The Engine-Scheduler Interface in the Aurora Or-Parallel Prolog System*

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

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

Abstract

Aurora is a prototype or-parallel implementation of the full Prolog language for shared memory multiprocessors, based on the SRI model of execution. The two major components of Aurora are the *engine*, which is responsible for the actual execution of the Prolog code, and the *scheduler*, which provides the engine component with work. This report describes the interface between the engine and scheduler components of Aurora.

The practical purpose of the interface is to enable different engine and scheduler implementations to be used interchangeably. The development of the interface has, however, contributed in great extent to the clarification of basic concepts in exploiting or-parallelism in Prolog. We believe that these issues are relevant to a wider circle of research in the area of or-parallel implementations of logic programming.

Keywords: Or-Parallel Execution, Multiprocessors, Implementation Techniques, Scheduling.

1 Introduction

Aurora is a prototype or-parallel implementation of the full Prolog language for shared memory multiprocessors, currently running on Sequent and Encore machines. It has been developed in the framework of the Gigalips project [10], a collaborative effort between groups at the Argonne National Laboratory in Illinois, the University of Bristol (previously at the University of Manchester) and the Swedish Institute of Computer Science (SICS) in Stockholm.

*This report is an extended version of a paper submitted for the North American Conference on Logic Programming 1990.

†*On leave from* SZKI, Donáti u. 35-45, Budapest, Hungary.

The issue of defining a clear interface between the engine and scheduler components of Aurora was raised in the early stages of the implementation effort. Ross Overbeek made the first attempt to formulate such an interface and Alan Calderwood produced the version [4] used in the first generation of Aurora (based on SICStus Prolog version 0.3).

A fundamental revision of the interface was necessitated by several factors. Performance analysis work on Aurora [12] has shown that some unnecessary overheads are caused by design decisions enforced by the interface. Development of new schedulers and extensions to existing algorithms required the interface to be made more general. The Aurora engine has also been rebuilt on the basis of SICStus Prolog version 0.6.

The new interface, described in the present report, is part of the second generation of Aurora. The major changes with respect to the previous interface are the following:

- execution is governed by the engine, rather than the scheduler;
- the set of basic concepts has been made simpler and more uniform;
- several potential optimisations are supported;
- the interface is extended to support transfer of information related to pruning operators [8].

The report is organised as follows. Section 2 summarises the SRI model and defines the necessary concepts. Section 3 gives a top level view of the interface. Section 4 presents the data structures involved in the interface, while Sections 5 and 6 describe engine-scheduler interactions in various phases of Aurora execution. Section 7 shows the extensions: handling of pruning information and various optimisations. Section 8 discusses the major issues involved in implementing the engine side of the interface. We end with a short concluding section.

A complete description of the interface is included in the Appendix.

2 Preliminaries

Aurora is based on the SRI model [13]. According to this model the system consists of several *workers* (processes) exploring the search tree of a Prolog program in parallel. Each node of the tree corresponds to a Prolog *choicepoint* with a branch associated with each alternative clause. A predicate can optionally be declared *sequential* by the user, to prohibit parallel exploration of alternative clauses of a predicate. Corresponding nodes are also annotated as sequential. All other nodes are *parallel.*

As the tree is being explored, each node can be either *live*, i.e. have at least one unexplored alternative, or *dead.* A node is a *fork node* if there are two or more branches below it; otherwise, it is a *nonfork node.* A fork node cannot be sequential. Live parallel nodes, and live sequential nodes with no branches below them, correspond to tasks that can be executed by workers. Each worker has to perform activities of two basic types:

- executing the actual Prolog code;

- finding work in the tree, providing other workers with work and synchronising with other workers.

In accordance with the SRI model each worker has a separate *binding array*, in which it stores its own bindings to potentially shared variables (conditional bindings). This technique allows constant time access to the value of a shared variable, but imposes an overhead of updating the binding arrays whenever a worker has to move within the search tree.

The or-tree is divided into an upper, *public*, part accessible to all workers and a lower, *private*, part accessible to only one worker. A worker exploring its private region does not have to be concerned with synchronisation or maintaining scheduling data; it can work very much like a standard Prolog engine. The boundary between the public and private regions changes dynamically. It is one of the critical aspects of the scheduling algorithm to decide when to make a node public, allowing other workers to share work at it. Normally the worker will make his *sentry* node, i.e. his topmost private node, public when all nodes above it have died, i.e. have no more alternatives to explore. This means that each worker tries to keep a piece of work on its branch available to other workers.

The exploration by a worker of its private region constitutes that worker's *assignment*, which normally terminates if the worker backtracks into the public part. The assignment terminates prematurely if the branch is *suspended*, or if it is *pruned* by some other worker.

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. A cut or a commit will not go ahead if there is a chance of 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. Work done in the scope of a pruning operator is said to be *speculative*.

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, or when a pruning operator is reached on a branch which could be pruned by a cut with a smaller scope, the execution must be suspended. Furthermore the scheduler may decide to suspend the current branch when less speculative work can be done somewhere else in the tree.

Four separate schedulers are currently being developed for Aurora. The Argonne scheduler [3] relies on data stored in the tree itself to implement a local strategy according to which live nodes "attract" workers without work. When several workers are idle they will compete to get to a given piece of work and the fastest one will win. The Manchester scheduler [5] tries to select the nearest worker in advance, without moving over the tree. It uses global data structures to store some of the information on available work and workers. The wavefront scheduler [2] uses a special distributed data structure, the *wavefront*, to facilitate allocation of work to workers. The Bristol scheduler [1] tries to minimise scheduler overhead by extending the public region eagerly: sequences of nodes are made public instead of single nodes, and work is taken from the bottommost live node of a branch.

3 The Top Level View of the Interface

The principal duty of the scheduler is to provide the engine with work. The thread of control thus alternates between the two components: the engine executes a piece of Prolog code, then the scheduler finds the next assignment, 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 [4].

We use a different approach in the current version of Aurora. The execution is governed by the engine: whenever it 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 upon 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 [12].

Figure 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 the convention that the names of all scheduler functions are prefixed with 'Sched_'.

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

- finding work (left side of Figure 1);
- communication with other workers during work (lower part of Figure 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 1), e.g. creation and destruction of nodes.

The four boxes on the left of Figure 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 backtracks to a public node;

`Sched_Be_Pruned` is invoked when the worker's current branch is pruned off by another worker;

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

These functions differ in their initial activities, but normally continue with a common algorithm for "looking for work" (see Section 5). 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

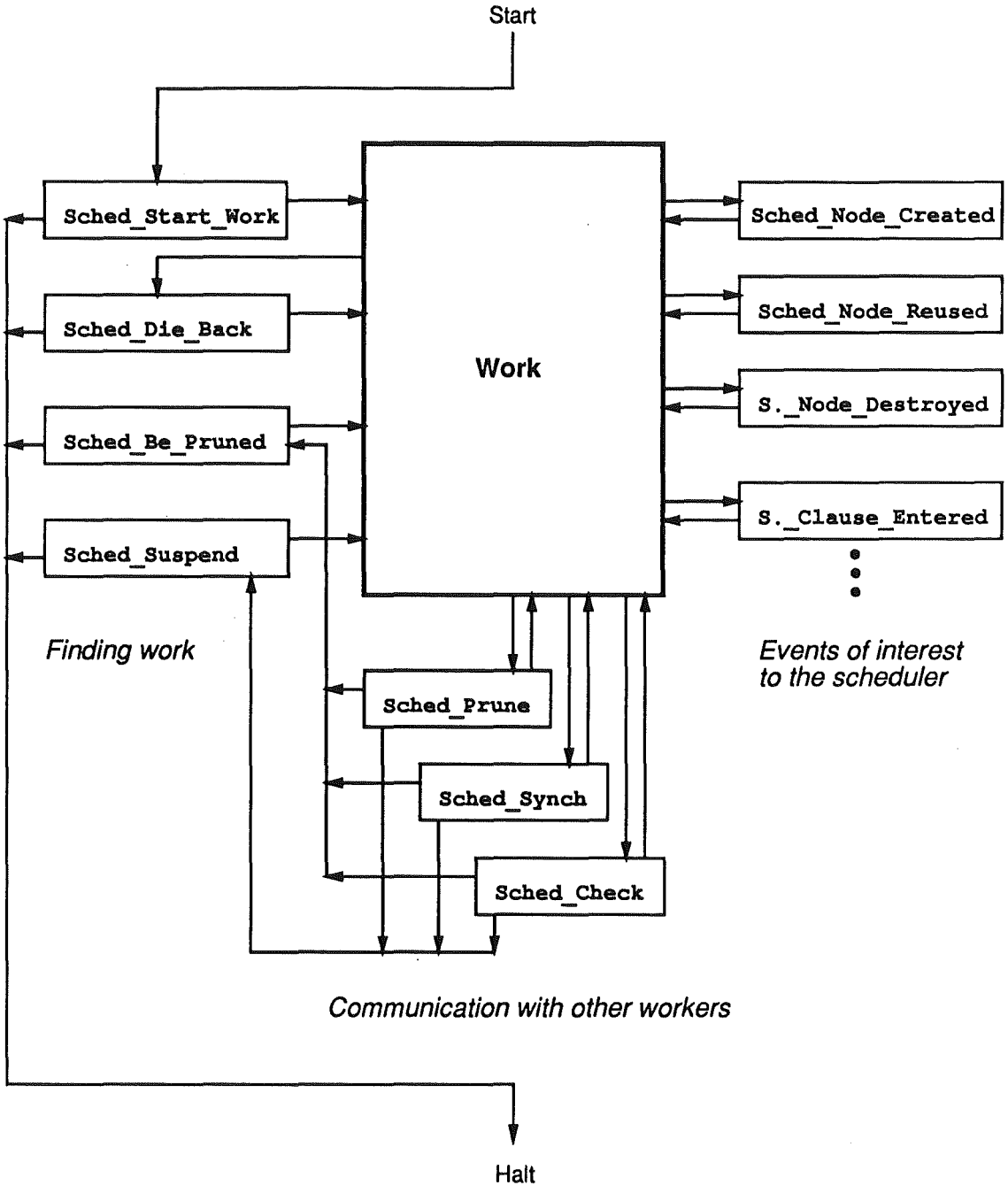


Figure 1: THE TOP LEVEL VIEW OF THE INTERFACE

boxes in Figure 1) leads back to work, while the other exit (left hand side) leads to the termination of the whole Aurora invocation.

The next group of interface functions provided by the scheduler is depicted at the bottom of Figure 1. These functions are called during work, when the engine may require some assistance from the scheduler (mainly in order to communicate with other workers):

`Sched_Prune` — when a cut or commit is executed;

`Sched_Synch` — when a predicate with side effects is encountered;

`Sched_Check` — at every Prolog procedure call (to check for interrupts).

The above functions have three exits. The normal exit (depicted by upwards arrows in Figure 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 (leftward and downward 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 to find the next assignment. See Section 6 for a more detailed description of the functions for communication with other workers.

The third group of functions shown in Figure 1 (right hand side) corresponds to some events during work that may be of interest to the scheduler. A common property of this group is that the interface does not prescribe any specific activity to be done by these functions: the scheduler is merely given an opportunity to do whatever is needed for maintaining its data structures. 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 7.2).

There are further groups of scheduler functions, not shown in Figure 1. These are used in the initialisation of the whole system, in handling keyboard interrupts (Section 7.3), and in the implementation of certain optimisations (Section 7.1).

The engine side of the interface consists of several groups of functions that support the scheduler algorithm:

- providing access to certain data structures (nodes and alternatives) maintained by the engine,
- extending the public region on the current branch of execution,
- positioning the engine (i.e. the binding array) in the search tree, while looking for work,
- notifying the engine of certain events, e.g. work being found.

The data structure aspects of the engine interface are presented in Section 4. Other interface functions provided by the engine will be described in Sections 5 and 6.

4 Common Data Structures

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 are provided in the interface:

`Node_Level` — the distance of the node from the root of the search tree,

`Node_Parent` — a pointer to the parent node in the tree,

`Node_Alternatives` — a pointer to the next unexplored alternative of the node.

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.

An additional common static data structure, the *alternative*, is introduced to allow the schedulers to keep static data related to clauses. This data structure is used in the Aurora engine to replace the `'try'`, `'retry'` and `'trust'` instructions of WAM [7]. Each clause of the user program is represented by an alternative, which stores a pointer to the code of the clause and a pointer to the successor alternative, if any. If a predicate is subject to indexing, the compiler may create several chains of alternatives to cater for different values in the indexing argument position. This means that several alternatives can refer to the same clause.

The scheduler may supply a number of fields to be included in the alternative structure, to accommodate any (static) information to be associated with clauses. The scheduler can derive this data from the information supplied by the engine when alternatives are created (`Sched_Alternative_Created`). There are two types of static data supplied by the engine:

- information about sequential predicates—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 (see Section 7.2).

The only engine field in the alternative structure that is of interest to the scheduler is the one pointing to the successor alternative (`Alternative_Next`). This field is used, for example, when the scheduler starts a new branch from a public node and needs to advance the next alternative pointer of the node.

5 Finding Work

Figure 2 shows the engine functions used by the scheduler while it is looking for work. The actual algorithms of the four functions for finding work will normally differ, but they all use the same set of engine support functions.

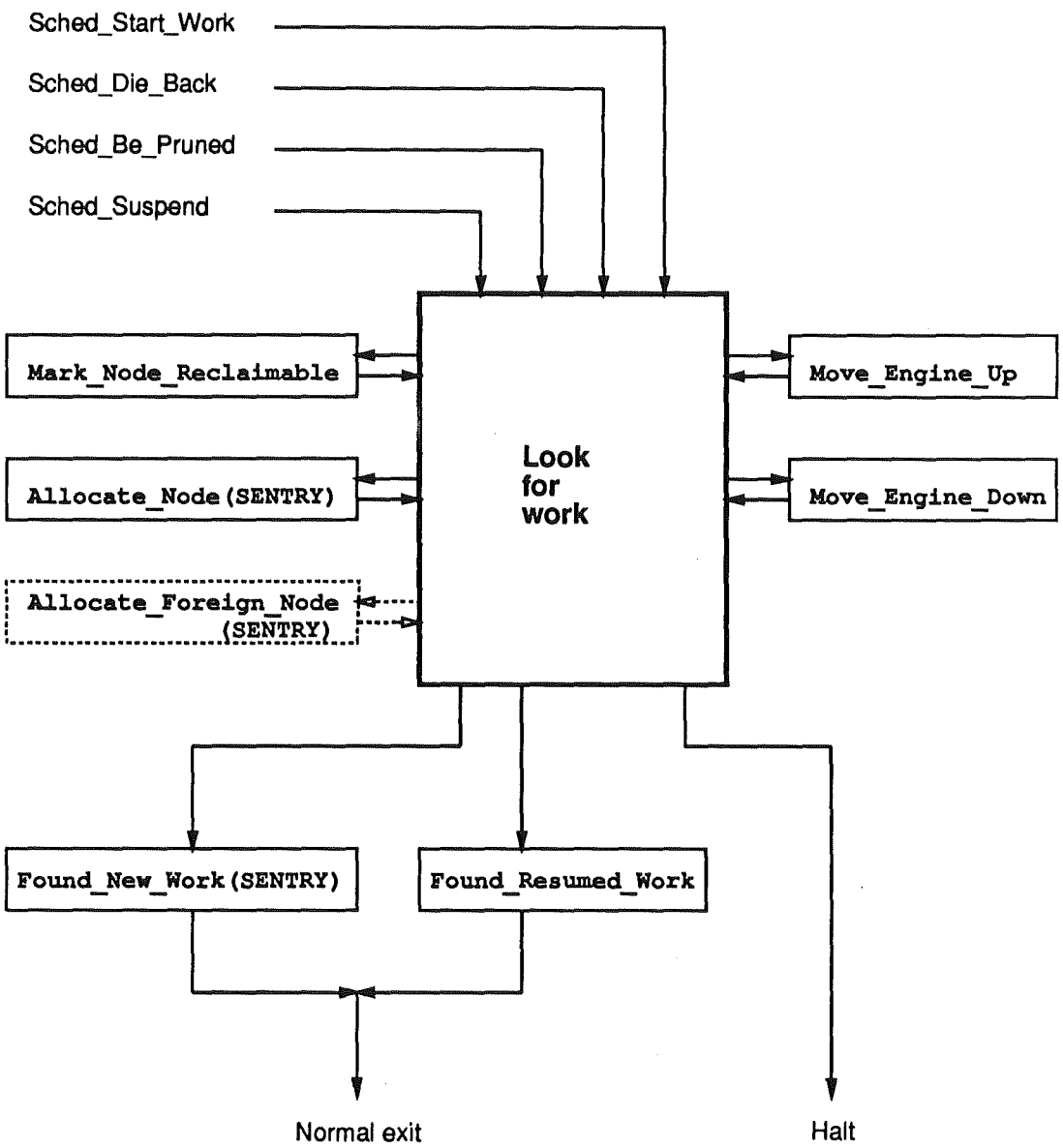


Figure 2: ENGINE FUNCTIONS IN LOOKING FOR WORK

Functions `Move_Engine_Up` and `Move_Engine_Down`, shown on the right hand side of Figure 2, instruct the engine to move the binding array up or down the current branch. Initially, the binding array is positioned at or below the youngest public node on the branch. Before returning, the scheduler has to position the binding array above the new sentry node.

Different schedulers employ different strategies in moving over the tree. The Argonne scheduler moves node-by-node, when approaching the potential work node. Other schedulers locate a piece of work from a distance and move the engine to the appropriate place in a few big jumps.

There is no need to move the engine if work is taken from the parent of the old sentry node. An additional entry point to the scheduler, `Sched_Get_Work_At_Parent` (see Section 7.1), has been provided for this special case.

The left hand side of Figure 2 shows the engine functions for memory management of the node stack. A worker may have to remove some dead nodes from the tree as it moves upwards. This involves deleting these nodes from the scheduler data structures (normally the sibling chain) and invoking the `Mark_Node_Reclaimable` engine function. As a special case, the old sentry node will have to be deleted from the tree at the beginning of `Sched_Die_Back` and `Sched_Be_Pruned`.

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 involves calling 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 related `Allocate_Foreign_Node` function is used if *another* worker allocates a node on the stack of the worker looking for work. This is used in the Manchester scheduler to implement handing work to an idle worker.

The new sentry node serves as a placeholder for the new assignment. The scheduler inserts the sentry into the search tree and simultaneously reserves an alternative to be explored by the new branch (by reading and advancing the `Node_Alternatives` field of the work node).

The bottom part of Figure 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 2 shows the `SENTRY` argument to highlight the fact that this argument should be the same as the one returned in `Allocate_...Node`.

6 Communication with Other Workers

The need for communication with other workers arises when a pruning operator or a built-in predicate with side effects is to be executed. In addition, a periodic check is needed to examine if there are communication requests from other workers.

The `Sched_Prune` function is invoked when a pruning operator is encountered. At this moment the engine has already executed the private part of the pruning. The scheduler receives a pointer to the cut node (showing the scope of pruning) and an argument indicating

the type of the pruning operator (cut, commit or cavalier commit). It has to check if the preconditions for pruning are satisfied: the current branch should not be pruned itself, and, except for the cavalier commit, it should not be endangered by cuts with a smaller scope, as discussed in [8]. The latter condition can be replaced by a requirement for the branch to be leftmost in the subtree rooted at the child of the cut node, if the scheduler does not maintain specific pruning information.

If the preconditions of pruning are not satisfied, `Sched_Prune` uses one of the abnormal exits (cf. Figure 1) to indicate that the branch has been killed or that it has to suspend (waiting to become leftmost). If the pruning operation can go ahead, the scheduler has to locate the workers that are in the pruned subtree and interrupt them. There may be branches in this subtree which have previously been suspended. A special engine function, `Mark_Suspended_Branch_Reclaimable`, is used for cleaning up such branches.

The `Sched_Synch` function is invoked when a call to a built-in predicate with side-effects is encountered. Normally such calls are executed only when their branch becomes leftmost in the whole tree. There are, however, some special predicates (e.g. those used to assert solutions in a `bagof`), for which the order of invocation is not significant: their execution can go ahead if not endangered by a cut within a specific subtree. The `Sched_Synch` function receives an argument encoding the type of the check needed, and a pointer to the root of the subtree concerned.

The third communication function, `Sched_Check`, is called at every Prolog procedure call. Frequent invocation of this function 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 (as is the case for the Manchester and Argonne schedulers).

The nature of requests to be handled by `Sched_Check` varies from scheduler to scheduler. There are, however, two common sets of circumstances: the worker may be requested to kill its assignment or to make some of its private nodes public (to make work available to other workers). The latter activity needs assistance from the engine: the function `Make_Public` extends the public region on the current branch down to a specified node.

7 Extensions of the Basic Interface

7.1 Optimisations

This section describes two important optimisations supported by the interface: simplified backtracking and bypassing.

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, rather than being marked as reclaimable and re-allocated. There is scope for a related optimisation in the scheduler: instead of deleting the old sentry from the sibling chain and then installing it as the last sibling, the scheduler can move the sentry node to the end of the sibling chain (or do nothing if the old sentry was the last child). The interface supports this important optimisation by a function `Sched_Get_Work_At_Parent`, called when the engine backtracks to a live public node. If the scheduler, following the necessary synchronisation

operations, still finds the node to be live, it can reserve an alternative from that node. If the scheduler cannot take work from the node in question, it returns to the engine, which will subsequently invoke `Sched_Die_Back` to acquire a new piece of work.

The `Sched_Get_Work_At_Parent` function also supports the contraction operation of the SRI model [13]. This operation removes a dead nonfork node after the last alternative has been taken from it. The node in question can be physically removed only if it is on the top of the stack of the worker executing the given branch. The engine informs the scheduler whether this last condition applies by passing an argument to `Sched_Get_Work_At_Parent`. If the scheduler decides to apply contraction, the public node from which the last alternative has been taken becomes the new sentry node.

If the last alternative is taken from a nonfork node, but the node in question is on another worker's stack, then the node cannot be fully reclaimed. This is because the segments of other WAM stacks (environment, heap and trail) that correspond to the node are still required, until the given branch of the tree finally dies back. As the memory management of stacks relies on the node stack, such dead nonfork nodes have to remain in the search tree, as viewed by the engine.

A similar situation arises when a worker dies back to a dead node leaving just one other branch below. The SRI model envisages that this dead nonfork node can now be removed from the tree (straightening). In Aurora, the effect of straightening and contraction can be achieved by *bypassing* the dead node, provided the scheduler maintains its own notion of parent, different from that of the engine. This means that a new node field, the *bypassed parent*, has to be introduced by the scheduler. This field is initialised to be equal to `Node_Parent`, but is subject to change if bypassing is applied.

Bypassing may be required in the private region, when *remote* nodes (i.e. nodes on other workers' stacks) are cut or trusted. Such remote nodes are created when a worker resumes a branch that has been suspended by another worker. The `Sched_Node_Destroyed` function, which is called whenever a private node is exhausted, receives an additional argument to indicate whether the engine is able to reclaim the node in question. This makes it possible for the scheduler to perform the bypassing operation in the private region, if necessary.

Two further functions are related to bypassing in the interface. The scheduler informs the engine whenever a node has been bypassed and is not needed any more (`Mark_Node_Bypassed`). It also supplies a function for accessing the bypassed parent field in nodes (`Sched_Node_Bypassed_Parent`), which may be of use to the engine.

An issue related to bypassing is that of the implementation of the `Node_Parent` field of nodes. The engine offers two alternative representations for `Node_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 use the default, fast implementation.

7.2 Pruning Information

Information about the presence of pruning operators in a clause may be needed by the scheduler to perform pruning more efficiently or to distinguish between speculative and

non-speculative work. Various algorithms related to pruning have been developed and discussed in [8]. When designing the interface, we tried to generalise and extend the format of pruning data as described in [8], so that other possible approaches (e.g. [11]) can be supported as well.

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[1]. For more exact pruning algorithms the number of occurrences of each pruning operator may be needed. The fact that a clause must fail, may also be of interest: when such a clause is entered, the pruning operators in the current continuation (i.e. in the previous resolvent) become inaccessible. 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 the future, global compile time analysis might discover this property for other calls).

The presence of disjunctions makes the situation more complicated. The Aurora compiler [6] replaces all disjunctions by so called internal predicates, 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 disjunction, making it impossible to predict the exact number of pruning operators to be encountered in a clause. As schedulers will mainly be interested in knowing whether a branch is endangered by cuts or commits, data on the maximal number of cuts and commits in a clause seems to be a suitable substitute.

Several algorithms in [8] involve maintaining a counter that shows the maximal number of outstanding pruning operators in the current resolvent. When an internal clause (i.e. a disjunct) is entered, this counter has to be adjusted by the difference between the number of operators assumed for the whole internal predicate and the actual data for the given clause. Pruning data for internal clauses will thus include the maximal number of cuts and commits in the whole internal predicate (of which the clause in question is a member).

The set of pruning data is completed by three Boolean values which indicate whether the clause in question is internal, whether it is a conditional expression (relevant to internal clauses only) and whether it ends in a call to `fail`.

A formal definition of the pruning data is given in the Appendix.

Maintaining a counter of outstanding pruning operators is facilitated by several interface functions. The pruning data supplied by the engine is stored by the scheduler in the alternative data structure (`Sched_Alternative_Created`). A pointer to the selected alternative is passed to `Sched_Clause_Entered`, allowing to adjust the counter as needed. The counter has to be decremented when a pruning operator is executed (`Sched_Prune`), saved in the node being created (`Sched_Node_Created`), and restored when control backtracks to a node (`Sched_Node_Reused` and `Sched_Node_Destroyed`).

[1]Note that data on cavalier commits is not included in the pruning information, as this operation is expected to be used only for handling exceptional circumstances.

7.3 Related Interfaces within Aurora

There are some further interfaces within Aurora, loosely related to the interface between the engine and the scheduler. There are two groups of functions, for locking and for shared memory management. Both groups are defined by the engine in the present implementation, and are used by both the engine and the scheduler side.

There is also a RIO (remote input/output) package [9], that allows multiple processes (workers) to read/write the same file and handle keyboard interrupts. The RIO package is embedded in the engine; the only issue that affects the scheduler is interrupt handling. When a keyboard interrupt is received, the engine asks the scheduler to block the execution of all workers (`Sched_Block`), while the user is being asked what to do. If the user requests an abort, the scheduler is required to kill all the workers and die back to a specified node (`Sched_Abort`). If the user wants the execution to continue, the scheduler is notified accordingly (`Sched_Unblock`).

8 Implementation Aspects of the Interface

The Aurora emulator [7] was produced by modifying the SICStus emulator to support the SRI model and by converting it from a stand-alone program to an Aurora worker component connected by an algorithmic interface to a scheduler component. The total performance degradation resulting from these changes has been found to be around 25%. In an earlier paper [10] we gave an overview of the changes imposed by the SRI model. In this section we concentrate on the impacts of the interface on the engine and on changes introduced in the new design.

8.1 Boundaries

The engine needs to maintain the boundary between the public and private regions. Within the private region, it must distinguish between *local* nodes, i.e. nodes adjacent to the top of the worker's own stack, and remote nodes. This is achieved by storing a pointer to the respective boundary nodes in certain registers. These registers are initialised when an assignment is started (`Found...Work`). They are updated when the public region is extended (`Make_Public`) or contracted (`Sched_Get_Work_At_Parent`), and when backtracking in the private region winds back to the worker's own stack. They are consulted to distinguish different cases of backtracking and pruning operations.

The algorithm for computing the top of the environment stack has to ensure that new environments are allocated in the current worker's stack even if the current environment is in another worker's stack.

8.2 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 check whether it is about to backtrack into the public region, in which case the scheduler must be invoked to perform public

backtracking (`Sched_Die_Back` or `Sched_Get_Work_At_Parent`). Private backtracking has to face the complication that the private region may extend to other workers' stacks, and possibly wind back to the worker's own back again. As explained earlier, remote nodes cannot be reclaimed when they are trusted; instead, `Mark_Node_Reclaimable` is invoked when dying back over a remote node.

Shallow backtracking is optimised in the private region, but only if the current node is on the top of the worker's own stack.

8.3 Memory Management

As stated earlier, the stack memory management relies on the node stack. While finding work, each worker maintains a pointer to the youngest node that has to be kept for the benefit of other workers. Such pointers are used and updated by the `Allocate...Node` functions. When an assignment is started (`Found...Work`) the top of stack pointers for the other WAM stacks are initialised from relevant fields of the node physically preceding the embryonic node of the new assignment, as these fields define how much of the other stacks has to be kept.

The old Aurora engine had no provision for handling stack overflows due to the problems of performing garbage collection and stack shifting on memory areas shared by several processes. Sufficient stack sizes had to be supplied by the user.

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.

8.4 Pruning Operators

Pruning operations must distinguish between (i) pruning local nodes only, (ii) pruning remote nodes, and (iii) pruning public nodes. In cases (i) and (ii), the node can be pruned right away, but the memory occupied by the pruned node can only be reclaimed in case (i). The trail must be tidied in all three cases, as explained in [10]. In case (iii), the scheduler is responsible for pruning the public nodes, but may decide to suspend or abort the current assignment instead, forcing the engine to invoke `Sched_Suspend` or `Sched_Be_Pruned`, respectively. Note that `Sched_Prune` is invoked in all three cases, to give the scheduler an opportunity to keep pruning information up to date.

To support suspension of cuts and commits, the compiler provides extra information about what temporary variables need to be saved until the suspended task is resumed. This extra information also encodes the type of the pruning operator.

8.5 Premature Termination

To suspend the current assignment when the scheduler uses the "suspend" exit in `Sched_Prune`, `Sched_Synch`, or `Sched_Check`, the engine creates an auxiliary node which stores the current state of computation and calls `Sched_Suspend`. It is up to the scheduler to decide when the suspended work may be resumed.

To abort the current assignment when the scheduler uses the "be_pruned" exit in the above functions, the engine deinstalls all conditional bindings made by the current assignment, marks all remote nodes as reclaimable except the sentry node, and calls `Sched_Be_Pruned`.

8.6 Movement

While executing Prolog code, the binding array is kept in phase with the trail stack: whenever a binding is added to or removed from the trail, the bound value is also stored or erased in the binding array. While finding work, the engine maintains a pointer to a node in the tree corresponding to the current contents of the binding array. When the scheduler asks the engine to "move" the binding array up to a new position (`Move_Engine_Up`), bindings which were recorded on the trail path between the current and the new position are deinstalled from the binding array, and the current position is updated. Similarly, `Move_Engine_Down` installs a number of trailed binding in the binding array and updates the current position.

When an assignment is started (`Found...Work`, the engine positions its binding array at the tip node of the new or resumed branch in order to get ready to start executing the Prolog code.

8.7 Notifying Events

As explained in Section 3, the engine notifies the scheduler whenever an event of interest occurs, for example when a choicepoint is created, backtracked to, or deleted, when a predicate is called, and when a clause is entered. Notification of such events is done merely by calling the appropriate scheduler function and does not involve any change in the emulator logic.

9 Conclusions and Future Work

We have described the engine-scheduler interface used in the second generation of the Aurora or-parallel Prolog system. We have defined a simple set of functions to cover the two basic areas of engine-scheduler interaction: finding work and communication between workers. We have identified those events during Prolog execution that may be of potential interest to schedulers, e.g. creation of nodes, entering clauses, etc. We have also developed a general characterisation of pruning properties of Prolog clauses that can be used both for scheduling speculative work and for improving the implementation of pruning operators.

The interface described in this report is fundamentally revised with respect to earlier versions. The new interface is designed to help avoid scheduling overheads, to make the set of

basic concepts simpler and more uniform, to give scope for potential optimisations including better memory management, improved treatment of pruning operations, and avoidance of speculative work.

The main purpose of the interface is to enable different engines and schedulers to be used interchangeably. To date, four separate schedulers have been written and connected to the Aurora engine by means of the interface. Perhaps more importantly, the evolution of the interface has helped clarify many basic concepts in exploiting or-parallelism in Prolog, such as straightening, contraction, bypassing, and handling of pruning information.

No detailed performance analysis work has been done for the new Aurora implementation yet. Preliminary measurements have been performed with the Manchester scheduler, on the benchmark suite introduced in the performance analysis of the earlier Aurora version [12]. There is an overall improvement of up to 60% in terms of absolute speed, partly due to the new, much faster engine. For some of the fine granularity benchmarks the relative speedups have deteriorated; this is because the increase in engine speed implies a relative increase in scheduler overheads. For benchmarks with coarse granularity, and especially for the ones with frequent suspension and resumption, the relative speedups have improved, showing the advantages of the new interface.

We also believe that the new interface plays a significant part in the good performance results of the Bristol scheduler. The Bristol scheduler has been designed with the new interface in mind, and, in spite of applying a very simple scheduling strategy, it outperforms the earlier schedulers on several benchmarks [1].

Work is in progress to extend Andorra-I [15] for or-parallel execution by connecting it via the interface to the Bristol scheduler. A successful outcome of that experiment would prove the relevance of the interface design outside the scope of or-parallel Prolog implementation.

The main outstanding issue which has not been treated in the interface is garbage collection. Patrick Weemeeuw [14] has addressed the problem of garbage collection of the public parts of the tree. Since such activities involve synchronisation between workers and possibly relocation of scheduler data, it is likely that the interface will have to be extended to support garbage collection.

10 Acknowledgements

The work on engine-scheduler interfaces was initiated by David Warren. Earlier versions of the interface were developed by Ross Overbeek and Alan Calderwood. The design of the new interface benefited from several discussions with Tony Beaumont, Per Brand, Bogumił Hausman and Ewing Lusk.

The authors are indebted to Feliks Kluźniak and Ewing Lusk for careful reading and valuable comments on drafts of this report.

This work was supported by ESPRIT projects 2471 ("PEPMA") and 2025 ("EDS").

References

[1] Anthony Beaumont, S Muthu Raman, and Péter Szeredi. Scheduling or-parallelism in Aurora with the Bristol scheduler. March 1990. Internal Report, Gigalips Project.

[2] Per Brand. Wavefront scheduling. 1988. Internal Report, Gigalips Project.

[3] Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek, and Rick Stevens. Scheduling OR-parallelism: an Argonne perspective. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 1590–1605, MIT Press, August 1988.

[4] Alan Calderwood. Aurora—description of scheduler interfaces. January 1988. Internal Report, Gigalips Project.

[5] Alan Calderwood and Péter Szeredi. Scheduling or-parallelism in Aurora – the Manchester scheduler. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 419–435, MIT Press, June 1989.

[6] Mats Carlsson. *A Prolog Compiler and its Extension for Or-Parallelism.* SICS Research Report R90006, Swedish Institute of Computer Science, 1990.

[7] Mats Carlsson and Péter Szeredi. *The Aurora Abstract Machine and its Emulator.* SICS Research Report R90005, Swedish Institute of Computer Science, 1990.

[8] Bogumił Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG.* PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[9] Ewing Lusk. Remote I/O handling package specification. October 1989. Internal Report, Gigalips Project.

[10] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[11] Raed Sindaha. Scheduling speculative work in the Aurora or-parallel Prolog system. March 1990. Internal Report, Gigalips Project.

[12] Péter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732, MIT Press, October 1989.

[13] David H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

[14] Patrick Weemeeuw. A la recherche de la mémoire perdue, or: Memory compaction for shared memory multiprocessors. March 1990. CW-Report, K. U. Leuven, Department of Computer Science.

[15] Rong Yang and Vítor Santos Costa. Andorra-I: a system integrating or-parallelism with dependent and-parallelism. March 1990. Internal Report, Gigalips Project.

Appendix: Definition of the interface

The Appendix gives a detailed description of the interface version 2.15. In Section A.1, the data types common to the engine and scheduler are described. In Section A.2 and Section A.3, 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 A.4, a description is given of the data related to static properties of clauses that is supplied to the scheduler by the engine. Sections A.5 and A.6 give further details that may be of interest for a prospective user of the interface.

This Appendix is a slightly extended version of Chapter 3 of [7].

A.1 Common Data Types

A.1.1 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. 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*.

A.1.2 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.

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 A.4). 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.

A.1.3 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

A.2 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.

A.2.1 Finding Work

This section presents the macros for finding work. 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 A.3.1) 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.

A.2.2 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:

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

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).

```
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.

A.2.3 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 A.4);
- the whole system being halted.

A.2.3.1 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 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.

A.2.3.2 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.

Note that this macro may be invoked with the node FROM being above the node UP_TO, if the pruning operator is superfluous, and no nodes are actually destroyed.

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.

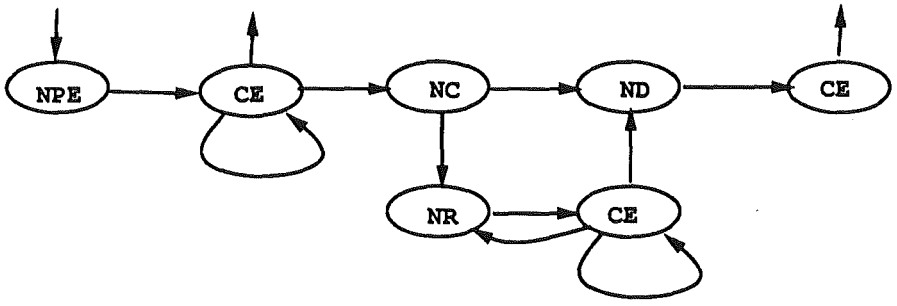
The invocation order of the node creation and clause entry macros is affected by the shallow backtracking optimisation. As creation of nodes is delayed, Sched_Node_Created will be called *after* the first Sched_Clause_Entered, while Sched_Node_Reused and Sched_Node_Destroyed will be called *before* the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

The graph in Figure 3 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.

A.2.3.3 Other Events

†void Sched_Halt(struct node *SENTRY)

This macro is called when the halt built-in is processed.



where

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

Figure 3: ORDER OF EVENTS RELATED TO NODE CREATION AND DESTRUCTION

```
†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 A.4 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.

A.2.4 Optimisations

This section describes two macros related to specific optimisations.

A.2.4.1 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.

†`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 A.2.3.2).

A.2.4.2 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.

A.2.5 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 node stack of the worker (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**.

A.2.6 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.

A.2.7 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.

A.3 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 (‡).

A.3.1 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.

A.3.2 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`.

A.3.3 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.

A.3.4 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.

A.3.5 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.

A.3.6 Further Node Handling

`struct node *My_Embryonic_Node()`

This macro returns a pointer to 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).

A.4 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 A.2.3.3):

```
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.

A.4.1 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.

A.4.2 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. 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).

The set of pruning data includes the following items for each clause:

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

`int ASSUMED_CUTS` and `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. it 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) ∧ *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, G2' → max_cuts(G1) + max_cuts(G2);
    B ≡ 'A1; A2' → max(max_cuts(A1), max_cuts(A2));
    otherwise → 0

max_commits(B) =
    commit_operator(B) → 1;
    B ≡ 'A -> C' → max_commits(C);
    B ≡ 'G1, G2' → max_commits(G1) + max_commits(G2);
    B ≡ 'A1; A2' → max(max_commits(A1), max_commits(A2));
    otherwise → 0
```

A.5 Related interfaces

This section describes two interfaces related to the engine-scheduler interface: macros for locking and for memory management. In the current version of Aurora these macros are defined by the engine.

A.5.1 Locking

`LOCKTYPE`

This macro provides a type for the declaration of locks. Locks are normally declared as fields of a (shared) data structure. Field selection expressions referring to such fields can be used in subsequent locking macros (see below).

`void Init_Lock(LOCKTYPE L)`

This macro initialises the lock `L`.

`void Lock(LOCKTYPE L)`

This macro obtains the lock `L`.

`void Unlock(LOCKTYPE L)`

This macro releases the lock `L`.

A.5.2 Shared Memory Management

```
char *Shared_Memory_Allocate(unsigned int SIZE)
```

This macro allocates a piece of shared memory of `SIZE` bytes. Returns a `NULL` pointer if no memory is available.

```
char *Shared_Memory_Reallocate(char *PTR, unsigned int OLD_SIZE,
    unsigned int NEW_SIZE)
```

This macro changes the size of a previously allocated block of size `OLD_SIZE` to `NEW_SIZE`, moving the contents if necessary. Returns a `NULL` pointer if no memory is available.

```
void Shared_Memory_Free(char *PTR, unsigned int SIZE)
```

This macro releases shared memory allocated by the above macros.

A.6 File Organisation

A.6.1 The Makefile

The scheduler is responsible for creating its "portion" of the makefile, in the file `'sch.makefile'`. This should define the following makefile-variables:

`$(SCH_OBJECTS)`

the list of object (`*.o`) files containing the scheduler,

`$(SCH_SOURCES)`

the list of source (`*.c *.h`) files containing the scheduler,

`$(SCH_INTERFACE_H)`

the list of scheduler macro files upon which the engine may depend. This list has to include `'sch.interface.h'`, `'sch.switches.h'`, `'sch.node.h'`, `'sch.alternative.h'` and any further files referred to by these files (directly or indirectly),

and it should make use of the following makefile-variable (defined by the engine):

`$(ENG_INTERFACE_H)`

the list of engine macro files upon which the scheduler may depend. This list has to include `'eng.interface.h'` and any further files referred to by this file (directly or indirectly).

The file `'sch.makefile'` should contain instructions for making each of the files listed in `$(SCH_OBJECTS)`.

A.6.2 New Built-in Predicates Defined by the Scheduler

New built-in predicates can be defined by the scheduler using the standard foreign language interface. The *ObjectFile* argument in `foreign_file` and the *ObjectFiles* argument in `load_foreign_files` should be made equal to `[]` for predicates included in the scheduler code. When booting, the engine consults file `'scheduler.pl'` to allow the scheduler to define the required new built-in predicates.

A.6.3 Files Provided by the Scheduler

`sch.switches.h`

contains the definition of the two static switches (Section A.2.7) to be provided by the scheduler.

`sch.interface.h`

contains all the remaining interface macros provided by the scheduler.

`sch.node.h`

fields to be included in **`struct node`**.

`sch.alternative.h`

fields to be included in **`struct alternative`**.

`sch.makefile`

the part of the makefile describing the scheduler.

`scheduler.pl`

a file to be consulted during the booting phase (to define scheduler specific built-in predicates).

A.6.4 Files Provided by the Engine

`eng.interface.h`

basic type definitions and macros provided by the engine.

Index