Or-Parallel Prolog Execution on Multicores Based on Stack Splitting

Rui Vieira Ricardo Rocha Fernando Silva

CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{revs,ricroc,fds}@dcc.fc.up.pt

Abstract

Many or-parallel Prolog computational models exploiting implicit parallelism have been proposed in the past. The Muse and YapOr systems are arguably two of the most efficient systems exploiting or-parallelism on shared memory architectures, both based on the environment copying model. Stack splitting emerged as an alternative model specially geared to distributed memory architectures as it basically splits the computation in such a way that no further, or just minimal, synchronization is required.

With the new multicore architectures, it just makes sense to recover the body of knowledge there is in this area and reengineer prior computational models to evaluate their performance on newer architectures. In this paper, we focus on the design and implementation of stack splitting in the YapOr system. Our aim is to take advantage of its robustness to efficiently implement stack splitting support using shared memory, and then be able to directly compare the original YapOr with the YapOr using stack splitting. We consider two splitting models, *vertical splitting* and *half splitting*, and have adapted data structures, scheduling and incremental copy procedures in YapOr to cope with the new models. Experimental results, on a multicore machine with 24 cores, show that YapOr using stack splitting is, in general, comparable to the original YapOr, obtaining in some cases better performance than with only environment copying.

Categories and Subject Descriptors D.1.6 [*PROGRAMMING TECHNIQUES*]: Logic Programming

General Terms Design, Languages, Performance

Keywords Logic Programming, Or-Parallelism, Stack Splitting

1. Introduction

Prolog programs, whose semantics is based on First Order Logic, naturally exhibit *implicit parallelism*. The advantage of implicit parallelism is that one can develop specialized run-time systems to transparently explore the available parallelism in programs, thus freeing the programmers from the cumbersome task of explicitly identifying it. One important source of parallelism arises from the simultaneous evaluation of a Prolog goal against all the predicate clauses that match that goal. This form of parallelism is called *or-parallelism*.

One basic problem with implementing or-parallelism is how to represent, efficiently, the multiple bindings for the same variable produced by the parallel execution of the alternative matching clauses. Two of the most prominent binding models that have been proposed, *binding arrays* [4] and *environment copying* [1, 6], have been efficiently used in the implementation of or-parallel Prolog systems on shared memory architectures.

Another main difficulty in the implementation of any parallel system is to devise an efficient strategy to assign computing tasks to idle workers waiting for work. A parallel Prolog system is no exception as the parallelism that Prolog programs exhibit is usually highly irregular. Achieving the necessary cooperation, synchronization and concurrent access to shared data structures among several workers during their execution is a difficult task. The *stack splitting* model [2, 5, 10] provides a simple, clean and efficient method to accomplish work sharing among workers. It successfully splits the computation task of one worker in two fully independent tasks, and thus was first introduced aiming at distributed memory architectures [7, 9].

Recent advances in parallel architectures have made our personal computers parallel with multiple cores sharing the main memory. The multicores and clusters of multicores are now the norm, and exploiting implicit parallelism in a transparent way is a quite relevant research direction to take. Although many parallel Prolog systems have been developed in the past [3], evaluating their performance or even the implementation of newer computational models specialized for the multicores is still open to further research.

In this paper, we focus on the design and implementation of stack splitting in the YapOr system [6], a Prolog system that exploits or-parallelism based on the environment copying model. Our approach is to benefit from prior research on the development of YapOr and extend it to efficiently support stack splitting on multicore architectures. The work developed allowed us to make some contributions, not only on the implementation of two stack splitting models, the *vertical splitting* and *half splitting* models, but also on the update and creation of new mechanisms, mainly related with the incremental copy technique [1]. Experimental results on a multicore machine with 24 cores, show that YapOr with stack splitting is, in general, comparable to the original YapOr based on environment copying, and, in some cases, even surpasses it.

The remainder of the paper is organized as follows. First, we introduce the general concepts of environment copying and stack splitting. Next, we describe the vertical and half splitting models and discuss the major implementation issues in YapOr. We then present experimental results using a set of benchmark programs

DAMP'12, January 28, 2012, Philadelphia, PA, USA.

widely used to assess the performance of or-parallel Prolog systems. Finally, we advance some conclusions and further work.

2. The YapOr System

In or-parallelism, alternative branches of the search tree may be executed in parallel and thus may result in conflicting bindings for shared variables. The YapOr system addresses this problem by implementing the environment copying model so that conflicting bindings can be easily discerned.

2.1 Environment Copying

In the environment copying model, each worker[1] keeps a separate copy of its own environment, but in an identical address space, thus enabling it to freely store assignments to shared variables without conflicts. Every time a worker shares work with another worker, all the execution stacks are copied to ensure that the requesting worker has the same environment state down to the search tree node[2] where the sharing occurs. To reduce the overhead of stack copying, an optimized copy mechanism, called *incremental copy*, was devised [1]. It takes advantage of the fact that the requesting worker may already have traversed a part of the path between the root node and the youngest common node of both workers. Therefore, it does not need to copy the stacks referring to the whole path from root, but only the stacks starting from the youngest common node of both workers.

As a result of the environment copying, each worker can proceed with the execution exactly as a sequential engine, with just minimal synchronization with other workers. Synchronization is mostly needed at *work sharing operations* to ensure that each untried alternative is only explored once. Shared nodes are represented by *or-frames*, a data structure that workers must access to obtain the untried alternatives, point in which mutual exclusion is enforced. All other data structures, such as the environment, the heap, and the trail do not require synchronization.

2.2 Work Sharing

Similarly to the Muse system [1], YapOr also uses a *bottommost scheduling strategy* that has proved quite effective. Work sharing takes place at deep choice points, and it has the effect of exposing the whole private region of a worker when it shares work. This seems to favor increased granularity of tasks, thus avoiding too frequent requests.

Whenever an idle worker Q makes a request for work to a busy worker P, the work sharing procedure is activated to share all private nodes of P with Q. P accepts the work request only if its private work load is above a given *threshold value*. Accomplishing this operation involves the following stages:

Sharing loop. This stage handles the sharing of the private nodes of P. For each private node, a new or-frame is allocated and the access to the unexplored alternatives previously done through the `cp_alt` pointer in the private choice points, is now done through the new or-frame which acts as an interface for work retrieval. The access to the alternatives is made through the or-frame field `OrFr_alt`, and the private `cp_alt` pointers are updated to a `getwork` pseudo instruction. All the private nodes have now a corresponding or-frame, which are sequentially chained through the fields `OrFr_next` and `OrFr_nearest_livenode`. The `OrFr_nearest_livenode` field is used to optimize the search for work in the shared region. The membership field in each of the chained or-frames is also updated to indicate that P and Q are sharing the corresponding choice points.

Updating old or-frames. Next, the old or-frames on P's branch are updated to include the requesting worker Q in the membership field in the frames starting from P's current `top_or_frame` til Q's `top_or_frame`.

Updating top or-frames. Finally, the new top or-frames in each worker are set, and since all shared work is available to both workers, both get the same `top_or_frame`. As we will see next, this is not the case for stack splitting, and the `top_or_frame` variable of Q is set accordingly to the splitting model being implemented.

3. Stack Splitting

Stack splitting is a model that was introduced to target distributed memory architectures, thus aiming to reduce the mutual exclusion requirements from both binding arrays and environment copying models when accessing shared branches of the search tree. It accomplishes this, by defining a work sharing strategy in which all available work is *fully divided* between the two sharing workers. In practice, it is a refined version of the environment copying model in which the synchronization requirement was removed by the preemptive split of all untried alternatives among both workers at the moment of sharing. The splitting is such that both workers will proceed, each executing its branch of the computation, without any need for further synchronization.

The original stack splitting approach [2] proposed two main models for dividing the work among the sharing workers P and Q: *vertical splitting* and *horizontal splitting*. Vertical splitting divides the choice points alternately between the two sharing workers P and Q. Horizontal splitting, on the other hand, alternately divides the unexplored alternatives in each choice point between the two workers P and Q.

In this paper, we will focus on the implementation of vertical splitting in YapOr as well as in the implementation of another splitting model [10], which we named *half splitting*. Both models split the choice points in two halves. This contrasts with the horizontal and diagonal [7] models, in which the split is based on the unexplored alternatives.

Figure 1 illustrates the effect of the vertical and half splitting models in a work sharing operation. The sharing starts, Figure 1(a), with the request for work made by the idle worker Q to a busy worker P. Figure 1(b) shows the effect of vertical splitting in which P keeps its current choice point and alternately divides with Q all remaining choice points up to the root choice point. Figure 1(c) illustrates the effect of the half splitting model after a sharing operation.The bottom half is for worker P and the half closest to the root is for worker Q. After completing the division process, the stacks are copied to Q and Q is set to begin its own execution.

4. Implementation

The implementation of stack splitting in the YapOr system requires modifications on some data structures and procedures, namely:

- Work chaining of or-frames. With stack splitting, each worker has its own work chaining sequence. We make use of the `OrFr_nearest_livenode` field stored in the or-frames to do this chaining. At work sharing, the sharing worker adjusts the `OrFr_nearest_livenode` fields so that two separate chains are built corresponding to the intended split of the work.
- Scheduling chaining of or-frames. In order to reuse the YapOr's general execution model, namely its scheduler, the or-frames are still chained through the `OrFr_next` fields.

[1] A worker corresponds to a system process executing a Prolog engine.

[2] A search tree node corresponds to a choice point in the stack.

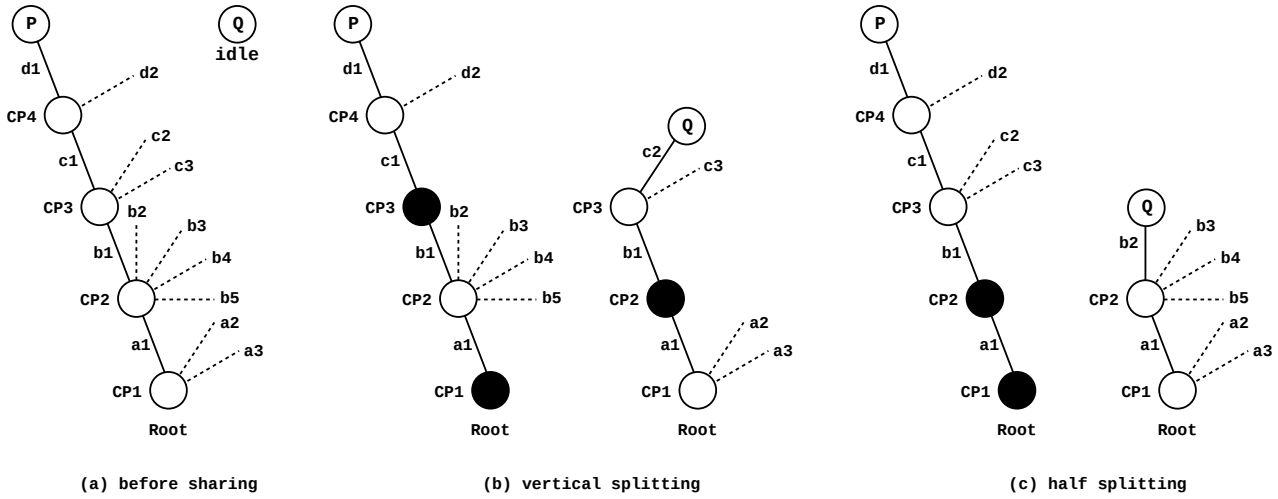

Figure 1. Vertical and half stack splitting models.

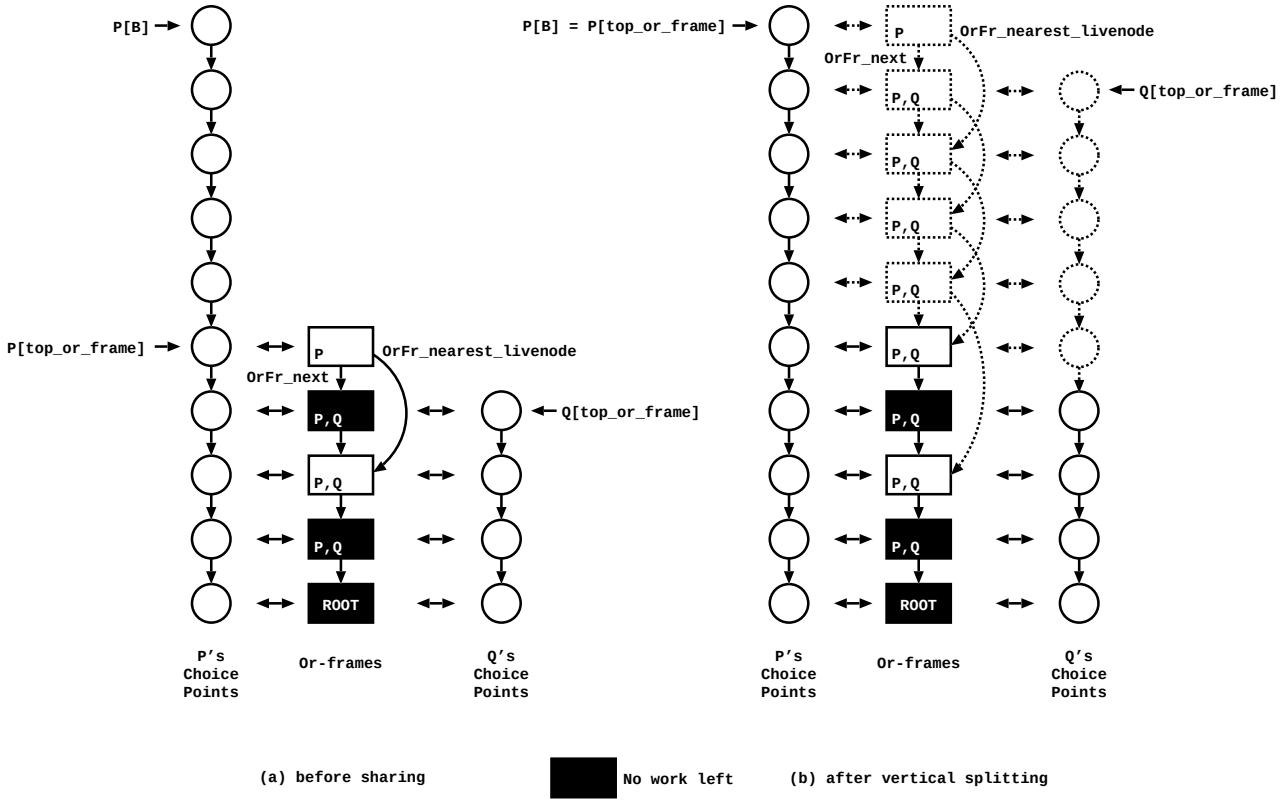

Figure 2. Work sharing with vertical splitting.

- Membership mechanism. YapOr uses a bitmap in every or-frame to define the set of workers that own or act upon the corresponding choice point. With stack-splitting, although the work is split and we could possibly avoid this membership information, we still keep it as it allows the reuse of YapOr's scheduler to best position a worker that is requesting work, and avoid to implement a new strategy.

Next we detail the implementation of vertical and half splitting as well as the incremental copy mechanism.

4.1 Vertical Splitting

The vertical splitting strategy follows a pre-determined work splitting scheme in which the chain of shared choice points is alternately divided between the two sharing workers. At the implementation level, we use the `OrFr_nearest_livenode` field in order to generate two alternated chain sequences in the or-frames, and thus divide the available work in two independent execution paths. Workers can share the same or-frames but they have their own independent path without caring for the or-frames not assigned to them. Figure 2 shows an example of work sharing with vertical splitting and Figure 3 presents the pseudo-code that implements the sharing stages described previously.

```
next_fr = NULL
nearest_fr = NULL
current_cp = P[B]  // P's youngest choice point
while (current_cp != P[top_cp])
  // P[top_cp] is P's youngest shared choice point
  current_fr = alloc_or_frame(current_cp)
  init_lock_field(OrFr_lock(current_fr))
  if (next_fr)
    OrFr_next(next_fr) = current_fr
    add_to_bitmap(P & Q, OrFr_member(current_fr))
  else
    add_to_bitmap(P, OrFr_member(current_fr))
  if (nearest_fr)
    OrFr_nearest_livenode(nearest_fr) = current_fr
  nearest_fr = next_fr
  next_fr = current_fr
  // move to the next choice point on stack
  current_cp = cp_b(current_cp)

// connecting with the older or-frames
// P[top_or_frame] is P's youngest or-frame
if (next_fr)
  if (P[top_or_frame] == ROOT_FRAME)
    OrFr_nearest_livenode(next_fr) = DEAD_END
  else
    OrFr_nearest_livenode(next_fr) = P[top_or_frame]
  OrFr_next(next_fr) = P[top_or_frame]
if (nearest_fr)
  if (P[top_or_frame] == ROOT_FRAME)
    OrFr_nearest_livenode(nearest_fr) = DEAD_END
  else
    OrFr_nearest_livenode(nearest_fr) = P[top_or_frame]

// continuing vertical splitting
if (next_fr = NULL)
  current_fr = P[top_or_frame]
nearest_fr = OrFr_nearest_livenode(current_fr)
while (nearest_fr != DEAD_END)
  OrFr_nearest_livenode(current_fr) =
      OrFr_nearest_livenode(nearest_fr)
  current_fr = nearest_fr
  nearest_fr = OrFr_nearest_livenode(current_fr)
```

Figure 3. Work sharing procedure for vertical splitting.

As illustrated in Figure 2, the first noticeable difference from the previous description is how the `OrFr_nearest_livenode` field is now connected. Instead of sequentially connected, the `OrFr_nearest_livenode` field is now double spaced connected during the or-frame creation process.

Starting from *P*'s youngest choice point, the work sharing procedure starts by traversing all *P*'s private choice points and creates a corresponding or-frame by calling the `alloc_or_frame()` procedure. In the pseudo-code, the `current_fr`, `next_fr` and `nearest_fr` variables represent, respectively, the or-frame allocated in the current step, the or-frame allocated in the previous step, which is used to link to the current or-frame by the `OrFr_next` field, and the or-frame allocated before the `next_fr`, which is used as a double spaced frame marker in order to initiate the `OrFr_nearest_livenode` fields. To continue the loop, the `nearest_fr` is updated to the `next_fr`, and the `next_fr` is updated to the `current_fr`.

The sequentially created or-frames are connected through the `OrFr_next` fields. Thus, if there is a defined `next_fr`, its `OrFr_next` field is made to point to the `current_fr`. Moreover, If `nearest_fr` is defined, then its `OrFr_nearest_livenode` is also assigned to the `current_fr`. For the top choice point, the or-frame is initialized with just the owning worker *P* in the membership bitmap. The other or-frames are initialized with both workers *P* and *Q*.

Next, follows the connection of the last newly allocated or-frames with the older and already stored or-frame structure. Here, consideration must be given to the condition of `P[top_or_frame]`, which points to *P*'s current top or-frame, being the root or-frame or just an ordinary or-frame. If it is a root or-frame, the `OrFr_nearest_livenode` fields of the auxiliary or-frames are assigned with the value `DEAD_END`. If not, they are assigned to *P*'s top or-frame. The `DEAD_END` assignment marks the ending point for unexplored work.

Finally, we need to decide where to continue the application of the vertical splitting algorithm for the old shared nodes. If no private work was shared, which means that we are only sharing work from the old shared nodes, the starting or-frame is *P*'s current top or-frame. Otherwise, if some new or-frame was created, the starting or-frame is the last created frame in the sharing loop stage, which was connected to *P*'s current top or-frame in the previous step. Either way, this serves the decision to elect the or-frame where the continuation of vertical splitting, guided through the `OrFr_nearest_livenode` field, should continue.

The procedure then traverses the old shared frames until reaching a `DEAD_END`. At each frame lies a reconnection process of the `OrFr_nearest_livenode` field. The `OrFr_nearest_livenode` of the starting or-frame (`current_fr`) is first saved to the `nearest_fr` variable. Then, while the `nearest_fr` variable is not a `DEAD_END`, the `nearest_fr`'s `OrFr_nearest_livenode` is assigned to the `current_fr`'s `OrFr_nearest_livenode`, and the process continues by moving the `current_fr` to the `nearest_fr`.

Upon completion of the sharing process, follows the stack copying phase. In some situations of stack splitting, there is no need for any copy at all, and a backtracking action is enough to place the requesting worker ready for execution.

4.2 Half Splitting

The half splitting model partitions the shared chain of choice points in two consecutive and almost equally sized parts, which are chained through the `OrFr_nearest_livenode` field of the corresponding or-frames. The choice points are numbered sequentially and independently per worker to allow the calculation of the *relative depth* of the worker's assigned choice points. In order to support this numbering of nodes, a new field, named *split counter*, was introduced in the choice point structure.

Figure 4 illustrates an example of work sharing with half splitting where the sharing worker *P* has six choice points in its path. Three of these choice points are then assigned to the requesting worker *Q*. *P* updates the split counter on its half of the choice points

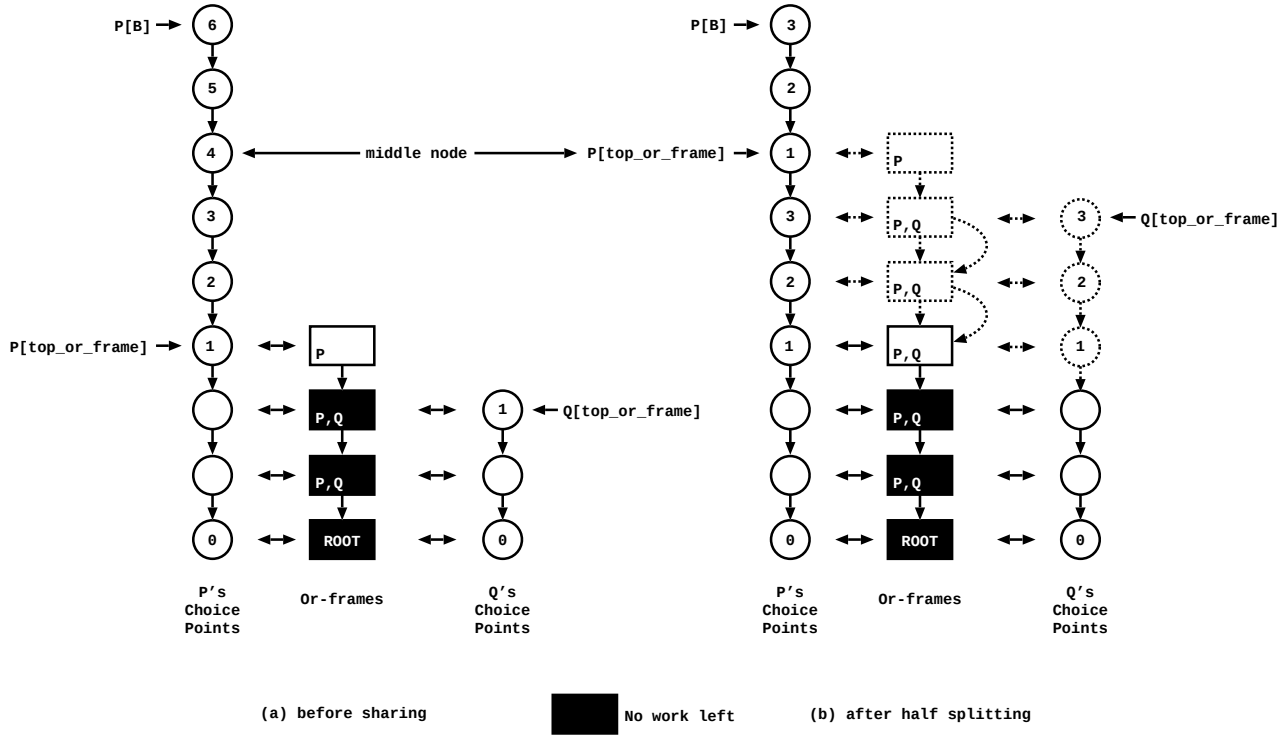

Figure 4. Work sharing with half splitting.

starting from *P*'s youngest choice point (`P[B]`), in decreasing order, until the middle choice point in *P*'s initial partition, which gets a split counter value of 1. Figure 5 shows the pseudo-code for this procedure.

```
current_cp = P[B]  // P's youngest choice point
// cp_sc is the split counter field
split_number = cp_sc(current_cp) / 2
while (cp_sc(current_cp) != split_number + 1)
  cp_sc(current_cp) = cp_sc(current_cp) - split_number
  // move to the next choice point on stack
  current_cp = cp_b(current_cp)
cp_sc(current_cp) = 1
```

Figure 5. Updating the split counter.

After updating the split counter, the `current_cp` variable points to the middle node. Here, we can distinguish two different situations. In the case where there are more old shared choice points than private in *P*'s branch, the middle node is already assigned with an or-frame. Thus, there is no need for the sharing loop stage, the middle frame is assigned to a `DEAD_END`, and the requesting worker *Q* is excluded from all or-frames from the top frame til the middle frame. The `DEAD_END` in the middle frame marks the end of *P*'s newly assigned work. Figure 6 shows the pseudo-code for this procedure.

The second situation occurs when the middle node is private in which the sharing loop stage is performed. Starting from the middle node, the remaining choice points are then updated to belong to *Q*, which includes allocating and initializing the corresponding or-frames (see Figure 4(b)).

Finally, the top frame of the sharing worker *P* is assigned to be the middle frame and, the top frame of the requesting worker *Q* is assigned to be the frame pointed by the middle frame's `OrFr_next` field.

```
// current_cp is the middle node
middle_frame = cp_or_fr(current_cp)
if (middle_frame)
  OrFr_nearest_livenode(middle_frame) = DEAD_END
  // P[top_or_frame] is P's youngest or-frame
  current_frame = P[top_or_frame]
  while (current_frame != middle_frame)
    remove_from_bitmap(Q, OrFr_member(current_frame))
else
  // sharing loop stage
```

Figure 6. Checking if the middle node is already shared.

4.3 Incremental Copy

We now introduce the practical aspects for implementing stack splitting with support for the incremental copy technique.

4.3.1 Copy Ranges

In YapOr, the incremental copy process includes copying everything in *P*'s stack segments that *Q* doesn't have. With stack splitting, we only need to copy the interval between *Q*'s top before and after sharing for the global and local stacks. For the trail stack, the process is similar to YapOr's implementation and the same interval of the trail stack is copied. Figure 7 shows the stack segments to be copied for our stack splitting implementation with the incremental copy technique.

In YapOr, the copy ranges can be defined before starting the work sharing procedure since *P*'s current state will be fully shared with *Q*. For stack splitting, only some of the copy ranges can be determined before starting the work sharing procedure, such as:

```
start_global = Q[old_top_cp->cp_h]
end_local = Q[old_top_cp]
start_trail = P[TR]
end_trail = Q[old_top_cp->cp_tr]
```

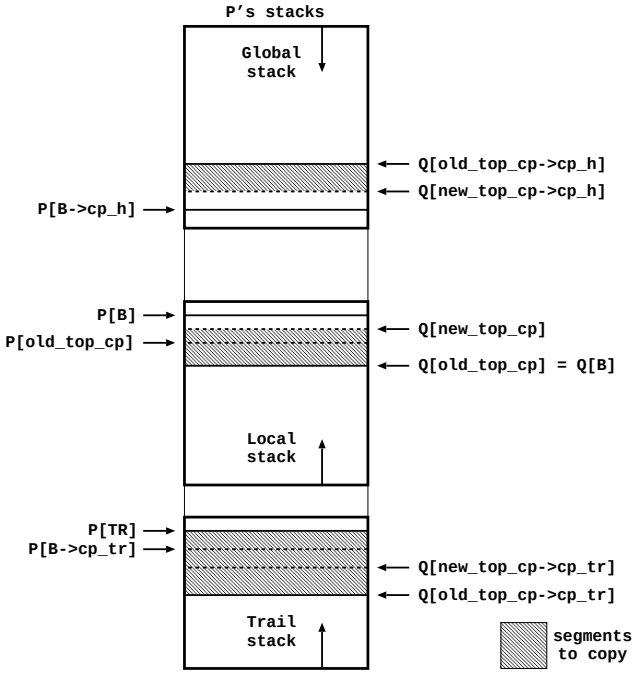

Figure 7. Stack segments to copy for stack splitting with incremental copy.

The other two ranges:

```
end_global = Q[new_top_cp->cp_h]
start_local = Q[new_top_cp]
```

can only be determined after the `new_top_cp` is known.

For vertical splitting, if P has private work, the `new_top_cp` is assigned with the second choice point in P's choice point set (`P[B->cp_b]`). If there is no private work, the `new_top_cp` is assigned with the choice point corresponding to the or-frame pointed by `OrFr_nearest_livenode(P[old_top_cp->cp_or_fr])`. For half splitting, the `new_top_cp` is always assigned with the choice point denoted by `P[middle_node->cp_b]`.

Note that for the trail, it is mandatory to copy the interval between `P[TR]` and `P[B->cp_tr]` in order to implement a new phase, named the *dereference phase*, necessary to correctly support stack splitting with incremental copy, as explained next.

4.3.2 Dereference Phase

Following YapOr's implementation, after copying the stack segments between the worker P and the worker Q, P continues its execution while Q starts the *installation phase*. Since the stack splitting work sharing process does not fully copy the stack segments of P, the installation phase of the variables in the trail may not be enough to correctly setup Q's stacks. A new phase, called *dereference phase*, must come before the installation phase. This is necessary in order to avoid the possibility of Q having incorrectly bounded variables in the copied segments. This may happen when P has instantiated variables belonging to the copied segments, i.e., in the execution path between `Q[new_top_cp]` and `Q[old_top_cp]`, that where bound in the execution path not copied to Q, i.e., between `P[B]` and `Q[new_top_cp]`.

The dereference procedure traverses the trail from `P[TR]` to `Q[new_top_cp->cp_tr]` looking for references to variables in the copied segments of the global and local stacks. If such a variable is found then the variable is dereferenced and becomes a free variable with no value assigned. Figure 8 illustrates a situation that shows why the dereference phase is necessary to correctly setup Q's stacks.

Starting from Q's assigned top choice point, **CP4**, notice how some variables in Q's global stack are not consistent with the computational state corresponding to the **CP4** choice point. One of them is variable **D** which was a free variable before **CP4** creation and is bound with the value three in Q's global stack after copying. This happens because **D** was instantiated by P only after **CP4** creation. The reference to **D** in the trail after **CP4** creation confirms such behavior. Thus, after the copying phase, the dereferencing procedure operates in order to reset such incorrectly bound variables in the copied stack segments.

4.3.3 Unbitmapping

We next discuss the situations where a requesting worker Q does not need to copy any stack segments from the sharing worker P in order to get new work. This may happen when the new top or-frame of Q, assigned after the sharing procedure, is older than its previous top or-frame (before sharing). In such situations, the requesting worker Q only needs to move up in the search tree in order to be consistent with the new assigned top or-frame.

In this movement, we may have to update the membership information for the or-frames corresponding to the backtracked path by removing Q from the bitmap field for such or-frames. We named this procedure as *unbitmapping*. As we will see next, for the half splitting model a *split counter checking phase* may also be needed for the backtracked frames.

4.3.4 Split Counter Checking Phase

The *split counter checking phase* is necessary in order to avoid incoherent values in the split counter fields for the choice points, in the requesting worker Q, not copied from P. We can say that such incoherency can be caused by the independent work sharing operations with different workers that make the common (not copied) stack segments of P and Q, namely the local stack's choice points split counter fields, to be inconsistent in Q. This checking phase is only applied within the half splitting model and is performed by the requesting worker Q after the work sharing and copying procedures.

5. Experimental Results

We evaluated the performance of our two stack splitting models with a set of well-known benchmarks widely used to evaluate or-parallel Prolog systems, and we make a comparison between the vertical splitting, half splitting and YapOr's original implementation based on environment copying.

5.1 Environment

The parallel platform for our experiments, was a machine with four AMD Six-Core Opteron TM 8425 HE (2100 MHz) chips (24 cores in total) and 64 (4x16) GB of DDR-2 667MHz RAM, running GNU/Linux (kernel 2.6.31.5-127 64 bits) with the Yap Prolog 6.2.0. The machine was running in multi-user mode, but no other users were using the machine. For the benchmarks, we used the following set of Prolog programs:

- **cubes7**. A program that consists of stacking 7 colored cubes in a column in such a way that no color appears twice in the same column for each given side.
- **ham**. A program for finding all the Hamiltonian cycles in a graph with 26 nodes, with each node connected to 3 other nodes.
- **magic**. A program to solve the Rubik's magic cube problem.

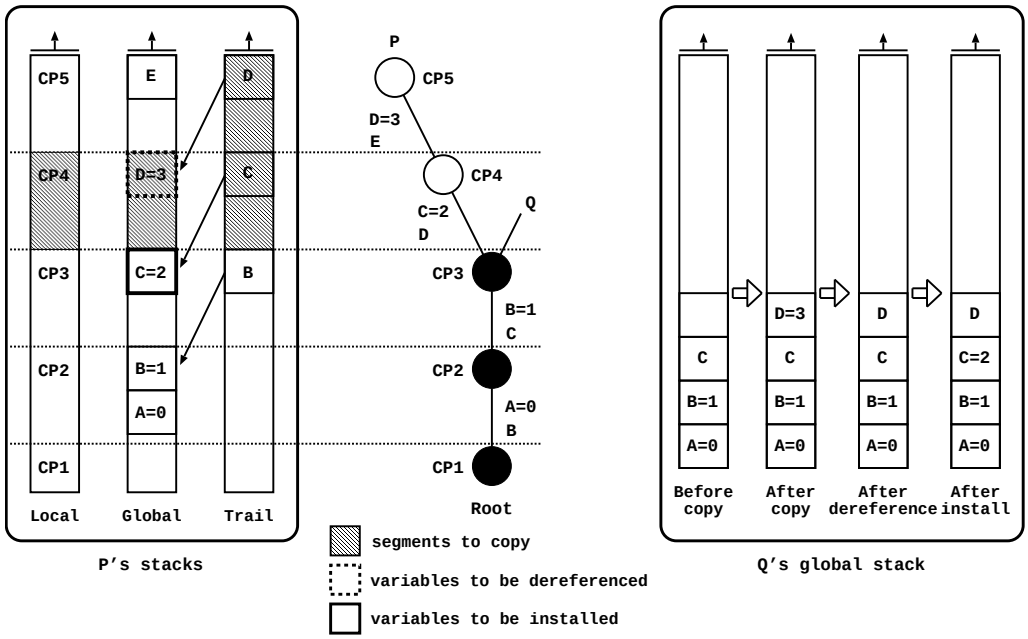

Figure 8. Dereference phase.

- **map**. A program for solving the problem of coloring a map of 10 countries with five colors in such a way that all two adjacent countries have different colors.
- **nsortN**. A program for ordering a list of N elements using a naive algorithm and starting with the list inverted.
- **puzzle**. A program that solves a version of the sudoku problem where the diagonals must add up to the same amount.
- **puzzle4x4**. A program that solves a maze problem in a 4x4 grid by moving an empty square.
- **queens13**. A program to solve the 13-queens problem that analyzes the board state at every step.

All the benchmarks find all the solutions for the given problem by simulating an automatic failure whenever a new solution is found. Each benchmark was executed twenty times and the results presented are the average of those twenty executions. To measure the execution time, we took advantage of YapOr's timing support and we used it in all models in the same way.

Next, we show the performance results for our stack splitting implementations in the YapOr system. We start by measuring the cost of the parallel model over the sequential system. Then, we evaluate the behavior of the vertical splitting and the half splitting implementations and compare with YapOr's original implementation. For shaping a fair comparison among all implementations, instead of considering the base execution times with 1 worker for each model, we considered the base execution times of the sequential implementation. All models were compiled with the same configuration parameters and using the same compiler.

5.2 Cost of the Parallel Models

Table 1 presents the execution times, in seconds, for the set of benchmark programs, when using the sequential version of the Yap system and the respective ratios, compared to Yap's sequential execution times, when using the several parallel models with one worker. In general, for all models, YapOr overheads result from handling the work load register and from testing operations that (i) verify whether the youngest node is shared or private, (ii) check for sharing requests, and (iii) check for backtracking messages due to cut operations [6].

Table 1. Execution times, in seconds, for Yap's sequential model and the respective ratios, compared to Yap's sequential execution times, for YapOr's implementation based on environment copying (EC), on vertical splitting (VS) and on half splitting (HS), all running with a single worker.

Programs	Yap	YapOr / Yap		
		EC	VS	HS
cubes7	0.202	1.044	1.038	1.059
ham	0.321	1.198	1.197	1.098
magic	45.990	0.985	0.986	0.901
map	22.434	1.130	1.130	1.141
nsort10	2.567	1.140	1.149	1.040
nsort11	28.239	1.135	1.133	1.028
nsort12	339.406	1.126	1.129	1.003
puzzle	0.154	1.152	1.151	1.106
puzzle4x4	9.875	1.032	1.030	0.958
queens13	48.220	1.061	1.063	1.001
Average		1.100	1.101	1.033

By observing the results on Table 1, we can say that, for these set of benchmarks, YapOr's vertical and half splitting models have on average, respectively, an overhead of 10.1% and 3.3% over Yap's sequential implementation. Notice also that YapOr's original implementation based on environment copying has, on average, an overhead of 10.0%, which is similar to the overheads observed previously [6, 8].

5.3 Parallel Execution

To assess the performance of the or-parallel models, we ran YapOr with a varying number of workers and we show the obtained speedups. For the speedups we used the obtained execution times and compared them against the execution times for the sequential implementation, thus reflecting the general improvement starting

from the sequential execution times. By doing that, when an obtained value is considered to be the best speedup value among all models, it really corresponds to the fastest execution time.

Tables 2 to 4 show the obtained speedups for each model. Each entry in these tables shows the speedups against the sequential execution time and for the two stack splitting models (Tables 3 and 4), in parenthesis, it shows the same speedups but without using the incremental copy technique. The best speedups among all implementations are marked with a gray background color.

From Table 2 we can see how YapOr's original implementation based on environment copying compares with the new stack splitting models. Each gray background entry illustrates the cases where environment copying is not surpassed by any stack splitting model, while the remaining entries correspond to cases where the results obtained with stack splitting is better.

In general, we can observe that stack splitting obtains better results for the cases with a smaller number of workers and that environment copying seems to perform better, on average, for the cases with 16 and 24 workers. In any case, for the 10 programs in analysis, environment copying only obtains better results in 4 and 5 programs for 16 and 24 workers, respectively.

Table 2. Speedups for YapOr's original implementation based on environment copying.

Programs	Workers			
	4	8	16	24
cubes7	3.27	5.66	7.62	7.43
ham	3.10	5.34	7.32	6.49
magic	4.05	8.08	16.08	23.95
map	3.58	7.11	13.92	20.32
nsort10	3.61	7.08	13.44	17.97
nsort11	3.71	7.37	14.63	21.63
nsort12	3.68	7.39	14.89	22.19
puzzle	2.96	4.68	5.94	5.03
puzzle4x4	3.90	7.77	15.32	22.44
queens13	3.76	7.50	14.93	22.22
Average	3.56	6.80	12.41	16.97

From Table 3, we can observe that the overall performance of vertical splitting is quite close to the performance of the original YapOr. By analyzing the speedups, it is also clear the improvement obtained with the incremental copy technique. On terms of average, the difference is noticeable in all worker cases. For example, for 4, 8, 16 and 24 workers, the speedup gain is 0.24 (from 3.31 without incremental copy to 3.55 with incremental copy), 0.77 (from 5.97 to 6.74), 1.65 (from 10.49 to 12.14) and 2.18 (from 14.26 to 16.44), respectively, which shows a clear positive tendency as the number of workers increases.

The only exception seems to be the **nsort12** program. Note that, for the **nsort11** program, the speedup gain already shows a huge reduction (from 19.93 without incremental copy to 21.16 with incremental copy for 24 workers), when compared with **nsort10**, where the speedup gain is clear (from 10.41 to 17.56 for 24 workers). We believe that this behavior is related to the balance between the overhead of copying unneeded stack segments, as happens without incremental copy, against the overhead of executing the dereference and installation phases, as necessary with incremental copy. In this particular case, it seems that the percentage of saving for using incremental copy and thus not copy the full set of stacks, starts to be considerable for the **nsort10** program, but then as we increment the size of the program, this percentage becomes less significant for the **nsort11** program and for the **nsort12** it seems to be irrelevant, making the overhead of executing the dereference and installation phases a potential cost.

Table 3. Speedups for YapOr's vertical splitting implementation with and without (in parenthesis) the incremental copy technique.

Programs	Workers			
	4	8	16	24
cubes7	3.33 (2.63)	5.52 (3.34)	6.98 (3.00)	6.05 (2.41)
ham	3.11 (2.39)	5.36 (3.21)	7.00 (3.29)	5.00 (2.94)
magic	4.04 (4.04)	8.07 (8.00)	16.04 (15.80)	23.79 (23.11)
map	3.59 (3.58)	7.13 (7.05)	13.96 (13.59)	20.36 (19.52)
nsort10	3.58 (3.52)	7.00 (6.52)	13.17 (9.81)	17.56 (10.41)
nsort11	3.66 (3.71)	7.26 (7.32)	14.33 (14.09)	21.16 (19.93)
nsort12	3.63 (3.69)	7.27 (7.39)	14.60 (14.85)	21.77 (22.05)
puzzle	2.93 (1.96)	4.52 (1.88)	5.23 (1.58)	4.27 (1.21)
puzzle4x4	3.90 (3.88)	7.76 (7.63)	15.32 (14.62)	22.46 (20.42)
queens13	3.75 (3.73)	7.46 (7.36)	14.77 (14.23)	21.93 (20.54)
Average	3.55 (3.31)	6.74 (5.97)	12.14 (10.49)	16.44 (14.26)

Finally, the speedups in Table 4 show that the improvements obtained with the incremental copy technique in the half splitting implementation are clear. On terms of average, the difference is overwhelming in all worker cases. For example, for 4, 8, 16 and 24 workers, the speedup gain is 0.94 (from 2.69 without incremental copy to 3.63 with incremental copy), 2.14 (from 4.54 to 6.68), 4.13 (from 7.20 to 11.33) and 6.11 (from 8.83 to 14.94), respectively, which shows again a clear positive tendency as the number of workers increases. We believe that these good results with incremental copy are also related to the percentage of saving achieved for not copying the full set of stacks. This advantage is more clear in the case of halt splitting since, by splitting the search tree in two halves and by sharing the older half, it reduces the stacks segments to be shared and thus to be copied, which augments the potential percentage of common stack segments that do not need to be copied.

Comparing to vertical splitting, on average, the overall performance of half splitting with incremental copy is not so close to the performance of the original YapOr with environment copying. For example, the average speedups for environment copying, vertical and half splitting are, respectively, 12.41, 12.14 and 11.33 for 16 workers and 16.97, 16.44 and 14.94 for 24 workers.

On the other hand, for the 10 programs in analysis, we can observe that half splitting with incremental copy obtains the best speedup results in 7, 6, 5 and 4 programs for 4, 8, 16 and 24 workers, respectively. Considering all combinations of programs and workers, half splitting obtains the highest number of best results among all implementations and owns the best average for the case of 2 workers.

Table 4. Speedups for YapOr's half splitting implementation with and without (in parenthesis) the incremental copy technique.

Programs	Workers			
	4	8	16	24
cubes7	3.03 (0.71)	4.67 (0.77)	5.18 (0.59)	3.65 (0.41)
ham	3.22 (1.52)	5.22 (1.85)	5.56 (1.90)	4.05 (1.63)
magic	4.44 (4.15)	8.80 (7.64)	17.44 (13.34)	25.86 (16.45)
map	3.35 (1.72)	5.36 (2.49)	5.89 (2.58)	4.86 (2.29)
nsort10	3.68 (3.27)	7.34 (5.76)	13.49 (8.45)	17.91 (8.95)
nsort11	3.78 (3.69)	7.58 (7.19)	14.90 (13.17)	22.06 (18.54)
nsort12	3.79 (3.76)	7.58 (7.47)	15.36 (14.68)	22.76 (21.18)
puzzle	2.96 (1.62)	4.48 (1.77)	5.01 (1.58)	4.46 (1.27)
puzzle4x4	4.13 (3.83)	8.08 (6.82)	15.60 (11.42)	22.93 (13.59)
queens13	3.91 (2.66)	7.69 (3.68)	14.82 (4.24)	20.90 (3.97)
Average	3.63 (2.69)	6.68 (4.54)	11.33 (7.20)	14.94 (8.83)

6. Conclusions and Further Work

We presented the design and implementation of two work sharing stack splitting models, namely vertical splitting and half splitting, in the YapOr system. The implementation of stack splitting required modifications and extensions to existing data structures, creation of new mechanisms, particularly in connection to the sharing work procedure and to the incremental copy technique.

Experimental results showed that YapOr with stack splitting is, in general, comparable to the original YapOr based on environment copying, obtaining in some cases better performance than with only environment copying. The overall performance of vertical stack splitting showed to be quite close to the performance of the original YapOr and, for the set of benchmarks used, the half splitting model performed better in 4 of 10 programs. The results attained allow us to also conclude that both stack splitting models clearly benefit from incremental copy. Globally, the results are quite encouraging as well given that in many benchmarks we achieved performances that are above a speedup of 20 on 24 cores.

Although stack splitting was initially proposed for distributed memory architectures, the results show that it is equally suitable for shared memory architectures. This is a clear advantage of stack splitting since we could use it as the basis for an hybrid execution model aiming at clusters of multicores. The idea is to combine workers into teams. A team of workers should run on shared memory and workers inside a team can distribute work using environment copying or stack splitting. Different teams should be assigned to different cluster nodes and share work performing stack splitting.

As further work, other models of stack splitting can be implemented and embedded in the work sharing procedure already implemented. Examples are the horizontal [2] and diagonal [7] stack splitting models.

Acknowledgments

This work has been partially supported by the FCT research project HORUS (PTDC/EIA-EIA/100897/2008) and the FCT research project LEAP (PTDC/EIA-CCO/112158/2009).

References

[1] K. Ali and R. Karlsson. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, 1990.

[2] G. Gupta and E. Pontelli. Stack Splitting: A Simple Technique for Implementing Or-parallelism on Distributed Machines. In *International Conference on Logic Programming*, pages 290–304. The MIT Press, 1999.

[3] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

[4] 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. Ciepielewski, and B. Hausman. The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems*, pages 819–830. Institute for New Generation Computer Technology, 1988.

[5] E. Pontelli, K. Villaverde, H.-F. Guo, and G. Gupta. Stack splitting: A technique for efficient exploitation of search parallelism on share-nothing platforms. *Journal of Parallel and Distributed Computing*, 66 (10):1267–1293, 2006.

[6] R. Rocha, F. Silva, and V. Santos Costa. YapOr: an Or-Parallel Prolog System Based on Environment Copying. In *Portuguese Conference on Artificial Intelligence*, number 1695 in LNAI, pages 178–192. Springer-Verlag, 1999.

[7] R. Rocha, F. Silva, and R. Martins. YapDss: an Or-Parallel Prolog System for Scalable Beowulf Clusters. In *Portuguese Conference on Artificial Intelligence*, number 2902 in LNAI, pages 136–150. Springer-Verlag, 2003.

[8] V. Santos Costa, I. Dutra, and R. Rocha. Threads and Or-Parallelism Unified. *Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue*, 10 (4–6):417–432, 2010.

[9] K. Villaverde, E. Pontelli, H. Guo, and G. Gupta. PALS: An Or-Parallel Implementation of Prolog on Beowulf Architectures. In *International Conference on Logic Programming*, number 2237 in LNCS, pages 27–42. Springer-Verlag, 2001.

[10] K. Villaverde, E. Pontelli, H. Guo, and G. Gupta. A Methodology for Order-Sensitive Execution of Non-deterministic Languages on Beowulf Platforms. In *International Euro-Par Conference*, number 2790 in LNCS, pages 694–703. Springer-Verlag, 2003.