COWL: Copy-On-Write for Logic Programs

Vítor Santos Costa
LIACC & DCC-FCUP
Universidade do Porto
vsc@ncc.up.pt

Abstract

In order for parallel logic programming systems to become popular, they should serve the broadest range of applications. To achieve this goal, designers of parallel logic programming systems would like to exploit maximum parallelism for existing and novel applications, ideally by supporting both and-parallelism and or-parallelism. Unfortunately, the combination of both forms of parallelism is a hard problem, and available proposals cannot match the efficiency of, say, or-parallel only systems.

We propose a novel approach to And/Or Parallelism in logic programs. Our initial observation is that stack copying, the most popular technique in or-parallel systems, does not work well with And/Or systems because memory management is much more complex. Copying is also a significant problem in operating systems where the copy-on-write (COW) *has been developed to address the problem. We demonstrate that this technique can also be applied to And/Or systems, and present both shared memory and distributed shared memory designs.*

1. Introduction

Logic programming systems are a good match for parallel computers. As parallelism in logic programs can be exploited implicitly, the programmer can be left free to concentrate on the logic of the program and on the control information necessary to obtain efficient algorithms. The two major forms of implicit parallelism are and-parallelism and or-parallelism. *Or-parallelism* (ORP) aims at exploring different alternatives to a goal in parallel, and is typical of search-based applications. *And-parallelism* consists in the parallel execution of two or more goals that cooperate in determining the solutions to a query. One very important form of and-parallelism is independent and-parallelism (IAP), where parallel goals do not share variables. This form of parallelism arises in divide-and-conquer algorithms, whereas dependent and-parallelism (DAP) in common in consumer-producer applications.

In order for parallel logic programming systems to become popular, they should serve the broadest range of applications. To achieve this goal, designers of parallel logic programming systems would like to exploit maximum parallelism for existing and novel applications, ideally by supporting both and-parallelism and ORP. Exploiting just one form of implicit parallelism requires sophisticated system design, and exploiting distinct forms of parallelism at the same time is even harder. Supporting Full Prolog semantics in a parallel environment requires further care.

One very successful approach to implement or-parallelism is *copying*, as originally implemented in Muse. The idea with copying based models is that work is shared by simply copying the execution stacks between different workers (or processors). The advantage is that this technique has very low overheads over sequential implementations. Unfortunately, attempts at using copying for an unified And/Or system have not been very successful. The extra complexity of stack management in the presence of IAP makes it very complex to select which chunks to copy.

In this work we present an efficient alternative to traditional copying based systems. The idea is to do virtual copying, by applying the copy-on-write technique that has proven so effective with Operating Systems. We name our idea COWL, or copy-on-write for logic programs. We prove that this leads to a rather direct solution for the problem of implementing And/Or parallelism, the αCOWL. Next, we demonstrate a more complex solution that improves and-parallelism, the βCOWL. This technique can be a good foundation for a distributed memory implementation based on shared virtual memory techniques, which we name the δCOWL.

2. Issues in And/Or Parallelism

We first briefly survey the issues in implementing purely or-parallel and and-parallel systems.

2.1. Or-Parallel Systems

In ORP models of execution, several alternative search branches in a logic program's search tree *can* be tried simultaneously. So far, quite a few models have been proposed. Most successful have been the multi-sequential models, where processing agents (workers in the Aurora [14] notation) select ORP work and then proceed to execute as normal Prolog engines. A fundamental problem in the implementation of ORP systems is that different or-branches may attribute different bindings to the same variable. In an ORP system, and differently from a sequential execution, these bindings must be simultaneously available. The problem is exemplified in Figure 1, where choice-points are represented by black circles and branches that are being explored by some worker are represented by arrows. The two branches corresponding to workers W_1 and W_2 see different bindings for the variable X.

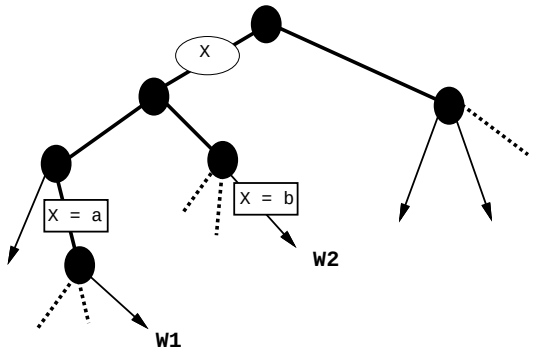

Figure 1. The Binding Problem in Or-parallelism

A large number of ORP models, including different solutions to these problems have been proposed (the reader is referred to Gupta [6] for a survey of several ORP models). The models vary according to the way they address the binding problem.

In shared space models all the workers share the stacks. They thus need to represent the different bindings for the or-branches. One example is the SRI model, where each Prolog variable has associated to it a cell in an auxiliary *private* data structure, the *binding array* [19]. A popular alternative solution to the binding problem is to have each worker operating on its part of the or-tree as independently from other workers as possible. In copy scheme based implementations, such as Ali and Karlson's [1] Muse, whenever a worker W_1 needs work from a worker W_2, it copies the entire stacks of W_2. Worker W_1 will then work in its tasks independently from other workers until it needs to request more work. To minimise the number of occasions at which copying is needed, scheduling in Muse favours selecting work at the bottom of tree.

Copy-based systems have become popular because they use the same data-structures as a sequential Prolog engine during ordinary execution. Thus they do not suffer any special overheads during ordinary execution. The simplicity and good parallel speedups possible with copying have made it most popular form of exploiting Or-parallelism. Copying has also been used with good results for distributed ORP systems such as Opera [2].

2.2. And-Parallel Systems

A second popular form of parallelism in logic programs is independent and-parallelism (IAP). This form of parallelism was originally proposed by DeGroot [5] as a scheme where only goals which do not have any run-time common variables are allowed to execute in parallel. Hermenegildo's &-Prolog system was a first implementation of IAP for Prolog [12]. It is based on an execution scheme proposed by Hermenegildo and Nasr [13] which extends backtracking to cope with IAP 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 corresponding 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 [12]. The instruction set of a PWAM is the WAM instruction set, plus instructions to test for independence between goals and instructions for parallel goal execution. Synchronisation between goals is implemented through the *parcall-frames*, data-structures that represent the parallel conjunctions and that are used to manage sibling and-goals. The &-Prolog system has very low overheads in relation to the corresponding sequential system, SICStus Prolog, and shows good speedups for the selected benchmark programs, including examples of linear speedups. The last few years have seen important developments on the efficient implementation of IAP, and several systems implementing a superset of independent and-parallelism are now available, such as Shen's DASWAM [17], and Pontelli and Gupta's &-ACE [15].

2.3. And/Or Parallel Systems

Interest in And/Or parallelism arose quite early in logic programming. Conery's AND/OR process model [3] was one of the most influential early models. In Conery's model, or-processes are created to solve the different alternative clauses, and and-processes are created to solve the body of a goal. Processes communicate through messages. One

traditional implementation of this model is to send in the message the goals to execute. Unfortunately this technique, known as *environment closing*, incurs significant overhead. An alternative approach to And/Or parallelism is to combine pure ORP with IAP. In the initial models, such as Gupta's Extended And-Or Tree model [9], Several solutions from the IAP computations are implemented through a special cross-product node, which can be quite complex to implement. These models are complex and thus hard to implement efficiently. Moreover, they do not adapt well to supporting traditional Prolog.

More recent proposals for combined And/Or parallelism use backtracking to obtain the several And-parallel solutions (thus, and as in &-Prolog, some recomputation is performed) [8]. Such systems should be easier to implement than PEPSys or the Extend And-Or Tree Model, as they can exploit the technology of &-Prolog and the ORP systems, and as they do not need to calculate cross-products. Such proposals include Shen's Fire [17] and the *C-Tree*, a framework for And/Or models that was proposed by Gupta et al. [8]. The extension essentially uses the idea of recomputing independent goals of a parallel conjunction, such as `a & b`. Thus, for every alternative of `a`, the goal `b` is computed in its entirety. Each separate combination of `a` and `b` is represented by a *composition* node (c-node for brevity). The extended tree for the above query, which we term the *Composition-tree* (C-tree for brevity), would appear as shown in Figure 2. Each composition node in the tree corresponds to a different solution for the parallel conjunction, and to represent the fact that a parallel conjunction can have multiple solutions we add a branch point (choice point) before the different composition nodes. The C-tree can represent ORP and IAP quite naturally—execution of goals within a single c-node gives rise to IAP while execution of different c-nodes and of untried alternatives which are not below any c-node, gives rise to or-parallelism. The C-tree of a logic program is dynamic in nature, and its shape may depend on the number of processors and how they are scheduled.

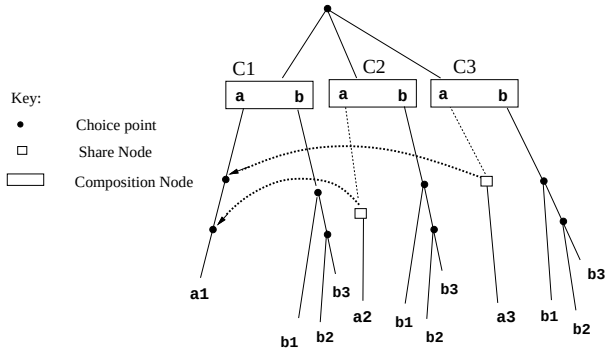

Figure 2. The C-Tree

Notice the topological similarity of the C-tree with the purely ORP tree shown in Figure 3. Essentially, branches that are shared in the purely ORP tree are also shared in the C-tree. This sharing is represented by means of a *share-node*, which has a pointer to the shared branch and a pointer to the composition node where that branch is needed (Figure 2). Due to sharing the subtrees of some IAP goals maybe spread out across different composition nodes. Thus, the subtree of goal `a` is spread out over c-nodes C1, C2 and C3 in the C-tree of Figure 2.

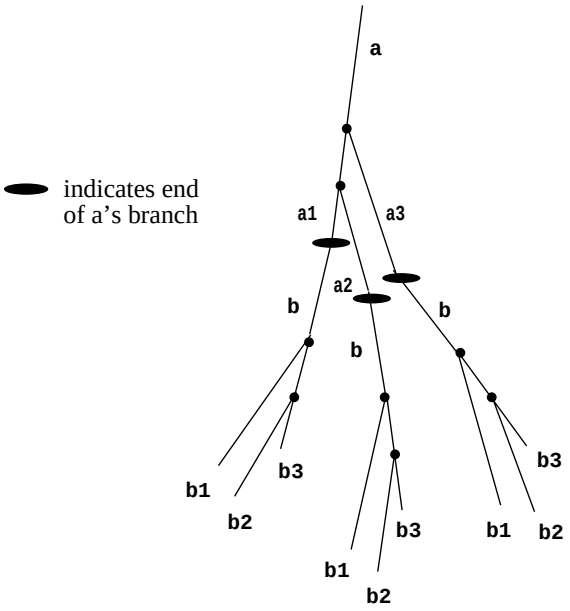

Figure 3. The C-Tree and the Search Tree

Proposed implementations of the C-tree framework include Gupta and Hermenegildo's ACE [7], a model that combines Muse and &-Prolog, and Gupta's PBA model [10], a model that combines Aurora and &-Prolog. One important advantage of these models is that they are quite suitable to the implementation of Full Prolog [11]. Work in the implementation of these models has shown several implementation difficulties, particularly in memory management. Correia and colleagues argue that to address these problems, C-tree based models must be redesigned to minimise interference between IAP and ORP, and proposed a novel data-structure towards this purpose, the Sparse Binding Array [4]. Work on SBA is progressing quickly, although a first And/Or implementation is still not available. The SBA is also the base for a distributed design, DAOS [16], that leverages distributed shared memory techniques to implement the shared stacks.

An alternative approach to combining and-or parallelism through recomputation has been proposed by Shen based in his PhD work in simulating And/Or parallelism [17]. The

FIRE model proposes to use hash tables instead of copying or binding arrays [18], thus avoiding memory management problems. Again, the price will be a significant overhead in one-processor execution.

3. The Need for COW

Copying has become the most popular model for Or parallelism. It has low overheads, and it is the simplest scheme to implement. Unfortunately, initial work in the copying based ACE system showed that applying straight copying incurred difficulties for And/Or systems, namely:

- The major advantage of copying is simplicity. In purely ORP systems it is sufficient to just copy contiguous chunks of stacks. Moreover, incremental copying can be used to reduce the amount of stacks to copy. With incremental copying, one just copies the most recent part of the stacks, instead of always copying the full stacks. With And-parallelism execution is not contiguous and is in fact distributed through several segments. Instead of a straightforward operation, copying now requires chasing through several segments. Hence, there is more to copy, and it is harder to do it.

- A second problem concerns memory management. The major trick with copying is that different workers have their stacks in the same virtual address. With and-parallelism, one has the problem that segments being exploited for one and-parallel goal in an or-branch may conflict with segments for another and-parallel goal in a different or-branch. In the worst case, this requires a conservative solution where different or-branches may not share and-goals at all, making the system more complex and slower.

The relative complexity of implementing ACE has led to losing interest in copying as an implementation technique for And/Or parallelism. Note that initial proposals for Binding Array models suffered from similar problems. Work on supporting binding arrays for And/Or parallelism has proved that, through a substantial redesign, one could obtain efficient solutions to this problem. Our design was motivated by similar considerations.

To design a model that can support copying efficient in And/Or parallel systems, we need to rethink major issues:

- In Muse, Prolog engines always execute in the same virtual addresses. In the presence of and-parallelism, we do not have continuous stacks, so this may not be an important requirement.

- In Muse, sharing simply involves a continuous block of bytes. The nature of and-parallelism means that this is impossible. If we do not want to physically copy, experience with Operating Systems has shown the best alternative technique to be use copy-on-write. That is, the pages created by one worker are to be copied (or duplicated) on demand.

The major insight of a COW based-system is:

- Copying is implemented by a very simple operation. One simply copies the other team's space into our own. This is implemented by the other team first first releasing their write rights for this space. Then, both teams obtain copy-on-write access to the space. At this point, sharing has been implemented. The Operating system next ensures that any further work will be private to each team.

We shall name this copy-on-write based execution scheme *COWL*, or, *Copy-On-Write for Logic programs*. We next discuss the possible implementations in some detail.

4. Making It Simple: The αCOWL

Two schemes of execution of a COW model are possible. In a traditional shared-memory thread-based COW implementation, teams are represented as Operating Systems processes, and workers are threads within teams. Note that workers within the same team therefore automatically share the same address space. We emphasise that a team corresponds to an alternative in the search tree. COWL and-workers exploit and-parallelism by running and-tasks in parallel, and these tasks belong to a single alternative.

Work is created by a worker within a team. Work may be a choice-point, exploitable by other teams, or a new goal, exploitable by a different worker within the same team. In COWL, workers generate work in their team's space. Note that, differently from Muse, COWL cannot maintain contiguous stacks for purely ORP work.

Figure 4. The Muse stacks

Figure 4 shows a typical Muse execution. The stacks drawn in black were copied from a different worker to the current branch's address space. New work then continues from the point where stacks were copied onwards. In Muse, ORP workers work on contiguous stacks.

Figure 5 shows how Muse implements copying between contiguous stacks, all at the same virtual address for every worker. The key is the Muse memory layout. To the right of picture it is shown a large chunk of address space, seen

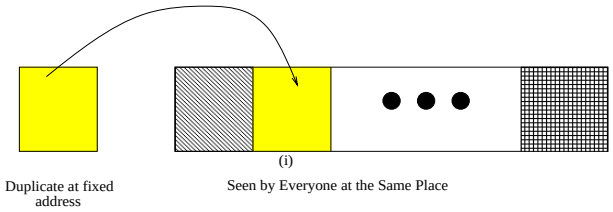

Figure 5. The Muse Stack Layout

at the same addresses by every worker. All copying is performed through this area. To the left of the figure, one can see that every worker (team) maintains an extra mapping of the stack segment. This stack segment is what the worker sees as its work stacks. As we have pointed out, Muse uses a very nice scheme where every workers sees its stacks at two different locations: at one location, its sees the area as Prolog stacks, to operate as sequentially as possible; at a different location, as a chunk of copiable memory.

Note that Muse requires good support from the host Operating System, namely an implementation of `mmap` or similar system call that allows for mapping the same file at two different locations in the same process. Arguably, it is taking the best advantage of Operating System functionality that made Muse efficient.

The αCOWL model proposes a very different way of looking at memory, based on using COW. The idea is simple: if it is complicated to just copy the relevant segments from an IAP computation, why not just *virtually* copy the whole IAP stacks.

This layout suggests a simple implementation of copying. Imagine team T_{i+1} has no more alternatives to exploit, and wants to exploit an alternative from team T_i. Why not make team T_{i+1} just restart from scratch? That is what we propose in αCOWL:

- *copy all the stacks from team T_i, and restart work from scratch.*

This is a very simple scheme, and is guaranteed to be compatible with Prolog. Copying is implemented through three direct system calls: the source team makes its current mapping COW; the destination team releases its own memory mapping, and then maps the source team's map as COW. Note that this "copying" does not involve actual copying of pages, at least until they become necessary.

5. Preserving And-Work: the βCOWL

There is one significant drawback to the αCOWL: when we copy to a team, when have to destroy every other work that was going on. Returning to our simple parallel conjunction `a & b`, whenever we fetch new alternatives for `a` we must kill-off work for `b`. We next research on whether it is possible to do more limited copying. This guarantees that long computations performed by other members of the team will not go to waste, and guarantees ideal execution for situations with fine-grained or-parallelism. The key idea is that we *will copy everything other teams have created, and just that.* We shall call this solution βCOWL.

To have access to work generated by other teams we need a more complex memory layout than what was required for the αCOWL, as described in Figure 6. The key idea is that a team works in its own area (area in (i) in Figure 6). We will refer to this area as the *team's workspace*. Within this area several workers may be processing in parallel, the figure shows three workers expanding stacks. All the other areas were generated by other teams, and have been accessed through copy-on-write.

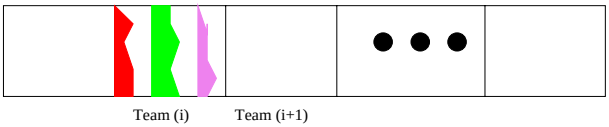

Figure 6. The βCOWL Stack Layout

One first observation results from the fact that in C-tree based models teams can only share completed and-work. As such:

- All and-tasks that have been executed in advance, that is, that do not belong to a shared or-branch, *have been executed in the team's workspace.*

Consider an initiated but not yet completed and-goal g_i. If the goal g_i was executed by the current team, than it must have been executed in the team's workspace. If it was executed by some other team, it has been shared with the current team, and if so it cannot have been executed in advance. If all advance and-work is in the team's workspace, then the only work of in interest in the remaining workspace must be shared-or work corresponding to the current or-branch. We shall name this important result the *localisation principle.*

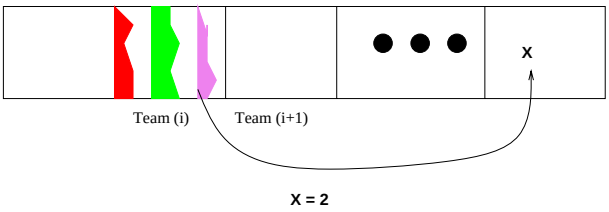

Figure 7. Writing Outside Own's Workspace

Unfortunately, the localisation principle does guarantee that we may not have update pages outside our team's workspace. The problem is presented in Figure 7. The

worker to right wants to attribute the binding `X = 2` to variable `X`, created in a different workspace. To do so, it must change the page where `X` is located. If a different worker in the same team shares external work, copies of the workspace will be remade and the binding may be lost. Note that new data-structures are written in the team's workspace. Updates to external spaces consist only of changing values of variables (or changing pointers), and will always be *registered in the trail*. We shall for simplicity's sake assume for the moment assume that the localisation principle is always followed, and discuss the consequences later.

The Local Workspace We would like to never copy work within the team's workspace. This creates a problem if other teams may be using work originally created by us.

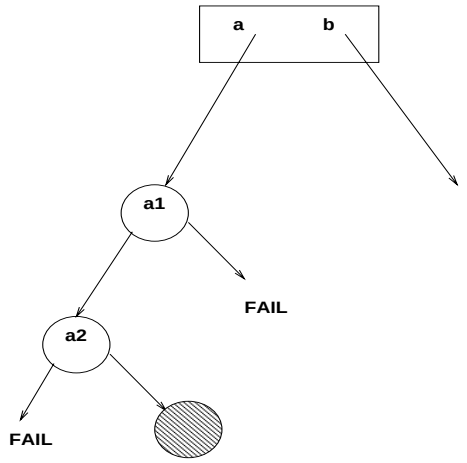

Figure 8. Copying Or Work Under And Work

To understand the problem, consider Figure 8. The figure shows an initial parallel conjunction, `a & b`. Suppose team T_0 takes the parallel conjunction, and that within this team worker W_{00} executes goal `a`, creating choice-points `a1`. Worker W_{00} continues execution, but receives a request for sharing from, say, team T_1. At this point, worker W_{00} gives choice-points `a1` and `a2` for sharing. Team T_1 goes on to work and creates the shaded choice-point.

The complexity of sharing ORP work arises when worker W_{00} backtracks and requests for shared work. Imagine worker W_{00} first looks at choice-point `a2`, and decides not to take work because the available alternative is taken by team T_1. It thus backtracks to the remaining alternative for `a1`, exploits the alternative, and eventually fails. At this point, worker W_{00} can either fail the conjunction, or take work from the shaded alternative, created by team T_1.

At this junction, team T_0 needs to fetch work from team T_1. We would like to do so without fully restarting team T_0. In other words, we need to have in our workspace node `a2`, even though we had previously backtracked to node `a1`. This condition guarantees we can do so safely:

- if a shared branch in the search-tree was created by a worker in a team T_i, the branch must be preserved within the team's workspace, even if does not belong to the branch currently being exploited by team T_i.

In conjunct with the localisation principle described before this condition guarantees that a team can obtain valid sharing with another team by copying all the team workspaces, except its original one:

- The localisation principle guarantees that all the and-work performed in advance is in the team workspace, and will not be affected by the copying operation (as we shall see, this is not entirely correct, and we discuss the consequences next).
- The shared part of the tree that is in other teams' workspace is fully copied.
- The shared part of the tree that is in the own team's workspace has been kept safe.

By induction, we can argue that if all sharings up to the current one have provided the necessary stacks, the next sharing will do so.

Localisation We have noticed that the localisation principle is not fully obeyed in βCOWL, as updates are in fact made to variables in external workspaces. We further commented that these updates must be registered in the trail, because they will always affect external variables.

This suggests a solution to the problem: all bindings for external variables for work performed in advance should be redone after sharing. In fact, this means that whenever we copy work we need to stop every worker in the team, reconsult their trails, and redo bindings to variables outside the team's workspace.

6. Distributing Work: the δCOWL

The βCOWL is particularly interesting as a stepping stone towards a distributed design for COWL, which we shall name δCOWL. This model is based on the notion of teams, that is, of workers sharing the same view of the virtual address space. The difference is that team members are now full, independent, processes, and that they are responsible to keep memory coherent within a team. In other words, whereas memory coherence in the threaded implementations was performed eagerly by the Operating System, in the distributed implementation it will be performed lazily, when needed, by the workers themselves.

Workers in a team become incoherent when a worker, say W_i, backtrack to shared work and virtually copy a different teams workspace. The key idea in δCOWL is that other workers in the team must be performing work in advance, and that as long as we guarantee this work is independent, *it cannot be affected by changes in variables for* W_i. Therefore the other works may gladly be incoherent with W_i. More in detail, supposing W_j is any other worker in the team:

- If W_i backtracked over a parallel conjunction that is an ancestor for W_j's and-task, then the task must be killed and W_j should backtrack. The process is described by Hermenegildo [12].

- if W_i did not backtracks over a parallel conjunction that is an ancestor for W_j's and-task, then we do not want to kill the task. Moreover, because of IAP, the task cannot share variables with any task we have backtracked over (otherwise it could not have been executed in parallel).

  From this we can reach the conclusion that remapping the shared stacks will not affect execution for W_j in this case. The variables accessed for W_j will have the same value before and after the sharing operation. Executing COW will only delay accesses, as it will result in more page fault operations.

This leads directly to a design for δCOWL: workers within a team *will not share the same address space*, but will instead have private copies, that can themselves be maintained through a Distributed Shared Memory mechanism. This will not affect execution, until the workers backtrack over each other. At this point, workers will synchronise by dropping invalid mappings to external workspaces.

One should note that the localisation problem that arises in the βCOWL scheme is not a problem in a distributed system. Whenever a worker within a team shares with some other team, it only updates its own page tables.

7. Discussion

Although COWL and Muse serve two different purposes, it is interesting to compare them. One first observation is that whereas Muse preserves a contiguous stack for each worker (workers are teams in Muse), βCOWL must freeze segments of stacks, and hence generates a fractured or cactus-like stack, in the style of Aurora. This is not a serious drawback of βCOWL for three reasons:

First, in the presence of IAP, the stack could never be contiguous, as the space required for parallel and-goals is hardly known in advance. One can hardly lose what one would never would have had. Second, the essential problem with descontiguous stacks is that copying becomes cumbersome, as one is now jumping over the stack. By using COW, this problem simply disappears: the copying will magically be performed by the operating system, on demand. Third, having a cactus-like stack results in more complex memory management, and thus will slow-down execution over a pure sequential execution. This is in fact true, if only to a small extent. In fact, several Prolog systems have evolved to a fragmented stack, if only for the extra flexibility.

A further argument in favour of a cactus-stack is that it simplifies suspension. Imagine a team needs to perform a side-effect. If the team is not leftmost in the tree, it must suspend. A cactus-like stack makes it possible for the worker that suspended to leave the stack and move to some other piece of work, while the rest of the team remains blissfully unaware.

The first proposal for a recomputation And/Or system based on copying was ACE [7]. ACE has an execution mechanism remarkably similar to αCOWL. The major problem with ACE is that IAP distributes stacks, thus copying is rather complex and overhead prone. αCOWL was specifically designed to make copying simple, by using the COW mechanism.

Note that ACE, as αCOWL, needs to fully restart a team whenever we copy. Again, the reason is that ACE does not guarantee memory independence. The Sparse Binding Array Model, or SBA [4], is a binding array based approach that guarantees both memory independence and localisation by placing all bindings outside the shared area. The SBA thus generates an execution similar to βCOWL. The SBA has one important advantages over COW based models: it supports full localisation, so and-parallelism does not need to be interrupted by sharing operations. There is a single shared space, in contrast to the several sets of workspaces used in COWL. On the other hand, each team must keep a binding array that is a full copy of the stacks. Further, Binding Array based systems do introduce a more significant overhead in a sequential execution.

Traditional distributed And/Or systems use environment closing, which is very overhead prone. DAOS [16] is an alternative approach based on a distributed SBA, and as such is close to δCOWL. Note that one advantage of SBA over βCOWL cannot not hold between DAOS and the δCOWL. Whereas αCOWL must force localisation, by its very nature δCOWL is localised.

8. Conclusions

We have demonstrated how to apply copying to And/Or parallelism. The idea is to use copy-on-write to replace the problem of what the copy by the, much simpler, reverse problem of what not to copy. We propose a direct application of this principle, the αCOWL, that may be of inter-

est for a simple, efficient implementation in shared-memory systems. The βCOWL provides a more flexible interface between And and Or-parallelism, but incurs extra complexity. It is of interest as a solution for shared memory systems, and as a stepping stone for the δCOWL, a generalisation of our method that addresses the problem of copying based And/Or execution in distributed systems.

Applying copying has been an elusive goal for the designers of And/Or parallel systems. We believe that our proposals preserve the two major advantages of copy: low overheads, and implementation simplicity. In the near future, we expect to implement this technique as a low overheads alternative for the SBA in shared-memory systems. We are also collaborating in the implementation of the distributed And/Or parallel systems, and we believe this technique is an useful alternative to DAOS.

One major advantage of copying is that it is quite opaque to what is being copied, more so than binding arrays. This simplifies parallelising extensions to logic programs. In our case, we are interested in the parallel execution of tabulated goals and in parallelising constraint based programming. These are fields with practical applications with have quite large execution times and which, we believe, can greatly benefit from both IAP and ORP.

Acknowledgments The author would like to acknowledge and thank the contribution and support from Eduardo Correia and Fernando Silva. The work has also benefitted from discussions with Luís Fernando Castro, Inês de Castro Dutra, Kish Shen, Gopal Gupta, and Enrico Pontelli. This work has been partly supported by Fundação da Ciência e Tecnologia and JNICT under the Melodia project (JNICT contract PBIC/C/TIT/2495/95).

References

[1] Khayri A. M. Ali and Roland Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *Proceedings of the North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[2] J. Briat, M. Favre, C. Geyer, and J. Chassin. Scheduling of or-parallel Prolog on a scaleable, reconfigurable, distributed-memory multiprocessor. In *Proceedings of Parallel Architecture and Languages Europe*. Springer Verlag, 1991.

[3] John S. Conery. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, Norwell, Ma 02061, 1987.

[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] 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.

[6] G. Gupta and B. Jayaraman. Analysis of or-parallel execution models. *ACM TOPLAS*, 15(4):659–680, 1993.

[7] Gopal Gupta, M. Hermenegildo, E. Pontelli, and Vítor Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP'94*, pages 93–109. MIT Press, 1994.

[8] Gopal Gupta, M. Hermenegildo, and Vítor Santos Costa. And-Or Parallel Prolog: A Recomputation based Approach. *New Generation Computing*, 11(3,4):770–782, 1993.

[9] Gopal Gupta and Bharat Jayaraman. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *Proceedings of the North American Conference on Logic Programming*, pages 332–349. MIT Press, October 1989.

[10] Gopal Gupta and Vítor Santos Costa. And-Or Parallelism in Full Prolog with Paged Binding Arrays. In *LNCS 605, PARLE'92 Parallel Architectures and Languages Europe*, pages 617–632. Springer-Verlag, June 1992.

[11] 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.

[12] 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.

[13] 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.

[14] Ewing Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, David H. D. Warren, A. Calderwood, P. Szeredi, Seif Haridi, P. Brand, M. Carlsson, A. Ciepelewski, and B. Hausman. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[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] Luís Fernando Castro, Cláudio Geyer, Manuel E. Correia, Ví tor Santos Costa, and Fernando Silva. Daos — distributed and-or in scalable systems. Technical note, LIACC, Universidade do Porto, 1998.

[17] Kish Shen. *Studies of AND/OR Parallelism in Prolog*. PhD thesis, University of Cambridge, 1992.

[18] Kish Shen. A New Implementation Scheme for Combining And/Or Parallelism. In *1997 Post ILPS Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming*, 1997.

[19] 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.