On Comparing Alternative Splitting Strategies for Or-Parallel Prolog Execution on Multicores

Rui Vieira, Ricardo Rocha, and 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 models exploiting implicit parallelism have been proposed in the past. Arguably, one of the most successful models is *environment copying* for shared memory architectures. With the increasing availability and popularity of multicore architectures, it makes sense to recover the body of knowledge there is in this area and re-engineer prior computational models to evaluate their performance on newer architectures. In this work, we focus on the implementation of splitting strategies for or-parallel Prolog execution on multicores and, for that, we develop a framework, on top of the YapOr system, that integrates and supports five alternative splitting strategies. Our implementation shares the underlying execution environment and most of the data structures used to implement or-parallelism in YapOr. In particular, we took advantage of YapOr's infrastructure for incremental copying and scheduling support, which we used with minimal modifications. We thus argue that all these common support features allow us to make a first and fair comparison between these five alternative splitting strategies and, therefore, better understand their advantages and weaknesses.

1 Introduction

Detecting parallelism is far from a simple task, specially in the presence of irregular parallelism, but it is commonly left to programmers. Research effort has been made towards making specialized run-time systems more capable of transparently exploring available parallelism, thus freeing programmers from such cumbersome details. Prolog programs naturally exhibit *implicit parallelism* and are thus highly amenable for automatic exploitation.

One of the most noticeable sources of parallelism in Prolog programs is called *or-parallelism*. Or-parallelism arises from the simultaneous evaluation of a subgoal call against the clauses that match that call. When implementing or-parallelism, a main difficulty is how to efficiently represent the *multiple bindings* for the same variable produced by the parallel execution of alternative matching clauses. One of the most successful models is *environment copying* [1, 2], that has been efficiently used in the implementation of or-parallel Prolog systems on shared memory architectures. Recent advances in computer architectures have made our personal computers parallel with multiple cores sharing the main memory. Multicores and clusters of multicores are now the norm and, although, many

parallel Prolog systems have been developed in the past, evaluating their performance or even the implementation of newer computational models specialized for the multicores is still open to further research.

Another major difficulty in the implementation of any parallel system is to design efficient *scheduling strategies* to assign computing tasks to 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 execution is a difficult task. For environment copying, scheduling strategies based on *bottommost dispatching of work* have proved to be more efficient than topmost strategies [3]. An important mechanism that suits bottommost strategies best is *incremental copying* [1], an optimized copy mechanism that avoids copying the whole stacks when sharing work. *Stack splitting* [4, 5] is an extension to the environment copying model that provides a simple, clean and efficient method to accomplish work splitting among workers. It successfully splits the computation task of one worker in two complementary sets, and was thus first introduced aiming at distributed memory architectures [6, 7].

In this work, we focus on the implementation of splitting strategies for or-parallel Prolog execution on multicore architectures and, for that, we present a framework, on top of the YapOr system [2], that integrates and supports five alternative splitting strategies. We used YapOr's original splitting strategy [2] and two splitting strategies from previous work [8], named *vertical* and *half splitting*, that split work based on choice points, together with the new implementation of two alternative stack splitting strategies, named *horizontal* [4] and *diagonal splitting* [7], in which the split is based on the unexplored alternative matching clauses. All implementations take full advantage of the state-of-the-art fast and optimized Yap Prolog engine [9] and share the underlying execution environment and most of the data structures used to implement or-parallelism in YapOr. In particular, we took advantage of YapOr's infrastructure for incremental copying and scheduling support, which we used with minimal modifications. We thus argue that all these common support features allow us to make a first and fair comparison between these five alternative splitting strategies and, therefore, better understand their advantages and weaknesses.

The remainder of the paper is organized as follows. First, we introduce some background about environment copying, stack splitting and YapOr's scheduler. Next, we describe the five alternative splitting strategies and discuss their major implementation issues in YapOr. We then present experimental results on a set of well-known benchmarks and advance some conclusions and further work.

2 Environment Copying

In the environment copying model, each worker keeps a separate copy of its own environment, 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[1] where the sharing occurs. To reduce the overhead of stack copying, an optimized copy mechanism called *incremental copy* [1] takes advantage of the fact that the requesting worker may already have traversed one part of the path being shared. Therefore, it does not need to copy the stacks referring to the whole path from root, but only the stacks starting from the youngest node common to both workers.

As a result of 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 when updating scheduling information and when accessing shared nodes in order to ensure that unexplored alternatives are only exploited by one worker. Shared nodes are represented by *or-frames*, a data structure that workers must access, with mutual exclusion, to obtain the unexplored alternatives. All other data structures, such as the environment, the heap, and the trail do not require synchronization.

3 Stack Splitting

Stack splitting was first introduced to target distributed memory architectures, thus aiming to reduce the mutual exclusion requirements of environment copying when accessing shared nodes of the search tree. It accomplishes this by defining simple, clean and efficient work splitting strategies in which all available work is statically divided in two *complementary sets* between the sharing workers. In practice, stack splitting is a refined version of the environment copying model, in which the synchronization requirement was removed by the preemptive split of all unexplored alternatives 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 when accessing shared nodes.

The original stack splitting proposal [4] introduces two strategies for dividing work: *vertical splitting*, in which the available choice points are alternately divided between the two sharing workers, and *horizontal splitting*, which alternately divides the unexplored alternatives in each available choice point. *Diagonal splitting* [7] is a more elaborated strategy that achieves a precise partitioning of the set of unexplored alternatives. It is a kind of mix between horizontal and vertical splitting, where the set of all unexplored alternatives in the available choice points is alternately divided between the two sharing workers. Another splitting strategy [10], which we named *half splitting*, splits the available choice points in two halves. Figure 1 illustrates the effect of these strategies in a work sharing operation between a busy worker P and an idle worker Q.

Figure 1(a) shows the initial configuration with the idle worker Q requesting work from a busy worker P with 7 unexplored alternatives in 4 choice points. Figure 1(b) shows the effect of vertical splitting, in which P keeps its current choice point and alternately divides with Q the remaining choice points up to the root choice point. Figure 1(c) illustrates the effect of half splitting, where

[1] At the engine level, a search tree node corresponds to a choice point in the stack.

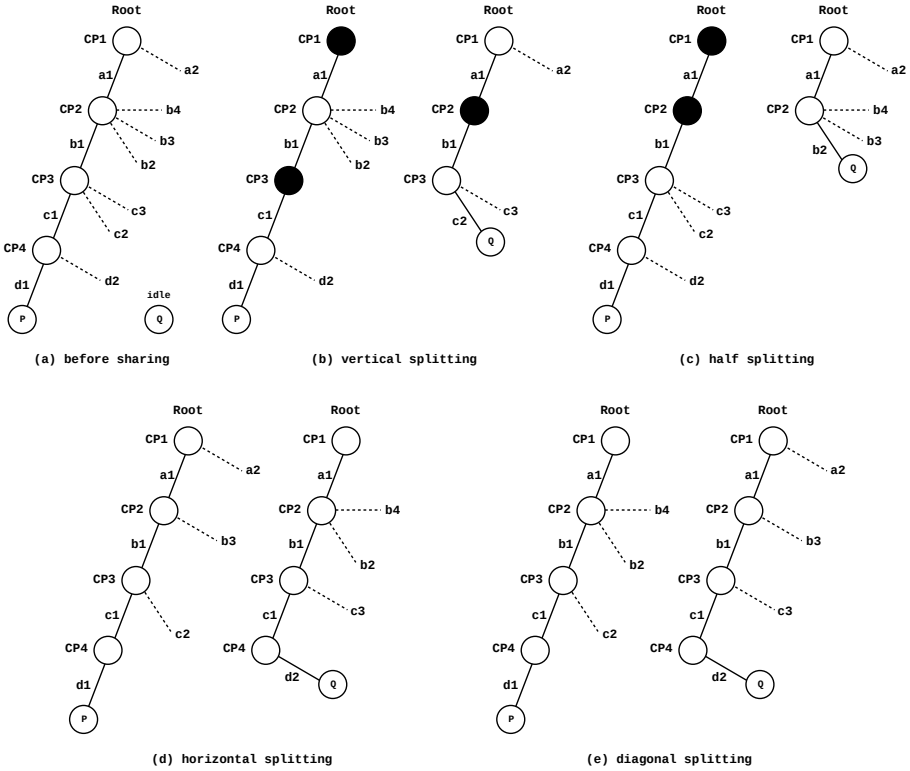

Fig. 1. Alternative stack splitting strategies

the bottom half is for worker P and the half closest to the root is for worker Q. Figure 1(d) details the effect of horizontal splitting, in which the unexplored alternatives in each choice point are alternately split between both workers, with workers P and Q owning the first unexplored alternative in the even and odd choice points, respectively. Figure 1(e) describes the diagonal splitting strategy, where the unexplored alternatives in all choice points are alternately split between both workers in such a way that, in the worst case, Q may stay with one more alternative than P. For all strategies, the corresponding execution stacks are first copied to Q, next both P and Q perform splitting, according to the splitting strategy at hand, and then P and Q are set to continue execution. As we will see, in some situations, there is no need for any copy at all, and a backtracking action is enough to place the requesting worker ready for execution.

4 YapOr's Scheduler and Original Splitting Strategy

We can divide the execution time of a worker in two modes: *scheduling mode* and *engine mode*. A worker enters in scheduling mode whenever it runs out of

work and calls the scheduler to search for available work. As soon as it gets a new piece of work, it enters in engine mode and runs like a sequential engine.

4.1 Work Scheduling

In YapOr, when a worker runs out of work, first the scheduler tries to select a busy worker with excess of *work load* to share work. The work load is a measure of the amount of unexplored alternatives in private nodes. There are two alternatives to search for busy workers in the search tree: search *below* or search *above* the current node where the idle worker is positioned. Idle workers always start to search below the current node, and only if they do not find any busy worker there, they search above. The main advantage of selecting a busy worker below instead of above is that the idle worker can request immediately the sharing operation, because its current node is already common to the busy worker, which avoids backtracking in the tree and undoing variable bindings.

When the scheduler does not find any busy worker with excess of work load, it tries to move the idle worker to a better position in the search tree. By default, the idle worker backtracks until it reaches a node where there is at least one busy worker below. Another option is to backtrack until reaching the node that contains all the busy workers below. The goal of these strategies is to distribute the idle workers in such a way that the probability of finding, as soon as possible, busy workers with excess of work below is substantially increased.

4.2 Work Sharing

Similarly to the Muse system[3], YapOr also follows a *bottommost work sharing strategy*. Whenever an idle worker Q makes a work request to a busy worker P, the work sharing operation is activated to *share all private nodes* of P with Q. P accepts the work request only if its work load is above a given *threshold value*. In YapOr, accomplishing this operation involves the following stages:

Sharing loop. This stage handles the sharing of P's private nodes. For each private node, a new or-frame is allocated and the access to the unexplored alternatives, previously done through the `CP_alt` fields in the private choice points, is moved to the `OrFr_alt` fields in the new or-frames. All 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 shared work. The membership field `OrFr_members`, which defines the set of workers that own or act upon a node, is also initialized to indicate that P and Q are sharing the corresponding choice points.

Membership update. Next, the old or-frames on P's branch are updated to include the requesting worker Q in the membership field (frames starting from P's current `top_or_frame` til Q's `top_or_frame`). In order to delimit the shared region of the search tree, each worker maintains two important

variables, named `top_cp` and `top_or_frame`, that point, respectively, to the youngest shared choice point and to the youngest or-frame[2].

Compute 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 strategy being considered.

5 Supporting Alternative Splitting Strategies in YapOr

Extending YapOr to support different stack splitting strategies required some modifications to the way unexplored alternatives are accessed. In more detail:

- With stack splitting, each worker has its own work chaining sequence. Hence, the control and access to the unexplored alternatives returned to the `CP_alt` choice point fields and the `OrFr_alt` and `OrFr_nearest_livenode` or-frame fields were simply ignored.
- For the vertical and half splitting strategies, the `OrFr_nearest_livenode` field was recovered as a way to implement the chaining sequence of choice points. At work sharing, each worker adjusts its `OrFr_nearest_livenode` fields so that two separate chains are built corresponding to the intended split of the work.
- In order to reuse YapOr's infrastructure for incremental copying and scheduling support, the or-frames are still chained through the `OrFr_next` fields and still use the `OrFr_member` fields for work scheduling.

Next, we detail the implementation of the vertical, half, horizontal and diagonal splitting strategies as well as the incremental copy technique.

5.1 Vertical Splitting

The vertical splitting strategy follows a pre-determined work splitting scheme in which the chain of available 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 presents the pseudo-code that implements the work sharing procedure for vertical splitting.

The work sharing procedure starts from P's youngest choice point (register `B`) and traverses all P's private choice points to create a corresponding or-frame

[2] Please note that the use of the naming *top* in these two variables can be confusing since, due to historical reasons, it refers to the top of the choice-point stack (where the root node is at the bottom) and not to the top of the search tree (where the root node is at the top). Despite this naming, our discussion keeps following a search tree approach with the root node always at the top.

```
next_fr = NULL
nearest_fr = NULL
current_cp = B  // B points to the youngest choice point
while (current_cp != top_cp)  // loop until the youngest shared choice point
  current_fr = alloc_or_frame(current_cp)
  add_member(P, OrFr_member(current_fr))
  if (next_fr)
    OrFr_next(next_fr) = current_fr
    add_member(Q, OrFr_member(current_fr))
  if (nearest_fr)
    OrFr_nearest_livenode(nearest_fr) = current_fr
  nearest_fr = next_fr
  next_fr = current_fr
  current_cp = CP_b(current_cp)  // next choice point on stack

// connecting with the older or-frames
if (next_fr)
  if (top_or_frame == root_frame)
    OrFr_nearest_livenode(next_fr) = DEAD_END
  else
    OrFr_nearest_livenode(next_fr) = top_or_frame
  OrFr_next(next_fr) = top_or_frame
if (nearest_fr)
  if (top_or_frame == root_frame)
    OrFr_nearest_livenode(nearest_fr) = DEAD_END
  else
    OrFr_nearest_livenode(nearest_fr) = top_or_frame

// continuing vertical splitting
if (next_fr = NULL)
  current_fr = 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)
```

Fig. 2. Work sharing with vertical splitting

by calling the `alloc_or_frame()` procedure. In Fig. 2, 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. For the youngest choice point, the or-frame is initialized with just the owning worker P in the membership field. The other or-frames are initialized with both workers P and Q.

Next, follows the connection with the older and already stored or-frames. Here, consideration must be given to the condition of P's current `top_or_frame`. If it is the root or-frame, the `OrFr_nearest_livenode` fields of the new or-frames are assigned to a `DEAD_END` value, which marks the ending point for unexplored work. Otherwise, they are assigned to P's current `top_or_frame`.

Finally, we need to decide where to continue the vertical splitting algorithm for the older 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 a `DEAD_END` is reached and, at each frame, lies a reconnection process of the `OrFr_nearest_livenode` field.

5.2 Half Splitting

The half splitting strategy partitions the chain of available choice points in two consecutive and almost equally sized parts, which are chained through the `OrFr_nearest_livenode` field of the corresponding or-frames. For that, 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 *split counter* field, named `CP_sc`, was introduced in the choice point structure. Figure 3 presents the pseudo-code that implements work sharing with horizontal splitting.

```
// updating the split counter
current_cp = B  // B points to the youngest choice point
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
  current_cp = CP_b(current_cp)  // next choice point on stack
CP_sc(current_cp) = 1  // middle choice point

// assign the remaining choice points to the requesting worker
middle_fr = CP_or_fr(current_cp)
if (middle_fr)
  OrFr_nearest_livenode(middle_fr) = DEAD_END
  current_fr = top_or_frame  // top_or_frame points to the youngest or-frame
  while (current_fr != middle_fr)
    remove_member(Q, OrFr_member(current_fr))
    current_fr = OrFr_next(current_fr)
else
  // sharing loop stage
```

Fig. 3. Work sharing with half splitting

The work sharing procedure starts from P's youngest choice point and updates the split counter on half of the choice points, in decreasing order, until reaching the *middle choice point* in P's initial partition, which gets a split counter value of 1. These are the half choice points that, after sharing, will be still owned by P. The other half will be assigned to the requesting worker Q.

After updating the split counter, we can distinguish two different situations. The first situation occurs when there are more old shared choice points than private in P's branch, in which case the middle choice point 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`, to mark the end of P's newly assigned work, and the requesting worker Q is excluded from all or-frames from the top frame til the middle frame. The second situation occurs when the middle choice point is private, in which case the remaining choice points are updated to belong to Q, which includes allocating and initializing the corresponding or-frames.

5.3 Horizontal Splitting

In the horizontal splitting strategy, the unexplored alternatives are alternately divided in each choice point. For that, the choice points include an extra field, named `CP_offset`, that marks the offset of the next unexplored alternative belonging to the choice point. When allocating a private choice point, `CP_offset` is initialized with a value of 1, meaning that the next alternative to be taken has a displacement of 1 in the list of unexplored alternatives. This is the usual and expected behavior for private choice points.

When sharing work, we follow YapOr's default splitting strategy where a new or-frame is allocated for each private choice point in P and then all or-frames are updated to include the requesting worker Q in the membership field. Next, to implement the splitting process, we double the value of the `CP_offset` field in each shared choice point, meaning that the next alternative to be taken in the choice point is displaced two positions relatively to the previous value. Finally, we adjust the first alternative at each choice point for the workers P and Q. Recall from Fig. 1 that P must own the first unexplored alternative in the even choice points and Q the first unexplored alternative in the odd choice points. Figure 4 shows the pseudo-code for this procedure.

```
// the sharing worker P starts the adjustment
if (sharing worker) adjust = TRUE else adjust = FALSE
current_cp = top_cp
while(current_cp != root_cp)  // loop until the root choice point
  alt = CP_alt(current_cp)
  if (alt != NULL)
    offset = CP_offset(current_cp)
    CP_offset(current_cp) = offset * 2
    if (adjust)
      CP_alt(current_cp) = get_next_alternative(alt, offset)
  current_cp = CP_b(current_cp)  // next choice point on stack
  adjust = !adjust
```

Fig. 4. Work sharing with horizontal splitting

5.4 Diagonal Splitting

Diagonal splitting is an alternative strategy that implements a better overall distribution of unexplored alternatives between workers. Diagonal splitting is based on the alternated division of *all* alternatives, regardless of the choice points they belong to. This strategy also follows YapOr's default splitting strategy and uses the same offset multiplication approach as presented for horizontal splitting, but takes into account the number of unexplored alternatives in a choice point to decide how the partitioning will be done in the next choice point.

When a first choice point with an odd number of alternatives (say $2n + 1$) appears, the worker that must own the first alternative (say Q) is given $n + 1$ alternatives and the other (say P) is given n. The workers then alternate and, in the next choice point, P starts the partitioning. When more choice points with an odd number of alternatives appear, the split process is repeated. At the end, Q and P may have the same number of unexplored alternatives or, in the worst case, Q may have one more alternative than P. The pseudo-code for this procedure is shown next in Fig. 5.

```
// the sharing worker P starts the adjustment
if (sharing worker) adjust = TRUE else adjust = FALSE
current_cp = top_cp
while(current_cp != root_cp)  // loop until the root choice point
  alt = CP_alt(current_cp)
  if (alt != NULL)
    offset = CP_offset(current_cp)
    CP_offset(current_cp) = offset * 2
    if (adjust)
      CP_alt(current_cp) = get_next_alternative(alt, offset)
    n_alts = number_of_unexplored_alternatives(alt) / offset
    if (n_alts mod 2 != 0)  // workers alternate
      adjust = !adjust
  current_cp = CP_b(current_cp)  // next choice point on stack
```

Fig. 5. Work sharing with diagonal splitting

5.5 Incremental Copy

In YapOr's original implementation, the incremental copy process copies everything in P's stacks that is missing in Q. With stack splitting, it only copies the segments between Q's `top_cp` before and after sharing for the global and local stacks. For the trail stack, the copy is the same since this is necessary to correctly implement the *installation phase* [2], where Q installs from P the bindings made to variables belonging to the common segments not copied from P.

Figure 6 illustrates the stack segments to be copied with incremental copy. For vertical splitting, if P has private work, Q's `new_top_cp` is assigned with the second choice point in P's choice point set (`P[CP_b(B)]`). If there is no private

work, the `new_top_cp` is assigned with the choice point corresponding to the or-frame pointed by `P[OrFr_nearest_livenode(CP_or_fr(old_top_cp))]`. For half splitting, the `new_top_cp` is always assigned with the choice point denoted by `P[CP_b(middle_cp)]`. For the horizontal and diagonal splitting, the assigning ranges are similar to YapOr's original implementation.

We next discuss the situations where Q's new `top_or_frame`, assigned during sharing, is older than Q's `top_or_frame` before sharing. In such case, Q does not copy any segment from P and 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 or-frames corresponding to the backtracked path by removing Q from the membership fields and by executing a *checking phase*. The checking phase is necessary to avoid incoherent values in the `CP_alt` fields in Q's choice points not copied from P. For half splitting, it also avoids incoherent values in the split counter fields for Q's choice points 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, to be inconsistent in Q.

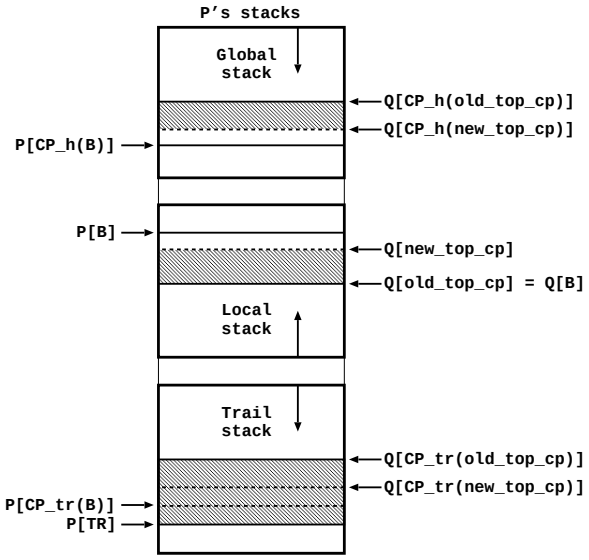

Fig. 6. Segments to copy with incremental copy

6 Experimental Results

In this section, we evaluate and compare the performance of the five splitting strategies on a set of well-known benchmarks. The environment for our experiments was a multicore machine with 4 AMD Six-Core Opteron TM 8425 HE (2100 MHz) chips (24 cores in total) and 64 GB of DDR-2 667MHz RAM, running Linux (kernel 2.6.31.5-127 64 bits) with Yap Prolog 6.3.2. The machine was running in multi-user mode, but no other users were using it. For the benchmarks, we used the following set of programs:

cubes(N) a program that consists of stacking N colored cubes in a column in such a way that no color appears twice in the same column for each side.

ham(N) a program for finding all the Hamiltonian cycles in a graph with N nodes, with each node connected to 3 other nodes.

magic(N) a program to solve the Rubik's magic cube problem in N steps.
maze(N) a program that solves a maze problem in N steps by moving an empty square in a 4x4 grid.
nsort(N) a program for ordering a list of N elements using a naive algorithm and starting with the list inverted.
queens(N) a program to solve the N-queens problem that analyzes the board state at every step.
puzzle a program that solves a puzzle problem where the diagonals must add up to the same amount.

All benchmarks find all the solutions for the given problem by simulating an automatic failure whenever a new solution is found. Each benchmark was executed 10 consecutive times and the results are the average of those executions.

We start by measuring the cost of the parallel strategies over the sequential system. Table 1 presents the execution times, in seconds, for the set of benchmark programs, when using the sequential version of Yap and the respective ratios 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 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.

Table 1. Execution times, in seconds, for Yap's sequential model and the respective overhead ratios for YapOr running 1 worker with YapOr's original splitting strategy (OS), vertical splitting (VS), half splitting (1/2S), horizontal splitting (HS) and diagonal splitting (DS).

Programs	**Yap**	**YapOr / Yap**				
		OS	**VS**	**1/2S**	**HS**	**DS**
cubes(7)	0.200	1.050	1.080	1.070	1.110	1.135
ham(26)	0.350	1.169	1.180	1.177	1.094	1.100
magic(6)	5.102	1.045	1.036	1.005	1.245	1.252
magic(7)	45.865	1.051	1.021	1.007	1.251	1.261
maze(10)	0.623	1.064	1.050	1.050	1.273	1.207
maze(12)	10.558	1.057	1.041	1.035	1.268	1.214
nsort(10)	2.775	1.124	1.155	1.096	1.074	1.072
nsort(12)	368.862	1.128	1.074	1.057	1.081	1.082
queens(11)	1.216	1.039	1.234	1.051	1.036	1.107
queens(13)	47.187	1.025	1.165	1.053	1.043	1.039
puzzle	0.153	1.157	1.235	1.144	1.176	1.157
Average		1.083	1.116	1.068	1.150	1.148

Results in Table 1 show that for these set of benchmarks, YapOr's overhead with each of the splitting strategies is small, between 6.8% and 15%. This is in-

Table 2. Speedups for YapOr running 16 and 24 workers with YapOr's original splitting strategy (OS), vertical splitting (VS), half splitting (1/2S), horizontal splitting (HS) and diagonal splitting (DS) without the incremental copy technique.

Programs	16 Workers					24 Workers				
	OS	**VS**	**1/2S**	**HS**	**DS**	**OS**	**VS**	**1/2S**	**HS**	**DS**
cubes(7)	6.45	4.65	0.61	5.26	5.12	6.66	3.92	0.46	4.76	4.54
ham(26)	6.14	4.86	2.34	4.11	5.14	6.36	4.79	2.07	3.97	5.14
magic(6)	14.33	14.25	8.35	11.67	11.70	20.40	19.77	7.76	16.51	16.35
magic(7)	14.97	15.51	12.18	12.29	12.31	22.24	22.96	16.17	18.39	18.43
maze(10)	9.58	10.74	4.82	7.78	7.98	11.32	11.98	4.20	9.16	8.41
maze(12)	14.44	15.06	11.55	12.50	12.56	21.03	21.81	14.89	17.80	17.68
nsort(10)	10.63	11.37	9.91	9.94	10.16	13.73	12.50	12.06	12.50	12.33
nsort(12)	14.37	14.71	14.72	14.43	14.52	21.16	21.47	21.62	20.93	20.78
queens(11)	12.66	7.84	1.68	11.05	11.15	16.21	8.94	1.60	13.07	12.93
queens(13)	15.66	14.05	4.10	15.08	15.16	22.14	20.54	4.12	22.20	22.42
puzzle	3.82	2.21	2.25	3.00	3.12	3.73	1.91	1.45	2.59	2.68
Average	11.19	10.48	6.59	9.74	9.90	15.00	13.69	7.85	12.90	12.88

line with the overheads observed previously for YapOr and some of the splitting strategies [2, 11, 8].

Next, we assessed the performance of the or-parallel models, by running YapOr with a varying number of workers, up to 24, although for simplicity here we only show results for 16 and 24 workers. For fairness in the comparison of all strategies, we use the sequential execution times as the base execution times, instead of considering the base execution times with 1 worker for each strategy. In this way, the speedups do reflect real gains from sequential execution times. The results are shown in Tables 2 and 3 and the best speedup value among all strategies, which corresponds to the fastest execution times, for each benchmark, is marked with a gray background color.

From Table 2 we can observe the overall performance of all strategies without resorting to incremental copy optimization. The results show reasonably good speedups with exception for half splitting. With 24 workers, YapOr's original splitting shows the best performance, followed by vertical splitting and then horizontal and diagonal splitting with minimal differences. For some benchmarks, such as the **cubes** and **queens** benchmarks, half splitting does pretty badly.

Table 3 shows the overall performance for all strategies, but now using the incremental copying optimization. The performance for all strategies improve significantly for all benchmarks. Again, half splitting is the worst performing strategy, on average, it performs about 14% less than the best performing strategy with 24 workers. Another observation is that vertical, horizontal and diagonal splitting perform slightly close to the original YapOr. The best overall performance with 16 and 24 workers is achieved with vertical splitting.

Table 3. Speedups for YapOr running 16 and 24 workers with YapOr's original splitting strategy (OS), vertical splitting (VS), half splitting (1/2S), horizontal splitting (HS) and diagonal splitting (DS) with the incremental copy technique.

Programs	16 Workers					24 Workers				
	OS	**VS**	**1/2S**	**HS**	**DS**	**OS**	**VS**	**1/2S**	**HS**	**DS**
cubes(7)	8.00	13.33	6.45	13.33	12.50	13.33	14.28	4.00	16.66	15.38
ham(26)	10.00	10.29	7.95	10.00	11.29	9.45	7.60	4.48	7.14	9.45
magic(6)	14.96	15.46	15.27	12.41	12.47	22.08	22.87	22.77	18.41	18.41
magic(7)	15.15	15.64	15.46	12.52	12.50	22.63	23.40	22.96	18.67	18.78
maze(10)	13.54	15.19	14.83	12.46	12.71	18.32	22.25	21.48	18.32	18.87
maze(12)	15.12	15.59	15.25	13.18	13.46	22.36	23.30	22.75	19.73	19.95
nsort(10)	14.15	14.60	14.60	14.15	14.08	20.25	20.70	21.34	19.96	20.40
nsort(12)	14.18	14.36	14.43	14.04	14.26	21.59	22.28	22.16	21.69	21.85
queens(11)	14.65	13.66	9.57	14.82	14.82	20.26	17.62	6.75	20.26	20.96
queens(13)	15.75	14.51	13.87	15.35	15.32	23.44	21.60	15.90	22.99	22.91
puzzle	9.00	10.20	11.76	11.76	11.76	9.56	10.20	15.30	10.92	12.75
Average	13.13	13.89	12.68	13.09	13.20	18.48	18.74	16.35	17.71	18.16

Instead of using the sequential execution times as the base reference, if one uses the execution times with 1 worker for each strategy, then the average speedups with incremental copying and 24 workers for the original, vertical, horizontal and diagonal splitting were very close and above 20.

7 Conclusions and Further Work

We have presented the integration of five alternative splitting strategies on top of the YapOr system for or-parallel Prolog execution on multicores. Our implementation shares the underlying execution environment and most of the data structures used to implement or-parallelism in YapOr.

Experimental results, on a multicore machine with 24 cores, showed that clearly incremental copying optimization pays off in improving real performance in all strategies. The results for all strategies are reasonably good and the average speedups over all benchmarks is reasonably close, with exception for half splitting that performs a little worse. However, these are preliminary results and further detailed statistics are necessary to enable us to explain some apparently inconsistent results. For example, half splitting performs badly with **cubes** and **queens** benchmarks, both with incremental and without incremental copying, but, on the other hand, it is the best performing on the **nsort(10)** and **puzzle** benchmarks with incremental copying. To explain these results, we need not only to gather low level statistics, during the execution, but also understand in which manner the splitting strategy influences the scheduling of work. A postmortem visualization of the search tree might also bring some insight in to this analysis.

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

Acknowledgments

We thank the referees for their valuable comments and suggestions. This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within projects PEst (FCOMP-01-0124-FEDER-022701), LEAP (PTDC/EIA-CCO/112158/2009) and HORUS (PTDC/EIA-EIA/100897/2008).

References

1. Ali, K., Karlsson, R.: The Muse Approach to OR-Parallel Prolog. International Journal of Parallel Programming **19**(2) (1990) 129–162
2. Rocha, R., Silva, F., Santos Costa, V.: YapOr: an Or-Parallel Prolog System Based on Environment Copying. In: Portuguese Conference on Artificial Intelligence. Number 1695 in LNAI, Springer-Verlag (1999) 178–192
3. Ali, K., Karlsson, R.: Full Prolog and Scheduling OR-Parallelism in Muse. International Journal of Parallel Programming **19**(6) (1990) 445–475
4. Gupta, G., Pontelli, E.: Stack Splitting: A Simple Technique for Implementing Or-parallelism on Distributed Machines. In: International Conference on Logic Programming, The MIT Press (1999) 290–304
5. Pontelli, E., Villaverde, K., Guo, H.F., Gupta, G.: Stack splitting: A technique for efficient exploitation of search parallelism on share-nothing platforms. Journal of Parallel and Distributed Computing **66**(10) (2006) 1267–1293
6. Villaverde, K., Pontelli, E., Guo, H., Gupta, G.: PALS: An Or-Parallel Implementation of Prolog on Beowulf Architectures. In: International Conference on Logic Programming. Number 2237 in LNCS, Springer-Verlag (2001) 27–42
7. Rocha, R., Silva, F., Martins, R.: YapDss: an Or-Parallel Prolog System for Scalable Beowulf Clusters. In: Portuguese Conference on Artificial Intelligence. Number 2902 in LNAI, Springer-Verlag (2003) 136–150
8. Vieira, R., Rocha, R., Silva, F.: Or-Parallel Prolog Execution on Multicores Based on Stack Splitting. In: International Workshop on Declarative Aspects and Applications of Multicore Programming, ACM Digital Library (2012)
9. Santos Costa, V., Rocha, R., Damas, L.: The YAP Prolog System. Journal of Theory and Practice of Logic Programming **12**(1 & 2) (2012) 5–34
10. Villaverde, K., Pontelli, E., Guo, H., Gupta, G.: A Methodology for Order-Sensitive Execution of Non-deterministic Languages on Beowulf Platforms. In: International Euro-Par Conference. Number 2790 in LNCS, Springer-Verlag (2003) 694–703
11. Santos Costa, V., Dutra, I., Rocha, R.: Threads and Or-Parallelism Unified. Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue **10**(4–6) (2010) 417–432