A New Implementation Scheme for Combining And/Or Parallelism

Kish Shen
Department of Computer Science
University of Manchester
Manchester M13 9PL
U.K.
kish@cs.man.ac.uk

Abstract

This paper presents the Fire model, an implementation scheme of the "or-under-and, no reusage" execution scheme for and/or parallelism in Prolog. Unlike previous schemes, the Fire model does not contain any private storage for the alternative bindings. Instead, the bindings are stored in data-structures associated with the search-space, and is accessible by all. The advantage is that task-switching and scheduling, one of the major area of complexities with previous schemes, becomes simple and constant time. Another property of the Fire model is that when either and- or or-parallelism alone is exploited, there are little extra overheads when compared to schemes which exploits only one of these forms of parallelism.

1 Introduction

There has been a lot of interest in the implicit parallel execution of Logic Programming languages, especially of Prolog, and many schemes which exploit various forms of and- and or-parallelism have been proposed and implemented. Although many of the early proposals exploited both forms of parallelism (e.g.[9, 20, 36, 11, 15]), many of the more recent schemes, especially those that have been successfully implemented, have tended to concentrate on either and- or or- parallelism (e.g. [21, 1, 18, 5, 28]), probably because of the complexities and difficulties in efficiently implementing both forms of parallelism together. One system that has been relatively successful in exploiting both forms of parallelism is Andorra-I [25], but the and-parallelism is limited to "determinate" goals only, with or- and and-parallelism exploited in distinct phases. Comparatively recently, various implementation schemes that exploits both forms of parallelism have been proposed ([13, 14, 16, 24]), which to a greater or lesser extent implements the high-level execution scheme "or-under-and, no goal reusage" used in a simulation study of and- and or-parallelism [26, 29, 31]. This scheme combined independent and-parallelism (IAP) with or-parallelism such that the and-goals are "recomputed" (or more accurately, separately computed); the scheme can be readily extended to cover dependent and-parallelism (DAP)[1] as proposed in the Prometheus scheme [26]. The "or-under-and" scheme provides a more flexible framework for combining and- and or-parallelism than the Basic Andorra Model, because the two forms of parallelism need not be exploited in different phases. At the same time, it also presents more challenging implementation problems because the and- and or-parallelism can be mixed.

Conceptually, the "or-under-and" scheme is simple and adheres quite directly to Prolog: essentially the search-tree is divided into chunks which are explored in and- and or-parallel, with no restrictions on where and- or or-parallelism can be exploited. However, the scheme does present significant implementational challenges: not only does it have to deal with the implementation complexities of and- and or-parallelism, but it also need to handle the combinations of both forms of parallelism, and implementational decisions suitable for one form of parallelism may not be very suitable for the other form. There is thus the question if a system exploiting and and or-parallelism is really needed, especially as to date not many programs have been shown to exploit very significant amounts of both and- and or-parallelism. I believe that such a system can still be useful, for the simple reason that the programmer would not want to switch to a different Prolog system simply because his or her programs are more suited for a particular form of parallelism. One way round this problem is to exploit both forms of parallelism, but place restrictions on where a particular form of parallelism could be exploited; examples of this approach include Conery's AND/OR process model [9], Fagin's PPP model [11], the "no or-under-and" scheme studied in my simulation

[1] The high-level execution scheme can be readily extended to cover DAP, but depending on the actual implementation scheme, extending the scheme to efficiently implement DAP can be more difficult.

study [29], and Andorra-I [25]. Such schemes are able to simplify some of the implementation complexities associated with combining and- and or- parallelism, but the restrictions imposed can exclude significant sources of one or the other form of parallelism: for example, Andorra-I cannot exploit non-deterministic and-parallelism; [26] showed that "no or-under-and" can significantly restrict the exploitation of or-parallelism in some programs, and the restrictions imposed on the PPP scheme can also constrain the exploitation of parallelism when compared to the "or-under-and". Lastly, the development of an "or-under-and" scheme leaves open the possibility that programs can exploit both forms of parallelism together: the lack of good example programs that contain significant amounts of both forms of parallelism does not necessarily mean that there are no applications that would be able to exploit both forms of parallelism, especially given that and/or parallel systems have not been widely used.

ACE and PBA are the first two proposed implementation schemes of the "or-under-and, no goal reusage" approach [13]. However, the actual development of these schemes have uncovered some difficulties and complications, which have to some extent delayed the development of actual prototype systems;[2] in fact, PBA is no longer being developed, instead, an alternative to the PBA – the SBA [24], which should simplify some of the complexities associated with PBA, is now being developed. Probably because of the implementation difficulties, neither schemes appear to have reached a state of development where an efficient implementation is available. In addition, for implementational reasons, both schemes imposed some restrictions on when parallelism can be exploited. In this paper, we propose *Fire*, a new alternative implementation scheme for the "or-under-and, no goal reusage"method for combining and- and or-parallelism. We believe that this scheme greatly simplifies some of the complexities associated with implementing ACE/PBA/SBA by making different trade-offs from these schemes: task switching (which appears to be at the heart of much of the complexities of these previous implementation schemes) is made simple in Fire, at the expense of some increase in complexity of the implementation of or-parallelism. It also lifts most of the scheduling restrictions associated with the previous proposals. Due to space limitation, in this paper we shall only discuss the model in relation to combining IAP and or-parallelism, and leave out the extra implementation complexities of accommodating DAP.

In the rest of this paper we shall first introduce the separate computation approach to combining and- and or-parallelism, and then introduce some terminologies and concepts that will be used through out the rest of the paper. This will be followed by an introduction to the Fire model, with a quick overview of the PEPSys or-parallel binding model, which forms the basis of the or-parallel model used by Fire. A discussion of various aspects of Fire is then presented, followed by a comparison to other models.

1.1 Separate computation versus reuse

Given sufficient freedom to combine IAP and or-parallelism, an opportunity for reducing the amount of computation performed by a program readily presents itself: the various instances of an and-goal in different branches of the search-tree, which would be computed repeatedly in sequential Prolog, always returning the same answers (because the goals have the same inputs), can be computed only once and then the results reused when and if needed. This idea of reusing the results of the same computation in different places in a program has its origins in the memo function [22]. The reusage of independent and-goals can be considered as a form of implicit memo function. In theory, it is possible to perform this implicit memo function in sequential Prolog or schemes which exploits only and- or or-parallelism, but the problem of identifying suitable goals, and the implementation challenge of efficiently storing such goals and reusing them at the appropriate places has meant it has not been proposed in such systems. It requires both and- and or-parallelism, with sufficient freedom in how the two forms of parallelism can be combined, before the reusage option seem attractive and obvious: the or-alternatives of the and-goals and the and-goals themselves have to be available simultaneously. Proposals which exploit only one form of parallelism, in which not everything is available simultaneously, have naturally followed sequential Prolog and did not reuse independent goals, and indeed even some of the early proposals to combine the two forms of parallelism (e.g. [9, 11]) did not have reusage, due to the severe restrictions on how and- and or-parallelism can be combined (and thus what would be available simultaneously). However, for all early schemes which did have sufficient freedom, reusage seemed so attractive that it was always included (e.g. [36, 20, 35, 15]).[3]

As part of a study of IAP and or- parallelism [29], I studied goal reusage by comparing it with an alternative scheme which did not implement reusage. This involved the development of an execution scheme that allowed great freedom in how and- and or-parallelism can be combined, but which did not have reusage. To the best of my knowledge, this "or-under-and, no reusage" scheme is the first proposed and/or parallel scheme with sufficient freedom in combining and- and or- parallelism that did not reuse independent goals. Instead, these goals were

[2] Personal communications with E. Pontelli, Sept. 1996 and March 1997; with V. Santos Costa, Nov. 1996.

[3] Another reason for the near universal proposal of reusage may be that the and/or tree abstract data model that was generally used to represent the and/or computation cannot represent the separate computation of and-goals simultaneously.

computed separately as in Prolog. The simulation study and subsequent experiences suggested that the non-reusage of and-goals may in fact be better than the reusage of and-goals, for the following reasons:

- The reduction of search-space via reusage, while large for some benchmark programs, does not have any effect (or very insignificant effect) on most programs. In the rare cases where it might be useful, predicates such as bagof, setof, or some special new predicates (for efficiency) can be used to provide the functionality (see for example [26] for details).
- An extra stage is needed to combine the reused goals to produce all the solutions. This is expensive and requires rather complex synchronisations. In addition, if the system provides the option of selective or-parallelism, then solution combining becomes even more difficult, as in PEPSys [3]. Furthermore, supporting reusage would carry a certain overhead, quite possibly even affecting the efficiency of execution where reusage does not bring any benefits.
- It is difficult to deal with side-effects.
- It is difficult to deal with dependent and-parallelism with reusage of goals: dependent and-goals cannot be reused, and if mixed with the execution of independent goals, the scheme for managing the reusage can become even more complex.

Since the proposal of the non-reusage execution model, various implementation schemes for the model (usually with various restrictions) have been proposed, e.g. ACE, PBA, SBA. As already discussed in the introduction, this paper introduces yet another alternative implementation scheme — Fire — which may have some advantages over previous schemes.

2 Preliminaries

2.1 The Annotated Search-Tree

The Annotated Search-Tree (ANT) will be used to represent the search-space explored by an and/or parallel system. An ANT also gives some indication of where parallelism may be exploited in this search-space. It is basically a Prolog search-tree with extra annotations to represent and indicate the parallelism. It was the basis for the data-structure used in the and/or parallel simulator [26]. Its advantage is that it clearly relates both and- and or-parallelism to where they occur in the search-tree, making it obvious what needs to be computed and at what point. An alternative representation to the ANT is the C-tree proposed by Gupta [13]; it is not used here because it is less easy to relate it to the search-tree, and it is less obvious to determine when recomputation should take place by looking at the tree, although it does have the advantage of being able to represent and-parallelism in a clearer way that the ANT.
The annotations on an ANT are:

or-node A normal search-tree node (a choice-point) which is a source of or-parallelism — the clause is allowed to execute in or-parallel. An or-parallel node is distinguished from a normal search-tree node to make the source of or-parallelism obvious, and also to allow for execution models which allow or-parallelism to be exploited at selected locations for efficiency reasons.

And-fork This indicates the start of a new source of and-parallelism, which corresponds to the start of a CGE in execution models such as &-Prolog [18] and DDAS [27].

And-node This indicates the start of an actual and-task, i.e. an individual and-goal in a CGE in &-Prolog and DDAS.

And-join This indicates the end of a source of and-parallelism, which corresponds to the end of a CGE.

Note that because an ANT represents the search-space of a parallel Prolog execution scheme, it sometimes does not correspond exactly to a sequential Prolog search-tree: due to the possibility of performing speculative computation in a parallel system, the ANT could have extra branches not found in the search-tree generated by sequential execution, representing the wasted computation performed by a parallel execution.

Figure 1 shows an example ANT and the program that generated it. For illustrating the correspondence between the two, goals which can potentially produce different results, such as `c` and 'm', are indicated in the ANT by

```
b :- (c & d), e.

c :- A=1, (g & h), m.
c.

d.

e.

g :- fail.
g.
g.

h.

m.
```

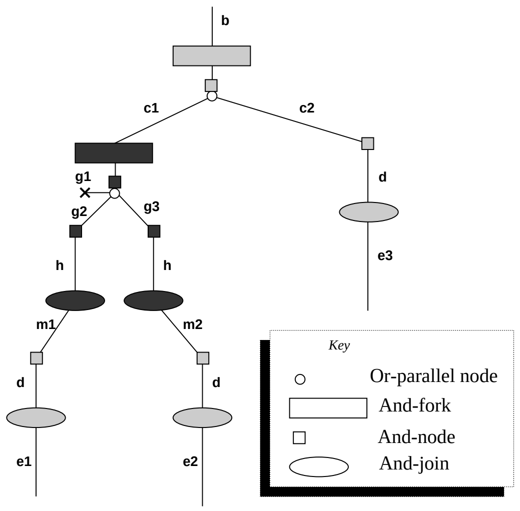


Figure 1: Example Program with its Annotated Search-Tree

appending a number to the name of the goal, e.g. `c1` and `c2`. Note that goals such as `d` occurs in different places in the tree because they compute identical results in these positions, and these are the goals that would be computed once and reused in a scheme that exploits reusage. Here, the separate computation of these goals are clearly shown by the tree, because they form different arcs of the ANT. Failures (which occurs for `g1`) are indicated by a cross. This program contains two CGEs, and the associated annotations (And-fork, And-node, And-join) with each CGE are indicated by different levels of grayness.

2.2 Some definitions

An *or-branch* of an ANT is a branch of the ANT that originates either from a non-leftmost branch of an or-parallel node, or from the root of the tree.
An *and-arc* of an ANT is a segment of a branch between two adjacent and-nodes in a CGE, or a segment from the rightmost and-node in a CGE to the corresponding and-join, or a segment of a branch originating from an and-node which leads to failure, the latter is also referred to as a *failing and-arc*. An and-arc corresponds to the search-space explored by an and-task (inclusive of any subtasks). Figure 2 shows examples of and-arcs in an ANT, indicated by double-headed arrows. Note that the some segment of the ANT can occur in more than one and-arc if there is a choice-point further down the segment, and that definitions of and-arcs are not affected if choice-points are or-parallel nodes or not.
A *complete and-thread* is a segment of a branch between an and-fork and a corresponding and-join of the same

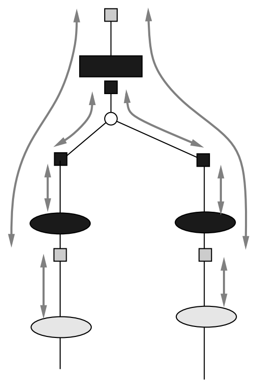


Figure 2: Examples of and-arcs

CGE. It consists of several connected (non-failing) and-arcs. A *failing and-thread* is a segment of a branch that originates in an and-fork, and which consists of several and-arcs which would be connected if completed, but at least one of which is a failing and-arc. Note that and-parallelism means that an and-thread can potentially be explored in and-parallel, and if the and-thread fails, then the branch may not be physically contiguous.
Sibling and-nodes are and-nodes which occur in the same (complete or failing) and-thread.

3 The Fire Model

In several previous work to combine and- and or-parallelism, the or-parallel part of the system was based either on the SRI-model [34] (e.g. Andorra-I [25], PBA [16], SBA [24]), or on the Muse model [2] (e.g. ACE [12]). In these or-parallel models, the or-parallel bindings are associated with the workers (private binding arrays in the SRI-model, copied stack in the Muse approach); when extended to include and-parallelism, these structures which are private to each worker are shared by a group of workers, known as a *team*, which are working on the same and-thread of an ANT. The maintenance of this shared structure requires co-ordination between the member workers of a team, which imposes some complications, such as restrictions on scheduling (e.g. workers cannot simply leave a team without considering other workers in the team) and implementation complications due to nested and/or parallelism. In order to stay within a team, a worker must be restricted to selecting the and-parallel work in the and-thread. This may limit the amount of parallelism exploited, especially if workers are fixed within a particular team, as is the case in many of the initial implementations of these models. An extra scheduler which partially overcomes this restriction by moving workers around teams was developed for Andorra-I [10], but this imposes extra overheads (especially in terms of memory usage[4]), and does not totally overcome the problem, because this is an extra layer of scheduling (which in itself adds further complications), and will only kick in when it is somehow decided (probably via some heuristics) that a worker is no longer doing fruitful work in a particular team, especially if moving worker to a different team is relatively expensive (as it is probably likely to be).

Another implication of using private (to a team) storage for the alternative bindings is that when a team of workers moves to new or-work, there must be a physical traversal of the ANT from where the team was to the new work, in order to disinstall and install bindings into the private storage area. The structures that needs to be maintained to do this was relatively expensive in an or-parallel only system, but extra complications exist for an and/or parallel system, e.g. the path between the old and new work may not yet be completely connected as there may be and-threads that have not yet completed. In both ACE and PBA, restrictions are imposed on scheduling so that such movements will always be between connected paths[5].

Another approach to dealing with the or-parallel binding problem is to associate the alternative bindings with the search-tree. In fact, some of the earliest proposals to deal with or-parallel bindings use this approach (e.g. [6, 8]). The main characteristic common to these models are that access to some bindings may be non-constant time (because the structures representing the search-tree may have to be searched to locate the correct binding), but at the same time, as the bindings are not associated with the worker, the worker can task-switch in constant time. In contrast, both the SRI and the Muse models can access all bindings in constant time, although task-switching would not be constant time. It has been argued that having non-constant time task-switching is inherently better

[4] Personal communication with I. Dutra, Nov. 1996.
[5] Personal communications with M. Correia and V. Santos Costa, Nov. 96; E. Pontelli, March, 97

than non-constant time variable access, because variable access occurs more often. However, I do not believe that the implementation experiences and the experimental evidence strongly support this, because the actual trade-off is quite complex, and depends on how frequently and how expensive the specific non-constant time event occurs, and how expensive it is to support the various different approaches. For example, in Aurora, significant overhead in task-switching is incurred, not because the number of bindings that need to be installed/disinstall – which was originally expected to be the main cost of task-switching – are great [33], but probably because in order to allow a worker to install and disinstall bindings from the binding arrays, and in particular if extra information (such as locations of individual workers in the search-tree) are retained to allow a worker to move to a "good" location, a lot of extra overhead is incurred. In fact, a simulation study [31] showed that even for a favourable program like N-queens, where there is extremely high potential parallelism and large granularity, the overhead for task switching can be very significant and have an impact on the speedup. Performance data obtained from PEPSys [7] and Aurora [33] shows that Aurora gives somewhat better performances, but the margin is not sufficient to conclusively show that the SRI model is inherently better than the PEPSys' binding scheme.

I believe that one advantage that the SRI and Muse models offer over PEPSys is the comparative simplicity of the abstract machine – the PEPSys or-parallel binding model requires more modifications to a sequential WAM, although it may potentially hold an advantage in some areas such as simpler task switching support. However, the simplicity advantage for the SRI and Muse models may not hold when the models are extended for and-parallelism. The Fire model is based on extending the PEPSys or-parallel binding model[6] to and-parallelism, and I believe it offers advantages over previous proposals in simplicity of the implementation and in the lack of restrictions on scheduling. PEPSys is used as the basis for handling or-parallelism in the Fire model because it appears to be the most efficient scheme which stores alternative bindings into the search-tree.

3.1 The PEPSys or-parallel binding model

For completeness, a very brief description of the PEPSys or-parallel binding model will be given here; the reader is referred to [8] for more details. Note that only the or-parallel binding scheme of PEPSys is used for the Fire model. The complete PEPSys model included and-parallelism via a reusage scheme, which to the best of my knowledge was never fully implemented, This scheme is very different to the way and-parallelism is handled in the Fire model.

In PEPSys' or-parallel binding model, alternative bindings are stored in a data-structure associated with each or-branch of the search-tree. This data-structure was referred to as a "hash-table", but it need not be implemented as a hash-table[7], so we use the more general term "binding area" (BA) in this paper. A new BA is created each time a new or-branch is taken by a worker, and linked to the BA in the original or-branch (the *parent or-branch*) that the new or-branch originates from. In addition, each time a new or- parallel node is encountered in a branch, an "age" counter is incremented. On any or-branch, if a variable which was created before the latest or-parallel node is bound, they may take on different values in different or-branches, and are referred to as *alternative bindings*. Such bindings are stored into the BA associated with the or-branch (otherwise it can be bound on the stack as normal), along with the current "age". Thus a BA holds the alternative bindings in the or-branch it represents. When a task (working on some or-branch) attempts to access an alternative binding, it has to first check its own BA to see if its has made a binding; if not, it needs to check the parent or-branch's BA to see if there is a valid binding there. A valid binding is one which is made before the or-branch split from its parent, i.e. if the "age" associated with the alternative binding is younger than the origin of the or-branch. This checking of BA needs to be continued (if a valid binding is not found) until the or-branch in which the variable was created. If no valid binding is found, then the variable is unbound. Once the value for a variable is looked up in this way, it can be then "cached" into the local BA, so that the next lookup for the variable can be found in the local BA.

In this scheme, because the alternative bindings are stored in BAs, which are associated with the search-tree, and not the worker, a worker can simply jump to a new task when it finishes an old one. A worker finishes its task when it backtracks to an or-parallel node where there are still or-branches that have not been completely explored. The cost for being able to task-switch in constant time is that access to some variable bindings may be non-constant time. Measurements with PEPSys suggested that there are relatively few of these bindings, and that the lookup usually do not involve accessing many BAs [7].

[6] Or more precisely, the CMOS or-parallel binding model developed independently by the author, but which was not published because it proved to be very similar to the published PEPSys model.

[7] For example, it might be implemented as a sparse array, using similar techniques to SBA (Sparce Binding Array) [24].

3.2 Extending PEPSys for and-parallelism in "or-under-and" scheme

In the Fire model, the PEPSys or-parallel binding model is extended to cover and- parallelism. As the or-parallel bindings are associated with the ANT which is accessible by all workers, there are no longer storage that are private to teams, and indeed, there is no real concept of teams. Conceptually, branches of the ANT which are actually executed in or-parallel has a BA associated with it, with the various and-parallel workers that are working on the branch using this BA. This corresponds to a "team" in the other approaches, but the workers do not need to manage any common data-structures together, as the BA is associated with the ANT, and not the workers.

It may be possible to implement this conceptual BA as one physical BA, but it is also possible to have separate physical BAs associated with each part of the tree where a task is running in parallel. This simplifies the management of the BAs, and will be the approach developed here. One disadvantage of this approach is that it creates more physical BAs, and potentially more BAs need to be searched for the valid bindings. Once a worker completes a task (or the task is suspended), it is free to take up any available work (i.e. work from either and- or or-nodes), anywhere in the computation, so no restrictions are placed on scheduling, unlike the team-based schemes. The selected new work can be started immediately, with the possible initialisation of a new BA: it is possible to use the same BA, if the new task is in the same conceptual BA, and is further down (i.e. would be computed later sequentially) the branch. The need for the new task to be further down the branch is due to the requirements for the age counter, as shall be explained shortly.

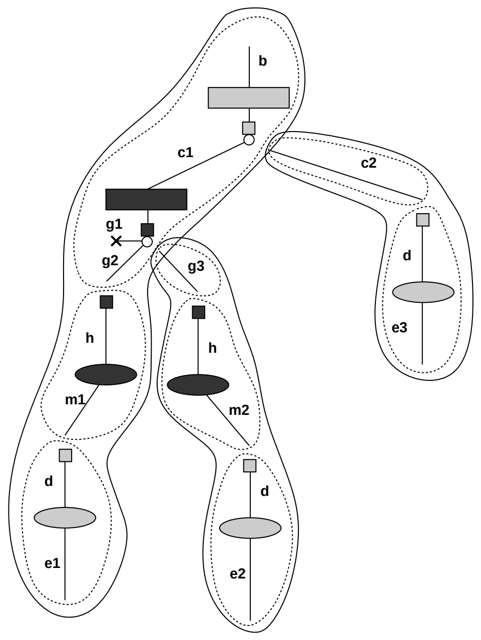
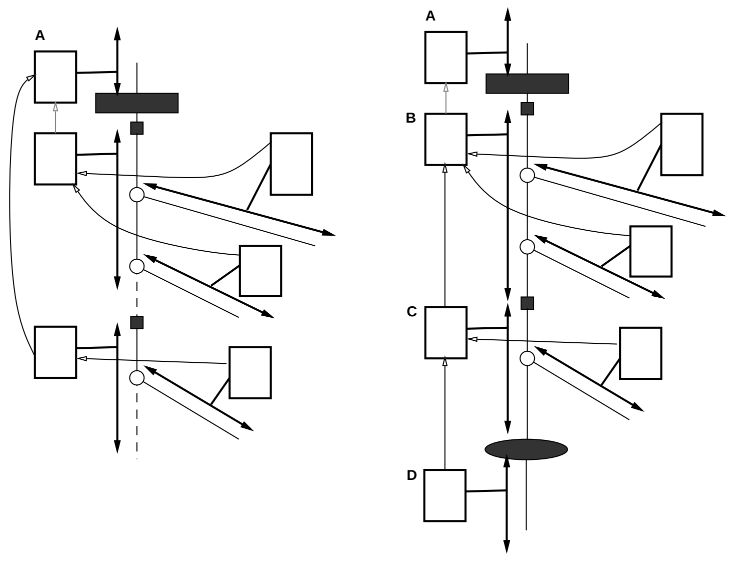


Figure 3: Physical and Conceptual Binding Areas

Figure 3 shows one possible arrangement for the BA in association with the ANT, running the program in Figure 1. For clarity, the parts of the tree are separated from each other if they are run as a separate parallel task, so for example, c2 and d were ran in and-parallel, with the worker executing d picking up the continuation after the CGE, executing e3. In the case of g1, the same worker picked up (in this instance) g2 after g1 failed, so g1 and g2 are executed as the same task.[8] The solid outlines surrounding the tasks represent the extent of each "conceptual" BA, with the dotted outline representing the extent of each physical BA.

For IAP, bindings created in one and-arc does not need to be visible to the other sibling and-arcs while the and-thread is not yet completed, thus the BAs of the sibling and-arcs do not need to be linked to each other before the and-thread completes. Figure 4 shows how the BAs are linked at two different stages of execution. In addition to the ANT, the physical BAs are shown as boxes with thick outlines, with the extent covered by each BA shown by a double-headed arrow. Figure 4a shows the situation when the and-thread is still executing, i.e. when at least one of the and-arcs of the and-thread have not yet completed (this is indicated by the dashed lines at the end of the and-arcs in the figure). In this case, the BAs of the two sibling and-arcs (B,C) are not linked to each other, but

[8] Note that in this figure, the continuation is always picked up by the rightmost and-goal. It is also possible for the continuation to be picked up by a non-rightmost and-goal if it is the last to finish; in this case, the physical BA would extend over two discontinuous areas in the ANT.



a) During execution of and-thread b) After completing and-thread

Figure 4: Linkage of physical Binding Areas

are both linked to the BA ‘A’, before the and-thread. The links between BAs are shown in the figure by the lighter arrows. Any or-branches would need its own BAs, which would be linked to the appropriate BA covering the area where the or-branch originates, much as in an or-parallel only system. The sibling and-arcs’ BAs have to be linked when the and-thread completes, and as any inner and-threads in a nested CGE must already have completed (and thus properly linked), only the local level of sibling and-arcs needs to be linked. In fact, as the physical BAs to be linked are all part of the same conceptual BA, the order in which the physical BAs are linked is not important. In Figure 4b, they are shown linked in a right to left order. Note that although there is a physical BA for each segment of the graph in Figure 4, this is only for clarity of the illustration. In practice, the number of physical BAs will be less than that shown, because successive tasks executed by the same worker that belongs to the same conceptual BA will not require the creation of a new BA, so at the very least, A and B will be a single physical BA, and D will be the same BA as the one used by the task that completes the and-thread.

3.3 Dealing with age of variables

At first sight there might seem to be a problem with the handling of the age for determining valid bindings. In an or-parallel only system, the age is incremented each time there is an or-parallel node in a particular search-tree branch, so the age increases going down the branch. With and-parallelism, different parts of this branch (segments forming and-arcs) would be explored in and-parallel, so the straight-forward incremental increase of age down a search-tree branch is no longer possible, unless and-parallelism is either restricted or not exploited. However, this is not a problem as this monotonic increase of age is not required. For the age to work, the following is required:

- Within a physical BA, the age should increase down the part of the branch the BA covers.
- For a variable stored in the BAs, three types of events can occur in different physical BAs (but the same conceptual BA) that requires some consistencies between the physical BAs: 1) the creation of the variable (this is needed to know when to stop looking for the variable’s value), 2) the binding of the variable, 3) the access of the variable’s value. What is needed is that the age for event 3 must be greater than or equal to the age for event 2, which in turn must be greater than or equal to the age of event 1. Thus if the three events occur in different physical BAs, then the ages in the BAs must maintain the above relationships.

For IAP, before an and-thread completes, a variable bound in one physical BA cannot be accessed in another physical BA of a sibling and-arc, because the variables are independent, so no age relationship needs to be enforced *between* the physical BAs of an and-thread, though the following relationship has to be enforced between them and the physical BAs before and after the and-thread: a) The age should be smaller (older) before an and-thread is started, to ensure that any variables created before the start of an and-thread would have an age smaller than the ages used in any physical BAs of the and-thread. b) The age after the and-thread has executed should be greater than the ages used in the and-thread. This is to ensure that any access to variables bound inside the and-thread would have a greater age than the binding.

To observe the needed relationship, the following scheme for the ages, adapted from the handling of dependent variables' age in DASWAM [26] is used: the age is incremented each time an or-parallel node is created as in the or-parallel case and, in addition, each time an and-thread is started or completed – i.e. the age in a branch is incremented each time it encounters an and-fork. The same physical BA can be used for different segments of a and-thread if the new segment is further down the and-thread than the old segment, because the age of the new segment will be greater than (if or-nodes were encountered) or the same age (if no or-nodes were encountered) as the old segment, as required. If a new physical BA is created when a sibling and-arc is started, then the age used at the start is the age set by the and-fork of the and-arc. When an and-thread is finish, the age for the continuation (i.e. the segment following an and-join) is set to be greater than the greatest age used inside the and-thread. This can be done by recording the greatest age for each and-arc of the and-thread as each and-arc is completed. When the last and-arc is completed and the continuation is to be executed, then the greatest age used by the and-arcs can be easily determined by checking the ages for each sibling and-arc.

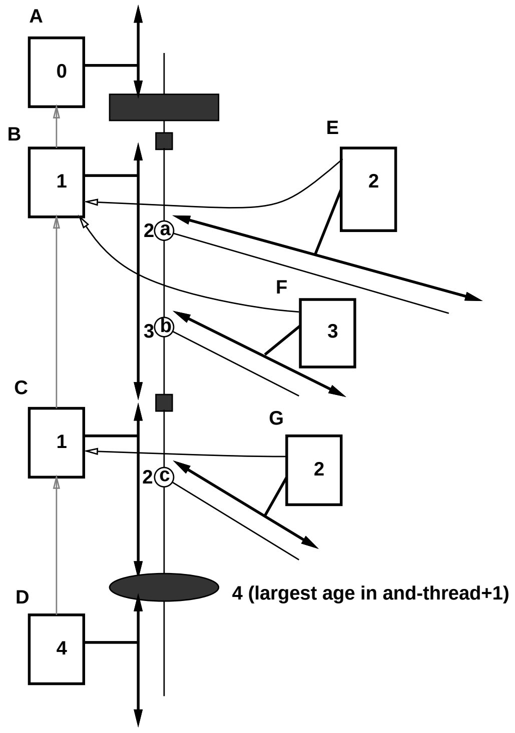


Figure 5: Arrangement of age in and/or parallel

Figure 5 shows the arrangement of the age inside a CGE with more than one and-thread. The number inside each BA indicating the age at the start of that BA; the numbers beside the or-parallel nodes are the ages of the nodes. Thus, the age before starting the CGE is 0, and this age is incremented to 1 for the start of the and-thread. At the or-parallel node a, the age is incremented to 2, and subsequently at node b, the age is incremented to 3, as in an or-parallel only system. If the second and-goal is picked up by a different worker, and a new physical BA has to be created, then the age again starts at 1, and increments to 2 at node c (instead of from 3 to 4 with only or-parallelism or if the same worker executes both and-goals). In this case, when the and-thread completes, the age of the continuation is set at 4, which is the greatest of the ages of the and-arcs of the and-thread (3 for the first and-arc, 2 for the second) + 1.

Consider a variable created in BA A of Figure 5, so that it's creation age is 0. If it is bound before or-node a, then the binding age will be 1, and if it is accessed in the or-branch covered by BA E, then the value is valid, because the starting age of E (2) is greater than binding age. If the variable is bound after or-node a, then its age will be greater than 1, and the binding would not be valid for BA E. The same variable cannot be accessed in BA C, the BA for the other and-arc, because only IAP is exploited. After the completion of the and-thread, the age is set to 4 initially for BA C, so all bindings made in BA B and C would be valid for BA D. Also, note that any search for the binding of the variable would stop once BA A is reached, as it has the same starting age as the variable's age. The search will not terminate early because the BAs following it (B to G) all have greater starting ages.

3.4 Arrangement of Markers

We shall assume that the Fire model will be using a marker-based distributed stack scheme [17, 30] for management of the stacks. With the introduction of or-parallelism, the parcall marker, at least as used in DASWAM, is not sufficient to manage the and-thread. In DASWAM, the parcall marker is used to co-ordinate the execution of goals in a CGE. With only and-parallelism, each CGE can have one and-thread being executed at any time; and in DASWAM, the and-thread is handled by a parcall marker. With the addition of or-parallelism, more than one and-thread can be executing, and modifications to DASWAM's marker scheme are needed to cope with this. One method to do this is to introduce alternative slots for the different and-arcs in the parcall marker in a similar way to alternative bindings, but this can be very complex to manage and implement, especially the way in which alternatives are represented in the Fire model. Another method is to use extra data-structures to represent the different and-threads, as was done in the author's and/or simulator. This can be done without the allocation of more markers by extending some of the existing markers to handle the extra functionality. For example, when a new or-branch is started inside a CGE, this means the creation of a new and-thread and, if the branch is picked up by a worker to run in parallel, then a new marker would be allocated before the new task is started. If this marker is extended with slots for remaining and-arcs that the and-thread has to compute separately, then it can act as a parcall marker for the part of this new and-thread that is not shared. The use of these *or-parcall markers* (OPC) is shown in Figure 6.

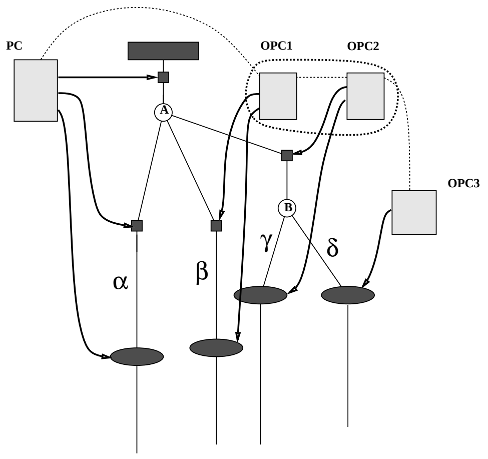


Figure 6: Or-parcall markers

In the Figure, the leftmost and-thread is managed by a parcall marker, marked "PC" in the figure, as in DASWAM. The or-parallel node A produces two new or-branches, and thus two new and-threads. If these are picked up by workers and executed in or-parallel, then or-parcall markers will be allocated. These are marked as "OPC1" and "OPC2" in the figure and are grouped together as they originate from the same or-parallel node. These OPCs have slots for the rest of their and-threads: the or-parallel node A occurs in the leftmost and-arc, and the and-thread contains two and-arcs. So for each new and-thread, a new and-arc is started and managed by the OPCs. (β for OPC1 and γ for OPC2). An or-parallel node B occurs in the and-arc γ, which starts a new and-thread, δ. This in turn is managed by a new or-parcall marker, OPC3, This only has to manage the the and-arc δ, because there are no further and-arcs in the and-thread.

The or-parallel markers are linked together in their left to right order, starting with the leftmost parcall marker for the CGE. This is shown in Figure 6 by dotted lines. Each task has a pointer to its or-parcall marker, and so when the task creates or-work, the new or-parcall marker can be added to the appropriate place in the chain in constant time. The connection of the or-parcall markers preserves the topology of the search-tree, allowing for side-effects which requires a notion of leftness of the tree, and will also allow the propagation of signals which may be needed when the CGE is killed.

With nested CGEs, something more is needed: as an or-branch started inside a CGE can continue beyond the and-thread. This can be handled by creating new parcall markers for each or-branch for each level of CGE, but it can also be handled by adding parcall markers slots to the join markers (join markers are allocated when an and-thread is completed). These extra slots are used to represent the remaining and-arcs in the outer and-thread, as shown in Figure 7, where there is a nesting of two CGEs, with an or-parallel node in the inner CGE, which results

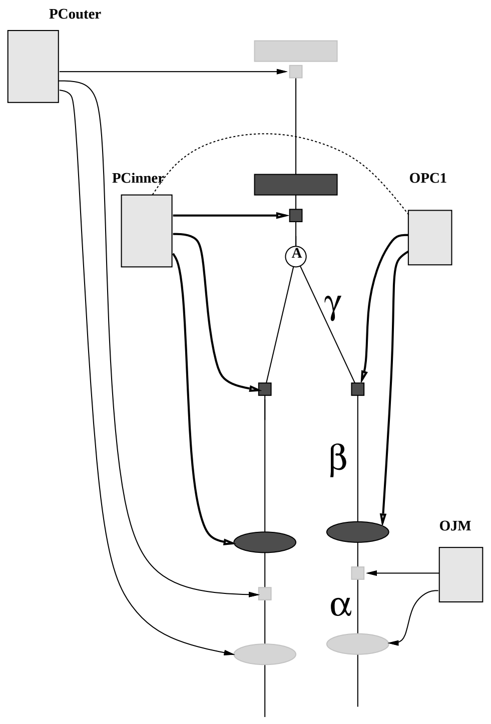


Figure 7: Join-parcall markers in nested CGEs

in two and-threads for both the inner and outer CGEs. The second and-thread in the inner CGE is handled by a or-parcall marker, OPC1, and the join marker of this and-thread (OJM) is used to handle the second outer and-thread. With this arrangement, and-parallel work in this second outer and-thread (the and-arc α in the figure) would not be available until the second inner and-thread completes and allocates the join marker. This and-parallel work could have been made available earlier in a more eager scheme, when the or-branch is started. This can be done by allocating the marker for the outer and-thread (the equivalent to OJM) as soon as the or-branch is taken from or-parallel node A, that is, in addition to allocating OPC1, the equivalent of OJM would be allocated as well. Note that there can be an unbounded number of markers allocated in such a scheme, one for each level of CGE that the or-work is nested in, and so starting an or-branch in such an eager scheme can be expensive. Also, these additional work can be highly speculative, as discussed in the next section. Therefore, the marker scheme developed here avoid starting such work early. In fact, although β can be started as soon as OPC1 is allocated, it seems useful in many situations to actually not execute β until the remainder of the second and-arc, γ is finished. This was the execution model used in the simulator [29], and the data from it indicated that this avoided much speculative work. It is also certainly the case that this can prevent useful parallelism from being exploited, so in Fire model, the possibility for exploiting this parallelism is left open.

3.5 Scheduling of work

Scheduling of work can be very simple in the Fire model, because of the constant-time task switching: as in an and-parallel only implementation such as &-Prolog and DASWAM, the stack space for the common and-arcs in an and-thread are shared through the distributed stacks, and the different or-alternatives (if any) are stored in BAs associated with the ANT as in PEPSys, so all that needs to be done for a worker when starting a new task is to link into the appropriate distributed stack segments as in an and-parallel only system, and to link to the appropriate parent BA: both done by setting a pointer.

As for scheduling itself, the same mechanism as used in RAP-WAM [17] can be used here: work (both or-work and and-work) can be pushed onto work stacks associated with the workers (if desired, each worker can have separate and- and or- work stacks, to allow for more sophisticated scheduling). Other workers can then steal work from these stacks. Unlike Aurora and Muse (and hence their derivatives ACE, SBA, and PBA), there is no need for a complex scheduler to select appropriate work. The motivations for the various (non-speculative) schedulers for

Aurora and Muse was to allow the selection of "good work" because task-switching is non-constant time. With a constant time task switching, there is no longer any strong reason to prefer one or-goal over another, and the simulation study of [32] support this in that the selection strategy made only small differences to the speedups, even in the presence of constant-time overheads.

The above does not consider the issue of speculativeness: e.g. or-speculativeness (work to the right of a cut, which might not be executed because of the cut), and-speculativeness (non-leftmost and-work, which might not be executed because of failures of goals to its left), and the additional speculativeness from the way and/or work are combined implementationally. These are separate issues from the fundamental scheduling problem (and may indeed require more than the selection of the "right" work to deal with them, as for example the most effective means of dealing with or-speculativeness to date in or-parallelism is to suspend speculative work and moving to less a speculative one [4]). These issues are generally best dealt with later when there is an implementation to experiment with. Note that the constant time task switching ability favours Fire model in that a speculative task can be suspended, and the worker moved to the less speculative task immediately without having to traverse the ANT to get there as in the non-constant time task switching models.

In addition, the Fire model does have properties that makes it much easier to deal with a form of speculativeness that is only found in nested and/or work. This will be discussed in the next section. Note that this should be considered an optimisation on the basic scheme, and is only presented here because it presents another potentially useful aspect of the Fire model.

3.6 Handling of speculative nested and/or work

Consider:

```
foo :- a(1000000).

a(0) :- or.
a(N) :- N1 is N - 1, a(N1) & b.

or :- fail.
or :- fail.
or.

b.
```

where an or-parallel node is created after a million level of nesting of CGEs: the node has three alternatives, two of which lead to failures. If these alternatives are taken, then in the more eager scheme, it would mean that and-work for each of the million CGEs enclosing this inner CGE would also be made available, although in two cases there are failures and the outer and-work (the million execution of `b`s) would be wasted. A study of the behaviour of and- and or-parallelism in Prolog programs [29, 31] suggests that there are many programs contain many or-branches inside CGEs of which only one lead to success, while the other branches are relatively short and fails. This suggests that if and-parallel work are started for such or-branches, especially ones that are deep inside many levels of CGEs, would be very speculative.

Another problem is that it is difficult in general to know in advance which or-branch would lead to failure, so in a team oriented approach, where a team is basically working on an and-thread, there can be considerable wastage. Consider the example program above where the first two or-branches of `or/0` fails. In a team based approach, the team working on the leftmost and-thread would have created a million `b/0` before failing, and the work done on these million `b/0` would be thrown away, even though they give identical results in the other and-threads because they are independent.[9]

In the Fire model, workers working on a particular and-thread, which could be considered as a team, are only very loosely bound together: they work on different parts of the and-thread, which have different physical binding areas associated with them. It is thus very easy to move a task, working on a and-arc, to a different and-thread, should failure occur elsewhere in the and-thread. Consider Figure 8, where there are two nested CGEs, with an or-parallel node A in the first and-arc. Assume that the left-branch of A leads to failure, but the right branch succeeds. If copying was not available, then all the work done on the rest of the and-thread associated with the failing branch

[9] Note that these are the goals that would be computed once in a reuse scheme. These goals are not really "reused" in the traditional sense, because they would only be computed once sequentially. An and/or parallel system may compute them more than once because of speculative computation.

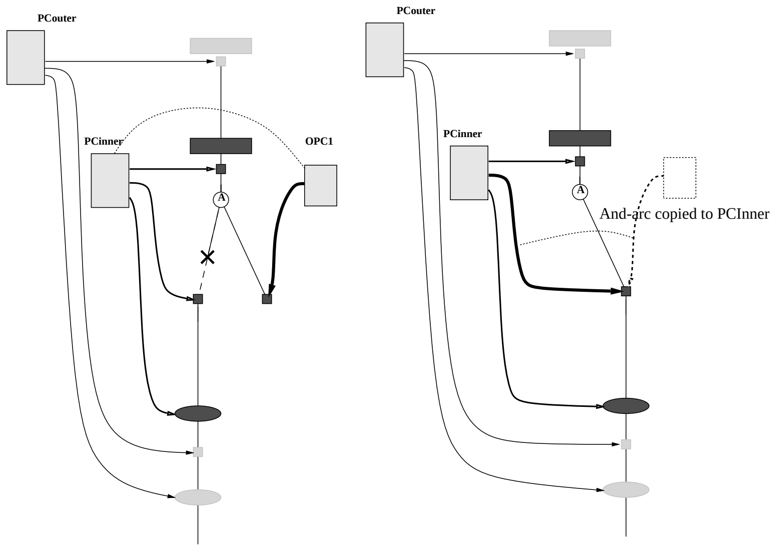


Figure 8: Moving and-arcs

(i.e. both PCinner and PCouter) would be wasted. However, if the task originally managed by OPC1 (indicated with a thicker arrow) can be moved to be manage by PCinner, replacing the failing and-arc, as shown in the right figure, then the work in PCinner and PCouter need not be thrown away. This can be done easily within the Fire model: only the pointers in the appropriate slot in OPC1 has to be copied to PCinner's appropriate slot. The effect would be as if all the other branches have moved (recall that the physical BAs are not yet linked at this point as the and-thread has not yet completed). Another change is needed: the task executing the moved and-arc has a pointer to OPC1, this has to be changed to point at PCinner. This last operation can be done with a single pointer change if an extra level of indirection is added to the pointer. Obviously, one possible problem is that there may be other and-arcs running in OPC1. This is however not such a big problem with the Fire model, because of the control of speculative computation such that in the non-leftmost and-thread, computations in the outer CGEs are not started immediately, as described in the previous section. Nevertheless, some synchronisation strategy would need to be developed.

3.7 Cost

If the system is running or-parallel goals only, then only one BA per or-branch would be created, so the system reduces to PEPSys' or-parallel binding model. With IAP goals only, only one and-thread would be executing at a time, so there is only one conceptual BA at a time, although physical BAs may need to be created during some of the task-switches, which would add a constant time overhead to initialise the BA. The BAs would not be used, because none of the variables will have alternative bindings, so again the cost will be reduced close to that of an and-parallel only system.

One potential problem is that the number of (physical) BAs are increased from the purely or-parallel PEPSys model, and it is possible that finding a binding would require the checking of many BAs. However, BAs can continue to be used for a new task if that task is simply further down the and-thread. In the limit, if only one worker is running the program, there will always be only one BA, no matter what the shape of the ANT is. It is likely that with only a few workers — and the most widely available parallel machines in the near future are likely to have only a few processors — the number of BAs that actually need to be created will be quite small. Also, the increased number of BAs will only be a serious problem if there are deep nesting of and and or-parallelism, and if bindings are usually not found locally. The caching of bindings into the local BAs after the first access should also reduce the impact of this problem. Actual measurements of behaviour of real programs in an actual or simulated system would be needed to see if this is a serious problem or not.

4 Comparison to Related Work

The Fire model is an implementation scheme of the "or-under-and, no goal reusage" execution model. As such, the actual execution model is very similar to previous proposed implementation schemes of the same execution model, such as ACE [14] and PBA [16], except for whatever scheduling restrictions that are imposed in one implementation scheme or the other due to implementation considerations. By associating the alternative bindings with structures representing the search-space rather than storing them with the team, workers can task-switch very quickly by simply jumping to a new piece of work once it is found. In ACE/PBA/SBA, a worker need to move from an old piece of work to a new one when task-switching, and the support needed for this, especially in the context of an and/or parallel system, introduces many overheads. Also, the association of the alternative bindings with a team of worker also complicates scheduling: although it is possible to move workers in and out of a team, deciding when to do so (and there may also be overheads associated with doing this) does add extra complications to the implementation. With the Fire model, these complications and overheads are avoided, at the expense of non-constant time access to *some* variable bindings. We believe that this trade off can be worthwhile, because much of the complexity in the previous implementation schemes are associated with handling the scheduling and managing the alternative bindings. As an example, both ACE and SBA restrict the selection of or-work to or-parallel nodes where the whole subtree on their left has been completed, because the complexity of dealing with copying unfinished segments of stack.[10]

In addition, as discussed in section 3.6, the Fire model may offer some advantage in the handling of speculative work in nested and/or situations. In a team-based system, it is much more difficult to move an and-arc from one and-thread to another, because the alternative bindings of an and-thread are stored in a common data-structure (for example, the binding arrays of PBA and SBA), so such movements will involve non-constant time operations of copying this information from one team to another. The author's previous simulation study have suggested that there can be a lot of speculative or-parallelism in these situations, thus the ability to handle such situations without generating too much wasted work is important.

The control of speculative computation is also important. This could avoid a lot of wasted work, and in addition make moving tasks easier. It is also possible to control speculative work via scheduling, as suggested by Santos Costa [11]. However, the work would still need to be made available, and this can involve an unbounded amount of work, in the allocation of markers for each nested CGE level. Also, this requires a smart (and presumably complex) scheduler, while the aim of the Fire model is to make scheduling as simple as possible, so that task switching can be done as quickly as possible.

As mentioned, a scheme to handle IAP via reusage of and-goals was proposed for PEPSys [3]. Because this implementation scheme is based on reusage, it is fundamentally different from the Fire model. The various or-parallel solutions to and-goals are combined to give the cross-product via data-structures called join-cells. These join-cells are also used to connect the binding areas. The join-cells can only handle two and-goals, and no description was given on how a CGE containing more than two and-goals would be handled (presumably via some form of multiple join-cells); also no description of how the ages (OBL) are to be handled across the different BAs (hash-windows) were given in [3]. Because or-parallelism is selective (i.e. not all search-tree nodes need be or-parallel nodes, as in the Fire model), the cross product has to be computed each time (with new join-cells) a non-or-parallel solution to an and-parallel goal is generated, with solutions to the right-hand and-goal computed again for the cross-product. Recomputation is used only in this very limited case, and the final solutions are still produced via cross-products, unlike the Fire model. It is also not clear how this computation model can be extended to CGEs with more than two and-goals, because the treatment of the left- and right- goals are asymmetric.

Compared to other and/or parallel execution schemes, the "or-under-and, no goal reusage" avoids many of the problems associated with goal reusage, as discussed in section 1.1, while at the same time, it does not place much restriction on when and- and or-parallelism can be exploited. This could be important, as it allows the great flexibility in how parallelism is exploited in a program. Although the author's simulation study and subsequent experience suggest that programs which contain significant amounts of both forms of parallelism are rare, restricting the situations where either forms of parallelism can be exploited can lead to situations where potentially significant parallelism exists, but is unexploitable by the model, because it is difficult to foresee how the two forms of parallelism are arranged. Another approach may be to allow the user to have more control over where various forms of parallelism would be exploited: one possibility might be to have and-parallelism only by default , with special predicates for exploiting or-parallelism, in which the way and- and or-parallelism is less general than the "or-under-and" model, and thus easier to implement. Of course, there can always be applications that contain both

[10] Personal communications with E. Pontelli, March 1997.
[11] Personal communications with V. Santos Costa, March, 1997

and- and or-parallelism — not many large scale programs' behaviour have been examined in parallel yet.

Another alternative to exploiting both and/or- parallelism together is to abandon Prolog, and do it in a different language from Prolog, and parallelism has to be programmed explicitly. Such an approach is taken with AKL[19]. While abandoning Prolog may have merits, especially if the aim is to design a language for concurrent programming, this is somewhat different from the aim of this research, which is to exploit parallelism in Prolog programs implicitly. In AKL, or-parallelism is achieved by copying, and in implementations of AKL [23], this has been shown to cause or-parallel programs to run significantly slower.

5 Conclusion

This paper outlined the Fire implementation model for combining and- and or-parallelism, which may have some advantages over existing models, because it simplifies task-switching, which is at the heart of much of the complexities associated with other models. Much remains to be done, including implementing the model, and extending it to deal with dependent and-parallelism.

6 Acknowledgments

The idea for the work discussed in this paper resulted from discussions about SBA with Eduardo Correia and Vítor Santos Costa during my visit to University of Porto in November, 1996. The work was helped by subsequent discussions with Eduardo, Vìtor, and Enrico Pontelli, especially during my visit to the State University of New Mexico in March, 1997. I would also like to thank Gopal Gupta for hosting me during the visit, where part of this paper was written.

References

[1] K. A. M. Ali and R. Karlsson. Full Prolog and Scheduling Or-Parallelism in Muse. *International Journal of Parallel Programming*, 19(6):445–475, December 1990.

[2] K. A. M. Ali and R. Karlsson. The Muse Approach to Or-Parallel Prolog. Technical Report SICS/R-90/R9009, Swedish Institute of Computer Science, 1990.

[3] U. C. Baron, J. Chassin de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, J.-C. Syre, and H. Westphal. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988, Volume 3*, pages 841–850, 1988.

[4] A. J. Beaumont and D. H. D. Warren. Scheduling Speculative Work on Or-parallel Prolog Systems. In *Logic Programming: Proceedings of the Tenth International Conference*, pages 135–149. The MIT Press, 1993.

[5] J. Bevemyr, T. Lindgren, and H. Millroth. Reform Prolog: the Language and its Implementation. In *Logic Programming: Proceedings of the Tenth International Conference*. The MIT Press, 1993.

[6] R. Butler, E. L. Lusk, R. Olson, and R. A. Overbeek. ANLWAM: A Parallel Implementation of the Warren Abstract Machine. Internal report, Argonne National Laboratory, U.S.A, 1986.

[7] J. Chassin de Kergommeaux. Measures of the PEPSys Implementation on the MX500. Technical Report CA-44, European Computer-Industry Research Centre, Arabellaastr. 17, D-8000 München 81, Germany, 1989.

[8] J. Chassin de Kergommeaux, P. Robert, and H. Westphal. An Abstract Machine for the Implementation of the PEPSys Model. Technical Report CA-26, European Computer-Industry Research Centre, 1987.

[9] J. S. Conery. *The AND/OR Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, University of California at Irvine, 1983. Available as technical report 204.

[10] I. Dutra. Strategies for Scheduling And and Or Work in Parallel Logic Programming Systems. In *Logic Programming: Proceedings of 1994 Internation Symposium*. The MIT Press, 1994.

[11] B. S. Fagin. *A Parallel Execution Model for Prolog.* PhD thesis, University of California at Berkeley, Nov. 1987.

[12] G. Gupta and M. V. Hermenegildo. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proceedings of ICLP'91 Workshop on Parallel Execution*, June 1991.

[13] G. Gupta and M. V. Hermenegildo. Recomputation based Implementation of And-Or Parallel Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992, Volume 2*, pages 770–782. Institute for New Generation Computing, June 1992.

[14] G. Gupta, M. V. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Logic Programming: Proceedings of the Eleventh Conference*, pages 93–110. The MIT Press, June 1994.

[15] G. Gupta and B. Jayaraman. Combined And-Or Parallelism on Shared Memory Multiprocessors. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference, 1989, Volume 1*, pages 332–349. The MIT Press, 1989.

[16] G. Gupta, V. Santos Costa, and E. Pontelli. Shared Paged Binding Arrays: A Universal Data Structure for Parallel Logic Programming. In *Proceedings of the NSF/ICOT Workshop on Parallel Logic Programming and its Environments*. University of Oregon, 1994.

[17] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, The University of Texas At Austin, 1986.

[18] M. V. Hermenegildo and K. J. Green. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In D. H. D. Warren and P. Szeredi, editors, *Logic Programming: Proceedings of the Seventh International Conference*, pages 253–268. The MIT Press, 1990.

[19] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In V. Saraswat and K. Ueda, editors, *Logic Programming: Proceedings of 1991 International Symposium*, pages 167–183. The MIT Press, 1991.

[20] L. V. Kalé. *Parallel Architectures for Problem Solving*. PhD thesis, State University of New York at Stony Brook, 1985.

[21] E. L. Lusk, R. Butler, T. Disz, R. Olson, R. A. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora Or-Parallel Prolog System. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988, Vol. 3*, pages 819–830. Institute for New Generation Computer Technology, 1988.

[22] D. Michie. "Memo" Functions and Machine Learning. *Nature*, 218:19–22, Apr. 1968.

[23] J. Montelius. *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages*. PhD thesis, Uppsala University, 1997.

[24] 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.

[25] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *Logic Programming: Proceedings of the Eighth International Conference*, pages 825–839, 1991.

[26] K. Shen. An Overview of DASWAM — An Implementation of DDAS. Technical Report CSTR-92-08, Computer Science Department, University of Bristol, 1992.

[27] K. Shen. *Studies of And/Or Parallelism in Prolog*. PhD thesis, Computer Laboratory, University of Cambridge, 1992.

[28] K. Shen. Initial Results from the Parallel Implementation of DASWAM. In M. Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*. The MIT Press, 1996.

[29] K. Shen and M. V. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In V. Saraswat and K. Ueda, editors, *Logic Programming: Proceedings of 1991 International Symposium*, pages 135–151. The MIT Press, 1991.

[30] K. Shen and M. V. Hermenegildo. Divided We Stand: Parallel Distributed Stack Memory Management. In E. Tick and G. Succi, editors, *Implementations of Logic Programming System*, pages 185–201. Kluwer, 1994.

[31] K. Shen and M. V. Hermenegildo. High-level Characteristics of Or- and Independent And-parallelism in Prolog. *International Journal of Parallel Programming*, 24(5):433–478, 1996.

[32] K. Shen and D. H. D. Warren. A Simulation Study of the Argonne Model for Or-Parallel Execution of Prolog. In *Proceedings of the Fourth Symposium on Logic Programming*. Computer Society Press of the IEEE, Sept. 1987.

[33] P. Szeredi. Performance Analysis of the Aurora Or-Parallel System. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference, 1989, Volume 2*, pages 713–732. The MIT Press, Oct. 1989.

[34] D. H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog – Abstract Design and Implementation Issues. In *Proceedings 1987 Symposium on Logic Programming*, pages 92–102. Computer Society Press of the IEEE, Sept. 1987.

[35] H. Westphal, P. Robert, J. Chassin de Kergommeaux, and J.-C. Syre. The PEPSys Model: Combining Backtracking AND- and OR-parallelism. In *Proceedings 1987 Symposium on Logic Programming*, pages 436–448. Computer Society Press of the IEEE, Sept. 1987.

[36] M. J. Wise. *Prolog Multiprocessors*. Prentice-Hall, 1986.