macro is used when the scheduler wants to extend the public region of the tree, making `NEW_SENTRY` the new sentry node. The relevant boundaries are adjusted. If there are any local nodes in the scope of this operation, they must first be completed by filling in the values of conditional bindings made in the corresponding stack segments.

```
{
  w->sentry_node = NEW_SENTRY;

  if (w->sentry_node->level > w->own_sentry_node->level)
  {
    TAGGED *tr = w->sentry_node->parent->trail_top;
```

```
    TAGGED *trlim = w->own_sentry_node->parent->trail_top;

    while (tr ≠ trlim)
      PublishBinding(tr);

    w->own_sentry_node = w->sentry_node;
  }
}
```

`struct node *My_Embryonic_Node()`

This macro returns the current embryonic node of the worker, while it is not engaged in public backtracking.

```
{
  return w->choice_top;
}
```

---

6. Instruction Set

6.1 Introduction

In this chapter we give semantics for the Aurora instruction set, which is materially the same instruction set as the one used in SICStus Prolog, which in turn is an extended subset of the WAM instruction set. One of the features of the SICStus emulator is that none of the original WAM indexing instructions is used. Instead, an indexing tree is incrementally built up per predicate as each clause is compiled, and the emulator uses such trees instead of executing indexing instructions. The compiler algorithms are described in [Carlsson 90].

The instruction set extensions can be divided into the following classes:

- *Support for full Prolog.* The original WAM instruction set does not cover the full language. Efficient implementation of operations such as pruning and arithmetic requires extensions so that such operations can be performed in-line. These extensions are provided by the `choice`, `cut`, `function`, and `builtin` instructions.

  A mechanism exists for calling dynamically loaded C functions. This mechanism requires further extensions to perform the necessary parameter conversions (see section 7.5, page 110).
- *Indexing support.* Although the WAM indexing instructions are not used, the general *get* instructions can sometimes be replaced by specialised versions which rely on the fact that the first argument register has been dereferenced and has been found to contain the desired value. This optimisation is provided by the `get...x0` instructions.
- *Memory management support.* At every procedure call, the emulator checks to see that a certain fixed amount of free memory exists beyond the global stack pointer. To allow for clauses whose first chunk might consume more than this amount, the compiler inserts before all other instructions a `heapmargin_call` instruction to ensure that a sufficient amount exists. Similarly for the first chunk of a clause continuation (always immediately preceded by a `call` instruction), a `heapmargin_exit` instruction is inserted before all other instructions if the chunk might consume more memory than the amount allowed by default.

  To support garbage collection, the implementation insists that there be no dangling or undefined pointers in any environment when a general procedure call occurs. This is ensured by initialising all permanent variables in the first chunk and replacing `put_variable`, `get_variable` and `unify_variable` instructions in subsequent chunks by `put_value`, `get_first_value` and `unify_first_value` instructions, respectively. The latter two instructions assume that the permanent variable is currently unbound and ensure by trailing that it will stay unbound if the execution could backtrack to a choicepoint created after the environment. See [Appleby et al. 88] for details about the garbage collection algorithm.

- *Shallow backtracking support.* The crucial instruction in the shallow backtracking implementation is the `neck` instruction, which marks the "commitment point" in the clause, i.e. where one must decide whether or not to allocate a choicepoint. There is one `neck` instruction per clause, and it must be placed before any *deep instruction*, where a deep instruction modifies such registers whose values need to be preserved over the shallow backtracking phase, in particular before any pruning operators, instructions that transfer control to another clause, or instructions that alter the head arguments. Whether or not an instruction is deep sometimes depends on the context, since `put` instructions sometimes alter head arguments and sometimes don't.

  The neck instruction is actually placed immediately after the last instruction that can cause backtracking (if any), but before any deep instruction. It is pointless to place it after an instruction that cannot fail, and placing it later than necessary may actually degrade the compiler's register allocation.

  To enhance the efficiency of shallow backtracking, the WAM `allocate` instruction has been split into `allocate1` and `allocate2` (see section 4.2.2, page 38), where `allocate2` is deep and `allocate1` is not. This measure ensures that the value of `w->frame` is preserved over the shallow backtracking phase.

  Finally, the `branch` instruction is introduced to support two entrypoints for some clauses, representing code streams to be used in the shallow and deep backtracking phase respectively. The point is that the `neck` instruction is a no-op in the deep backtracking phase, and may degrade register allocation, so the second code stream can be significantly more efficient than the first. At some point there is a merger between the code streams, implemented by the `branch` instruction.

- *SRI model support.* The impact on the instruction set by the SRI model is quite small. The most important change is the support for assigning variable numbers to local variables and generalising environment trimming to apply to the binding array. This is supported by the extra `VarCount` argument in `call` instruction, and a similar extra argument in `put_variable` and `put_void` instructions.

  Pruning operators compile to `cut` instructions, which acquire an extra argument to distinguish the type of pruning operator (cut or commit). Since pruning operators can cause suspension, `cut` instructions also need an extra argument denoting which argument registers are currently live.

  Finally, resumption of suspended branches is supported by the `deallocate_and_jump` instruction. This instruction is really a pseudo-instruction whose sole purpose is to aid in bringing the engine back to a previous state, and is not strictly necessary. It only occurs in the formal alternative of the tip node of suspended branches, and never in compiled clauses. See section 5.4.2, page 76.

6.2 Support Macros

The following primitive operations implement dereferencing and variable binding, and are used by the various instructions and by the general unification algorithm.

`TAGGED RegisterValue(TAGGED V)`

This evaluates to the value of a temporary or permanent variable denoted by `V`, where `V` is an operand of some instructions and is written as '`x(N)`' or '`y(N)`', respectively.

For temporary variables, this is simply its contents. For permanent variables, `V` is an environment stack location, which could be the value cell of an unbound permanent variable. This macro is really a *meta macro* which has been introduced to make the descriptions of the instructions more readable.

```
{
  if (V is written as x(N))
    return X(N);
  else                              /* V is written as y(N) */
    return RefLocation(&Y(N));
}
```

`TAGGED RefLocation(TAGGED *L)`

This evaluates to the contents of the global or environment stack location `L`, or to a reference to that location, if it contains an unbound variable.

```
{
  TAGGED t1 = *L;

  if (TagOf(t1) ≡ VNO)
  {
    t1 = CondBinding(t1);
    if (t1 ≡ 0)
      t1 = Tag(REF, L);
  }
  return t1;
}
```

`TAGGED RefNext()`

This evaluates to the contents of the global stack location `w->structure`, or to a reference to that location, if it contains an unbound variable. `w->structure` is incremented.

```
{
  TAGGED t1 = RefLocation(w->structure);

  w->structure++;

  return t1;
```

```
}
```

`TAGGED UnsafeValue(TAGGED V)`

This evaluates to the dereferenced value of the variable reference `V`, unless it dereferences into the current environment, in which case it is globalised, i.e. bound to a new global variable.

```
{
  TAGGED t1 = DerefValue(V);

  if (TagOf(t1) ≡ REF ∧ *ValueOf(t1) ≥ w->frame2->local_var)
  {
    TAGGED t2 = NewVAR();

    BindVAR(t1, t2);
    return t2;
  }
  else
    return t1;
}
```

`TAGGED DerefValue(TAGGED T)`

Evaluates to the dereferenced value of `T`.

```
{
  TAGGED last_ref;

start:
  switch (TagOf(T))
  {
  case REF:
   last_ref = T;
   T = *ValueOf(T);
   goto start;
  case VNO:
   T = CondBinding(T);
   if (T ≡ 0)
     return last_ref;
   else
     goto start;
  case CON:
  case LST:
  case STR:
    return T;
  }
}
```

`void WriteLocalValue(TAGGED X)`

Stores the term `X` on the global stack, binding it to a new global variable if necessary.

```
{
  TAGGED t1 = DerefValue(X);

  if (TagOf(t1) ≡ REF ∧ *ValueOf(t1) ≥ Tag(VNO, 0))
    BindVAR(t1, NewVAR());
  else
    *w->global_top++ = t1;
}
```

`void BindVAR(TAGGED U, TAGGED V)`

The variable `U` is bound to the term `V`.

```
{
  if (CondVAR(U))
  {
    TrailBinding(w->trail_top, U);
    CondBinding(U) = V;
  }
  else
    *ValueOf(U) = V;
}
```

The general unification algorithm is implemented as the Boolean function `unify_terms(u, v)` and can be expressed as a decision table, indexed by the tags of the two arguments. The terms to be unified are first dereferenced and compared. If the two tagged pointers are found equal, unification succeeds, otherwise the table is consulted.

```
BOOL unify_terms(TAGGED u, TAGGED v)
{
  TAGGED a = DerefValue(u);
  TAGGED b = DerefValue(v);
  if (a ≡ b)
    return TRUE;
  else
    perform case analysis on TagOf(a) and TagOf(b):
}
```

	REF	CON	LST	STR
REF	1	2	2	2
CON	2	5	5	5
LST	2	5	3	5
STR	2	5	5	4

where the numbers in the table describe the action taken:

1. Two variables are unified by first forming a ordered pair <*x*,*y*> with respect to the following ordering: younger local variables precede older local variables, which precede younger global variables, which precede older global variables. Then *x* is bound to *y*.

```
{
  TAGGED t1 = *ValueOf(a);
  TAGGED t2 = *ValueOf(b);

  if (t1 > t2 ∧ t2 ≥ Tag(VNO, 0) ∨
      t1 ≥ Tag(VNO, 0) ∧ Tag(VNO, 0) > t2 ∨
      t1 < t2 ∧ t2 < Tag(VNO, 0))
    BindVAR(a, b);
  else
    BindVAR(b, a);
  return TRUE;
}
```

2. A variable is bound to the other term.

```
{
  if (TagOf(a) ≡ REF)
    BindVAR(a, b);
  else
    BindVAR(b, a);
  return TRUE;
}
```

3. Both arguments are lists. The algorithm recurses on their arguments.

```
{
  int i=0;
  BOOL rc=TRUE;

  while (i<2 ∧ rc)
  {
```

```
    if (¬unify_terms(RefLocation(ValueOf(a)+i),
                     RefLocation(ValueOf(b)+i)))
      rc = FALSE;
    i++;
  }
  return rc;
}
```

4. Both arguments are structures. If the two compound terms have the same functor, the algorithm recurses on their arguments, otherwise it fails.

```
{
  int i=1;
  BOOL rc = (*ValueOf(a) ≡ *ValueOf(b));

  while (i ≤ ArityOf(a) ∧ rc)
  {
    if (¬unify_terms(RefLocation(ValueOf(a)+i),
                     RefLocation(ValueOf(b)+i)))
      rc = FALSE;
    i++;
  }
  return rc;
}
```

5. Unification fails.

```
{
  return FALSE;
}
```

6.3 Put Instructions

These instructions correspond to body arguments. They prepare arguments for the next procedure call or inline goal.

`put_void(x(`*i*`),_)`

This represents an *i*th goal argument that is a singleton variable.

```
  X(i) = NewVAR();
```

`put_void(y(`*n*`),`*m*`)`

This represents a permanent variable Y(*n*) which does not occur in the first chunk, and

so must be explicitly initialised there. The variable is the mth variable in the current environment to be explicitly initialised.

```
Y(n) = w->local_var+m;
```

`put_variable(x(`n`),`i`,_)`

This represents an ith goal argument that is the first occurrence of the temporary variable X(n).

```
X(i) = X(n) = NewVAR();
```

`put_variable(y(`n`),`i`,`m`)`

This represents an ith goal argument that is the first occurrence of the permanent variable Y(n), if it occurs in the first chunk. The variable is the mth variable in the current environment to be explicitly initialised.

```
X(i) = Tag(REF,&Y(n));
Y(n) = w->local_var+m;
```

`put_value(`Vn`,`i`)`

This represents an ith goal argument that is a subsequent occurrence of the variable Vn which cannot point (even before dereferencing) to a portion of the environment stack which is about to be deallocated.

```
X(i) = RegisterValue(Vn);
```

`put_unsafe_value(`Vn`,`i`)`

This represents an ith goal argument that is a subsequent occurrence of the variable Vn which might point to a portion of the stack which is about to be deallocated, i.e. it might need globalising.

```
X(i) = UnsafeValue(RegisterValue(Vn));
```

`put_constant(`C`,`i`)`

This represents an ith goal argument that is the constant C.

```
X(i) = C;
```

`put_nil(`i`)`

This represents an ith goal argument that is the empty list.

```
X(i) = '[]';
```

`put_structure(`F`,`i`)`

This represents an ith goal argument that is a structure whose functor is F. The instruction is followed by a sequence of `unify` instructions.

```
X(i) = Tag(STR, w->global_top);
*w->global_top++ = F;
w->structure = NULL;
```

`put_list(`i`)`

This represents an ith goal argument that is a list. The instruction is followed by a sequence of `unify` instructions.

```
X(i) = Tag(LST, w->global_top);
w->structure = NULL;
```

6.4 Get Instructions

These instructions correspond to head arguments and to certain inline goal arguments. They match head arguments against actual arguments and values computed by inline goals against other terms.

`get_variable(x(`n`),`i`)`

This represents an ith head argument or the result of an inline goal that is the first occurrence of the temporary variable X(n).

```
X(n) = X(i);
```

`get_variable(y(`n`),`i`)`

This represents an ith head argument or the result of an inline goal *in the first chunk* that is the first occurrence of the permanent variable Y(n).

```
Y(n) = X(i);
```

`get_first_value(y(`n`),`i`)`

This represents an ith head argument or the result of an inline goal that is the first occurrence of the permanent variable Y(n) occurring after the first chunk.

```
BindVAR(Tag(REF,&Y(n)), X(i));
```

`get_value(`Vn`,`i`)`

This represents an ith head argument or the result of an inline goal that is a subsequent occurrence of the variable Vn.

```
if (¬unify_terms(RegisterValue(Vn), X(i)))
  backtrack();
```

`get_constant(`C`,`i`)`

This represents an ith head argument or the result of an inline goal that is the constant C.

```
if (¬unify_terms(C, X(i)))
  backtrack();
```

`get_nil(`i`)`

This represents an ith head argument or the result of an inline goal that is the empty list.

```
if (¬unify_terms('[]', X(i)))
  backtrack();
```

`get_structure(`F`,`i`)`

This represents an ith head argument or the result of an inline goal that is a structure with the functor F. The instruction is followed by a sequence of `unify` instructions.

```
{
  TAGGED t1 = DerefValue(X(i));

  switch (TagOf(t1))
  {
  case REF:
    BindVAR(t1, Tag(STR, w->global_top));
    *w->global_top++ = F;
    w->structure = NULL;
    break;
  case STR:
    if (*ValueOf(t1) ≠ F)
      backtrack();
    else
      w->structure = ValueOf(t1)+1;
    break;
  case CON:
  case LST:
    backtrack();
```

```
  }
}
```

`get_list(`*i*`)`

This represents an *i*th head argument or the result of an inline goal that is a list. The instruction is followed by two `unify` instructions.

```
{
  TAGGED t1 = DerefValue(X(i));

  switch (TagOf(t1))
  {
  case REF:
    BindVAR(t1, Tag(LST, w->global_top));
    w->structure = NULL;
    break;
  case LST:
    w->structure = ValueOf(t1);
    break;
  case CON:
  case STR:
    backtrack();
  }
}
```

6.5 Unify Instructions

These instructions correspond to arguments of compound terms. Each one has two modes of operation, read and write. In read mode, the structure pointer (`w->structure`) points to the next structure argument that the instruction should match. In write mode, the structure pointer is `NULL` and the next structure argument is written to the top of the global stack.

`unify_void`

This represents a structure argument that is a singleton variable.

```
  if (w->structure)
    w->structure++;
  else
    NewVAR();
```

`unify_variable(x(`*n*`))`

This represents a structure argument that is the first occurrence of the temporary variable X(*n*).

```
if (w->structure)
  X(n) = RefNext();
else
  X(n) = NewVAR();
```

`unify_variable(y(`n`))`

This represents a structure argument that is the first occurrence of the permanent variable Y(n), *occurring in the first chunk.*

```
if (w->structure)
  Y(n) = RefNext();
else
  Y(n) = NewVAR();
```

`unify_first_value(y(`n`))`

This represents a structure argument that is the first occurrence of the permanent variable Y(n), occurring after the first chunk.

```
if (w->structure)
{
  BindVAR(Tag(REF,&Y(n)), RefNext());
}
else
{
  BindVAR(Tag(REF,&Y(n)), NewVAR());
}
```

`unify_value(`Vn`)`

This represents a structure argument that is a subsequent occurrence of the variable Vn which cannot (even before dereferencing) be pointing to the stack.

```
if (w->structure)
{
  if (¬unify_terms(RegisterValue(Vn), RefNext()))
    backtrack();
}
else
  *w->global_top++ = RegisterValue(Vn);
```

`unify_local_value(`Vn`)`

This represents a structure argument that is a subsequent occurrence of the variable Vn which could be pointing to the stack, i.e. it could need globalising.

```
if (w->structure)
{
  if (¬unify_terms(RegisterValue(Vn), RefNext()))
```

```
    backtrack();
  }
  else
    WriteLocalValue(RegisterValue(Vn));
```

`unify_constant(`C`)`

This represents a structure argument that is the constant C.

```
  if (w->structure)
  {
    if (¬unify_terms(C, RefNext()))
      backtrack();
  }
  else
    *w->global_top++ = C;
```

`unify_nil`

This represents a structure argument that is the empty list.

```
  if (w->structure)
  {
    if (¬unify_terms('[]', RefNext()))
      backtrack();
  }
  else
    *w->global_top++ = '[]';
```

`unify_structure(`F`)`

This represents a last structure argument that is a structure with the functor F.

```
  if (w->structure)
  {
    TAGGED t1 = DerefValue(RefNext());

    switch (TagOf(t1))
    {
    case REF:
      BindVAR(t1, Tag(STR, w->global_top));
      *w->global_top++ = F;
      w->structure = NULL;
      break;
    case STR:
      if (*ValueOf(t1) ≠ F)
        backtrack();
      else
        w->structure = ValueOf(t1)+1;
      break;
    case CON:
    case LST:
      backtrack();
```

```
  }
else
{
  TAGGED t1 = Tag(STR, w->global_top+1);

  *w->global_top++ = t1;
  *w->global_top++ = F;
}
```

`unify_list`

This represents a last structure argument that is a list.

```
if (w->structure)
{
  TAGGED t1 = DerefValue(RefNext());

  switch (TagOf(t1))
  {
  case REF:
    BindVAR(t1, Tag(LST, w->global_top));
    w->structure = NULL;
    break;
  case LST:
    w->structure = ValueOf(t1);
    break;
  case CON:
  case STR:
    backtrack();
  }
else
{
  TAGGED t1 = Tag(LST, w->global_top);

  *w->global_top++ = t1;
}
```

6.6 Procedural Instructions

These instructions correspond to the head and goals of a clause. They deal with recursive predicate calls and the data structures necessary for them. A call to a predicate D is implemented by the procedure `Execute(`D`)`, described below. The effect of this procedure is to update the program counter to a value that the emulator should proceed from, after giving the scheduler the opportunity to attend to any outstanding parallel business.

Predicates are normally compiled, but can also be interpreted, in which case they are interpreted by the compiled predicate `meta_interpreter/1`, or built into the emulator, in which case they are

implemented as C functions. Such functions retrieve their arguments from the argument registers `X(...)`.

```
Execute(D)
{
  Sched_Check(w->sentry_node, goto kill_task;, goto suspend_task;);
  if (D is built into the emulator as the function F)
  {
    if (¬F())
      backtrack();
    else
      w->insn = w->next_insn,
      w->frame2 = w->frame;
  }
  else if (D is not a compiled predicate)
  {
    X(0) = construct_goal(D);
    Execute(meta_interpreter/1);
  }
  else
  {
    struct alternative *alt = filter_clauses(D);

    w->previous_node = w->choice_top;
    w->insn = alt->insn;
    if ((w->next_alt = alt->next) ≠ NULL)
      make_node_partial();
    Sched_Clause_Entered(w->sentry_node, alt, TRUE);
  }
}
```

where `construct_goal(D)` constructs the current goal as a term on the global stack, in preparation for calling the meta-interpreter, and `filter_clauses(D)` evaluates to a pointer to a chain of alternative clauses that might match the current goal, filtered out by indexing on the first argument.

Thus for predicates implemented as C functions, the program counter is set to the value of the continuation pointer, and for compiled predicates it is set to the beginning of the first possibly matching clause. A partial choicepoint is made if there are candidates for alternative matching clauses. Finally, the scheduler is notified about the alternative being entered.

`allocate1`

This appears before the first reference to a permanent variable in a recursive clause. It must be matched by a `deallocate` instruction. Space for a new environment is allocated on the stack.

```
w->frame2 = w->local_top;
```

`allocate2`

This appears just before the first procedure call of a recursive clause, i.e. is always followed by a `call(`*Def*`,`*EnvSize*`,`*VarCount*`)` instruction. The new environment is completed. The environment stack pointer and local variable number are updated by accessing the operands of the `call` instruction. The actual emulator code uses a fused `allocate2 + call` instruction.

```
w->frame2->frame = w->frame;
w->frame2->local_var = w->local_var;
w->frame2->next_insn = w->next_insn;
w->frame = w->frame2;
w->local_top += EnvSize;
w->local_var += VarCount;
```

`deallocate`

This appears after computing the arguments of the last goal in a recursive clause. The current environment is deallocated, updating the current environment and continuation registers, and conditionally the environment stack pointer and local variable number.

```
w->frame = w->frame2->frame;
w->next_insn = w->frame2->next_insn;
if (w->local_uncond < w->local_var)
  w->local_top = w->frame2,
  w->local_var = w->frame2->local_var;
```

`call(`*Def*`,`*EnvSize*`,`*VarCount*`)`

This corresponds to a procedure call that does not terminate a clause. The argument *EnvSize* is the number of variables that are active in the environment; *VarCount* is the number of active variables in the environment which were explicitly initialised before the first procedure call. *Def* is the address of a data structure representing a predicate.

```
w->next_insn = w->insn;
if (w->local_uncond < w->local_var)
  w->local_top = w->frame2+EnvSize,
  w->local_var = w->frame2->local_var+VarCount;
Execute(Def);
```

The actual implementation avoids the test and update of `w->local_top` and `w->local_var` if this is the *first* `call` instruction in the clause, as the preceding `allocate2` instruction has already updated them.

`execute(`*Def*`)`

This corresponds to a procedure call that terminates a clause. *Def* is the address of a data structure representing a predicate.

```
Execute(Def);
```

`proceed` This terminates a clause not terminated by a procedure call.

```
w->insn = w->next_insn;
w->frame2 = w->frame;
```

`fail` This causes backtracking to an alternative clause.

```
backtrack();
```

6.7 Indexing Instructions

These instructions interact with the clause indexing mechanism which in the emulator in built into the procedure call mechanism. They also deal with choicepoint handling.

`get_constant_x0(`*C*`)`

This represents a first head argument that is the constant *C*. X(0) is already dereferenced and is either uninstantiated or instantiated to *C*.

```
if (IsVar(X(0)))
  BindVAR(X(0), C);
```

`get_nil_x0`

This represents a first head argument that is the empty list. X(0) is already dereferenced and is either uninstantiated or instantiated to the empty list.

```
if (IsVar(X(0)))
  BindVAR(X(0), '[]');
```

`get_structure_x0(`*F*`)`

This represents a first head argument that is a structure whose functor is *F*. X(0) is already dereferenced and is either uninstantiated or instantiated to a structure whose functor is *F*. The instruction is followed by a sequence of `unify` instructions.

```
if (IsVar(X(0)))
{
  BindVAR(X(0), Tag(STR, w->structure));
  *w->global_top++ = F;
  w->structure = NULL;
}
else
  w->structure = ValueOf(X(0))+1;
```

`get_list_x0`

This represents a first head argument that is a list. X(0) is already dereferenced and is either uninstantiated or instantiated to a list. The instruction is followed by a sequence of `unify` instructions.

```
if (IsVar(X(0)))
{
  BindVAR(X(0), Tag(LST, w->global_top));
  w->structure = NULL;
}
else
  w->structure = ValueOf(X(0));
```

`branch(`*Offset*`)`

This occurs at a point in a clause after a `neck(_)` instruction and indicates a branch from the nondeterministic head code stream to a shared body code stream.

```
w->insn += Offset;
```

`neck(`N`)` This represents a point in a clause between the head and the first procedure call, where it is appropriate to decide whether a choicepoint needs to be allocated or updated. The clause belongs to a predicate of N arguments.

```
if (w->next_alt)
{
  if (w->node ≠ w->choice_top)
  {                              /* complete choicepoint */
    w->node->next_alt = w->next_alt;
    w->next_alt = NULL;
  }
  else
  {                              /* partial choicepoint */
    make_node_complete();
  }
}
```

6.8 Utility Instructions

These instructions deal with cuts, inline goals, stack overflow checks and suspensions.

`choice(`*Vn*`)`

This represents the presence of a pruning operator, either an explicit one or one implicitly introduced by a control structure. A pointer to the oldest choicepoint that the operator must delete is stored in the variable *Vn*.

```
Vn = TagChoicepoint(w->previous_node);
```

`cut(`*Vn*`,`*NRegs*`,`*Op*`)`

This represent a pruning operator, either an explicit one or one implicitly introduced by a control structure, which occurs in a subsequent chunk. The operator resets the node stack pointer from the variable *Vn*. Argument registers `x(0)`, ..., `x(`*NRegs*`)-1` are live at this point, and *Op* determines the type of the pruning operator.

```
w->next_alt = NULL;
prune_tree(ChoicepointOf(Vn), Op);
```

`cut(`*NRegs*`,`*Op*`)`

Equivalent to `cut(`*Vn*`,`*NRegs*`,`*Op*`)` if the operator occurs in the same chunk as the matching `choice(`*Vn*`)` instruction. If there are no more uses of *Vn*, the `choice` instruction can be deleted. Argument registers `x(0)`, ..., `x(`*NRegs*`)-1` are live at this point, and *Op* determines the type of the pruning operator.

```
w->next_alt = NULL;
prune_tree(w->previous_node, Op);
```

`function(`*Name*`,`*k*`,`*i*`)`
`function(`*Name*`,`*k*`,`*i*`,`*j*`)`

These correspond to an application of a builtin function X(*k*) = *Name*(X(*i*)[,X(*j*)]). All other argument registers are preserved.

```
X(k) = Name(X(i)[,X(j)]);
```

`builtin(`*Name*`,`*i*`)`
`builtin(`*Name*`,`*i*`,`*j*`)`
`builtin(`*Name*`,`*i*`,`*j*`,`*k*`)`

These correspond to an application of a builtin predicate *Name*(X(*i*)[,X(*j*)[,X(*k*)]]). The builtin predicate preserves all argument registers and must not create choicepoints.

```
if (¬ Name(X(i)[,X(j)[,X(k)]]))
  backtrack();
```

`heapmargin_call(`*i*`,`*j*`)`

This represents the fact that a first chunk of a clause may require that at least *i* cells remain in the global stack. The clause belongs to a predicate with arity *j*. If less than *i* cells remain in the global stack, a stack overflow routine is invoked.

`heapmargin_exit(`*i*`)`

This represents the fact that a subsequent chunk of a clause may require that at least *i* cells remain in the global stack. If less than *i* cells remain in the global stack, a stack overflow routine is invoked.

`deallocate_and_jump`

This instruction is responsible for partially restoring the engine state as it was at an earlier point when a suspension occurred. It is never emitted by the compiler, and only occurs in the single, formal alternative of the tip node of suspended branches.

The original values of the `w->insn` register is restored, and the dummy environment created when the branch was suspended is deallocated:

```
w->insn = w->next_insn;
w->frame2 = w->frame;
w->frame = w->frame2->frame;
w->next_insn = w->frame2->next_insn;
if (w->local_uncond < w->local_var)
  w->local_top = w->frame2,
  w->local_var = w->frame2->local_var;
```

7. Miscellaneous Implementation Details

Many details about the Aurora implementation have been omitted from this exposition to prevent it from being overly complicated. In this chapter we shall summarise some of these details.

7.1 Constrained Variables

Aurora has inherited from SICStus Prolog the ability to *block* goals until their argument are sufficiently instantiated [Naish 85]. This ability is implemented by *delay primitives*, and involves a new kind of variable terms, *constrained variables*, which associate the variable with some goal to be executed when the variable is bound. The concept of a blocked goal is completely independent of the concept of a suspended branch of the search tree.

Constrained variables are stored on the global stack and are identified by a unique tag value. The implementation actually uses different tag values for local and global variables too, making a total of three different tag values to represent variables, replacing the `REF` tag value. Thus all occurrences of the `REF` tag value in the previous chapters corresponds to occurrences of some of these three tag values. This makes certain case analyses more complex, but also leaves some scope for optimisation, for example when local variables need to be distinguished from global and when testing whether a variable may be unconditionally bound.

The following points summarise the machinery needed for supporting constrained variables. It is inherited almost unchanged from the implementation reported in [Carlsson 87].

- Constrained variables are created by certain delay primitives and by *wait declarations*. In both cases, the constrained variable points at a tuple on the global stack consisting of a value cell and a term, representing the blocked goal. A special "birth" item is also pushed on the trail stack, but is ignored by all operations that use the trail except at the termination of a query, as described below.
- When a constrained variable is bound, the binding is always made conditional, and a counter is incremented to the effect that a goal has become unblocked. The counter is reset to zero upon backtracking.
- The unblocked goal counter is tested at every procedure call, and if nonzero, the relevant constrained variables are found near the top of the trail stack, and the Prolog execution stream is rearranged so that the unblocked goals are executed before the current procedure call is allowed to proceed.

- While collecting the unblocked goals from the trail stack, the corresponding constrained variables are reconsidered to see whether they should be promoted, in analogy with the `tidy_trail()` procedure. That procedure has to cater for the case when the unblocked goal counter is nonzero, in which case it must preserve some number of constrained variables near the top of the trail stack, in the correct order.
- At the end of the execution of a query, the Prolog top level code scans the trail for "birth" items corresponding to goals that are still blocked. Warning messages are issued for such goals.

7.2 Backtrackable Destructive Assignment

Aurora has also inherited from SICStus Prolog a *setarg* feature which makes it possible to destructively modify an argument of a compound term in such a way that the modification is automatically undone upon backtracking. Although foreign to the ideals of logic programming, this feature can be used as a primitive for efficiently implementing significant extensions to Prolog, such as finite domain constraint solvers [Haridi 89].

In SICStus Prolog, this feature is built on a more general feature allowing any Prolog goal to be executed upon backtracking. Aurora does not inherit this more general feature, since is rather hard to reconcile with the SRI model; instead, special machinery is provided in Aurora for the *setarg* feature.

The key idea behind the Aurora *setarg* design is due to Péter Szeredi. By restricting the scope of the *setarg* feature to locations initialised as the value cells of unbound variables, we can achieve a nice property: destructive modifications in one branch are invisible to sibling branches, even if the modified location is being shared between branches. This property is achieved by performing the modifications in the binding array rather than in the global stack, thus treating them somewhat like conditional variable bindings. If the modifications were performed in the global stack, they would have to be synchronised (by suspending until leftmost in the tree), and so would all accesses to modified locations. This could seriously limit the parallelism whereas the present design avoids this problem.

The following points summarise the machinery needed for supporting destructive modifications of compound terms:

- Destructive modifications are recorded on the trail by *setarg items* with value parts that point to *setarg records* on the global stack. A *setarg record* is a tuple <*location*,*old_value*,*new_value*>, where *location* is the global stack location being modified; it must contain a variable number. However, if a setarg record for the same location exists in the current trail segment, the existing setarg record is simply updated instead of creating a new record. This measure speeds up backtracking and conserves space, but can take unbounded time since it involves scanning the current trail segment, and so represents a questionable optimisation.

- Pruning operations extend the embryonic segment group by collapsing it with zero or more adjacent segment groups. Thus several setarg records for the same location can exist in the embryonic trail segment after a pruning operation. The `tidy_trail()` procedure is extended to try to coalesce these, as in the previous item.
- Setarg items are deinstalled by storing the *old_value* (instead of 0) at the proper offset in the binding array. They are installed by storing the *new_value* at the proper offset in the binding array.
- Special care is needed when installing bindings in `Move_Engine_Down()`, since it traverses the trail in reverse chronological order (from the tip towards the root), and could encounter more than one setarg item for the same location, in which case the most recent *new_value* must be installed. This complication is handled by the following method:
  - Setarg items for which the binding array location has a designated bit set are simply ignored, and represent a subsequent occurrence of a setarg item for the same location.
  - If the bit is not set, the *new_value* is installed and a counter is incremented.
  - The trail is traversed in a second pass which turns off the bit in the relevant binding array locations and decrements the counter, until the counter value reaches zero.

Certain operations, such as asserting dynamic clauses and saving a solution to an all solutions predicate, involve copying a term from the global stack into the static area. The copy is represented as a sequence of instructions to reconstruct the saved term when executed. When constrained variables occur in such terms, their associated constraints are preserved in the copy. The instruction set has been extended by an instruction which creates a constrained variable:

`get_constraint(`*i*`)`
: This represents an argument of a compound term that is a constrained variable. The instruction is followed by a `unify` instruction which matches the blocked goal.

7.3 Dynamic Predicates

As described in the previous section, dynamic clauses are represented as sequences of instructions to reconstruct the source code when executed.

The implementation of dynamic predicates is inherited from the SICStus Prolog, which implements the "logical view", or virtual copy, semantics of dynamic code updates [Lindholm and O'Keefe 87]. The key idea is to insulate an invocation of a dynamic predicate against modifications of the same predicate, while there remains a possibility of backtracking to that invocation. In other words, it is as if every procedure, when called, creates a virtual copy of its definition for backtracking purposes. Lindholm's implementation uses a method based on timestamps, and Aurora uses an adapted version of that idea. In particular, the Aurora version allows the clauses of a dynamic predicate to be explored in parallel.

To preserve Prolog semantics, it is necessary to synchronise by suspension until leftmost in the tree all dynamic code updates. Since this can seriously limit the available parallelism of an application, we have introduced as an experimental extension the ability to declare dynamic predicates as *immediate*. Such predicates obey the "immediate update" semantics, which means that all updates take immediate effect, and may change the set of outstanding alternatives that exist for current invocations. Updates of immediate predicates are not synchronised, and consequently the implementation must use locks to guarantee the atomicity of updates. We are still debating whether they should suspend if in the scope of a pruning operator or be completely "cavalier". Currently, they never suspend.

7.4 All Solutions Predicates

The implementation of the all solution predicates `setof` and `bagof` is inherited from the public domain Prolog library [O'Keefe 83]. The sequential implementation is based on a primitive `findall`, which collects all solutions of a goal, ignoring quantifiers, and is implemented materially as:

```
findall(X, Goal, _) :-
        asserta('$findall'([])),
        call(Goal),
        asserta('$findall'([X])),
        fail.
findall(_, _, List) :-
        findall([], List).

findall(Sofar, List) :-
        retract('$findall'(Item)), !,
        (   Item=[X] -> findall([X|Sofar], List)
        ;   Sofar=List
        ).
```

Thus the first clause of `findall/3` represents a sub-computation which finds all solutions to *Goal* and stores them in a dynamic relation `$findall`. This implementation would be unacceptable in Aurora for performance reasons, since a single dynamic relation is used for temporarily storing the solutions of all `setof` and `bagof` calls, synchronised to be leftmost in the entire tree.

There are two approaches to the semantics of `findall` and `bagof` in a parallel context. One would be to insist, as we have in most situations, that the parallel program produce the exact same output as the sequential program, including the ordering of the lists returned by `findall` and `bagof`. (`setof` is non-controversial because the order of its solutions is strictly defined.) We take the position that even in the sequential case, the ordering of the list of solutions is undefined (rather than implicitly defined by the procedural semantics of sequential Prolog), and so we are free

to allow it to vary from the sequential order, and even to vary from one execution of the program to another.

Two changes have been made to make the implementation acceptable in Aurora.

- Each invocation of `findall` creates its own, temporary and anonymous, dynamic relation, instead of using a global one. Updates to that relation can occur only locally within the sub-computation. This enables invocations of `findall` to run in parallel.
- The restriction on the order of solutions produced by `findall` is lifted by allowing a solution to be stored even if it not leftmost within the subcomputation, as long as it is not in the scope of a cut (implemented by `Sched_Synch`).

7.5 Foreign Function Interface

Aurora has inherited from Quintus Prolog [Quintus 87] (via SICStus Prolog) a mechanism for dynamically loading compiled C functions and associating them with a corresponding interface predicate. The only change necessary for porting this mechanism to a parallel implementation is that the functions must be loaded into shared memory.

The interface predicates are represented as emulated predicates, each with a single clause. The instruction set has been extended slightly to provide a mechanism for passing arguments to and from the C function:

`ci_inarg(`*i*`,`*j*`)`
: Sets up input argument number *i* for a foreign language call according to type specifier *j*.

`ci_outarg(`*i*`,`*j*`)`
: Sets up output argument number *i* before a foreign language call according to type specifier *j*.

`ci_call(`*i*`,`*j*`)`
: Calls a foreign language function with arity *i* and symbol table index *j*.

`ci_retval(`*i*`,`*j*`)`
: Treats output argument number *i* after a foreign language call according to type specifier *j*.

where a *type specifier* encodes the C type of the argument in question. In addition to the above instructions, the interface predicate may contain `get_variable` instructions to reorder the argument list and a `heapmargin_call` instruction to ensure that a sufficient amount of free space exists in the global stack. The area beyond the global stack pointer is used as a scratch pad area for passing parameters to the C function.

7.6 Saved States

Another mechanism inherited from Quintus Prolog (via SICStus Prolog) makes it possible to save the entire execution state as a file which can be restored at some later time. Although it would be possible in principle to create a snapshot of all currently active workers and their states, we have settled for a somewhat simpler approach. When a state is saved, all Prolog computation ceases and all workers terminate except a designated "master" worker. The master worker is responsible for saving the state by writing out the shared memory image to the file. The other workers are restarted when the state has been saved, and the usual top level loop is re-entered.

7.7 Input-Output

Proper handling of I/O was long an outstanding problem in Aurora. In the Echo version, each worker performed I/O, but a UNIX stream opened by one worker process cannot be accessed by another. The solution to this problem has been to introduce a dedicated UNIX process which serves I/O requests issued by the workers. The I/O process is also responsible for maintaining the name of the current working directory, and listens to keyboard interrupts. The I/O process was implemented by Ewing Lusk at Argonne National Laboratory.

Keyboard interrupts are handled by the following scenario, due to Péter Szeredi. See section 3.3.6, page 24.

- When an interrupt is noticed, the I/O process blocks the screen/keyboard streams, flushes their buffers, notifies the scheduler (`Sched_Block()`) and then queries the user about what action to take,
- Workers reaching `Sched_Check` after a `Sched_Block()` have been received will engage themselves in a busy waiting loop,
- The user gives one of the following answers:

  `a` (for abort)
  `e` (for exit)

  For these two answers, the scheduler is notified that all workers should die back to a certain node N (`Sched_Abort(`N`)`, and the I/O process starts discarding screen/keyboard requests, until the engine has reentered the normal toplevel loop.

  `c` (for continue)
  `t` (for trace)
  `d` (for debug)

  For these three answers, the scheduler is notified that workers should continue (`Sched_Unblock()`), and the I/O process will continue normally.

7.8 Performance

A detailed performance study of the Echo implementation has been carried out by Szeredi [Szeredi 89]. His study showed that the overhead on purely sequential execution with respect to SICStus Prolog 0.3, caused mainly by the SRI binding scheme, varied between 25% and 30%.

A similar performance study of the Foxtrot implementation is yet to be undertaken, but using the same benchmark programs as in Szeredi's study, we have determined that the overhead with respect to SICStus Prolog 0.6 has stayed in the same range. Using the *gprof* profiling tool [Graham et al. 82], we also determined that the block oriented organisation of the various stacks has not caused any measurable overhead. To our surprise, the pruning operators turned out to consume up to 11% of the execution time for some programs. We are currently tuning that part of the implementation.

One of the goals of the Foxtrot design has been to perform better on fine granularity benchmarks, i.e. benchmarks where the mean size of an assignment is small. Somewhat to our disappointment, the absolute performance has improved for such benchmarks, but the relative speedup has decreased, compared to the Echo version. The reason for this is that for these benchmarks a significant amount of the time is spent in the scheduler, which is basically the same as in the Echo version. The engine becoming faster naturally causes speedups to decrease. We had hoped that the new interface, in particular the engine being "on top" of the scheduler, and the `Sched_Get_Work_At_Parent` optimisation, would reduce scheduler overhead, but that has not yet proved to be the case.

On the other hand, coarse granularity benchmarks have displayed improved absolute performance and increased relative speedup. In this case, the increased relative speedup must be due to the improved interface.

Another goal of the new design has been to reduce suspension overheads, by introducing the remote region. We ran a formulation of the Mastermind puzzle that causes many suspensions due to pruning operations into the public section. Table 7-1 slows execution time (in seconds) and relative speedups (in parentheses):

Goal	*Workers*			
	1	*4*	*8*	*11*
mm/echo	20.5	11.1(1.85)	6.69(3.06)	5.02(4.08)
mm/foxtrot	14.6	8.54(1.71)	3.70(3.95)	3.38(4.31)

Table 7-1: Times (in seconds) and speedup for Echo and Foxtrot

and we tentatively conclude that the increase in relative speedup (for more than four workers) is due to the more efficient suspension mechanism of the Foxtrot implementation.

Symbol table access represents a bottleneck in the system. There are two global symbol tables for constants, and these are accessed e.g. when a floating-point number has been computed or when an atom has been constructed. Workers cannot access these tables concurrently, and experiments performed at the University of Bristol have shown that this can effectively kill the parallelism if a large number of floating point operations are executed on parallel branches [Kluźniak 89].

We are contemplating changing the mechanism for floating-point numbers so that they are stored on the global stack instead of in a symbol table. This would make it somewhat more expensive to check whether two numbers are identical, and multiple copies of the same number could be created. However, these disadvantages are certainly outweighted by the increase in available parallelism. The atom symbol table is a less serious problem, as it is mainly used when reading terms, an activity which is sequential in nature.

The Foxtrot implementation inherits the shallow backtracking optimisation from SICStus Prolog 0.6. The Echo implementation, being based on an older sequential Prolog engine, did not optimise shallow backtracking. In [Carlsson 89] we measured the efficiency of this optimisation on four benchmark programs.

By changing the compiler and the emulator, we modified the new Aurora implementation to suppress the shallow backtracking optimisation, and undertook the same comparison as in [Carlsson 89]. Table 7-2 summarises the results, running with a single worker. Although some discrepancies exist, we note that the speedups are in the same range. More importantly, the shallow backtracking optimisation helps reduce scheduler overheads since the scheduler is notified less often that a new node has been created.

Program	*Speedup (SICStus)*	*Speedup (Aurora)*
PLWAM	1.11	1.02
CHAT	1.09	1.04
TP	1.18	1.09
PLM compiler	1.11	1.19

Table 7-2: Shallow backtracking speedup for SICStus and Aurora

8. Conclusions

Aurora is an or-parallel Prolog system whose two main components are the engine and the scheduler, with an algorithmic interface defining their communication. This report has described the engine component with emphasis on its side of the algorithmic interface and its instruction set.

The engine was produced in four steps. The first step was to take a sequential Prolog system and replace the WAM memory model by the SRI variable binding scheme, introducing a binding array. This step involved adding some fields to the WAM environments and choicepoints. Environment trimming was generalised by adding operands to certain instructions, and promotion of variables after a pruning operation became compulsory.

The second step was to change the layout of choicepoints to one which more naturally matches the abstract SRI model. In particular, a WAM register holding the top of stack was added, and serves as a natural representation of the embryonic node. In the new layout, choicepoints are explicitly linked by pointer fields instead of implicitly by address calculation based on arity.

The third step was to change the basic memory model so that each execution stack could be stored as a chain of memory blocks, instead of having to be stored as a single large memory block. This change made it possible in Aurora for the stacks to grow without having to relocate any pointers, which would require synchronisation between all workers.

The final step was the transition from a stand-alone Prolog engine to an Aurora worker component. This involved further changes in the choicepoint layout to implement cactus stacks. The most profound change involved the backtracking routine, which must now take into consideration remote and public backtracking, the latter being the main point of communication with the scheduler component. The pruning operators were also radically changed for the same reason. A suspension mechanism was introduced, and scheduler communication was added, as dictated by the interface definition.

The described engine has been implemented and connected to the Manchester and Argonne schedulers, forming a working Aurora prototype. This prototype and its precursors have run application programs with significant speedups [Lusk et al. 88], [Szeredi 89], showing beyond any doubt that the approach based on the SRI model is feasible for implementing a practical or-parallel Prolog system.

The algorithmic scheduler/engine interface has been an invaluable vehicle for keeping the modularity of the Aurora design. It has made it possible to develop the two components separately, by different people, a necessity when developing a system in an international collaboration.

Directions

On the engine side, the Aurora technology is rather mature. The main outstanding problem is memory reclamation. In Aurora, the cactus stack scheme introduces holes in the stacks, corresponding to ghost nodes. These holes are currently not reclaimed while they are still buried under non-ghost nodes. Preliminary findings by Weemeeuw indicate that this may be a serious problem.

As in sequential Prolog systems, garbage accumulates in the global stack. This can be tolerated for some applications but not for all.

Finally, fully implementing straightening and contraction involves eliminating dead nodes and promoting bindings in the interior of the tree, and represents opportunities for memory reclamation. It remains to be assessed how much memory one can hope to reclaim this way.

Acknowledgements

The authors are indebted to Seif Haridi, Andrzej Ciepielewski, Feliks Kluźniak, Ewing Lusk, and other members of the Gigalips Project for careful reading and valuable comments on drafts of this report.

The interface design is based on an earlier design by Alan Calderwood, and was influenced by discussions with Bogumil Hausman, Per Brand, Tony Beaumont and Ewing Lusk.

This work was supported in part by the U.K. Science and Engineering Research Council, under grant GR/D97757, and in part by ESPRIT project 2471 ("PEPMA").

References

[Ali 89] K. Ali, *Incremental Garbage Collection for Aurora*, Proc. Gigalips Workshop, SICS, Stockholm, 1989.

[Appleby et al. 88] K. Appleby, M. Carlsson, S. Haridi, D. Sahlin, *Garbage Collection for Prolog Based on WAM*, Communications of the ACM vol. 31 no. 6 pp. 719–741, 1988.

[Brand 88] P. Brand, *Wavefront scheduling*, Internal Report, Gigalips Project, 1988.

[Butler et al. 88] R. Butler et al. , *Scheduling OR-Parallelism: an Argonne Perspective*, Proc. Fifth International Conference on Logic Programming, pp. 1590–1605, MIT Press, 1988.

[Calderwood 88] A. Calderwood, *Aurora Prolog—Description of Scheduler Interfaces and Implementation*, internal report, Gigalips Project, July 1988.

[Calderwood and Szeredi 89] A. Calderwood and P. Szeredi, *Scheduling Or-parallelism in Aurora—the Manchester Scheduler*, Proc. Sixth International Conference on Logic Programming, pp. 419–435, MIT Press, 1989.

[Carlsson 87] M. Carlsson, *Freeze, Indexing and Other Implementation Issues in the WAM*, Proc. Fourth International Conference on Logic Programming, pp. 40–58, MIT Press, 1987.

[Carlsson 89] M. Carlsson, *On the Efficiency of Optimising Shallow Backtracking in Compiled Prolog*, Proc. Sixth International Conference on Logic Programming, pp. 3–16, MIT Press, 1989.

[Carlsson 90] M. Carlsson, *A Prolog Compiler and its Extension for Or-Parallelism*, SICS Research Report R90006, Swedish Institute of Computer Science, 1990.

[Graham et al. 82] S.L. Graham, P.B. Kessler, M.K. McKusick, *gprof: A Call Graph Execution Profiler*, Proc. SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices, vol. 17, no. 6, pp. 120-126, 1982.

[Haridi 89] S. Haridi. Personal communication, 1989.

[Hausman et al. 88] B. Hausman, A. Ciepielewski, A. Calderwood, *Cut and side-effects in or-parallel Prolog*, Proc. International Conference on Fifth Generation Computer Systems, pp. 841–849, ICOT, Tokyo, 1988.

[Hausman 90]
B. Hausman, *Pruning and Speculative Work in OR-Parallel PROLOG*, Ph.D. thesis, Department of Telecommunication and Computer Systems, The Royal Institute of Technology, Stockholm, 1990.

[Kernighan and Ritchie 78]
B. W. Kernighan, D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[Kluźniak 89]
F. Kluźniak (ed.), *Applications of Or-Parallel Prolog Systems (A Progress Report)*, Department of Computer Science, University of Bristol, 1989.

[Lindholm and O'Keefe 87]
T. Lindholm, R. A. O'Keefe, *Efficient Implementation of a Defensible Semantics for Dynamic PROLOG Code*, Proc. Fourth International Conference on Logic Programming, pp. 21–39, MIT Press, 1987.

[Lusk et al. 88]
E. Lusk, D.H.D. Warren, S. Haridi et al. , *The Aurora Or-Parallel System*, New Generation Computing vol. 7 nos. 2–3 pp. 243–271, 1990.

[Naish 85] L. Naish, *Negation and Control in Prolog*, Ph.D. thesis, Department of Computer Science, University of Melbourne, 1985.

[O'Keefe 83]
R. A. O'Keefe, *SETOF.PL*, file contributed to the public domain Prolog library in 1983.

[Quintus 87]
Quintus Prolog Reference Manual version 10, Quintus Computer Systems, Inc., Mountain View CA, February 1987.

[SICS 88] *SICStus Prolog User's Manual*, SICS Research Report R88007B, 1988.

[Szeredi 89]
P. Szeredi, *Performance Analysis of the Aurora Or-parallel Prolog System*, Proc. North American Conference on Logic Programming, pp. 713–734, MIT Press, 1989.

[Warren 83]
D.H.D. Warren, *An Abstract Prolog Instruction Set*, Technical Note 309, SRI International, 1983.

[Warren 87a]
D.H.D. Warren, *Or-Parallel Execution Models of Prolog*, Proc. TAPSOFT'87, The 1987 International Joint Conference on Theory and Practice of Software Development, pp. 243–259, Springer-Verlag, 1987.

[Warren 87b]
D.H.D. Warren, *The SRI Model for Or-Parallel Execution of Prolog—Abstract Design and Implementation Issues*, invited talk, Proc. Symposium on Logic Programming, pp. 92–102, IEEE Computer Society, 1987.

[Weemeeuw 89]
P. Weemeeuw, *Garbage Collection for the Aurora System*, internal report, Gigalips Project, July 1989.

Appendix: Synopsis of the Pseudocode Language

This appendix contains a synopsis of the subset of the C programming language [Kernighan and Ritchie 78] that we use in this report.

Type Expressions

C is a weakly typed language. The types of arguments, variables and functions are specified as *type expressions*. This report will use the following type expressions:

`int` A signed integer.

`unsigned int`
An unsigned integer.

`unsigned short`
An unsigned halfword integer.

`char` A quarterword integer.

defined_type
An instance of type *type*, where *defined_type* has been defined by:

```
typedef type defined_type
```

`struct` *record_name*
An instance of a record type whose layout has been defined by:

```
struct record_name {
  type_1 field_1
  ...
  type_n field_n
}
```

type `*` A pointer to an instance of type *type*.

type`[]` An array of instances of type *type*.

`void` This type definition is used for *procedures*, i.e. functions that perform some side-effect instead of computing a value.

The expression `NULL` is commonly used to designate a "null pointer", or "end-of-list", etc.

Functions

The basic code entity is the *function*, and most basic operations are defined as functions, written as:

```
type name(type_1 arg_1, ..., type_n arg_n)
{ statement }
```

where each formal parameter *arg_i* is written as an identifier, each *type_i* is a type expression, and the *name* begins with a lowercase letter.

Statements

The following kinds of *statement* are used:

expression;
: An expression which performs some side-effect.

`while` (*condition*) *statement*
: A while loop.

`if` (*condition*) *statement*
`if` (*condition*) *statement* `else` *statement*
: If statements.

`{` *statement*; ...; *statement* `}`
: A grouped statement.

`switch` (*expression*) `{`
`case` *Const_1*: *statement_1* ...
`case` *Const_n*: *statement_n* ...
`}`
: A case statement dispatching on the value of *expression*.

`break;`
: Exit from the current `switch` statement.

`return` *expression*;
: Exit from the current function with the value of the *expression*.

`goto` *Label*;
: A goto statement.

where grouped statement may contain local variable declarations, optionally initialised:

```
type identifier;
type identifier = expression;
```

Expressions

Expressions of the following types have both an *lvalue* and an *rvalue*, i.e. can occur on both sides of an assignment, whereas others have an *rvalue* only. The lvalue of each of the following expressions denotes a memory location, and its rvalue denotes the contents of that location:

variable A local or global variable.

struct_pointer`->`*field_name*
: A field of a record.

array_pointer`[`*integer_expression*`]`
: An element of an array.

`*` *scalar_pointer*
: The scalar value that the pointer points at.

Expression of the following types have an rvalue but no lvalue:

function_call
: Evaluates to the value which is `return`:ed by the function.

expression_1`,` *expression_2*
: Evaluates to the value of *expression_2*.

lvalue `=` *expression*
: Evaluates to the value of *expression*, which is stored in the location denoted by *lvalue*.

`--` *lvalue* The contents of the *lvalue* is decremented by one. The expression evaluates to the decremented value.

lvalue `++` The contents of the *lvalue* is incremented by one. The expression evaluates to the original value.

lvalue `-=` *expression*
lvalue `+=` *expression*
: The contents of the *lvalue* is decremented or incremented by the value of *expression*, respectively. The whole expression evaluates to the updated value.

`(`*condition* `?` *expression_1* `:` *expression_2*`)`
: Evaluates to the value of *expression_1* if the *condition* evaluates to a nonzero value, otherwise to the value of *expression_2*.

`&` *lvalue* Evaluates to the location denoted by *lvalue*.

`(`*type*`)` *expression*
: Evaluates to the value of *expression*, cast to some other *type*.

pointer_expression `+` *integer_expression*
pointer_expression `-` *integer_expression*
: Evaluates to the value of *pointer_expression* incremented or decremented by $i*s$, where i is the value of *integer expression* and s is the size of the elements pointed at.

expression_1 `+` *expression_2*
expression_1 `-` *expression_2*
: Addition and subtraction, respectively.

expression_1 relop expression_2
: where *relop* is one of '$\equiv$ $\not\equiv$ `<` $\leq$ `>` $\geq$', evaluates to 1 (true) or 0 (false).

expression_1 $\vee$ *expression_2*
expression_1 $\wedge$ *expression_2*
$\neg$ *expression_2*
: The "or else", "and then", and "not" operators, respectively.

Text Macros

In this report, many operations are defined by *text macros* as opposed to functions. An occurrence of a text macro call is to be understood as textually replacing the macro call by the macro definition, substituting formal parameters for actual parameters.

The interface between the scheduler and engine Aurora components is defined almost entirely in terms of macros.

The introduction of text macros was motivated for convenience and efficiency. They provide a mechanism for abbreviating and naming complex expressions, making the code easier to read, and avoid the overhead of procedure calls. They can always be replaced by functions, except in three cases which occur relatively frequently:

- when the macro definition updates its formal arguments,
- when the lvalue of the macro call is needed, e.g. in the left hand side of an assignment statement, and
- when a formal argument is a C statement rather than an expression.

The C language proper does not have text macros, but in this report we shall deviate somewhat from the standard notation and define text macros just as functions with names beginning with an *uppercase* letter. If the macro is used in contexts where its lvalue is needed, the function definition is simply written as `return` *expression*`;` and the macro call is to be understood as textually replaced by the *expression*.

Type Definitions

Basic Type Definitions

```
typedef unsigned short INSN;      /* bytecode instructions */
typedef unsigned int TAGGED;      /* terms */
typedef int BOOL;                 /* false or true */
```

Abstract Machine Registers

```
struct wam {
  INSN *insn;                   /* program counter */
  INSN *next_insn;              /* continuation pointer */
  struct frame *frame;          /* environment pointer */
  struct node *node;            /* current logical choicepoint */
  struct frame *local_top;      /* environment stack pointer */
  TAGGED *trail_top;            /* trail pointer */
  TAGGED *global_top;           /* global stack pointer */
  TAGGED *structure;            /* structure pointer */
  TAGGED global_uncond;         /* lowest cond. global variable */
  TAGGED local_uncond;          /* lowest uncond. local variable  */
  struct node *previous_node;   /* embryonic node at predicate entry */
  struct frame *frame2;         /* permanent variable base */
  struct alternative *next_alt; /* alt. clause during shallow backtracking
*/
  TAGGED global_var;            /* lowest used global variable number */
  TAGGED local_var;             /* lowest unused local variable number */
  struct node *choice_top;      /* embryonic node */
  struct node *own_node;        /* current choicepoint in own stack */
  struct node *own_sentry_node; /* remote/local boundary */
  struct node *sentry_node;     /* public/remote boundary */
  TAGGED term[];                /* argument registers */
};

  struct wam *w;
```

Environments

```
struct frame {
  struct frame *frame;          /* continuation frame pointer */
  INSN *next_insn;              /* continuation program pointer */
  TAGGED local_var;             /* local variable number */
  TAGGED term[];                /* permanent variables */
};
```

Nodes

```
struct node {
  struct node *own_node;        /* physical predecessor node */
  TAGGED *trail_top;            /* top of trail stack */
  TAGGED *global_top;           /* top of global stack */
  struct alternative *next_alt; /* alternative clause */
  struct frame *frame;          /* environment pointer */
  INSN *next_insn;              /* continuation */
  struct frame *local_top;      /* environment stack pointer */
  TAGGED global_var;            /* global variable number */
  TAGGED local_var;             /* local variable number */
  int level;                    /* distance from root */
  struct node *parent;          /* parent node */
#include "sch.node.h"
  TAGGED term[];                /* saved argument registers */
};
```

Alternatives

```
struct alternative {
  int arity;                    /* arity of predicate */
  struct alternative *next;     /* next alt. in chain, if any */
  BOOL parallel;                /* TRUE if parallel alts. may be explored */
  INSN *insn;                   /* program address of clause */
#include "sch.alternative.h"
}
```

Memory Blocks

```
struct blockheader {
  unsigned int size;            /* total size in bytes */
  unsigned int used;            /* bytes used if stacktop in previous* */
  struct blockheader *previous; /* NULL or previous segment */
  struct blockheader *next;     /* NULL or next segment */
};
```

Table of Contents