Issues in Implementing ACE: A Stack Copying Based And-Or Parallel System

Enrico Pontelli and Gopal Gupta
{epontell,gupta}@nmsu.edu

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA

Manuel V. Hermenegildo
herme@fi.upm.es

Facultad de Informatica
Universidad Politecnica de Madrid
28660-Boadilla del Monte, Madrid, SPAIN

Abstract. We discuss several issues involved in the implementation of ACE, a model capable of exploiting both And-parallelism and Or-parallelism in Prolog in a unified framework. The Or-parallel model that ACE employs is based on the idea of stack-copying developed for Muse, while the model of independent And-parallelism is based on the distributed stack approach of &-Prolog. We discuss the organization of the workers, a number of sharing assumtions, techniques for work load detection, and issues relaed to which parts need to be copied when a flexible and-scheduling strategy is used.

Keywords: Independent and-parallelism, or-parallelism, stack copying, implementation issues.

1. ACE

The ACE (And-Or/Parallel Copying-based Execution) model [4] uses stack-copying [1] and recomputation [3] to efficiently support combined Or- and Independent And-parallel execution of logic programs. ACE represents an efficient combination of Or- and independent And-parallelism in the sense that penalties for supporting either form of parallelism are paid only when that form of parallelism is actually exploited. Thus, in the presence of only Or-parallelism, execution in ACE is *exactly* as in the Muse [2] system—a stack-copying based purely Or-parallel system. In the presence of only independent And-parallelism, execution is *exactly* like the &-Prolog [7] system—a recomputation based purely And-parallel system. This efficiency in execution is accomplished by introducing the concept of *teams of processors* and extending the stack-copying techniques of Muse to deal with this new organization of processors. This paper, after giving a brief overview of ACE, concentrates on discussing the several implementation issues that arise in combining or-parallelism and independent and-parallelism using this model.

2. Independent And-parallelism

ACE uses the model of independent and-parallelism adopted in &-Prolog which includes several ways of expressing parallelism including a parallel conjunction operator ("&"), wait primitives, and Conditional Graph Expressions (CGEs). Such parallelism can be automatically uncovered by a parallelizing compiler [?] or annotated by the user [7]. For simplicity, in this discussion we will assume that, in particular, CGEs are being used to express and-parallel execution of a set of literals. CGEs are annotations of the form:

$$\cdots, (< conditions > \Rightarrow literal_1 \& \cdots \& literal_n), \cdots$$

where <*conditions*> is used to check the independence of the literals in the CGE. Whenever this condition is satisfied the literals belonging to the CGE can be executed in and-parallel.

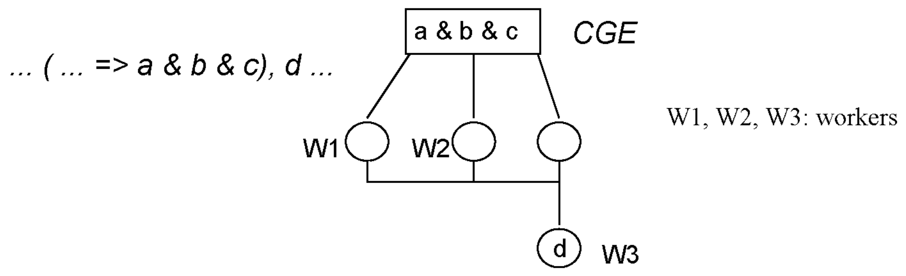

Fig. 1. Execution of a CGE

In implementing ACE techniques used for implementing &-Prolog can be adopted without much change [4]. In fact, ACE adopts &-Prolog's execution model, as implemented in the P-WAM abstract machine [7, 11], itself an evolution of the original RAP-WAM [6]. One of the characteristics of this execution model is that backtracking into a CGE is forced to strictly follow the Prolog semantics. The literals are scanned from right to left in search of a new solution and once this has been found all the literals on the right are recomputed in parallel.

However, we have made some modifications to the P-WAM model that facilitate incorporation of or-parallelism. While in P-WAM the continuation of a CGE is assigned to the worker who finishes the last goal in the conjunction, in ACE we assign such continuation to the worker executing the rightmost sub-goal of the CGE.

This approach leads to the following advantages:

a. It always avoids the use of wait-markers on the choice point stack - necessary sometimes in the P-WAM to separate the CGE subgoals from the continuation subgoals of the CGE for a proper backtracking order;
b. it makes backtracking from outside into the CGE always immediate, without requiring any further synchronization activity.

It should be noted that this approach has the disadvantage of sometimes leaving the stack set associated with the rightmost goal blocked waiting for other goals to complete (the agent or process itself can migrate to a different stack set, however).

As in the P-WAM the backtracking process is not controlled by a single worker (the same which started the CGE), but is distributed among the workers executing the CGE. The same worker which fails takes care of killing the rest of the CGE (inside failure) or of propagating the backtracking to the subgoal on its left (outside backtracking). Also adopted from the P-WAM is that the And-scheduler does not put any restriction on the subgoals that can be taken from another worker; this is different from the original proposal of RAP-WAM [6], in which only subgoals satisfying a specified ordering relation are considered, in order to avoid backtracking on trapped subgoals. Thus in P-WAM and ACE the trapped subgoal problem has been tackled by keeping the physical and logical structure of the choice point stack distinct (i.e. a logical contiguous part of the stack may be actually spread in different parts of the physical stack). This approach may cause holes to appear in the stack. A hole will be reclaimed when everything above it has been reclaimed. Thus, though this approach is more expensive in space, it is relatively simple to implement. It has been adopted in other systems (e.g. Aurora [8], where the holes are called *ghost nodes*).

3. Or-parallelism

The Or-Parallel model that ACE employs is based on the idea of stack copying, developed for the Muse system [1]. The different workers involved in the computation operate using an identical logical address space (i.e. each worker allocates its stacks at the same logical addresses); this allows ACE to reduce the job

sharing operations (stealing of or-parallel work) to a simple copying of contiguous chunks of memory from one worker to the other (without any form of pointer relocation). In this way the multiple environments that are needed for or-parallelism are maintained by keeping separate copies of the execution stacks.
In the stack copying approach, whenever a worker runs out of work it tries to detect another worker which has unexplored alternatives and copies its complete choice point stack, reactivating the desired alternative by backtracking.

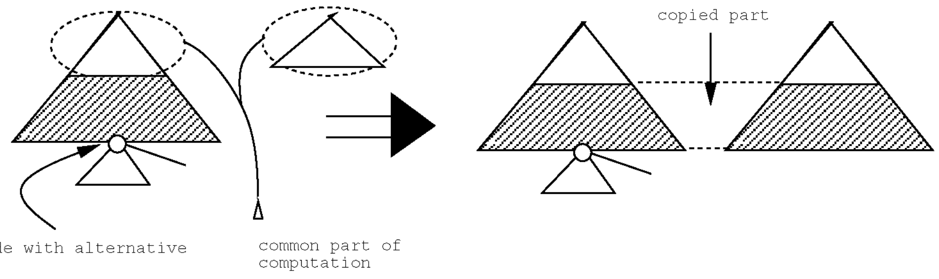

Fig. 2. Stack-copying during Or-parallel execution

A choice point from which different workers may take alternatives is said to be *shared.* For each shared choice point a data structure, called shared frame, is built. This is actually the only real data structure shared among different workers. For the rest each worker has its own private execution environment and, except for the sharing operations, it acts like a standard sequential engine.

4. ACE

4.1 Organization of the Workers

Conflicting requirements are imposed by the two models adopted respectively for or- and and-parallelism, in terms of the visibility of structures: Or-parallelism requires a strict separation among the execution environments of different workers, while and-parallelism requires complete visibility of the execution environment of each worker.
This situation is solved by organizing the workers (individual WAM-like engines) into teams. Each team, which is created by an initial worker (called *master* of the team) contains a certain number of workers which execute a certain branch of the computation tree in and-parallel. The different teams are assigned to execute different branches of the computation tree in or-parallel.
This organization of workers into teams requires an appropriate configuration of the memory, as illustrated in Figure 3.

It is important to notice that each team sees its own logical address space starting from address 000 and going to address xyz; this approach allows the possibility of performing copying among teams without any address relocation.

4.2 Sharing Assumptions

ACE tries to stay as close as possible to Prolog semantics. Thus, it recomputes independent goals in a CGE, rather than reusing their solutions. Recomputation of independent goals not only simplifies the

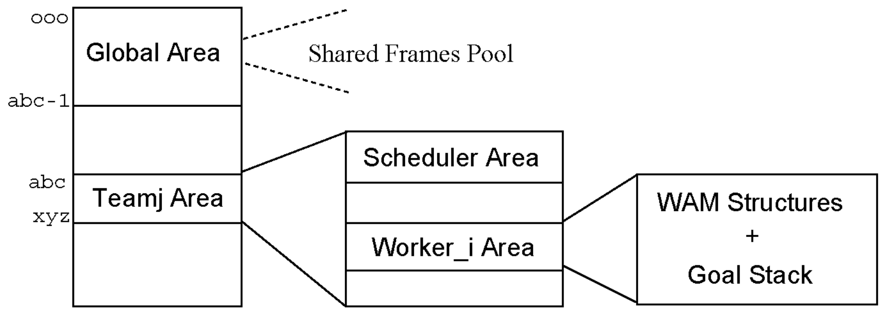

Fig. 3. Memory Organization

implementation, it also allows the inclusion of side-effects and extra logical predicates in a natural way [3, 5].

This choice of recomputing goals has an influence in the Or-scheduling phase, both for

1. detecting when an alternative is to be made visible for sharing;
2. detecting which parts of the computation tree should be copied during the sharing.

The basic sharing assumption that we adopted can be described by the following rule: an untried alternative may be stolen from a choice point C iff

a. C is not in the scope of any CGE;
or
b. if C is below the nested CGEs E_1, ..., E_n and s_j indicates the subgoal of the CGE E_j through which the branch leading to C is passing, then
∀i: $1 \leq$ i $\leq$ n, ∀j: j $\leq s_i$, branch(E_i, j) is completed

The main idea behind this concept is to disallow sharing of an alternative with another team as long as the computation on the left in the tree is not terminated.

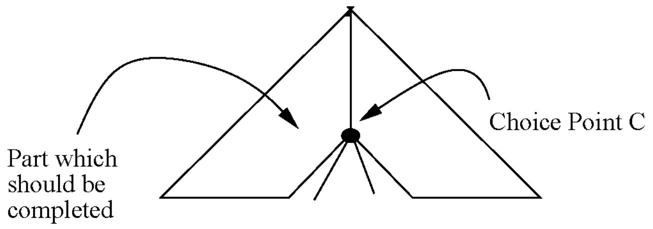

Fig. 4. Sharing Assumptions on a computation tree

Together with simplifying the implementation of side-effects and extra-logical predicates, the sharing assumption

- simplifies identification of stack parts to be copied;
- simplifies the reactivation of execution in the team which is picking up work;
- furthermore, allowing a copy when the subgoals on the left have not terminated yet may lead to situations in which the cost of the sharing is not balanced by a satisfactory level of parallel execution.

The example in Figure 5 is perhaps an extreme case but illustrates the idea. Sharing of the choice point shown would lead to an expensive copying of the data corresponding to the dark shaded part. If this huge computation fails, the whole computation will fail and the stack-copy operation will turn out to be a useless overhead. Essentially, one should avoid picking work from goals to the right since the more to the right a goal is in a CGE the more speculative it is going to be (unless all goals to its left in the CGE have finished or it can be somehow determined that they will not fail).

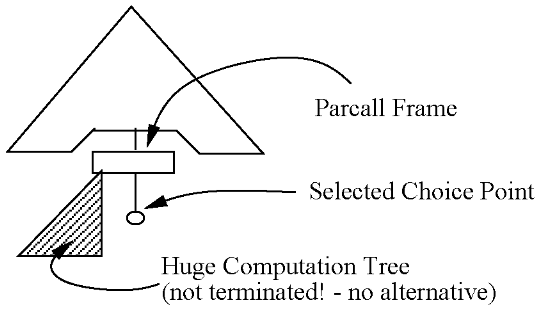

Fig. 5. Computation Tree with wide failing subtree

4.3 Work Load Detection

Inside a team each worker keeps track of the work that it can make available to others for and-parallel execution. This information can be automatically obtained from the current value of the pointer to the goal stack, in which and-parallel work (pointers to untried subgoals of the CGEs) is recorded. More complex is keeping track of the global amount of (or-parallel) work available for sharing that is present inside a team.

The available or-parallel work of a team is given by the number of unexplored alternatives located in the private part (i.e. the part of the tree which has not been made public or made available for sharing) of the tree computed by the team, counting only the choice points satisfying the sharing assumptions. This is similar to the idea of keeping track of *private richness* in MUSE, where richness indicates the amount of Or-parallel work available (in terms of number of alternatives currently untried) at any given moment in the private part of the tree.

In order to support this approach for detecting or-parallel work, some fields have been added to the CGE's descriptor (the PARCALL frame of [6]). Each subgoal of a CGE has an associated field in which the amount of work currently existing in the subtree rooted at such subgoal is accumulated, and a mark which specifies if the computation on the left has already been completed. Whenever a new choice point is created and it falls in the scope of a CGE, the new alternatives are recorded in the descriptor of the subgoal and, if the mark is set, the richness of the subtree is added to the current richness of the CGE. Whenever a subgoal of a CGE completes execution, we check if any goals to the right have finished, and if so, their richness is added to the richness of the CGE, and so on.

4.4 Memory Copying

In the usual Or-parallel systems based on stack copying, detecting the part of stack to be copied is quite straightforward, since the private part of the computation tree is completely stored between the current top of stack and the top of stack at the time of the previous sharing operation.

When and-parallelism is introduced, the situation becomes more complex, since the private part of the tree is spread over different processors. Furthermore, the fact that no restrictions are imposed on the selection of subgoals at the and-scheduler level may lead to arbitrary interleaving of private and shared parts of the tree on the worker's stacks.

A consequence of this is that, since irrelevant parts of the tree may be intermixed with relevant parts (w.r.t. the sharing operation) on a worker's stack, during the copying operation some of these irrelevant parts may be copied. The irrelevant parts are either holes in the stack, or parts that need to be backtracked over by the copying processor before it can continue execution with an untried alternative.
The sharing operation involves three successive steps:

1. the stealing team (called *sharer* team) detects the best[1] team with available work (called *sharee* team) and figures out the parts of sharee's stacks to be copied. This is done by first detecting the choice point at which the last sharing between the two teams took place.
2. the actual copy operation is performed. The relevant parts detected in step 1 are transferred and the copying is performed in parallel among different workers of the two teams. Conditional bindings are removed by backtracking or by resetting variables in the copied trail, depending on the copying policy adopted (as explained later).
3. the stealing team reactivates the computation. This is realized by a mechanism that is already in place in P-WAM / RAP-WAM. If the choice point from which we are taking a new alternative is below the scope of a CGE, then as soon as a solution for the current subgoal is determined all the subgoals on the right in the CGE are reactivated in And-parallel.

The main problems occur in the first stage. Clearly stack frames that existed in the sharee at the time this last sharing was done also exist in the sharer and need not be copied. The sharer first backtracks to this choice point where sharing took place last, and then it finds out how much to copy from the sharee's stacks. Here, there are two possibilities: either the sharer copies everything from the sharee's stacks above the last shared choice-point (including those parts that are after the choicepoint from which an untried alternative will be subsequently picked up by the sharer) or it copies only those parts of the stacks that are between the last shared choicepoint and the choicepoint from where the sharer will finally pick work.

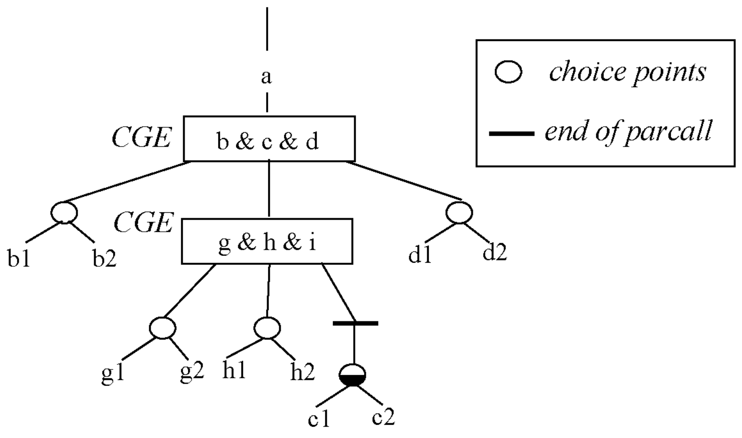

Fig. 6. Computation Tree with CGEs and Or-nodes

Consider the tree generated by ACE shown in Figure 6. If, during a sharing operation, the node g is assumed as the starting point for copying, i.e. it is the last shared choicepoint, then in the first approach

[1] The criteria for choosing the best team are a straightforward extension of those used in Muse.

stack frames corresponding to h, i, c and d will be copied (even though the solution for these goals may not have been reached yet). The advantages/disadvantages of this approach are the following:

- it is very simple and does not require a previous traversal of the computation tree (in sharee's stacks);
- the time for which the two teams need to synchronize will probably be minimum (recall that when a team is sharing work with another, the sharee cannot do any useful execution);
- a successive traversal of the computation tree should be made by the sharee team in order to detect the choice point (satisfying the sharing assumption) from which an alternative can be taken;
- the copying is blind and may lead to the transfer of large amounts of information which will be immediately marked as rejected (i.e. will be immediately backtracked over and reclaimed).

In the second approach, the branches to the right of g (h, i, c and d) should be traversed and the position of such computations in the workers' stacks determined. Only those parts of the stacks of the sharee will be copied that correspond to state of the sharee's stacks when the choicepoint up to which we want to do the new sharing operation came into existence. Thus, in this approach the sharer needs to know in advance the choice point from where it will be picking the alternative to compute the limits of the stacks in the sharee that should be copied. In this way only the part of the computation tree between the starting point and the chosen choice point is copied. Note that the trail stack of branches h, i, c and d should also have to be copied, in order to remove conditional bindings in the copied parts (in the first approach this implicitly happens during backtracking). The advantages/disadvantages of this approach are perfectly symmetrical to the ones described above.

In the first approach, where the difference between the sharee and sharer's stacks is blindly copied, we have to grapple with the following question. Suppose the branch i is not complete but branch d is. The question is whether also the choice points in d's branch should be made public or not. Because the choice points that will be produced later by the sharee in i will be, of course, private, which will result in arbitrary mixing of public and private parts in the logical structure of the tree (they are already mixed in the physical stacks, as pointed out earlier). ACE does not allow the mixing of public and private parts in the logical tree. Indeed, this is what is enforced by the sharing assumption.

In order to obtain a reasonable trade-off between the two described policies, the following heuristic has been introduced: if the sharee is currently executing below the scope of any CGE, then the second policy is used, otherwise the first policy is adopted. The key observation is that whenever a team is outside the scope of the CGEs, all the branches of the parcalls are completed and the sharing assumption are obviously satisfied. This makes less probable the risk of copying large sections of the computation tree to be subsequently discarded because of non-terminated branches in the middle. This approach constitutes the most natural extension of the **incremental copying** adopted in the stack copying implementations of Or-parallelism. If however the team is inside a CGE, with unfinished goals in this CGE, then the blind copying of the difference may copy too much irrelevant stuff to be immediately backtracked over.

5. Conclusions

In this paper, we have discussed several issues involved in the implementation of ACE, a model capable of exploiting both And-parallelism and Or-parallelism in Prolog in a unified framework. The Or-parallel model that ACE employs is based on the idea of stack-copying developed for Muse, while the model of independent And-parallelism is based on the distributed stack approach of &-Prolog. We have discussed a number of problems that arise in the implementation of ACE when in combining independent and-parallelism *a la* &-Prolog with the stack-copying approach of Muse, along with our solutions to them.

ACE is currently being implemented on a 20 Processor Sequent Symmetry in a collaboration between New Mexico State University and the Technical University of Madrid (UPM). Starting from the MUSE engine implementation we are adding the machinery to support independent and-parallelism and the mechanisms described in this paper, to obtain an implementation for ACE. Since the automatic parallelization task is almost identical in ACE and in &-Prolog, the ACE compiler incorporates large parts of the &-Prolog compiler, which is itself also evolving.

Acknowledgments

We would like to thank Vitor Santos Costa and Kish Shen for many discussions. The ideas presented in this paper are based on earlier papers that introduced the ACE Model [4]. Research reported here is supported by NSF (Grant CCR9211732), Sandia National Labs (Grant AE-1680), Oak Ridge Associated Universities (Grant CK004448), ESPRIT Project 6707 PARFORCE, CICYT proyect number TIC93-0976-CE, and NATO (Grant CRG921318).

References

1. K. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *Proceedings of the 1990 North American Conference on Logic Programming*, MIT Press.
2. K. Ali and R. Karlsson. Full Prolog and Scheduling Or-parallelism in Muse. In *International Journal of Parallel Programming*, 1991, Vol. 19, No. 6, pp. 445-475.
3. G. Gupta and M.V. Hermenegildo. Recomputation Based And-Or Parallel Execution of Logic Programs. In FGCS '92, ICOT.
4. G. Gupta, M.V. Hermenegildo, E. Pontelli, and V. Santos Costa. The ACE Model: And/Or-Parallel Copying-based Execution of full Prolog. Technical Report NMSU-TR-92-CS-13, New Mexico State University.
5. G. Gupta and V. Santos Costa. And-Or Parallel Execution of full Prolog based on Paged Binding Arrays. In *Proceedings of the 1992 Conference on Parallel Languages and Architectures Europe (PARLE '92)*, Springer Verlag, LNCS 605, June 1992.
6. M.V. Hermenegildo. An Abstract Machine for Restricted And Parallel Execution of Logic Programs. In *1986 International Conference on Logic Programming*, London, LNCS, Springer-Verlag.
7. M.V. Hermenegildo and K.J. Greene. &-Prolog and its performance: Exploiting Independent And- Parallelism. In *Proceedings of the 1990 International Conference on Logic Programming*, MIT Press, pp. 253-268.
8. E.Lusk, D.H.D.Warren, S.Haridi et. al. The Aurora Or-Prolog System. In New Generation Computing, Vol. 7, No. 2,3, 1990, pp. 243-273.
9. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.
10. Pontelli E. and G. Gupta. ACE: Or + Independent And Parallelism. Internal Report, New Mexico State University.
11. Shen K. and Hermenegildo M. Scheduling and Memory Management in And-Parallel Execution of Logic Programs Revisited. Internal Report, Technical University of Madrid.