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

CICLOPS 2012
Budapest, Hungary, September 2012

Why Parallelism in Prolog?

Efficient sequential implementations

- There are many efficient sequential implementations of Prolog, mostly based on the **Warren Abstract Machine (WAM)**.

Potential for implicit parallelism

- Prolog programs **naturally exhibit implicit parallelism** and are thus highly amenable for **automatic exploitation**.
- This makes parallel logic programming **as easy as** logic programming.

Parallelism in Prolog

Or-parallelism

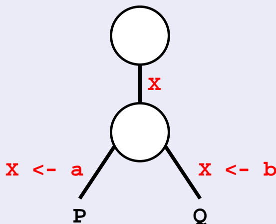
- One of the most successful sources of parallelism in Prolog programs is called **or-parallelism**.
- Or-parallelism arises from the simultaneous evaluation of a subgoal call against the clauses that match that call.

```
path(X,Z) :- path(X,Y), edge(Y,Z).
path(X,Z) :- edge(X,Z).
```

Or-Parallelism: Implementation Challenges

Multiple bindings

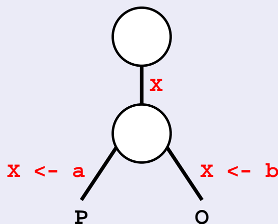
- How to efficiently represent the **multiple bindings** for the same variable produced by the parallel execution of alternative matching clauses.



Or-Parallelism: Implementation Challenges

Multiple bindings

- 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**:
  - Each worker maintains a **separated copy** of its environment.
  - Sharing is done by **copying the execution stacks** between workers.

Or-Parallelism: Implementation Challenges

Scheduling

- How to efficiently achieve the necessary cooperation, synchronization and concurrent access to shared data structures among several workers during parallel execution.

Or-Parallelism: Implementation Challenges

Scheduling

- How to efficiently achieve the necessary cooperation, synchronization and concurrent access to shared data structures among several workers during parallel execution.
- For environment copying, scheduling strategies based on **bottommost dispatching of work** have proved to be more efficient than topmost strategies.
- An important technique that fits bottommost strategies best is **incremental copying**, an optimization that avoids copying the whole stacks when sharing work.

Or-Parallelism: Implementation Challenges

Scheduling

- How to efficiently achieve the necessary cooperation, synchronization and concurrent access to shared data structures among several workers during parallel execution.
- For environment copying, scheduling strategies based on **bottommost dispatching of work** have proved to be more efficient than topmost strategies.
- An important technique that fits bottommost strategies best is **incremental copying**, an optimization that avoids copying the whole stacks when sharing work.
- **Stack splitting** is an extension to the environment copying model that provides a simple and efficient method to split work in which the available work is **statically divided in two complementary sets** between the sharing workers.

Our Goal

Comparing splitting strategies on multicores

- Benefit from prior work on the development of the **YapOr system** and extend it to efficiently support five alternative splitting strategies, on multicore architectures:
  - YapOr's **original splitting** strategy;
  - Two stack splitting strategies, named **vertical splitting** and **half splitting**, that split work based on choice points [DAMP'12];
  - Two stack splitting strategies, named **horizontal splitting** and **diagonal splitting**, that split work based on the unexplored matching clauses.

Our Goal

Comparing splitting strategies on multicores

- Benefit from prior work on the development of the **YapOr system** and extend it to efficiently support five alternative splitting strategies, on multicore architectures:
  - YapOr's **original splitting** strategy;
  - Two stack splitting strategies, named **vertical splitting** and **half splitting**, that split work based on choice points [DAMP'12];
  - Two stack splitting strategies, named **horizontal splitting** and **diagonal splitting**, that split work based on the unexplored matching clauses.
- Our implementation shares the underlying execution environment and most of the data structures used to implement or-parallelism in YapOr. We thus argue that all these common support features allow us to make a first and fair comparison between these five alternative splitting strategies.

The YapOr System

Execution Model

- YapOr is based on the **environment copying model**:
  - Each worker maintains a **separated copy** of its environment.
  - Sharing is done by **copying the execution stacks** between workers.

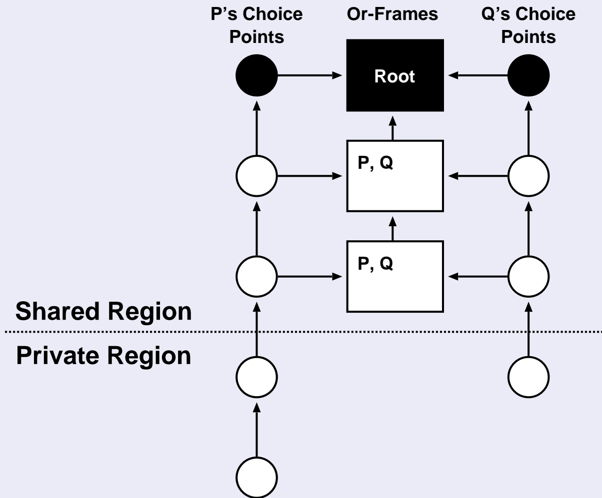
The YapOr System

Execution Model

- YapOr is based on the **environment copying model**:
  - Each worker maintains a **separated copy** of its environment.
  - Sharing is done by **copying the execution stacks** between workers.
- YapOr's original splitting strategy is based on **bottommost dispatching of work** and **dynamic sharing**:
  - Shared nodes are represented by **or-frames**, a data structure that workers must access, with **mutual exclusion**, to obtain the unexplored alternatives.
  - Synchronization is needed to ensure that each alternative is **explored only once**.

The YapOr System

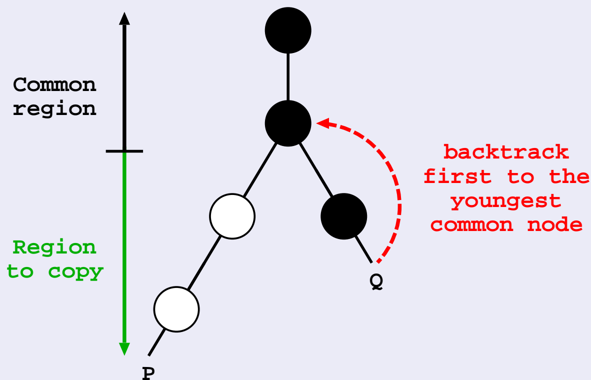
Data structures



Environment Copying

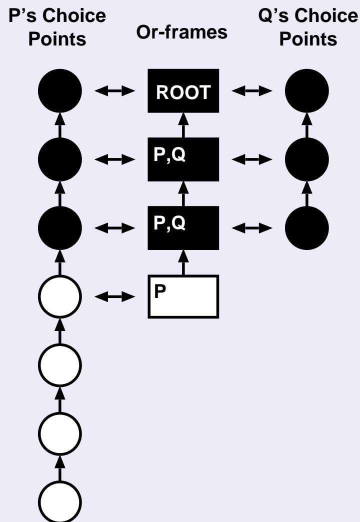
Incremental Copy

- Aims to **minimize the amount of data** copied between P and Q.
- It copies only the **state difference** between workers P and Q.



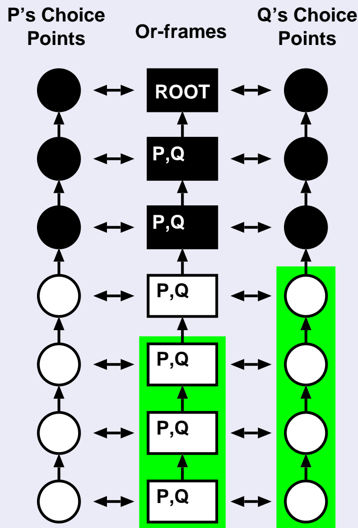
Environment Copying

Before sharing



Environment Copying

After sharing



Environment Copying

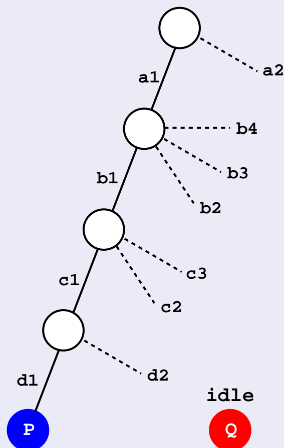
Stack splitting

- Introduced to target **distributed memory architectures**.
- Aiming to **avoid the mutual exclusion requirements** when accessing shared branches of the search tree.
- Defines a work sharing strategy in which the available work is **statically divided in two complementary sets**.
- The splitting is such that both workers can continue executing its branch of computation **independently**, without any need for further synchronization.

Stack Splitting

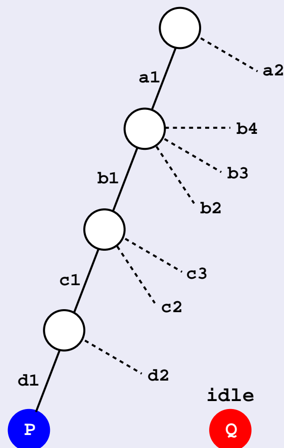
Vertical splitting

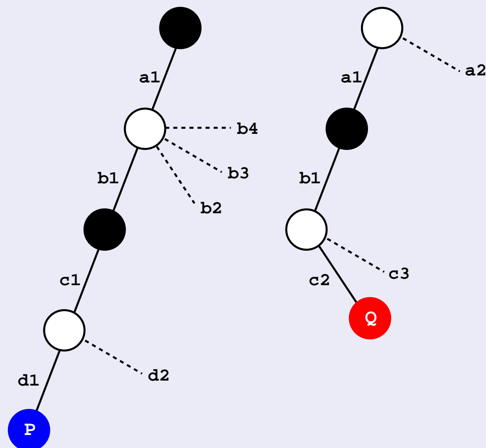
(a) before sharing



Stack Splitting

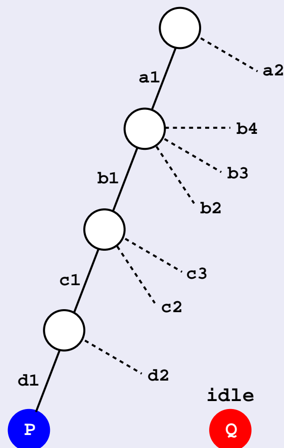
Vertical splitting

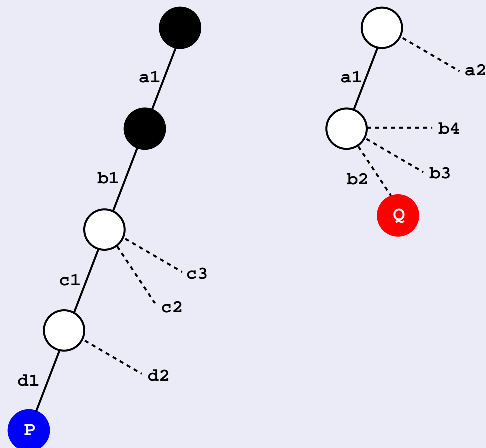




Stack Splitting

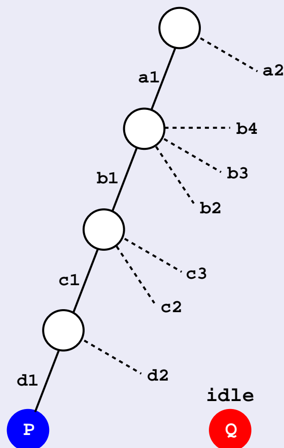
Half splitting

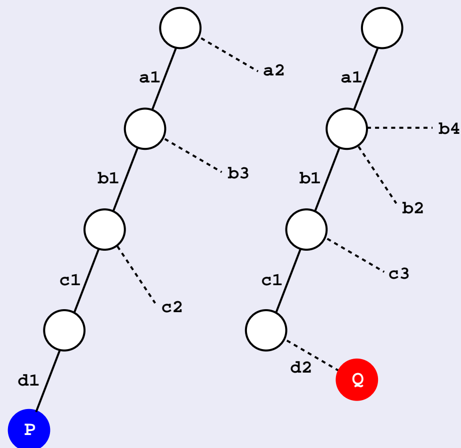




Stack Splitting

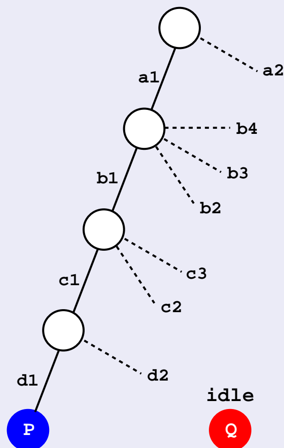
Horizontal splitting

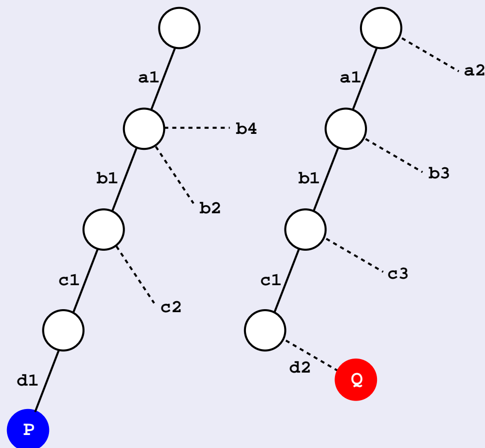




Stack Splitting

Diagonal splitting





Supporting Stack Splitting Strategies in YapOr

All splitting strategies

- Since, with stack splitting, each worker has its own work chaining sequence, the control and access to the unexplored alternatives returned to the choice points and some of the or-frame fields were ignored.
- In order to reuse YapOr's infrastructure for incremental copying and scheduling support, we still use the or-frames fields related with such support.

Supporting Stack Splitting Strategies in YapOr

Vertical and half splitting

- We use the or-frame field **OrFr_nearest_livenode** as a way to implement the chaining sequence of choice points. At work sharing, each worker adjusts its fields so that two separate chains are built corresponding to the intended split of work.
- For half splitting, a new choice point field **CP_depth** supports the numbering of nodes in order to allow the efficient calculation of the relative depth of the worker's assigned choice points.

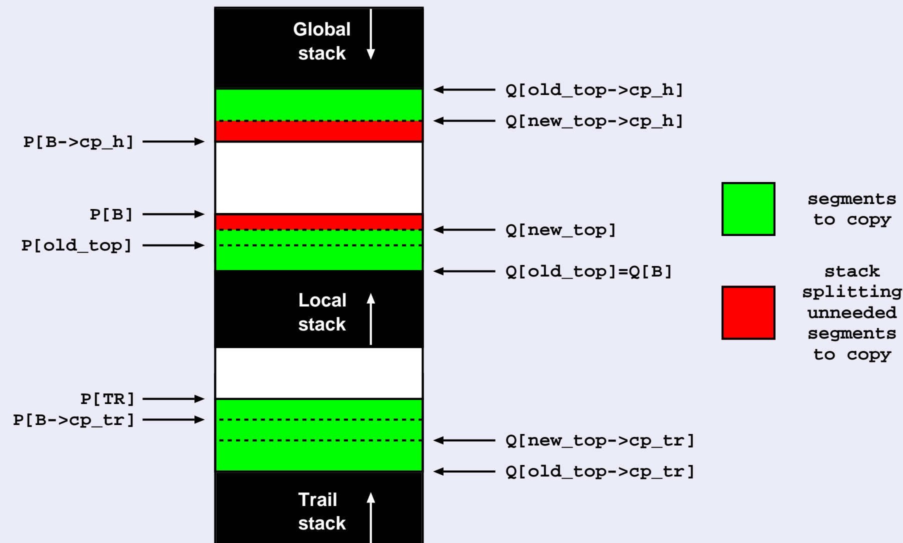
Supporting Stack Splitting Strategies in YapOr

Horizontal and diagonal splitting

- A new choice point field **CP_offset** marks the offset of the next unexplored alternative belonging to the choice point.
- To implement the splitting process, we double its value for each shared choice point, meaning that the next alternative to be taken is displaced two positions relatively to the previous value.

Supporting Stack Splitting Strategies in YapOr

Copy ranges



Experimental Results

Environment

- A machine with 4 AMD Six-Core Opteron TM 8425 HE (2100 MHz) chips (24 cores in total) and 64 (4x16) GB of DDR-2 667MHz RAM.
- All benchmarks find all the solutions 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 10 executions.
- We used the sequential execution times as the base reference for computing speedups (instead of considering the times with 1 worker for each strategy). In this way, the speedups do reflect real gains from sequential execution times.

Experimental Results

Cost of the splitting strategies (1 worker)

Programs	Yap (s)	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

Experimental Results

Speedups for 24 workers without incremental copy

Programs	OS	VS	1/2S	HS	DS
cubes(7)	**6.66**	3.92	0.46	4.76	4.54
ham(26)	**6.36**	4.79	2.07	3.97	5.14
magic(6)	**20.40**	19.77	7.76	16.51	16.35
magic(7)	22.24	**22.96**	16.17	18.39	18.43
maze(10)	11.32	**11.98**	4.20	9.16	8.41
maze(12)	21.03	**21.81**	14.89	17.80	17.68
nsort(10)	**13.73**	12.50	12.06	12.50	12.33
nsort(12)	21.16	21.47	**21.62**	20.93	20.78
queens(11)	**16.21**	8.94	1.60	13.07	12.93
queens(13)	22.14	20.54	4.12	22.20	**22.42**
puzzle	**3.73**	1.91	1.45	2.59	2.68
Average	**15.00**	13.69	7.85	12.90	12.88
Average (1w)	**16.25**	15.28	8.38	14.84	14.79

Experimental Results

Speedups for 24 workers with incremental copy

Programs	OS	VS	1/2S	HS	DS
cubes(7)	13.33	14.28	4.00	**16.66**	15.38
ham(26)	**9.45**	7.60	4.48	7.14	**9.45**
magic(6)	22.08	**22.87**	22.77	18.41	18.41
magic(7)	22.63	**23.40**	22.96	18.67	18.78
maze(10)	18.32	**22.25**	21.48	18.32	18.87
maze(12)	22.36	**23.30**	22.75	19.73	19.95
nsort(10)	20.25	20.70	**21.34**	19.96	20.40
nsort(12)	21.59	**22.28**	22.16	21.69	21.85
queens(11)	20.26	17.62	6.75	20.26	**20.96**
queens(13)	**23.44**	21.60	15.90	22.99	22.91
puzzle	9.56	10.20	**15.30**	10.92	12.75
Average	18.48	**18.74**	16.35	17.71	18.16
Average (1w)	20.01	**20.91**	17.46	20.37	20.85

Conclusions and Further Work

Conclusions

- 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. In particular, we took advantage of YapOr's infrastructure for incremental copying and scheduling support, which we used with minimal modifications.

Conclusions and Further Work

Conclusions

- Experimental results, on a multicore machine with 24 cores, showed that incremental copying clearly 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.

Conclusions and Further Work

Further Work

- These are preliminary results and further work is still necessary to better explain some apparently inconsistent results. We plan to gather low level statistics an use visualization tools of the search tree to bring some insight into this analysis.
- After that, we plan to have all strategies working together (i.e., we can have workers sharing work using different strategies) and let the scheduler decide which strategy to use accordingly to some heuristics.