IAP for Dummies: The YAP Design

Manuel Eduardo Correia   Vítor Santos Costa

Technical Report Series: DCC-2000-1

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150-180 Porto, Portugal

Tel: +351+22+6078830 – Fax: +351+22+6003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

IAP for Dummies: The YAP Design

Manuel Eduardo Correia	Vítor Santos Costa
LIACC, DC-FCUP	COPPE/Sistemas
Universidade do Porto	UFRJ
Portugal	Brasil
mcc@ncc.up.pt	vitor@cos.ufrj.br

Abstract

One of the advantages of logic programming is the fact that it offers several sources of implicit parallelism. One particularly interesting form of And-Parallelism is Independent And-Parallelism (IAP). Most work on the implementation of IAP is based on Hermenegildo's RAP-WAM. Unfortunately there are some drawbacks associated with the classical approaches based on the use of parcalls and markers. One first observation is that the introduction of parcall frames significantly slows down sequential execution. Moreover, it may result in fine-grained parallel work. We found these problems to be particularly significant in the context of the implementation of combined AND/OR systems.

In this paper we take a fresh look at this issue. Our goal is to start from a standard sequential Prolog implementation and try to discover the minimal number of changes that would be required for an efficient implementation of IAP. The key ideas in our design are to (i) to always take advantage of analogy between or-parallelism and IAP; (ii) to avoid creating new structures by adapting already existing WAM data-structures wherever possible; and (iii) to avoid major changes to the compiler.

1 Introduction

One of the advantages of logic programming (LP) is the fact that one can exploit implicit parallelism in logic programs. Implicit parallelism reduces the effort required to speedup logic programs through parallelization. Moreover, implicit parallel systems alleviate the user from the actual details of work management, which can be quite difficult to program for the irregular problems commonly addressed in LP applications. Logic programs have two major forms of implicit parallelism: or-parallelism (ORP) and and-parallelism (ANDP). Given an initial query to the logic programming system, ORP results from trying several different alternatives simultaneously. In contrast, ANDP stems from dividing the work required to solve the alternative between the different processors.

One particularly interesting form of And-Parallelism is Independent And-Parallelism (IAP) [6, 9], where one only runs goals in parallel if the goals do not share variables. Non-Strict Independent And-Parallelism [12] gives a more general definition of independence between goals that allows for some variable sharing.

Most work on the implementation of IAP is based on Hermenegildo's RAP-WAM [9], originally implemented in &-Prolog [10]. The RAP-WAM is based on an execution scheme proposed by Hermenegildo and Nasr [11] which extends backtracking to cope with independent and-parallel goals. As in Prolog, the scheme recomputes the solutions to independent goals if previous independent goals are nondeterminate.

The &-Prolog language extends Prolog with the parallel conjunction and goal delaying. One objective of this system was to have sequential execution as close to Prolog as possible. To do so, &-Prolog maintains much of the Prolog data structures. &-Prolog programs are executed by a number of PWAMs running in parallel [9]. The instruction set of a PWAM is the SICStus Prolog instruction set, plus instructions that include the CGE tests and instructions for parallel goal execution. Synchronization between goals is implemented through the parcall-frames, data-structures that represent the CGEs and that are used to manage sibling and-goals. Moreover, start-markers and end-markers are used to perform space management. The resulting system has some overheads in relation to one processor execution but shows good speedups for the selected benchmark programs, including examples of linear speedups. Work based on these principles includes Shen's DASWAM prototype for DDAS [18]. Pontelli and Gupta at New Mexico State University have also used similar principles in their &-ACE prototype [15], a very successful implementation. Both these systems extend IAP to also support dependent and-parallelism.

Unfortunately there are some drawbacks associated with the classical approaches based on the use of parcalls and markers. One first observation is that the introduction of parcall frames significantly slows down sequential execution. Moreover, it may result in fine-grained parallel work. Work in granularity has obtained good results in minimizing these problems [7, 13, 19]. Moreover, arguably these models are quite complex to implement. This is partly because of the intricacies of backtracking through parallel goals, but also because of the efforts required to manage efficiently parallel conjunctions. We found these problems to be particularly significant in the context of the implementation of combined AND/OR systems.

We next take a fresh look at this issue. Our goal is to start from a standard sequential Prolog implementation and try to discover the minimal number of changes that would be required for an efficient implementation of IAP. Ultimately, we aim at a novel IAP proposal that can be naturally extended to support ORP through the SBA [3] or other approaches, such as the αCOWL [16].

The key ideas in our design will be (i) to always take advantage of the analogy between ORP and IAP [14]; (ii) to avoid creating new structures by re-utilizing WAM data structures wherever possible; and (iii) to avoid major changes to the compiler. We shall assume that the underlying system is WAM based [20], like YAP [5, 17].

Next we propose our model in a stepwise fashion, starting from the simplest case, forward execution, and then studying the requirements to support outside, inside backtracking and side-effects.

2 Principles of Forward Execution

Independent And-Parallelism corresponds to taking goals from a conjunction and running them in different computing elements:

$$g \quad \texttt{:-} \quad a(X) \ \& \ b(Y) \ \& \ c(Z).$$

We name a set of goals that run in parallel a parallel conjunction. At any moment, a number of parallel conjunctions may be present, so more parallel goals than processors may exist. If so, a scheduler will select which processors will run a specific goal. Following Hermenegildo, we say that the scheduler should follow the "no-slowdown rule", that is, goals should be selected in such a way as to guarantee at most a constant-bound slowdown over a sequential execution. We shall name this bound the fixed overhead. One form of following this rule is to guarantee that all parallel goals "to the left", that is all goals such there exist no parallel neighbors that would have already executed in the sequential ordering, will execute first. More concisely, we define a partial order for all parallel goals, say $<$, such that $g_i < g_j$ if g_i and g_j belong to the same parallel conjunction and g_i is written to the left of g_j. The no-slowdown rule is guaranteed when g_j can only execute if g_i is actively executing or if it has executed.

The no-slowdown rule can be seen as an analogue of the sequential exploitation of alternatives in ORP (see Pontelli and Gupta for an in-depth discussion [14]). Moreover, the parallel conjunction represents a subset of a clause's body, which is represented as an environment in the WAM [20]. Clearly, this begs the question: why not represent parallel execution within an environment. We shall thus define next the first implementation principle for our system:

- Information regarding and-parallel execution of goals should be, whenever possible, part of the environment.

Actual implementation shows that IAP execution behavior is more subtle. Indeed, to actually implement forward execution two major problems must be addressed: memory management and scheduling.

2.1 Memory Management

Continuous execution of one or several parallel goals forms an *and-task*. An and-task will start when a processor takes an and-goal from the scheduler, and will end when the processor calls again the scheduler, moves into inside or outside backtracking, or suspends. Our goal in a parallel execution is to have several and-tasks progressing simultaneously.

Unfortunately, these tasks cannot be represented within the simple WAM stack based approach. Most, if not all, parallel systems that address this problem (both for IAP and ORP) have followed the same approach: give each processor a chunk of the available space, and allow them to expand it independently. For efficiency, while doing forward execution, each processor should see its space as a set of expanding stacks, as in the WAM. Thus we can give a precise definition for an and-task:

- A set of parallel and sequential and-goals such that their execution can follow the WAM's, that is, such they follow the total sequential ordering.

Because a processor will jump from and-task to and-task, its space will have memory segments for several tasks. The memory map of the system will thus be a network of task segments, several per processor space.

In the RAP-WAM, memory management for a task segment is performed by two markers, the start-marker and the end-marker. Memory management for the analogue ORP system Aurora is quite different. An or-task is a set of or-parallel nodes. Memory management for an or-task is performed by a special choice-point, the sentry-node.

We will take advantage of this analogy next, to propose using the choice-point start marker, `SM` as indicating the beginning of a new and-task (trail-frames are an obvious alternative). Our reasons for choosing choice-points instead of special terms in the global or new environments are largely pragmatic and have to do (i) with IAP/ORP integration; and (ii) the use of choice-points for trailing.

Next, we give a broad description of what should be stored in a `SM` when creating a new and-task:

- Pointers to WAM stacks, as in any other choice-points.
- Pointer to the worker's doubly linked list of segments, ordered by addresses. These pointers are required to recover space.
- Pointers to a doubly linked list of segments, ordered by the total sequential ordering. These pointers are required for backtracking.
- Pointer to the last choice-point for this and-task; this is not strictly necessary, but speeds up maintaining a list of choice-points.

Note that we shall not place information on actual tasks in a `SM`. `SM` exist for, and only for memory management. When they can be recovered, the space they were protected will be recovered. Figure 1 shows an example of the use of `SM`s for a two processor execution of the following program:

```
a(X)  :-  b(X)  &  c(X).

c(X)  :-  d(X)  &  e(X).
```

We assume that procedures for `b/1`, `d/1`, and `e/1` are sequential. Worker W_1 executes $a(X)$, $b(X)$ and then $e(X)$. Worker W_2 executes $c(X)$ and $d(X)$. The figure only shows the choice-point stacks. The labeled gray boxes correspond to `SM`s. Each `SM` corresponds to a different and-task. The initial `SM` is labeled with $a(X)$ but in practice

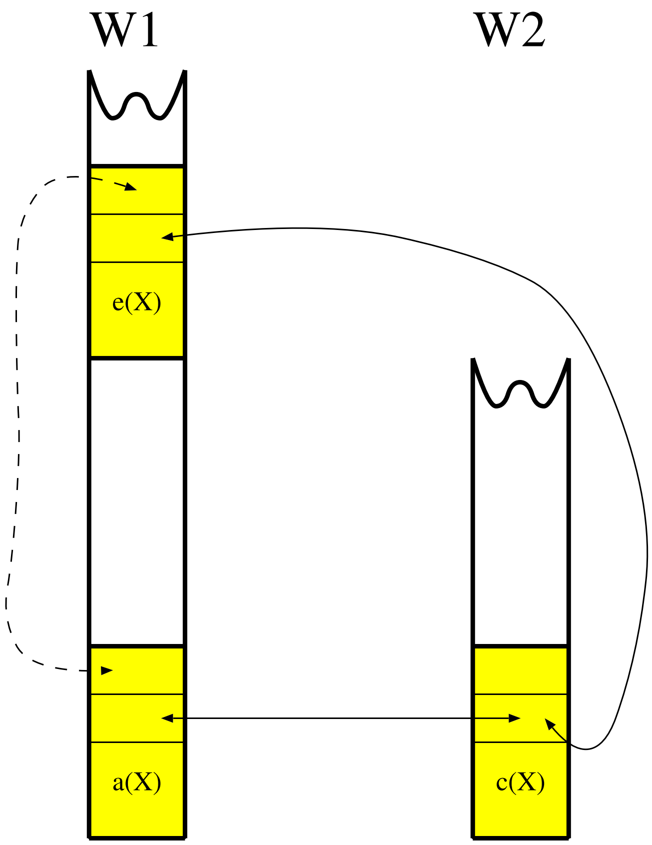

Figure 1: Connecting And-Tasks.

it would have created by W_1 long before executing $a(X)$. The second and-task is labeled by $e(X)$ and actually starts at $e(X)$. Worker W_2 has a single and-task. The dotted line shows the physical pointers in the stacks for W_1. These pointers are used to reclaim space: whenever the stacks for a task can be deleted we remove its `SM` for this doubly-linked list, if the `SM` had another `SM` above, or we physically recover the space, if this one was the top `SM`. The other lines represent the actual left-to-right connection between tasks. This connection will be followed when backtracking or when determining whether a task is leftmost. They are also required when we publish work in an AND-OR system [4, 2].

2.2 Scheduling

Traditionally, scheduling is performed very differently in IAP systems and in ORP systems. IAP systems have represented each parallel goal to be scheduled as a "goal record", that is stored in one or several run-queues. In contrast, ORP systems walk through the ORP tree looking for nodes with available alternatives. It is arguable which approach is best. We will select the run queue approach here, largely because we already implemented a IAP scheduler based on run-queues. We can also use an hybrid approach, where goals can be picked directly from the search tree or from the run-queues. For full details please consult [2].

Having selected a scheduling mechanism, we must define what is a goal record. We will define it to contain two elements:

- A pointer to the environment computation slot where work is to be found;
- A pointer to the code that we must execute to start the goal.

3 Control Flow in the Forward Execution

We have presented the main data structures that describe computation. It is time to define our basic control algorithm for the exploitation of IAP in Prolog-based systems:

Default Execution will be sequential, as in other Multi-Sequential systems. In other words, default execution will try to execute all parallel and sequential goals in a single and-task. When finding a parallel conjunction, the corresponding environment will be suitably extended to support parallel execution, but execution will at first ignore these fields.

Work Publishing is to be performed on demand. After receiving requests from a worker, the environment chain for a worker will be walked through, and environments with parallel alternatives will be initialised by setting up the necessary fields and adding the yet not executed parallel goals to the run-queue.

Returning Into a Parallel Environment will happen if we execute successfully after having published goals. At this point, we will have an option:

1. If the next goal to execute is yet to be executed, we should take it and continue to expand our current and-task. In this case, we should cancel the corresponding goal-record.
2. If the next goal to execute is being executed by some processor, or if it has been fully executed, our and-task must be terminated. We will thus therefore close our current and-task, and again we have a choice:
   (a) If all parallel goals in our parallel conjunction have been completed, the new and-task should start from the continuation of this conjunction.
   (b) Otherwise, we should call the scheduler for new and-parallel work.
3. If there are no more goals to execute (we are ending the rightmost goal) and if all the other goals in the environment are fully terminated, the continuation can be executed within the same and-task. Otherwise the current and-task must be terminated and the scheduler should be called for new and-parallel work.

Scheduling is performed when no more tasks are available from the current parallel conjunction, and some goals are still being executed. In our design the scheduler will look for work from the run-queues and start a new and-task.

These four states describe the state of during forward execution. We shall later describe how to implement backtracking and side-effects in the system.

3.1 Data-Structures for Forward Execution

We are now in a position to account for the data-structures required for IAP forward execution:

- Goal-Frames will store a pointer to the environment and a pointer to the code to execute next.
- `SM`s will be linked into two lists and will contain pointers to the base code for their stacks.
- Environments will contain an extra pointer for each parallel goal.

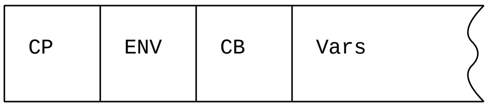

Figure 2: A Sequential Environment.

We next discuss in more detail how environments work. Figure 2 shows a sequential environment in YAP [5, 17]. Each environment contains three important fields: the Continuation Pointer, `CP`, storing the code to execute after completing execution of the environment, the parent environment, `E`, and the current choice-point, `CB`, used to implement cut.

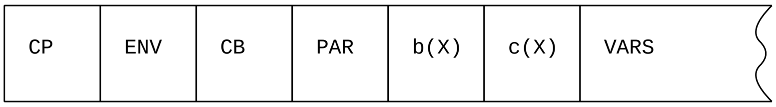

Figure 3: A Parallel Environment.

Figure 3 shows a parallel environment in our scheme for the clause:

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

Originally, the environment contains three uninitialized extra fields. One will point at `SM` for the task which originally created this environment and the other two are used to store information on the goals $b(X)$ and $c(X)$.

Note that there is no field explicitly saying whether an environment is sequential or parallel. The work publishing step detects whether an environment is parallel or sequential by testing the `CP` field of the child environment. If this field points to a special and-parallel instruction that we will discuss in the compilation section, the environment is parallel. Otherwise, it will be sequential. The child environment also informs of how many goals in the conjunction have already been execution.

The work publishing phase goes through an uninitialized environment, and readies goals for parallel execution:

- At least one goal, the current one, will always be marked as *Running*.
- Goals to its left will be marked as *Done*.
- Goals to its right will be marked as *Available*. Such a goal will be entered into the run queue.

Last, the work publishing phase initializes the `CSM` to point to the current task.

We have seen that initially parallel goals may be either initialized, that is *Free*, or in one of the *Running*, *Done*, or *Available* states. In more detail:

1. *Free* : the system is still trying to execute this conjunction sequentially. In this case the goal-field will be uninitialized (garbage).
2. *Done* : either the goal was executed sequentially, that is, before the environment was published or the goal is executed within the same and-task as the previous goal; this is represented by binding the field to the atom `$done$`.
3. *Available* : the goal is yet to be executed; this is represented by making the variable point to the goal record, which is tagged as a list.
4. *Running* : the goal is being executed; this is represented by pointing to the activation structure, tagged as an application term (compound term). The structure will have a field `Completed` that when bound will indicate that the goal has completed.

3.2 Picking Up New Work

Picking up work will take place at two points:

1. We have completed the previous goal, G_i, and we take the next goal, G_{i+1}. In this case, we shall not mutate our own state to executed, but we shall mutate goal G_{i+1}'s state to `$done$`.
2. We are initial-Ising a new task, either because (i) goal G_{i+1} was being executed but another goal from the same environment, G_j, $j <= i+1$ was still available, or because (ii) we found this goal in the scheduler. In both cases we must start a new and-task.

We argue that in most interesting problems good scheduling should result in most conjunctions being kept private, and therefore the total number of these operations should be much smaller than the total number of continuations executed.

Starting a New And-Task Starting a new and-task is only required when we fetch new work from the scheduler. To initiate a new and-task, we must create a new `SM`, as described in section 2.1. To do so, we would like to know how to insert our new `SM` in the chains of `SM`s. Inserting into the physical chain is straightforward, but inserting in the left-to-right chain requires some thought. The problem arises because a goal may be executed in several and-tasks, so when we start a goal we do not know which and-task will complete it. To address this problem thus requires collaboration between the worker that starts a goal and the worker that completes this task.

The key idea in our design is that the environment slot for a goal in the state *Running* is bound to a term with two fields:

- `InitialSM`: our new start marker, initialized by the new and-task;
- `LastSM`: our final start marker, initially an unbound variable; this field being unbound also indicates the goal is yet to complete.

Moreover, a new and-task must search the goals to the left according to the following algorithm:

1. If the goal is `$done$`, skip.
2. If the goal is in *Available*, stop.
3. If the goal is in *Running* and has completed, link its `LastSM` to our `InitialSM`, make their last choice-point the parent to our `InitialSM`, and then stop.

Last, when completing a new and-task one should set `LastSM` to the value of the current `SM`, which in general will be different from `FirstSM`. Moreover, one should propagate this to the goals to the right according to the following algorithm:

1. If the goal is `$done$`, skip.
2. If the goal is *Available*, stop.
3. If the goal is in *Running* and has completed, link its `FirstSM` field with our `LastSM`, make our last choice-point the parent to their `FirstSM`, and then stop.

In section 3.1 we argued that goals in the state *Running* would need an extra field, `Completed`, that when bound would indicate that the task has completed. In practice, the functionality of this field may be given by the `LastSM` field, as this field is only initialized when a task completes.

3.3 Completion of a Parallel And-Task

We are now in a position to say when a parallel conjunction has completed. Parallel conjunctions complete when all of their goals are either:

- in the `$done$` state; or
- they are in the *Running* state and `LastSM` is bound.

Note that if the goal that completes the parallel conjunction is the rightmost goal, execution can continue within the same and-task, otherwise a new and-task must start.

3.4 Compilation of Forward Execution

We shall only consider the simpler case of unconditional IAP first in this discussion. IAP systems have always required extra instructions to implement parallelism. In contrast, ORP parallel systems do not require specific instructions to implement parallelism, even though most will indeed use them.

The ORP-IAP analogy breaks here because of a major difference between choice-points and environments:

- Must of activity of ORP management takes place at backtracking, which is not managed explicitly in traditional Prolog implementations.
- Normally a new goal is to be started when we reenter a parallel conjunction. Therefore, it is important to check for the status of the conjunction before actually creating the data structures for the goal. This always requires extra instructions.

We therefore propose the following simple changes to the compiler and instruction set:

- The `allocate` instruction must be informed there are a few extra fields corresponding to a parallel execution. For this purpose we shall define a new instruction, `allocate_pcall`. In systems such as YAP that do initialize environments, even this instruction is not strictly necessary.
- A new instruction must be inserted before a parallel goal is started. We shall name this instruction `check_par`. Its task is to detect whether the goal is being run privately or shared. If the goal is parallel the instruction will detect completion.

  Our goal is for the `CP` to a parallel environment to always point to a `check_par` instruction.
- The `deallocate_if_not_parallel pcode` instruction is executed to deallocate the cge-environment in the eventuality that the CGE is completely executed within the same task and no release of and-work occurs during its lifetime.
- A new `close_pcall` instruction must be included as the continuation of the parallel environment.

In what follows we describe in some more detail the arguments and usage of these new instructions by showing the compilation of two simple cge-clauses.

3.4.1 Compilation examples.

Consider the following CGE :

$$a(X) \ \leftarrow \ (\ b(X) \ \& \ c(X) \ \& \ d(X) \ ).$$

In our new compilation scheme this will be compiled into :

```
   get_var Y6,A1
   allocate_pcall 3,1
   call b/1
First_par:
   check_par 2, next_par, 3
   put_val Y6,A1
   call c/1
Next_par:
   check_par 3, PCP, 3
   put_val Y6,A1
   deallocate_if_not_parallel Pcode,3
   execute d/1
Pcode:
   call d/1
PCP:
   close_pcall 3
   proceed
```

In this simple example we observe the use of the new machine instructions we have introduced to support the new execution scheme. There are four new instructions, namely :

1. **allocate_pcall n_goal_slots, n_environment_variables** – This instruction allocates a cge-environment. Observe that at this point the cge-environment is not yet parallelized. This is indicated by having all goal-slots initialized with the zero (SEQ) value.

2. **check_par goal_slot, next, n_slots** – This instructions verifies if the goal-slot GOAL_SLOT is still sequential (initial zero value when the cge-environment is first allocated). If this is the case this is a NOP operation. If not it checks to see the kind of value that is held in the goal-slot. if it is a pointer to a goal-frame in a run-queue it tries to grab by incorporating the goal into its current task.

   If it fails, the engine enters in a special *and-search* mode of operation and proceeds with the execution of the instruction labeled by **next**. This can be another *check_par* instruction or a *close_pcall* instruction.

   Each **check_par** instruction is associated with a goal-slot that is called the *IGS* (Instruction Goal Slot).

3. **deallocate_if_not_parallel pcode, n_goal_slots** – This instruction is executed to deallocate the cge-environment in the eventuality that the CGE is completely executed within the same task. If this is the case the cge-environment can be treated as a regular wam environment and deallocated. Otherwise this instruction results in a jump to the instruction labeled by *pcode*.

4. **close_pcall n_goal_slots** – if this instruction is reached in the normal mode of execution, we have to deal with several situations :

   (a) All other cge goals are completed. In this case, if the just completed goal is the rightmost, the continuation of the cge-environment can be incorporated (bypassed) into the current sequential task by closing the cge-environment and continuing execution with the next instruction.
   (b) All other cge goals are completed but the just completed goal is not the rightmost cge goal. In this case the and-agent closes the current task and begins a new task with the continuation.
   (c) If there are still unfinished goals within the cge, then the and-agent closes the current task and enters the and-scheduler to look for further and-work to do.

   If this instruction is reached in the special *and-search* mode of execution then the and-agent checks to see if the cge is completed. If this is the case it starts a new sequential task for the cge continuation. If on the other hand the cge is not yet completed the and-agent closes the current sequential-task and enters the and-scheduler to look for further and-work to do.

For an example of a CGE with a continuation within the same clause consider the following :

$$a(X) \leftarrow (\ b(X)\ \&\ c(X)\ \&\ d(X)\ ),\ e(Y).$$

```
   get_var Y6,A1
   allocate_pcall 3,2
   call b/1
First_par:
   check_par 2, next_par, 3
   put_val Y6,A1
   call c/1
Next_par:
   check_par 3, PCP, 3
   put_val Y6,A1
   call d/1
PCP:
   close_pcall 3
   put_val Y7,A1
   deallocate_if_not_parallel nil,3
   execute e/1
```

Observe that in this case there is a sequential goal (`e(Y)`) in the continuation of the cge that still needs to access the environment. In this case we simply cannot try to deallocate the environment before the cge-closure in *close_pcall*, as illustrated in the previous case.

If any sequential goal follows the cge, the environment deallocation can only be attempted with *deallocate_if_not_parallel* just before the execution of the last goal of the CGEs clause. Observe also the special label **nil**. It indicates no jump in case the cge-environment fails to be deallocated because it turned out to be parallel.

These extra instructions do not prevent the application of environment trimming, because in a cge-environment goal-slots precede local variables.

4 Implementation of Outside Backtracking

The key idea for our implementation of outside backtracking is based on two principles:

- All choice-points should have a parent according to the total sequential ordering. Outside backtracking implements backtracking by following this order, but must allow for parallel goals that have been fully executed to be redone.

- Parallel goals will be restarted by trailing the $Available \Rightarrow \{Done, Running\}$ transitions. When we backtrack, the transition will be undone, and new goals will be added back to the run-queue.

  It is interesting to remark that goal slots perform as mutable variables [1, 5]. In fact, the same implementation techniques will be used in both cases.

- Space is marked as free when backtracking into a `SM`. The alternative for a `SM` will point to the memory manager release stack code.

Note that if we backtrack over a parallel conjunction it makes no sense to have run-queue entries pointing at it. To address this problem we must know whether the choice-point we are backtracking to is younger or older than the environment for these goals.

Figure 4 shows the problem of outside backtracking in some more detail. Squares represent environments and round boxes choice-points. We are backtracking from after a parallel conjunction to the circle, and we want to kill all the shadowed area, including any parallel conjunctions we had created. Note that all goals for these conjunctions have been created and have therefore left the run-queues.

Fortunately, the key observation to know which environments have goals to restart is straightforward. We first notice that the conjunction `OUT_ENV` must have a common ancestor of `B_ENV` that we shall call `COMMON_ENV`. All environments that are between `B_ENV` and `COMMON_ENV` are still active, and transitions for these environments must be reset. All other environments have been killed and transitions for these environments must be ignored.

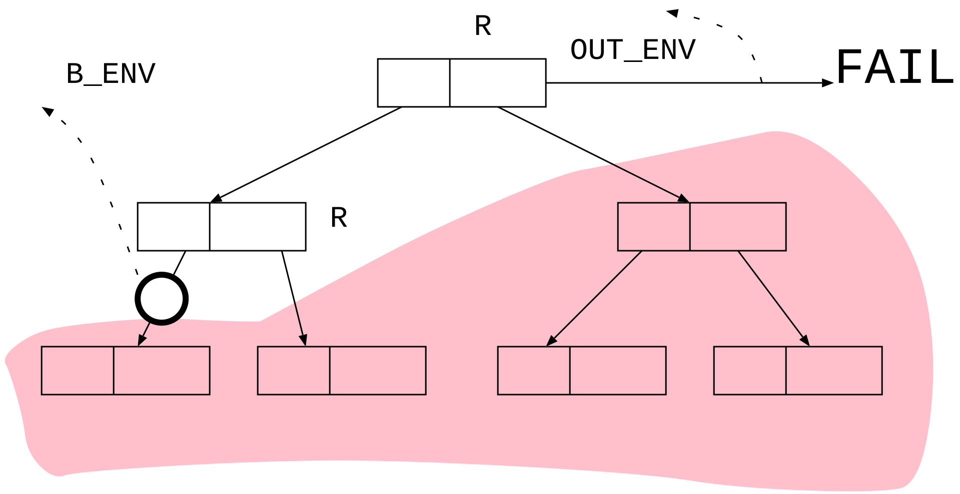

Figure 4: Outside Backtracking.

5 Implementation of Inside Backtracking

Inside Backtracking (IB) is the most complex feature to implement in IAP systems. With IB, we must tackle two problems:

1. Are we doing IB or OB?
2. What goals to undo?

To implement item (1) we need to step through the environment and detect whether all goals have completed. Alternatively, we can use an extra field in the environment, to informs us whether we have done the parallel conjunction or not. If we have not, and we are undoing a $Available \Rightarrow \{Done, Running\}$ transition, then we have entered IB.

To detect which calls to undo, we shall use the following invariants:

- All goals that have completed have pointers to their initial and final `SM`, so we can undo all these computations by going through segments.
- All goals that have yet not completed have pointers to their initial and-task. We can imagine that this and-task forms the root of a tree. If a leaf in the tree has completed it will have a pointer to another node. If a leaf is still being executed, this pointer will not have been initialized, and there may be a worker there. Our algorithm should visit the tree and find all workers at leaves of the tree. These workers should then be signalled and enter IB themselves until the whole of the tree is recovered.

Figure 5 shows the problem of inside backtracking in some more detail. We notice that the `FAIL` stems from a deterministic goal where the other goals have created new

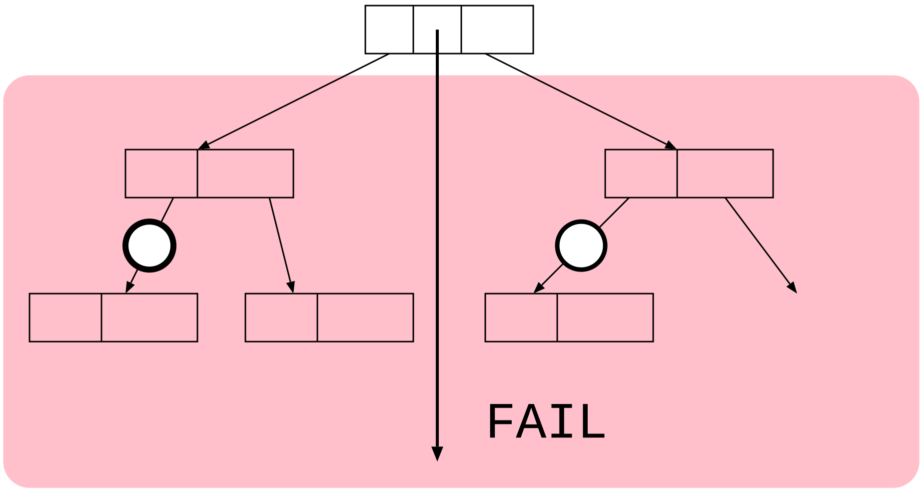

Figure 5: Inside Backtracking.

environments and choice-points. In this case we may use one the two techniques discussed above: walking through the and-tasks created including and since the original environment and forcing them to backtracking; or sending a signal and asking workers to check for their current position.

A detailed discussion of the data-structures required to fully implement backtracking can be found in [2].

6 Implementation of Side-Effects

To implement side-effects it is sufficient to see whether the current and-task is the leftmost and-task, or in other words, to see whether all and-tasks to left are completed. More precisely, by following the left-to-tight chain of pointers we should reach the top-level task [8].

If we expect this operation to be performed often, we can optimize the algorithm by keeping a global variable with the current leftmost and-task.

7 Conclusions

We have presented a novel scheme for the complete implementation of Independent And-Parallelism. Our scheme is designed for fast sequential Prolog systems, such as YAP [17]. Moreover, we assumed in our design that ultimately the system will be a part of an integrated And/OR system such as the SBA [3, 2], thus all data-structures presented should be seen either as choice-points, trail entries, or logical or mutable variables.

Work on the implementation of our model has started at the Universidade do Porto. We are adapting YAP to support threads and the new instructions required to support

IAP. In this work we will benefit from previous work in integrating the SBA into IAP.

Note that optimizations to this are still possible. More specifically, it is possible to compact and-tasks if no choice-points were created between.

8 Acknowledgements

The authors would like to acknowledge and thank the contribution and support from Fernando Silva. The work has also benefitted from discussions with Luís Fernando Castro, Inês de Castro Dutra, Kish Shen, Gopal Gupta, and Enrico Pontelli. Our work has been partly supported by Fundação da Ciência e Tecnologia and JNICT under the projects Melodia (JNICT/PBIC/C/TIT/2495/95) and Dolphin (PRAXIS/2/2.1/TIT/1577/95).

References

[1] Johan Andersson, Stefan Andersson, Kent Boortz, Mats Carlsson, Hans Nilsson, Thomas Sjoland, and Johan Widén. SICStus Prolog User's Manual. Technical report, Swedish Institute of Computer Science, November 1997. SICS Technical Report T93-01.

[2] Manuel Eduardo Correia. *On the Implementation of AND/OR Parallel Logic Programming Systems*. PhD thesis, Universidade do Porto, 2000.

[3] Manuel Eduardo Correia, F. M. A. Silva, and Vítor Santos Costa. The SBA: Exploiting orthogonality in OR-AND Parallel Systems. In *Proceedings of the 1997 International Logic Programming Symposium*, pages 117–131. MIT Press, October 1997. Also published as Technical Report DCC-97-3, DCC - FC & LIACC, Universidade do Porto, April, 1997.

[4] Manuel Eduardo Correia, F. M. A. Silva, and Vítor Santos Costa. The SBA: Exploiting orthogonality in OR-AND Parallel Systems. In *ILPS97*, pages 117–131. MIT Press, October 1997.

[5] L. Damas, V. Santos Costa, R Reis, and R. Azevedo. *YAP User's Guide and Reference Manual*, 1998. http://www.ncc.up.pt/~vsc/Yap.

[6] Doug DeGroot. Restricted and-parallelism. In Hideo Aiso, editor, *International Conference on Fifth Generation Computer Systems 1984*, pages 471–478. Institute for New Generation Computing, Tokyo, 1984.

[7] Pedro López García and Manuel Hermenegildo. A Technique for Dynamic Term Size Computation via Program Transformation. Research Report, Facultad de Informática, Universidad Politécnica de Madrid, TR CLIP 8/93.1(94), March 1994.

[8] Gopal Gupta and Vítor Santos Costa. Cuts and Side-Effects in And-Or Parallel Prolog. *Journal of Logic Programming*, 27(1):45–71, April 1996.

[9] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[10] M. V. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[11] M. V. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 40–54. Imperial College, Springer-Verlag, July 1986.

[12] M. V. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[13] Andy King, Kish Shen, and Florence Benoy. Lower-bound Time-complexity Analysis of Logic Programs. In Jan Małuszyński, editor, *Logic Programming: Proceedings of the 1997 International Symposium*, pages 261–276. The MIT Press, 1997.

[14] E. Pontelli and G. Gupta. On the duality between or-parallelism and and-parallelism in logic programming. *Lecture Notes in Computer Science*, 966:43–??, 1995.

[15] E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995.

[16] Vítor Santos Costa. Cowl: Copy-on-write for logic programs. In *Proceedings of the IPPS/SPDP99*, pages 720–727. IEEE Computer Press, May 1999.

[17] Vítor Santos Costa. Optimising bytecode emulation for prolog. In *LNCS 1702, Proceedings of PPDP'99*, pages 261–267. Springer-Verlag, September 1999.

[18] Kish Shen. Exploiting And-parallelism in Prolog: the Dynamic Dependent And-parallel Scheme (DDAS). In *Proceedings of the 1992 Joint International Conference and Symposium on Logic Programming*, 1992.

[19] Kish Shen, Vítor Santos Costa, and Andy King. Distance: a New Metric for Controlling Granularity for Parallel Execution. *Journal of Functional and Logic Programming*, 1999(Special Issue 1), April 1999.

[20] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.