The SBA: Exploiting Orthogonality in AND-OR Parallel Systems

Manuel E. Correia   Fernando Silva   Vítor Santos Costa

Technical Report Series: DCC-97-3

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 Porto, Portugal

Tel: +351+2+6001672 – Fax: +351+2+6003654

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

The SBA Exploiting Orthogonality in ANDOR Parallel Systems[1]

Manuel E. Correia, Fernando M. A. Silva, V. Santos Costa
Laboratório de Inteligência Artificial e Ciência de Computadores,
Universidade do Porto
R. do Campo Alegre 823, 4150 Porto, Portugal
{mcc,fds,vsc}@ncc.up.pt

Abstract

One of the advantages of logic programming is the fact that one can exploit *implicit* parallelism in logic programs, such as and-parallelism and or-parallelism. Recently, research has been concentrated on integrating the different forms of parallelism into a single combined system. In this work we concentrate on the problem of integrating or-parallelism and independent and-parallelism for parallel Prolog systems. We contend that previous data structures require *pure recomputation* and therefore do not allow for orthogonality between and-parallelism and or-parallelism. In contrast, we submit that a simpler solution, the sparse binding array, does guarantee this goal, and explain in detail how independent and-parallelism and or-parallelism can thus be efficiently combined.

1 Introduction

One of the advantages of logic programming is the fact that one can exploit *implicit* parallelism in logic programs. The two major forms of implicit parallelism are and-parallelism and or-parallelism. *Or-parallelism* has been exploited successfully in systems such as Aurora [18] and Muse [1]. *And-parallelism* can be classified as independent, as exploited in &-Prolog [15], &-ACE [11], and *dependent*, as exploited in the committed-choice languages [22], in Andorra-I [21] or the DDAS [23]. More recently, research has been concentrated on integrating the different forms of parallelism into a single combined system. In this work we concentrate on the problem of integrating or and independent and-parallelism.

Or-parallelism is arguably one of the most successful forms of parallelism available. It can obtain very good speedups for a large range of applications, such as applications that require search. It is therefore one of the main components to include in combined systems. Designers of Or-parallel systems must address two main problems, scheduling and variable binding representation. As we shall see later, the scheduling problem can be addressed rather orthogonally to the other forms of parallelism. In contrast, binding representation affects all forms of parallelism.

The binding representation problem arises because the same variable may have several different bindings in different or-branches. A number of approaches [12] have been presented to tackle the problem. Two successful ones are environment copying, as used in Muse [1], and binding arrays, as used in Aurora [18]. In the copying approach, each worker maintains its own copy of the path in the search tree it is exploring. Whenever work needs to be shared, the worker that is moving down the tree copies the stacks from the worker that is giving the work. In this approach, data sharing between workers only happens through an auxiliary data structure associated with choice-points.

In contrast, in the binding array approach work stacks are shared. To obtain efficient access, each worker maintains a private data structure, the binding array, where it stores its conditional bindings. To allow for quick access to the binding of a variable the binding array is implemented as an array,

[1]accepted for the International Logic Programming Symposium (ILPS), Port Jefferson, Long Island N.Y., October, 1997.

indexed by the number of variables that have been created in the current branch. The same number is also stored in the variable itself, thus giving constant-time access to private variable bindings.

Independent and-parallel systems use the same binding representation scheme as sequential Prolog, but require extensive changes to memory allocation, so that several workers can operate in and-parallel. They further require support for maintaining a pool of available work, for detecting successful completion of an and-parallel conjunction, and for propagating failure.

To fully exploit and-or parallelism, alternatives created by workers working in and-parallel must be available to or-parallel execution. This introduces problems to the environment representation schemes developed for pure or-parallelism. In the case of copying, the main problem is now what to copy. Whereas in systems that exploit pure or-parallelism, execution for each branch proceeds in a contiguous stack, support for and-parallelism requires distributing execution of goals through different data areas. The ACE system [11] proposed the first copying-based solution for this problem, based on the C-tree model [10]. ACE must tackle a few difficult problems. The major one is that at copy-time ACE must find out which data areas to copy, making copying more complex than, say, for Muse. It is also likely that more copying than for pure or-parallel systems will be necessary. Moreover, and as we shall see later, ACE requires pure recomputation, that is, it prevents any reuse of independent and-parallel goals between or-parallel branches.

The binding array cannot be directly used for combined systems. The problem is that the same binding array has to be shared between computations that execute in and-parallel, creating a complex memory allocation problem. The first solution to this problem was the paged binding array PBA [13]. The PBA uses operating system technology to provide an elegant solution to the problem, at the cost of extra complexity in dereferencing bindings and in managing the BA. As we shall see later, the PBA also requires pure recomputation.

In this paper we describe how these schemes fair against simultaneous exploitation of OR-IAP Parallelism. The keystone of our work is the principle of *orthogonality*: or-parallel execution should be unaware of independent and-parallel execution, and independent and-parallel execution should be able to ignore the very existence of or-parallelism in the system. Orthogonality is required for simpler solutions, with the potential for easier implementation and lower overheads. We contend that *pure recomputation* does not allow for orthogonality. In contrast, we submit that a simpler solution, the sparse binding array, does guarantee this goal, and explain how independent and-parallelism and or-parallelism can thus be efficiently combined.

The paper is organized as follows. We first describe in more detail the binding array based schemes. Next, we concentrate on the orthogonality problem and the need for memory independence between or-parallel branches. We then describe how the data structures and algorithms for independent and-parallelism must be changed to support or-parallelism, and vice-versa. We last present the conclusions and future work.

2 From Or-Parallelism to And-Or Parallelism

We first define some terminology. Computations are executed by processing agents, that we will call *workers*, usually implemented as processes or threads running on separate processors.

Two major approaches have been proposed for combining independent and-parallelism with or-parallelism. In *reuse-based* models, the computation is seen as an and-or tree. Solutions for each goal in a conjunction are stored and then combined with the solutions from the other goals through a special operation, the cross-product calculation. This solution is very attractive because it can reduce the search space of logic programs, and was the basis for original proposals such as Conery's AND/OR Process model [4] and Kalé's Reduce Or Process model [17].

An alternative to this approach are the *recomputation* based models where and-goals may be re-computed in quite the same way as Prolog would. One major example of such models is the C-tree model [10]. In this model, groups of workers, or *teams*, work together in and-parallel to explore branches from the search tree. Workers within the team execute and-parallel tasks in advance, as they would in pure and-parallel systems, such as &-Prolog. Note that as a result, different teams may be computing the same and-parallel goal in advance, and all generate the same solutions to this goal. Thus the model recomputes goals in much the same way as Prolog or &-Prolog would. Recomputation-based models have important advantages. First, their semantics are similar to Prolog's, thus enabling the support of side-effects in the presence of parallelism. Second, they avoid the overheads associated with implementing cross-product nodes and keeping the corresponding solutions. Often, the original programmers have written their programs in such a way as to avoid recomputing solutions, resulting in little gains for reuse. Also, recomputation allows producer-consumer dependent and-parallelism [23].

The last major advantage of recomputation based approaches is that they *simplify* extending or-parallel systems to incorporate and-parallelism. The two main proposals for doing so are copying, used in ACE, and binding arrays, that we describe in detail next.

2.1 Binding arrays for or-parallelism

The SRI model [25] was the first model to propose using binding arrays for or-parallelism. In this model, the stacks are shared by all the or-agents and form a search tree whose branch components are dynamically distributed between the stacks of all the workers participating in the computation. In this way, cactus stacks are formed for each of the heap, environment stack, choice-point stack and trail stack.

Each worker keeps a private set of bindings for shared variables. In the SRI model these bindings are stored in an array, the *binding array*. We say that a worker's binding array *shadows* the shared variables. The key idea in the binding array concept is that for each variable in the stacks there is a single, uniquely identified, binding array slot that is the same for any or-agent. This allows for fast access to a variable local binding. Figure 1 presents the concept. In this case, variable `X` receives two different bindings in the two different branches exploited by worker W_1 and worker W_2. The location of the variable is shared, so the bindings must be stored in a private location, in this case each worker's BA.

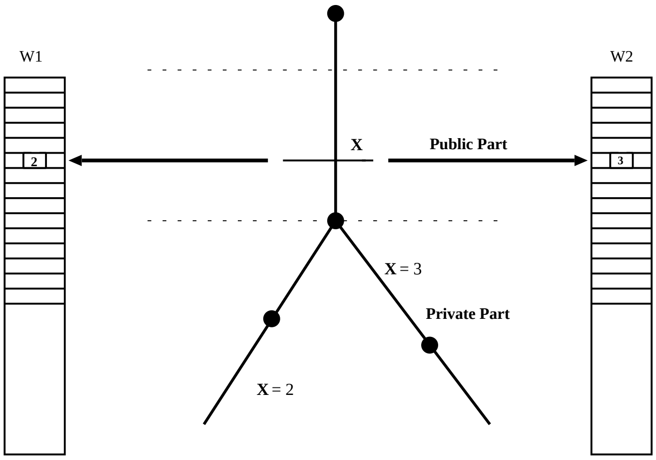

Figure 1: Binding Arrays in or-parallelism.

The SRI model implements the BA by "binding" each variable X to the number of variables N between X and the root of the computation tree. To find out the private binding of the variable X at $SHARED\_PTR$, it is sufficient to search for the following value:

$$PRIVATE\_VALUE = BA\_BASE[*(SHARED\_PTR)]$$

where $PRIVATE\_VALUE$ represents the value we are looking for, BA_BASE represents the origin of the corresponding binding array, and $SHARED\_PTR$ represents the position of the shared variable, with $*(SHARED\_PTR)$ giving the value X. This is a constant time operation, giving fast access. As an added benefit, the fact that $*(SHARED\_PTR)$ has the variable's age, gives a fast way to compare ages of variables, necessary to implement trailing and to guarantee that newer variables always bind to older variables.

2.2 Binding Arrays for And-Or Parallelism

The key idea of recomputation-based systems is that a set of workers, a *team*, works in *and-parallel* to exploit a branch of the search tree. The team thus becomes the or-agent that computes a given branch of the search tree. Workers in a team will share the same bindings for the variables of the branch, and must thus share the same binding array. Or-parallelism is exploited by having these teams running in or-parallel, as in Andorra-I [21], or as in the C-tree model [10].

The question now arises of how to manage a binding array in the presence of several workers. The main problem is that workers may want to place their variables in the same binding array positions. Ideally, we would like to partition the binding array in correctly sized chunks and allocate them to the different independent and-tasks that are executed concurrently during the computation of the current or-branch. Unfortunately, it is impossible to know beforehand how many variables are going to be created in a IAP computation.

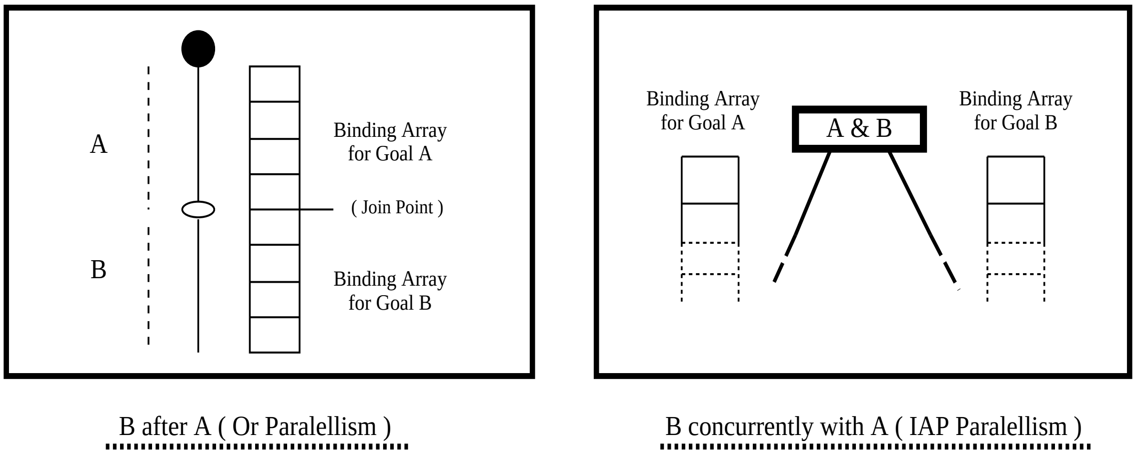

Figure 2: Using the binding array in and-parallelism.

The problem is illustrated in Figure 2. On the left we see the situation for which the binding array has been developed, sequential execution within an or-branch. In this case the index on the binding array for the first variable of goal `B` is known because goal `A` was completed before goal `B` is called. In the case on the right, `A` executes in IAP-parallel with `B`. Since it is not possible to known beforehand how many variables goal `A` is going to create, there is no efficient and easy way to guess an index in the binding array for the first variable created during the computation of goal `B`.

We could simply lock the entire access to the binding array each time a new variable is created, but this defeats the idea of having independent and-computations going on in parallel inside the team at maximum speed. It would also make reusing BA entries very difficult if a worker backtracked within a team. A similar problem arises in operating systems when physical memory has to be shared between different processes. The solution is to paginate the memory and have a memory page table for each process that translates the virtual address the process tries to access to a real memory address where the page containing the data to be accessed is located. This way several processes access the same physical memory while maintaining private address spaces. The same can be done with the binding array of a team. This was the main inspiration for paged binding arrays.

In the PBA each shared variable is divided into two fields, *PG_NUMBER* and *OFFSET*. The position in the BA is determined by obtaining the page offset from the page table, *PT*, and adding the variable offset within the page.

PBA structures thus uses indirection to tackle the binding array management problem. The BA is paginated, with page size being implementation defined. A page table manages pages with synchronized access among the elements of the team. Each time an and-worker creates a variable and there is no more space in its current page, or is just starting execution, a new page of size K is allocated from the binding array, a new page entry J is reserved from the page table and a trail entry is pushed indicating the allocation of this page slot. The first two operations must be atomic within the team. The value of the variable now becomes a pair <P,I> where P is the index in the Page Table where the beginning address of the page is kept and I is an offset within that page where the teams value for the variable is kept. Private values are obtained as follows:

$$PRIVATE\_VALUE = BA\_BASE[PAGE\_TABLE[P] + I]$$

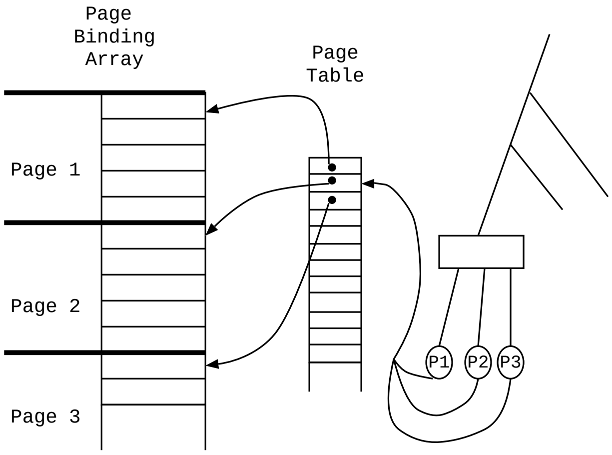

Figure 3: The Paged Binding Array.

PBA management is illustrated in Figure 3. Note that each time a new page is allocated a special marker in the and-worker local trail must be entered, indicating which entry in the page table has been allocated. This allows for page table entries to be reused in backtracking, and for the correct page table entries to be allocated and initialized when another team installs or-work from this same branch.

To preserve variable age in this scheme, in the continuation of the Parcall Frame a new page must be allocated and its address saved in an available table entry whose index is bigger then all the page indexes used in the previous parcall frame. As a consequence the page table will have holes during execution, if variable age is to be efficiently maintained.

3 Designing for And-Or Parallelism

We have noticed that a major goal in the design of recomputation based systems is to take the best advantage of research in and-parallel and or-parallel systems. To do so, one should aim at obtaining *orthogonality* between each component of the system. In an orthogonal design, or-parallel execution should be unaware of independent and-parallel execution, and vice-versa. Orthogonality implies that all the communication, synchronization, and data structures required to combine the two forms of parallelism can be designed independently. Moreover, orthogonality means that *And* Scheduling and *Or* Scheduling can be unaware of each other. This simplifies initial implementation and allows access to the substantial body of work available in scheduler design, while in no way compromising the ability to eventually design more sophisticated scheduling strategies.

In the case of independent-and plus or-parallelism, orthogonality means that workers should only be aware of or-parallelism when they backtrack to shared public choice points. The converse is that even if a worker backtracks to a public part of the search tree, *the other workers in the team should be able to continue independent computations.* The same principle applies to extensions of independent and-parallelism that support dependent and-parallelism.

One important consequence of this principle is that workers working in and-parallel should be unaware of the fact that other workers have backtracked to choice-points, *even if they are shared.* This seems to be a form of reuse, as and-parallel computations that are produced while in a branch of the search-tree will actually be used when the team moves to another branch. We claim that this form of reuse is in fact *needed* for efficient execution, because otherwise searching for or-work will result in killing legitimate and-work, and will very much increase the overheads of moving any non-trivial and-parallel task in the search tree.

To be able to reuse work, we must be able to guarantee that all data structures used to store local values, allocated by another team doing work that we may want share, *must never conflict with any of our own data-structures.* We call this property *memory independence.*

3.1 Memory Independence Among Teams

The requirement of orthogonality, and thus of memory independence, was not a major consideration in the original C-tree models, such as the PBA and ACE. The question thus arises of whether we can guarantee this property with these models.

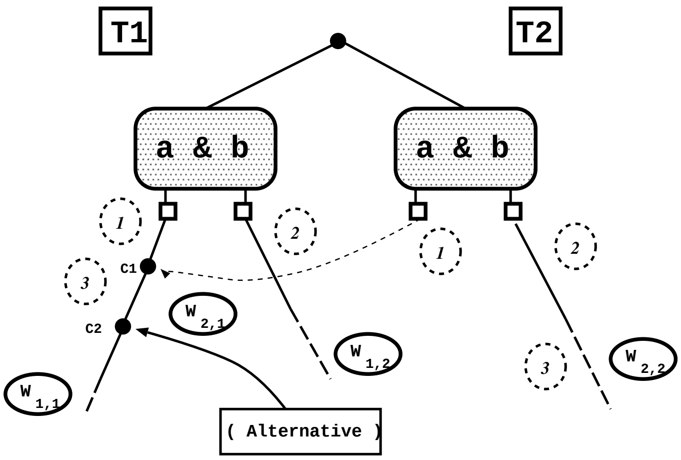

Figure 4: Structural problem in PBA.

Memory Independence in the PBA. We first study an example for using the PBA. Consider the situation shown in Figure 4. Team 1 (T_1) consists of two workers, $W_{1,1}$ and $W_{1,2}$. T_1 was the first team to exploit work from the parallel conjunction `a & b`. $W_{1,1}$ took the goal `a` and created the choice-points C_1 and C_2, whereas $W_{1,2}$ took the goal `b`. Note that initially $W_{1,1}$ allocated page 1 (as shown by the dotted circle) and that $W_{1,2}$ allocated page 2. Between creating choice-points C_1 and C_2 $W_{1,1}$ asked for a new binding-array page, receiving page 3.

Execution for team T_2 proceeds differently. The team also has two workers. $W_{2,1}$ first shares choice-point C_1 from T_1, whereas $W_{2,2}$ restarts execution of `b`. In this case, $W_{2,2}$ requires more variables more quickly than $W_{2,1}$ and asks for a new page receiving page 3.

At this point $W_{2,1}$ backtracks (an inside backtrack) to node C_1 and the Or-Scheduler decides that the worker will take from the closest node, in this case C_2. Notice that from node C_2 to C_1, T_1 used page table entry 3 that is at the same time being used by $W_{2,2}$ in its computation of goal `b`.

To move to node C_2, team T_2 must therefore rewind its computation of `b` in order to be able to install page 3 without conflicts. Notice that in general the problem may affect not just active active computations, but in fact all and-parallel computations that were performed in advance.

Memory Independence in ACE. The copying scheme used in ACE follows basic memory allocation principles similar to the PBA, and as such does not guarantee memory independence between teams.

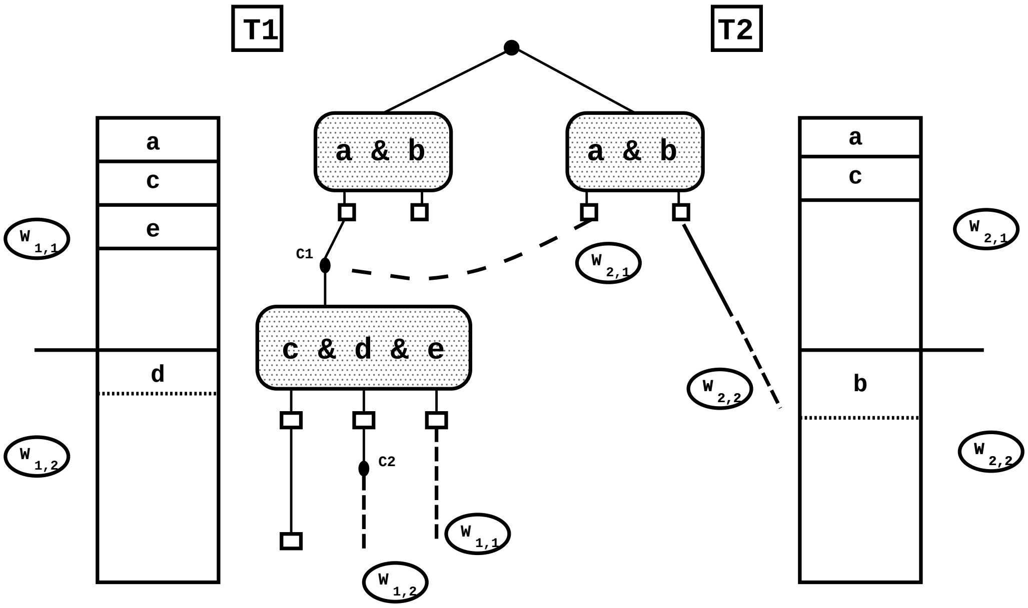

Figure 5: Structural problem in ACE.

Consider the situation shown in Figure 5.As in the the previous case T_1 was the first team to exploit work from the parallel conjunction `(a & b)`. In this case, $W_{1,1}$ took the goal `a` and created the choice point C_1. After a while, node C_1 is made public and team T_2 comes and takes work from this node relaunching goal `b`. $W_{2,1}$ takes goal `a` and $W_{2,2}$ takes goal `b` for execution. Meanwhile, worker $W_{1,1}$ starts parallel execution of `(c & d & e)` and takes for execution goal `c`. Worker $W_{1,2}$ moves to help and takes goal `d` for execution, eventually creating choice point C_2 in the process. Computation for goal `c` succeeds quickly and as a result node C_2 is made public. After succeeding goal `c`, $W_{1,1}$ takes goal `e` for execution. At this point arises the situation depicted in Figure 5. Worker $W_{2,1}$ has failed and performs *inside backtracking* to node C_1. Since this is a public node the Or-Scheduler is called and it decides to install work from the now public node C_2.

The problem in ACE appears when T_2 tries to install work from C_2. In the Figure 5 this is easily seen by observing that in the stacks for team T_1 the space used for goal `d` can conflict with the space being used by worker $W_{2,2}$ when executing goal `b`. As a result $W_{2,2}$ has to rewind and deallocate the computation of `b` before the installation of or-work from node C_2 can proceed. This is an inherent problem of ACE.

3.2 Obtaining Memory Independence

We have shown simple examples where teams conflict in their memory allocation. Much more complicated situations can and do arise. In general, correct execution using the previous data representations requires a team to kill and fully rewind at least some of the independent-and private computations that are under the sharing choice point before installing public or-work, even if these independent computations have or are going to succeed. This means that other and-workers would have to backtrack and discard any and-work performed in advance, a complex operation in independent and-parallel systems. Moreover, to find out which computations to kill we would need to compare stacks or binding arrays allocation tables, another expensive operation.

We would like to devise a data structure for variable bindings that does not impose any restrictions in the way a team is allowed to allocate memory, share or-work, and combine in an efficient way the solutions generated by independent and-computations.

One possibility is to extend the PBA to make it have centralized page management, thus avoiding conflicts and guaranteeing maximum flexibility. The idea was first proposed as the extended Paged Binding Array, or *SPBA* [9], a data structure that guarantees maximum flexibility in binding array management, at the cost of extra overheads. A second alternative would be to use hash tables, as originally proposed for ECRC's Pepsys [2] and then suggested by Shen for Prometheus [23]. Because hash tables are allocated in shared space, they trivially guarantee memory independence. On the other hand, hash tables do not guarantee constant time binding access. In pure or-parallel systems they do guarantee constant-time task switching, but this is not possible in and-or parallel systems, because of the need to reinitialize and-work. Last, hash-table management in an and-or system is rather more complex than for or-parallelism.

We would like a solution that does allow support for or- and and- parallelism, at a minimum overhead over pure or-parallel or and-parallel solutions. In the next section we present what we believe is such a solution, the sparse binding array.

4 The Sparse Binding Array

The *sparse binding array* (hereafter SBA), is derived from the original binding array used in SRI based models. The key idea is that we can guarantee independence if the index associated with a variable is unique in the whole search tree. In this way, if several and-workers within a team create variables at the same time, they are guaranteed never to use the same binding array slot. Unique indexes also guarantee orthogonality, as variable slots allocated within the different teams. that is or-agents, can never interfere with each other when public or-work is being installed by a team.

In shared-stack parallel logic systems, memory objects are built with blocks taken from a pool of common shared memory. This initial shared address space is then used to build each worker's stacks. Variables, among other objects, are located in these stacks. The simplest way to implement the SBA is to allocate for each team a private virtual address space with the same size has the system common memory pool. This address space is mapped at a fixed location for each team in the system. Bindings private to a team will be allocated there, whereas unconditional bindings can be placed in the shared data stacks. We say that the SBA *shadows* the system shared address space.

To obtain private values, SBA based systems can use a direct mapping from the shared memory pool to the team's SBA. The mapping is from addresses in the shared address space to addresses in the corresponding team's SBA as follows:

$$SHARED\_PTR \Rightarrow SHARED\_PTR + (SBA\_BASE - STACK\_BASE)$$

where $STACK\_BASE$ is the base address of the initially allocated System shared memory pool, and SBA_BASE is the base address of the corresponding SBA. Most operating systems allow

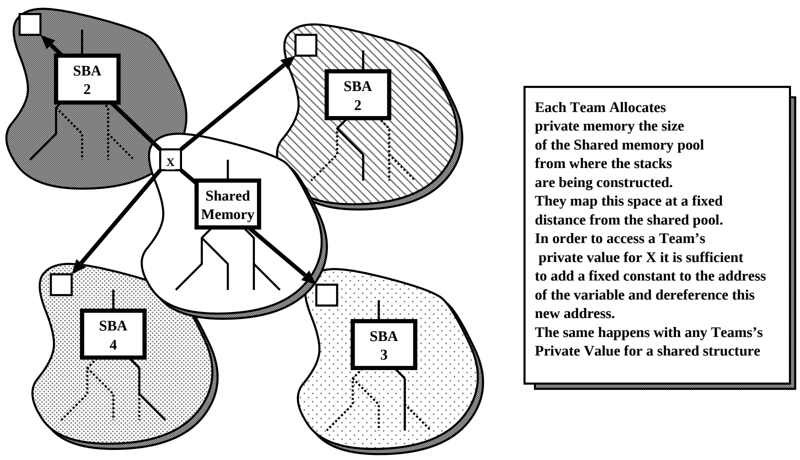

Figure 6: The Sparse binding array.

mapping memory at user-defined addresses. If so, we can guarantee the difference $SBA\_BASE - STACK\_BASE$ to be the same for all the teams the system, and known at compile-time. This is illustrated in Figure 6. In these systems, the private value of a variable can be obtained as follows:

$$PRIVATE\_VALUE = SHARED\_PTR[FIXED\_DISTANCE]$$

where $FIXED\_DISTANCE$ is a constant value known at compile-time. Variable access is in this case a constant-time operation, giving fast access. In fact, variable access becomes faster than for the traditional binding array, as we just need to add a constant.

The SBA approach solves the problem of memory allocation within teams, and guarantees faster access than for the BA. Notice that the value of the variable cell is no longer needed to find the binding array slot for its value, although it can still be used for variable age maintenance. One option is to use the same scheme as for Aurora. In our implementation of the SBA [20] we introduced an alternative scheme based on the notion of *level*, where a level corresponds to the number of choice points and parallel conjunctions above a stack object in the search tree.

In the SBA, each cell in the local and global stacks will have at least as many "virtual" copies as there are teams. We use the word *virtual* because we are referring to virtual memory, managed by the operating system. Most of this memory will never be used during execution and will just be kept as reserved backing store. In fact, modern operating systems only allocate memory when really needed. The question thus becomes of how much more memory will our scheme *use* over traditional binding arrays, and whether this is an acceptable overhead for supporting and-parallelism. In a previous analysis for pure or-parallel systems [20], we demonstrated that, most cells in the environment stack are variable cells, and also a large percentage of heap cells are variables. Thus the number of pages used by an or-parallel SBA systems was not much higher than for the corresponding BA approach, and the SBA-based system had better execution times.

Shadowing the whole stack and not just variable cells has an important advantage. Any object whose memory as been allocated from the initial common shared memory pool can have a private value in the SBA in exactly the same way a variable's private value has. Just use the address of the object, add to this address a constant offset (the distance between the SBA and the initial address of the common memory pool) and access that address. The SBA thus also provides the system implementor with a very flexible and easy way of maintaining other local values for objects in the search tree that are not variables. As we shall see that is fundamental for implementing the data structures associated with and-parallelism.

5 Data Structures for a Combined System

In our approach or-work is explored by teams of and-agents and iap-work is done by each individual and-agent inside a team. The organization of computing agents into teams has the following advantages:

- When workers fetch and-work they do not need to load their binding arrays, hash tables, or copy their new environment from other workers. For example, if the system is executing `(a & b),c` , any member of the team can take `c`. In systems where there is no concept of team, the worker that is going to execute `c` has to load the environments created during execution of `a` and `b` first.
- The team concept complies with the principle of orthogonality described previously. As such it allows the implementor to reuse code and ideas from previous implementations of other and-system and or-systems.
- Systems that are designed to support full Prolog need to know whether side-effects builtins can go ahead. The concept of a team allows us to clearly divide the leftmost search into and-leftmost and or-leftmost searches [8].

One can see a team as an address space shared by several workers and composed of two parts, (i) the shared system memory pool, which is in fact shared by every team in the system, and (ii) the SBA, which is private to a team. One can also see the team as formed by and-agents (in the spirit of &-Prolog or &ACE), or *threads*, that together form a team, or *process*. This organization gives a natural mapping to operating systems processes and threads.

Note that there is *no need for all threads in a team to be active at all times*. In fact, each team can have a pool of suspended threads (that is, not using computational resources) that are woken up when necessary. This provides a simple mechanism to implement reconfiguration [7]: whenever a team has excess of processors. it just has to yield a processor to a different team. This can be seen as a migration of and-workers from a team to another. If the team that receives the processor had no active workers, this mechanism can also be seen as reconfiguring from and-parallelism to or-parallelism. Notice that waking up a thread in another team can be done very efficiently, for instance, just by using operating system semaphores,

To summarize, a team is implemented as an operating system process with the shared system memory pool and private SBA. Together with an internal array of system threads some of which are suspended, waiting for computational resources to be freed by some other team, and some of which are actively doing iap-work in an or-branch of the search tree.

5.1 The Life of a Worker

The life of a worker is shown in Figure 7. The white-background nodes correspond to purely and-work, and the shadowed area corresponds to occasions where the worker must be aware of or-parallelism. The worker interacts with the rest of the team by sending kill signals or publishing work. Note that when a worker is moving in an area shared with other teams the rest of the team will be working in and-parallel unaware of the fact that the worker is backtracking to and fetching work from the shared part of the search tree.

The distinction between shared and non-shared parts of the search tree corresponds to the distinction between public and private work. Workers must interface with the or-parallel system when they move through public work. Following the Aurora engine-scheduler interface [24] there are three basic interface operations The *MakeNodePublic* operation is called to make private work a part of the logically shared tree; the *MoveUP* routine is called whenever a team backtracks in the search tree; the *MoveDown* routine is called when a team moves to new alternatives in the search tree.

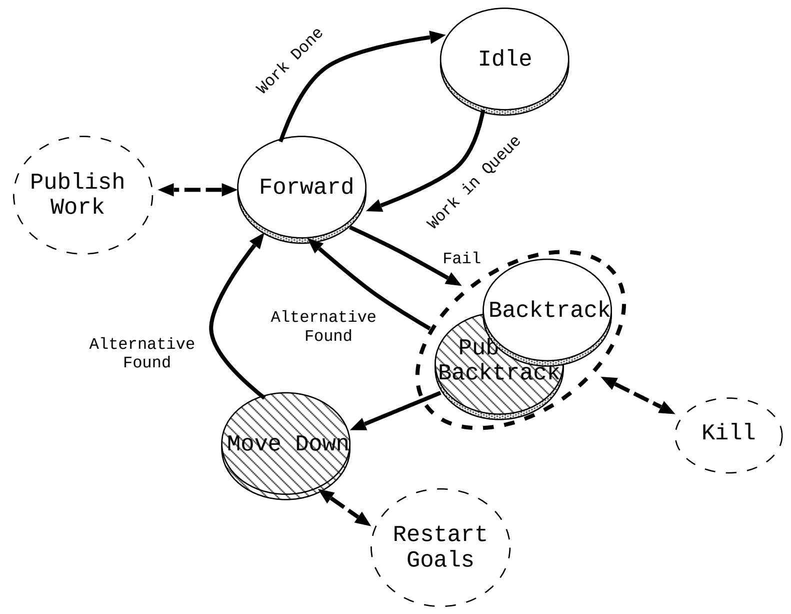

Figure 7: Worker Lifecycle.

The *MakeNodePublic* routine is called each time a choice point node is made public. As an engine routine it can have first-hand knowledge of this new data structures without interfering with the or-scheduler.*MakeNodePublic* must perform the same tasks as in the Aurora system, and perform actions related to the integration of IAP parallelism within the OR system. The major task is updating choice-point fields used by the or-scheduler to inform the work is shared. To do so the routine has to navigate through a choice-point stack that will also include markers and parcall-frames.

The *MoveUP* engine routine must be made aware of the new choice point stack objects and of the way memory is physically and logically connected so that it can be reused on backtracking. Observe that moving up the public tree involves decrementing the number of references of all the choice point objects found. The basic scheme uses a reference counter for each object in the choice point stack. Each time a team moves down over an object its reference counter is atomically incremented. When a team moves up over an object its reference counter is atomically decremented. When that reference counter reaches the value zero then the object is no longer referenced by any team in the system and can be logically deallocated.

By logically deallocating we mean marking some memory area that is not in the top of a stack owned by the worker doing the deallocation as free, and effectively making a hole of unused memory. Only in backtracking the worker who owns that piece of memory will be able to reuse it. This is the principle used in the Aurora system, proviso that now we have more objects in the choice point stack that need to be taken care of.

The *MoveDown* routine has to cope with the copying of computation slots from the shared heap to the SBA and update the slot pointers from the corresponding trail entries. The key idea here is that the routine actually moves from the bottom to the top of the shared tree. Thus, the first time it finds a parallel conjunction it has to initialize the SBA parallel conjunction values, copy the state of the computations of the goals to its left [2], and restart all and-parallel goals to the right of the current goal, allocating new computations state data structures in the process. If the parallel conjunction is visited again, no further work is required.

[2] Only complete work can be shared, as such the correct values for this computations have already been copied to the shared stacks by the team that published work

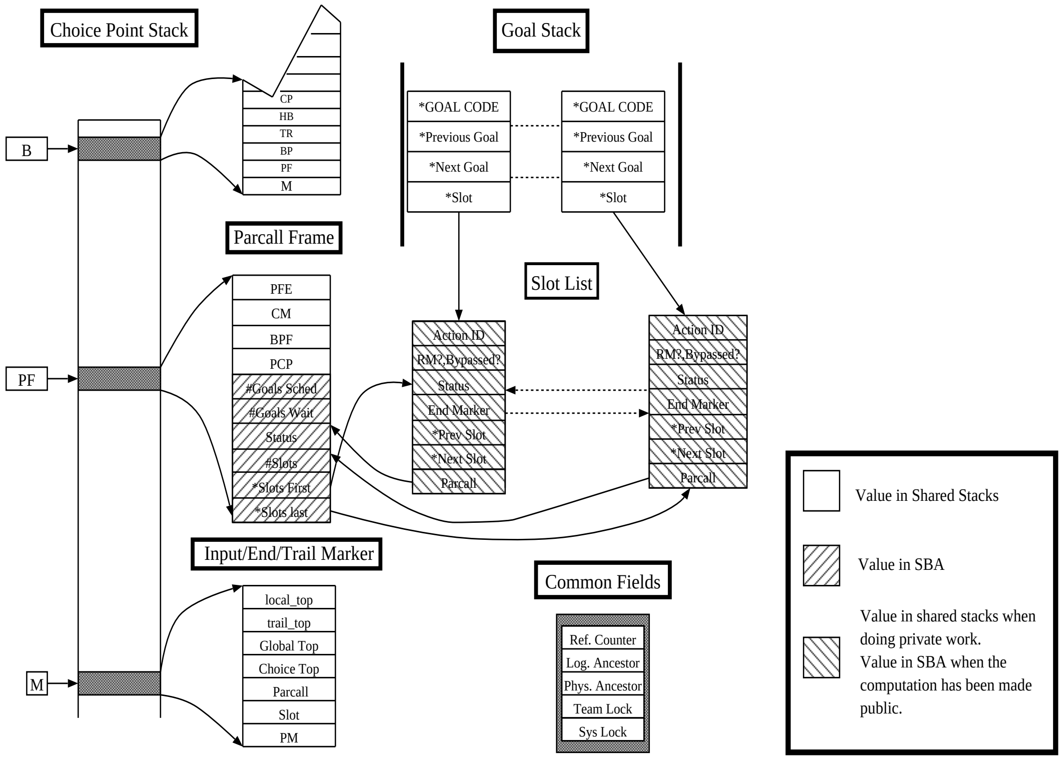

Figure 8: Data structures for or- and iap-and Parallelism.

5.2 The Execution Stacks

We have given an overview of how independent and-parallelism and or-parallelism interact in our proposal. Now, we are going to explain how to adapt the data structures and execution models used in the implementation of SRI derived or-parallel systems (Aurora) and iap-parallel RAP-WAM [14, 16] derived systems (&-Prolog and &ACE) to obtain an SBA recomputation based or-iap parallel execution model for full Prolog.

In Figure 8 we can see the major data structures introduced by our or-iap SBA based execution model. We extend the Or-parallel WAM, as used in the Aurora system, and combine it with the IAP-parallel mechanisms developed for &-Prolog and/or &ACE, using the SBA for environment representation. We concentrate on the issues relevant for the combined system, please consult [3, 14] for the issues in or-parallel and and-parallel systems.

The abstract machine registers closely follow the WAM registers. We introduce two new registers for independent and-parallelism. The *PF* register always points to the current parcall frame should a failure occur. The *M* register points to the Input Marker of the current iap-computation.

The two registers must be saved in choice-points only for shared work, as they are fixed for private work. Hence we may always save them in the `try` operations, or delay until a worker in a team makes the choice-point public. In the latter case it will be up to that worker that is publishing the work to give these fields the correct values.

We next concentrate on the data structures required for and-parallelism.

Parcall Frames Parcall frames are created whenever the conditions in a CGE [14] succeed and and-parallelism is started. The corresponding structure contains several fields. The Parcall-Frame Environment *PFE* is a pointer to the local environment when the parallel goals were activated, and is necessary to continue execution. The Continuation Input Marker *CM* is used to save the value of the M register at the time the Parcall is created. The previous parcall frame pointer, *BPF*, is used to propagate failure. The continuation of the Parcall Frame, *PCP*, is a pointer to the code to execute when the parcall-frame completes. Several other fields control current execution status: *#Goals_Sched* gives the number of goals still to be scheduled, *#Goals_Wait* gives the number of goals that are executed but not yet reported success, *Parcall_Status*, is used to detect whether we are in inside or outside backtracking, *#Slots* is the number of goals in the parcall frame, *SlotFirst* and *SlotLast* point to the first and last computation slot.

As we explained before, parcall frames are shared. We therefore must keep private copies for the values that will be different for different teams. As shown in Figure 8 the fields that change during execution are *#Goals_Sched*, *#Goals_Wait*, and *Parcall_Status*.

The &ACE system includes a powerful optimization, last parallel call optimization (LPCO) [6]. When launching a new set of and-parallel goals if the continuation returns directly to the previous parcall-frame, and if the intervening computation was deterministic, we can include these goals in the previous parcall-frame instead of creating a new one. The optimization results in impressive performance improvements for both space and time and should be considered in the combined system.

Two solutions are possible, we refer the reader to [5] for the full analysis. In the first alternative, we assume that the fields *#Slots*, *SlotFirst* and *SlotLast* will vary during execution, as parcall-frames receive new goals, and that they should be therefore private to each team. To update these fields as we search for work it will be required to keep special entries in the trail.

An optimization to this scheme is based in the observation that while the private computation is deterministic the and-agent can use the computation slot in the heap directly instead of using its "shadow" in the SBA. In this case LPCO can directly update the pointers in the heap slot list without trailing. Only after a choice point is created applications of LPCO need to be trailed.

While during private and-work manipulations of computation slots can proceed in the shared heap, when this private part of the branch is made public the value of the computations slots in the branch are required to be copied to the SBA and any further changes made in the SBA not in the heap.

As a further optimization if the computation completes and is deterministic then publicizing does not need to copy the computation slots from the heap to the SBA as their state will not be changed in the future by any team.

The Slots Each and-parallel goal is represented by a *Computation Slot*. This is a data structure allocated in the heap that has the following fields. The Id of the agent working within the goal or that finished the computation is stored in *ActionID*. The *RM?Bypassed?* flag indicates whether the computation is rightmost and whether it has been bypassed [6] by the computation to its left. The *Status* field gives the computation status. The *End Marker* slot points to the end marker of the current computation, if it exists. The *PrevSlot* and *NextSlot* maintain a doubly linked list of slots, and the *Parcall* field gives fast access to the parcall-frame owning the slot.

If LPCO is being used in the system then all these fields may sooner or later happen to have to be saved in the SBA. If this is not the case then *PrevSlot*, *NextSlot* and *Parcall* can be public.

The *End Marker* field requires special processing. Whenever an alternative is taken from a completed branch by another team the end-marker for this computation, if it succeeds, is going to be necessarily different from the end-marker of the former computation. As a general case when exploring or-parallelism for each computation slot there could exist more then one end-marker currently active, one in each team where alternatives for the computation succeeded.

To solve this problem an entry for this field is stored in the trail when the goal completes. This way when or-work is taken by another team, that includes this branch, the proper value for the end-marker is stored in the team's SBA location for the computation slot.

The Goal List The goal list provides a way for workers to share work. In &-Prolog each worker has a goal list, which is visible to other workers in the team. Each element of the list consists of four fields. The *Goal Code* field is a pointer to the code to execute, the *Slot* field is a pointer to the corresponding slot, and the *Next Goal* and *Prev Goal* fields maintain a doubly linked list.

The list is maintained by two registers, *FG* and *GL*, private to each worker. The *FG* register points to private work, and the *GL* register points to work shared and where access must be synchronized.

Goal lists are structures for and-parallelism. They are only affected by or-parallelism when a team moves down the search tree to fetch new work and relaunches new computations for goals to the right in each parcall frame it finds when installing work. This goals are introduced in the goal list of the and-worker that is moving down the tree.

Common Fields To support and-or parallelism all choice point stack objects contain extra entries labeled "Common Fields". These fields are shown in Figure 8 and are required to support synchronization and scheduling under the and and or execution mechanisms. The fields are:

- *Ref. Counter*- This field is used to describe how many teams are at the moment "under" that object. A node in the search tree is public if and only if its Ref. Counter is a positive integer, and private if zero. If a node is public, all stacks segments between this node and its logical ancestor must also be public. The field is needed for shared memory management under or-parallelism, as only when it gets to zero can the space in the stacks associated with the node be logically deallocated,
- *Log. Ancestor*- A pointer to the immediate ancestor object in the search tree.
- *Phys. Ancestor*- A pointer to the immediately physically preceding object in the worker stack. This is required for memory management, so that when allocating a new Object a check can be made for holes and memory effectively reused.
- *Team Lock*- A synchronization lock private to a team. For example when taking a goal from a parcall frame the counters must be updated. Exclusive access must be obtained. Note that to avoid unnecessary synchronization this lock should really be located in the SBA.
- *System Lock*- A synchronization lock global to the system. It is used for example when updating the reference counter of an object.

6 Conclusions and Future Work

One of the major challenges in current logic programming research is how to exploit maximum parallelism in logic programs efficiently. We presented in this paper a model and data structures that allow the efficient combination of two major forms of parallelism, or-parallelism and independent and-parallelism. Our model is based on recomputation of independent and-parallel goals, as originally proposed in the C-tree. We show that to obtain orthogonality between or- and and-parallelism some reuse of goals between search branches is required. We thus propose a new data structure, the Sparse Binding Array, that guarantees this orthogonality by simplifying binding array management, and demonstrate that by using this approach the main data-structures for both or-parallelism and and-parallelism can be naturally adapted to support a combined system.

Work has already started in developing systems based on the sparse binding array. We have adapted the Aurora system to use this principle, and first results show that for or-parallelism only this approach gives better single-processor performance and similar speedups as traditional binding arrays [20]. In the continuation, and in collaboration with the group at New Mexico State University, we are integrating the &ACE system with our or-parallel prototype. We expect that the combined system will not just exploit or- and independent and-parallelism, but it will also be able to support dependent and-parallelism as in the extensions to &ACE [19].

We are also continuing research on the data structures for supporting combined parallelism. We have demonstrated that the sparse binding array offers a very clean solution for recomputation based systems. In the future we plan to investigate whether the extra complexity of more space efficient solutions is worthwhile, by researching approaches such as the shared paged binding array [9], or using approaches that compress the sparse binding array, such as replacing the array by a fully-associative cache. We also plan to investigate on how to design copying based solutions that have memory independency.

Acknowledgments

The authors would like to acknowledge and thank for the contribution and support that Gopal Gupta, Enrico Pontelli, K. Shen, and Manuel V. Hermenegildo gave to this work. This work has been supported by the PRAXIS PROLOPPE project, the JNICT MELODIA project, and by the NATO MAPLE project.

References

[1] K. A. M. Ali and R. 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] U. Baron, J. Chassin de Kergommeux, M. Hailperin, M. Ratcliffe, P. Robert, J.-C. Syre, and H. Westphal. The Parallel ECRC Prolog System PEPSys: an Overview and Evaluation Results. In *International Conference on Fifth Generation Computer Systems 1988*, pages 841–850. ICOT, Tokyo, Japan, November 1988.

[3] M. Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine*. SICS Dissertation Series 02, The Royal Institute of Technology, 1990.

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

[5] M. E. Correia, F. M. A. Silva, and V. S. Costa. SBA: Exploiting orthogonality in OR-AND Parallel Systems. Technical report, LIACC, University of Oporto, April 1997.

[6] G. G. D. Tang, E. Pontelli and M. Carro. Last Parallel Call Optimization and Fast Backtracking in and-parallel systems. In *Workshop in Parallel and Data-Parallel Execution of Logic Programs.*, pages 93–109, 1994.

[7] I. Dutra. Strategies for Scheduling And- and Or-Work in Parallel Logic Programming Systems. In *Logic Programming: Proceedings of the 1994 International Symposium*, pages 289–304. MIT Press, 1994.

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

[9] G. Gupta, V. S. Costa, and E. Pontelli. Shared Paged Binding Arrays: A Universal Data-structure for Parallel Logic Programming. Proc. NSF/ICOT Workshop on Parallel Logic Programming and its Environments, CIS-94-04, University of Oregon, Mar. 1994.

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

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

[12] G. Gupta and B. Jayaraman. On Criteria for Or-Parallel Execution Models of Logic Programs. In *Proceedings of the North American Conference on Logic Programming*, pages 604–623. MIT Press, October 1989.

[13] G. Gupta and V. 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.

[14] M. V. Hermenegildo. An Abstract Machine for Restricted And-Parallel Execution of Logic Programs. In E. Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 25–39. Springer-Verlag, July 1986.

[15] 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.

[16] 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.

[17] L. V. Kalé. The REDUCE OR process model for parallel evaluation of logic programming. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 616–632. University of Melbourne, "MIT Press", May 1987.

[18] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. 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.

[19] E. Pontelli and G. Gupta. Non dependent and-parallelism revisited. Technical Report 1, Laboratory for Logic, Databases and Advanced Programming, New Mexico State University, Las Cruces, New Mexico, USA, February 1996.

[20] V. Santos Costa, M. E. Correia, and F. Silva. Performance of Sparse Binding Arrays for Or-Parallelism. In *Proceedings of the VIII SBAC-PAD*, 1996.

[21] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[22] E. Shapiro. The family of Concurrent Logic Programming Languages. *ACM computing surveys*, 21(3):412–510, 1989.

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

[24] P. Szeredi and M. Carlsson. The engine–scheduler interface in the aurora or–parallel prolog system. Technical Report TR-90-09, University of Bristol, Computer Science Department, April 1990.

[25] D. 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.