Novel Models for Or-Parallel Logic Programs: A Performance Analysis

Vítor Santos Costa[1] Ricardo Rocha[2] Fernando Silva[2]

[1] COPPE Systems Engineering, Federal University of Rio de Janeiro, Brazil
vitor@cos.ufrj.br
[2] DCC-FC & LIACC, University of Porto, Portugal
{ricroc,fds}@ncc.up.pt

Abstract. One of the advantages of logic programming is the fact that it offers many sources of *implicit* parallelism, such as and-parallelism and or-parallelism. Binding Arrays and Environment Copying are two models that efficiently solve the problem of assigning multiple bindings for shared variables to support or-parallelism. Recent research has led to two new binding representation approaches that also support independet and-parallelism: the Sparse Binding Array; and the Copy-On-Write binding models. In this paper, we investigate whether these newer models are practical alternatives to copying for or-parallelism. We based our work on YapOr, an or-parallel copying system using the YAP Prolog engine, so that the three alternative systems share schedulers and the underlying engine.

1 Introduction

One of the advantages of logic programming (LP) is the fact that one can exploit *implicit* parallelism in logic programs. Implicit parallelism reduces the programmer effort required to express parallelism and to manage work. Logic programs have two major forms of implicit parallelism: *or-parallelism* (ORP) and *and-parallelism* (ANDP). Given an initial query to the logic programming system, ORP results from trying several different alternatives simultaneously. In contrast, ANDP stems from dividing the work required to solve the alternative between the different processors. One particularly interesting form of ANDP is independent and-parallelism (IAP), found in divide-and-conquer problems.

Arguably, ORP systems, such as Aurora [14] and Muse [2], have been the most successful parallel logic programming (PLP) systems so far. One first reason is the large number of logic programming applications that require search, including structured database querying, expert systems and knowledge discovery applications. Parallel search can be also useful in constraint logic programming. Two major issues must be addressed to exploit ORP. First, one must address the *multiple bindings* problem. This problem arises because alternatives being exploited in parallel may give different values to variables in shared branches of the search tree. Several mechanisms have been proposed for addressing this problem [13]. Second, the ORP system itself must be able to divide work between processors. This *scheduling* problem is made complex by the dynamic nature of work in ORP systems.

Most modern parallel LP systems, including SICStus Prolog [5], Eclipse [1], and YAP [12] use copying as a solution to the multiple bindings problem. Copying was made popular by the Muse ORP system, a system derived from an early release of SICStus Prolog. The key idea for copying is that workers work in shared memory, but in separate stacks. Essentially, whenever a processor, say W_1, wants to give work to another, say W_2, W_1 simply copies its own stacks to W_2. In contrast to other approaches, Muse [3] showed that copying has a low overhead over the corresponding sequential system. On the other hand, copying has a few drawbacks. First, it is expensive to exploit more than parallelism than just ORP with copying. The problem is that the efficiency of copying largely depends on copying contiguous stacks, but this is difficult to guarantee in the presence of ANDP [?]. A second issue is that it copying makes it more expensive to suspend branches during execution. This is a problem when implementing cuts and side-effects.

Recent research in the combination of IAP and ORP has led to two new binding representation approaches: the SBA (Sparse Binding Array) [8] and the αCOWL (copy-on-write design) [9]. The SBA is an evolution of Warren's Binding Array (BA) representation [18]. In BA systems, the stacks form a cactus-tree representing the search-tree, and processors expand tips of this tree. Bindings that may be different are stored in a local data-structure, the binding array. The αCOWL scheme uses a copy-on-write mechanism to do lazy copying. Both of these approaches elegantly support IAP and ORP.

The question remains of how these systems fare against copying for ORP, in order to verify whether they are indeed practical alternatives to copying. To address this question, we experimented with YapOr, an ORP copying system using the YAP engine [16], and we implemented the SBA and the αCOWL over the original system. The three alternative systems share schedulers and the underlying engine: they do only differ in their binding scheme. We then used a set of well known ORP all-solutions benchmarks to evaluate how they perform comparatively.

The paper is organised as follows. We first review in more detail the three models. Next, we discuss their implementation. We then present and discuss experimental results. Last, we make some concluding remarks.

2 Models for Or-Parallelism

A goal in our research is to develop a system capable of exploring implicitly all forms of parallelism in Prolog programs. A key point to achieve such a goal is to determine a binding model that simplifies the exploitation of the combined forms of parallelism. In this paper we concentrate in three binding models: environment copying, sparse binding arrays and copy-on-write. We assume a multisequencial-system, where the computational agents, or *workers*, do not initially inform the system that they have created new alternatives, and thus have exclusive access to these alternatives. This is called *private work*. At some point choice-points may be made available to other workers, and we say they become *public* work.

The *Environment copying* model was introduced by Ali and Karlson in the Muse system [2]. In this model each computing agent (or worker) maintains a separate environment, almost as in sequential Prolog, in which the bindings it makes are independently recorded, hence solving the multiple bindings problem. When a worker becomes idle as it has no work, it searches for a busy worker from whom to request work. Sharing work among workers thus involves the actual copying of the computation state (WAM stacks) from the busy worker to the requester. After copying both workers have exactly the same state, and will diverge by executing alternative branches at the choice-point the work sharing took place.

Efficient implementations of copying depend on *Incremental copying* to much reduce the overheads of copying. With this technique, one just copies the parts of the execution stacks that are different among the workers involved. The scheduler plays an important role here by guiding idle workers to request work from the nearest busy workers. Bottom-most scheduling strategies have been very successful with this binding model, because they increase the number of choice-points shared between workers, thus preventing unnecessary copying.

The *Copy-On-Write* model, or αCOWL, was proposed by Santos Costa [9] towards supporting and/or parallelism. In the αCOWL, similarly to environment copying, each computing agent (worker) maintains a separate environment. Moreover, whenever a worker `Q` wants to share work from a different worker `P`, it also *logically* copies all execution stacks `P`. The insight here is that although stacks will be logically copied, they should be *physically* copied only on demand. To do so, the αCOWL applies the Copy-On-Write mechanism provided by most modern Operating Systems.

The αCOWL has two major advantages. First, we can copy anything. We can copy standard Prolog stacks, the store of a constraint solver, or a set of stacks for ANDP. Indeed, we might not even have a Prolog system at all. Second, because copying is done on demand, we do not need to worry about the overheads of copying large stacks. This is an important advantage for ANDP computations. The main drawback of the αCOWL is that the actual setting up of the COW mechanism can be itself quite expensive, and in fact, more expensive than just copying the stacks. In the next sections we discuss an implementation and its performance results.

The *Sparse Binding Array* (SBA) derives from the binding arrays model. Binding arrays were originally proposed by Warren for the SRI model [18]. In this model execution stacks are distributed over a shared address space, forming the so-called cactus-stack. In more detail, workers expand the stacks in the parts of the shared space they own, whilst they also can access stacks originally created by other workers. Note that the major source of updates to public and private work are bindings of variables. Bindings to the public part of the tree are tentative, and in fact different alternatives of the search tree may give different values, or even no value, to the same variable. These bindings are called *conditional* *bindings*, and WAM-based systems will also stored them in the *Trail* data-area, so that they can later be undone. Conditional bindings cannot be stored in the shared tree. Instead, in the original BA scheme workers use a private array data structure associated with each computing agent to record conditional bindings.

The Sparse or Shadow Binding Array (SBA) [8] is a simplification of the BA designed to handle IAP. In the SBA is that each worker has a private virtual address space that fully shadows the system's shared address space. This address space is mapped at the same fixed location for each worker in the system. Data structures and unconditional bindings are still stored in the shared address space. Conditional bindings are stored in the shadow area, which is consulted before consulting the shared area.

3 Implementation Issues

The literature includes several comparisons of copying-based versus BA-based systems, and paricularly of Aurora vs. Muse [4, 7]. One problem with these studies is that Aurora and Muse have very different implementations: it is quite difficult to know whether the differences stem from the model or from the actual implementation.

In contrast, we experimented with the three models by implementing them over the same YapOr system [16]. The system is derived from the Yap engine [10]. This is one of the fast emulator-based Prolog systems currently available, and only 2 to 3 times slower than systems that generate native-code. We would expect Yap to be between 2 to 4 times faster than the sequential Aurora engine on the same hardware.

3.1 YapOr With Copying

The YapOr system was originally designed to implement copying. The system is based on the Yap engine. The main changes were required on the instructions that manipulate choice-points. Other changes are in the initialisation code for memory allocation and worker creation, some small changes in the compiler provide extra information for managing ORP, and lastly a change designed to support built-in synchronisation.

In a nutshell, the adapted engine communicates with YapOr through a fixed set of interface macros and through two special instructions. The macros are activated when choice-points are activated, updated, or removed. The two instructions are activated whenever a worker backtracks to the shared part of the tree and they call the *scheduler* to do work search. One instruction processes parallel choice-points, and the other sequential choice-points (such that alternatives must be explored in order).

The scheduler is the major component of YapOr. Work is represented as a set of *or-frames* in a special shared area. Idle workers consult this area and the `GLOBAL` and `LOCAL` data structures, which contains data on work and the status of each worker, until they find work. If there is no work in the shared tree, idle workers try to share work with a *busy* worker. This sharing is implemented by two model dependent functions: `p_share_work()`, for the busy worker, and `q_share_work()`, for the idle one. After sharing the previously idle worker will backtrack to a newly shared choice-point, whereas the previously busy worker continues execution from the same point. Note that before sharing, workers will try to move up in the tree to simplify incremental copying. In copying, sharing is implemented by the following algorithm:

```
       Busy Worker P            Signals           Idle Worker Q

-------------------------- ---------------- --------------------------
Compute stacks to copy                      Wait sharing signal
.                          ----sharing----> .
Share private nodes                         Copy trail ?
.                                           Copy heap ?
.                                           Wait nodes_shared signal
.                          --nodes_shared-> .
Help Q in copy ?                            Copy local stack ?
.                          <---copy_done--- .
.                          ---copy_done---> .
Wait copy_done signal                       Wait copy_done signal
Back to Prolog execution                    Install conditionals
.                          <-----ready----- .
Backtrack to shared node ?                  Fail to top shared node
Wait ready signal                           .
```

Initialy, the idle worker waits for a sharing signal while the busy worker computes the stacks to copy. Next, the busy worker prepares its private nodes for sharing whilst the idle worker performs incremental copying. The busy worker may help in the copying process to speed it up. The two workers then synchronise to determine the end of copying. At the end, the busy worker goes back to Prolog execution and the iddle worker installs the conditional bindings from the busy worker that correspond to variables in the maintained part of the stacks. To guarantee the correctness of the installation step, the busy worker can not backtrack to a shared node until the idle worker do not completes installation.

3.2 αCOWL

To support sharing of work in the αCOWL we had to change `p_share_work()` and `q_share_work()`. We applied the main function where Unix-style Operating Systems implement COW: the `fork()` function. The idea is that whenever a worker `P` accepts a work request from another worker `Q`, worker `P` forks a child process that will assume the identity of worker `Q`, whilst the older process executing `Q` exits. At this point, the new process `Q` has the same state as that of `P`. The process `Q` then is forced to backtrack to the same choice-point. Note that scheduling is realised in exactly the same way as for the environment copying model, that is through the use of a public tree of or-frames in shared space.

The synchronisation algorithm for sharing work in αCOWL is as follows:

```
       Busy Worker P            Signals           Idle Worker Q

-------------------------- ---------------- --------------------------
.                                           Wait sharing signal
```

```
.                                ----sharing----> .
fork()                                            exit()
.                                                 Child takes Q's id
Back to Prolog execution                          Fail to top shared node
```

Note that `fork()` is a rather expensive operation. For programs which have parallelism of high granularity, one expects that the workers will be busy most of the time and the number of sharing operations be small. In this case the model is expected to be efficient. On the other hand, we would expect worse results for fine-grained applications. Note that one could use the `mmap()` primitive as an alternative to `fork()`, but we felt `fork()` provided the most elegant solution.

3.3 Sparse Binding Arrays

Supporting the SBA requires changes to both the engine and the sharing mechanism. The main changes to the engine affect pointer comparison, and variable and binding representation.

As regards *pointer comparison*, the Yap system assumes pointers in the stacks follow a well-defined ordering: the local stack is above the global stack, the local stack grows downwards, and the global stack grows upwards. These invariants allows one to easily calculate variable age and are useful for trailing and recovering space. Unfortunately, they are not valid in the SBA, as the cactus stack is fragmented. Aurora uses the BA offset as a means for calculating age, but such a counter is not immediately available. The Aurora/SBA implementation used an age counter that records the number of choice-points above. We do not maintain such counters, and instead follow the rule:

1. the sequential invariant is guaranteed to hold for private data;
2. shared data in the cactus-stack are protected as regards recovering space, and age follows the simple rule: smaller is older.

To implement this rule, each worker manages the so-called *frozen* registers that separate its private from the shared parts of the tree. Moreover, an extra register, *BB*, replaces the WAM's *B* register when detecting whether a binding is conditional. Note that these same problems must be addressed to support IAP.

The second issue we had to address is *variable representation*. In the original WAM a variable is represented as a pointer to itself. This is unfortunate, because we would need to initialise the whole of the BA. BA based systems (with the exception of Andorra-I [11]) thus assume unbound variables are ultimately null cells. In Aurora, the a new variable is initialised as a tagged pointer to the BA, itself null. In the SBA we do not need pointers to the BA, as it is sufficient to calculate the offset we are at in the shared space, and add it to the SBA base. Aurora/SBA thus initialises a new variable as a tagged age field.

We decided to optimise for the sequential overhead in the YapOr/SBA implementation. To do so, a new cell is initialised as a null field. Moreover, and in contrast to previous BA-based systems, conditional bindings will only be moved to the SBA *when they are made public, and only then*. This means that private execution in our scheme will not use the SBA at all.

As bindings are made public they will be copied to the SBA. Moreover, the original cell will be made to point to the SBA. Thus the variable dereferencing mechanism is unaware of the existence of the SBA. Note that the pointer that is placed in the original cell is independent of workers, although it points at a private structure.

The changes to the engine are therefore quite extensive. As regards the changes to `p_share_work()` and `q_share_work()`, the new algorithm is as follows:

```
         Busy Worker P               Signals            Idle Worker Q
--------------------------- ---------------- --------------------------
Compute stacks to share                      Wait sharing signal
Share private nodes                          .
.                           ----sharing----> .
.                                            Install conditionals
Back to Prolog execution                     Fail to top shared node
```

4 Performance Evaluation

In order to compare the performance of these three models we experimented the 3 systems in two parallel architectures: a Sun SparcCenter2000 with 8 CPUs and 256MB of memory, running Solaris2.7, and a PC server with 4 PentiumPro CPUs/200MHz/256KB caches and 128MB of memory, running Linux2.2.5 from standard RedHat6.0. Each CPU in the PC server is about 4 times as fast as each cpu in the SparcCenter. All systems used the same compilation flags.

We used a standard set of all-solutions benchmarks, widely used to compare ORP logic-programming systems [17]. We preferred all-solutions benchmarks because they are not susceptible to speculative execution, and our goal was to compare the models. The benchmarks include the n-queens problem, the puzzle and cubes problems from Evan Tick's book, an hamiltonian graph problem and a naïve sorting resolution. Table 1 shows the execution time, in seconds, for Yap Prolog and the overhead, in percentage over Yap Prolog, introduced by each or-parallel model when executing with one worker.

Programs	Yap Prolog		YapOr		αCOWL		SBA	
	PC	**Sparc**	**PC**	**Sparc**	**PC**	**Sparc**	**PC**	**Sparc**
cubes5	0.216	0.753	2%	13%	4%	10%	9%	21%
cubes7	2.505	9.042	1%	7%	3%	5%	7%	17%
ham	0.435	1.537	4%	20%	4%	24%	25%	57%
nsort	34.810	142.161	1%	14%	1%	21%	22%	55%
puzzle	2.145	8.411	2%	22%	2%	18%	22%	39%
queens10	0.703	2.809	3%	8%	4%	7%	12%	17%
queens12	20.921	84.600	2%	4%	3%	4%	10%	15%
Average			2%	13%	3%	13%	15%	32%

Table 1. Overheads Yap Prolog/Or-Parallel Models with one worker.

The overhead for copying (YapOr) confirmed previous results, and is of the order of 2% or 13% on PC/Linux and Sparc/Solaris, respectively. The overhead obtained for αCOWL is equivalent to copying. The results are consistent, and the variations are quite above the noise in our measures. We expected performance to be about the same, as for a single processor we execute quite the same code: the systems only differ in their scheduling code, and this is never activated.

The overhead for SBA is, as expected, higher but not very much so, only of 15% or 32%. We believe this good result stems from the optimisations discussed in the previous section. In fact, SBA vs. YapOr performs relatively better than Aurora vs. Muse. We believe this result supports continuing research on the SBA.

Table 2 shows speedups for the PC Server, and Table 3 for the SparcCenter. We use **Copy** for copying and **COW** for the αCOWL. The results show that the best speedups are obtained with copying. The SBA follows quite closely, but the speedups are not as good for higher number of workers. We believe this is partly a problem with the SBA optimisations. As work becomes more fine-grained, more bindings need to be stored in the Binding Array. Execution thus slows down as the system needs to follow longer memory references and touches more cache-lines and pages.

	2 workers			**3 workers**			**4 workers**		
Programs	**Copy**	**COW**	**SBA**	**Copy**	**COW**	**SBA**	**Copy**	**COW**	**SBA**
cubes5	1.99	1.88	1.98	2.97	2.60	2.97	3.95	2.69	3.98
cubes7	1.99	1.98	1.99	2.99	2.92	2.99	3.99	3.83	3.98
ham	1.97	1.90	1.99	2.93	2.64	2.97	3.82	2.79	3.87
nsort	2.03	2.00	1.96	3.06	2.98	2.93	4.08	3.90	3.90
puzzle	1.97	1.93	1.95	2.96	2.41	2.92	3.94	3.20	3.88
queens10	1.99	1.88	1.99	2.96	2.12	2.97	3.92	2.42	3.92
queens12	2.00	1.99	1.99	3.00	2.86	2.99	4.00	3.82	3.98
Average	1.99	1.94	1.98	2.98	2.65	2.96	3.96	3.24	3.93

Table 2. Speedups for the three models on the PC Server.

The results for the αCOWL are quite good, considering the very simple approach we use to share work. The αCOWL performs well for smaller number of processors and for coarse-grained applications. As granularity decreases the overhead of the `fork()` operation becomes more costly, and in general system performance decreases versus other systems. As implemented, the αCOWL is therefore of interest for parallel workstations or for applications with large running times, which are indeed the goal for this study.

5 Conclusions

We have discussed the performance of 3 models for the exploitation of ORP in logic programs. Our results show that copying has a somewhat better perfor-

Programs	2 workers			4 workers			6 workers			8 workers		
	Copy	COW	SBA	Copy	COW	SBA	Copy	COW	SBA	Copy	COW	SBA
cubes5	2.00	1.83	1.96	3.95	2.72	3.70	5.79	2.87	4.88	7.35	2.32	6.02
cubes7	2.01	1.97	1.99	3.98	3.79	3.87	5.97	5.03	5.53	7.74	5.87	7.40
ham	1.98	1.78	1.90	3.79	2.54	3.98	5.57	3.00	5.33	6.97	2.15	7.29
nsort	1.94	1.97	2.02	3.83	3.77	4.01	5.69	5.42	5.88	7.42	6.03	7.77
puzzle	2.02	1.92	1.91	3.94	3.08	3.64	5.91	3.79	5.12	7.68	3.79	7.08
queens10	2.03	1.85	1.93	3.97	2.36	3.82	5.83	2.73	5.48	7.47	2.43	6.95
queens12	2.01	1.95	1.97	4.01	3.74	3.92	5.97	5.13	5.89	7.77	5.81	7.66
Average	2.00	1.90	1.95	3.92	3.14	3.85	5.82	4.00	5.44	7.49	4.06	7.14

Table 3. Speedups for the three models on the SparcCenter.

mance for all-solution search problems. The results confirm the relatively low overheads of copying for ORP systems.

Our results confirm that the SBA is a valid alternative to copying. Although the SBA is slightly slower than copying and cannot achieve as good speedups, it is an interesting alternative for the applications where copying does not work so well. As an example we are using the SBA to implement IAP.

Our implementation of the αCOWL shows good base performance, but suffers heavily as parallelism becomes more fine-grained. Still, we see the αCOWL as a valid alternative for the good reason that the applications that interest us the most have very good parallelism. The αCOWL has two interesting advantages for such applications: it facilitates support of extensions to Prolog, such as sophisticated constraint systems, and it largely simplifies the implementation of garbage collection, that in this model can be performed independently by each worker. The next major challenge for the αCOWL will be the support of suspension, required for single-solution applications.

We would like to perform low-level simulation in order to better quantify how the memory footprints and miss-rates differs between models. Work on these models is progressing apace. We are working on better application support for constraint and inductive logic programming systems. Moreover, we are using copying as the basis for parallelising tabling [15], useful say for model-checking, and the SBA as the basis for IAP [6], that has been used in natural language applications.

Acknowledgments

The authors would like to acknowledge and thank the contribution and support from Eduardo Correia. The work has also benefitted from discussions with Luís Fernando Castro, Inês de Castro Dutra, Kish Shen, Gopal Gupta, and Enrico Pontelli. Our work has been partly supported by Fundação da Ciência e Tecnologia and JNICT under the project Dolphin (PRAXIS/2/2.1/TIT/1577/95) and by Brazil's CNPq under the NSF-CNPq project $CLoP^n$.

References

1. A. Aggoun and et. al. *ECLiPSe 3.5 User Manual*. ECRC, December 1995.
2. K. A. M. Ali and R. Karlsson. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, April 1990.
3. K. A. M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *NACLP'90*, pages 757–776. MIT Press, October 1990.
4. A. Beaumont, S. M. Raman, P. Szeredi, and D. H. D. Warren. Flexible Scheduling of OR-Parallelism in Aurora: The Bristol Scheduler. In *PARLE'91*, volume 2, pages 403–420. Springer Verlag, June 1991.
5. M. Carlsson and J. Widen. SICStus Prolog User's Manual. SICS Research Report R88007B, Swedish Institute of Computer Science, October 1988.
6. L. F. Castro, V. S. Costa, C. Geyer, F. Silva, P. Kayser, and M. E. Correia. DAOS: Distributed And-Or in Scalable Systems. In *EuroPar'99*. Springer-Verlag, LNCS, August 1999.
7. M. E. Correia, F. Silva, and V. S. Costa. Aurora vs. Muse; A Performance Study of Two Or-Parallel Prolog Systems. *Computing Systems in Engineering*, 6(4/5):345–349, 1995.
8. M. E. Correia, F. Silva, and V. S. Costa. The SBA: Exploiting Orthogonality in AND-OR Parallel Systems. In *ILPS'97*, pages 117–131. The MIT Press, 1997.
9. V. S. Costa. COWL: Copy-On-Write for Logic Programs. In *IPPS'99*. IEEE Press, May 1999.
10. V. S. Costa. Optimising Bytecode Emulation for Prolog. In *PPDP'99*. Springer-Verlag, LNCS, September 1999.
11. V. S. Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *ICLP'91*, 1991.
12. L. Damas, V. S. Costa, R. Reis, and R. Azevedo. *YAP User's Guide and Reference Manual*, 1998. `http://www.ncc.up.pt/~vsc/Yap`.
13. G. Gupta and B. Jayaraman. Analysis of or-parallel execution models. *ACM TOPLAS*, 15(4):659–680, 1993.
14. E. Lusk and et. al. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2,3):243–271, 1990.
15. R. Rocha, F. Silva, and V. S. Costa. Or-Parallelism within Tabling. In *PADL'99*, pages 137–151. Springer-Verlag, LNCS 1551, January 1999.
16. R. Rocha, F. Silva, and V. S. Costa. YapOr: an Or-Parallel Prolog System based on Environment Copying. In *EPIA'99*, pages 178–192. Springer-Verlag, LNAI 1695, September 1999.
17. P. Szeredi. Performance Analysis of the Aurora Or-parallel Prolog System. In *NACLP'89*, pages 713–732. MIT Press, October 1989.
18. D. H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog—Abstract Design and Implementation Issues. In *SLP'87*, pages 92–102, 1987.