Three Amigos: A Tale of Three Execution Models for Or-Parallelism

Vítor Santos Costa Ricardo Rocha Fernando Silva

Technical Report Series: DCC-99-2

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+226078830 – Fax: +351+226003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

Three Amigos: A Tale of Three Execution Models for Or-Parallelism

Vítor Santos Costa
COPPE/Sistemas, UFRJ
Brasil
vitor@cos.ufrj.br

Ricardo Rocha, Fernando Silva
Universidade do 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. A major problem that a parallel model must address consists in represent the multiple values that shared variables can be binded to when exploited in parallel. Binding Arrays and Environment Copying are two or-parallel models that efficiently solve that problem. Recently, research in combining independent and-parallelism and or-parallelism within the same system has led to two new binding representation approaches: the Sparse Binding Array (an evolution of binding arrays) and the Copy-On-Write binding models.

In this paper, we investigate whether for or-parallelism the newer models are practical alternatives to copying. To address this question, we experimented with YapOr, an or-parallel copying system using the YAP Prolog engine, and we implemented the Sparse Binding Array (SBA) and the Copy-On-Write (αCOWL) over the original system. The three alternative systems share schedulers and the underlying engine; they differ only in their binding scheme. We compared their performance on a set of well-known all-solutions benchmarks.

1 Introduction

One of the advantages of logic programming is the fact that one can exploit *implicit* parallelism in logic programs. Implicit parallelism reduces the effort required to speedup logic programs through parallelism. Moreover, implicit parallel systems alleviate the user from the actual details of work management, which can be quite difficult to program for the irregular problems commonly addressed in logic programming applications. 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, and ANDP stems from dividing the work required to solve one alternative.

Arguably, *or-parallel* systems, such as Aurora [LBD+90] and Muse [AK90a], have been the most successful parallel logic programming 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. Also, parallelising search can be quite useful for an important extension of Prolog, the constraint-based systems, commonly used for decision-support applications.

Two major issues must be addressed to exploit ORP. First, one must address the *multiple bindings* problem. This problem consists in efficiently represent the multiple values that variables in shared branches of the search tree can be binded to when exploited in parallel. Several mechanisms have been proposed for addressing this problem [GJ93]. Second, the ORP system itself must be able to divide work between each computing agent. This *scheduling* problem is made complex by the dynamic nature of work in ORP systems.

Most modern parallel logic programming systems, including SICStus Prolog [CW88], Eclipse [AA95], and YAP [DSCRA98] 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 each *worker* (or computing agent; or engine; or processor; or process)

work in shared memory, but in separate stacks. Whenever a worker, say P, wants to give work to another, say Q, P simply copies its own stacks to Q. As we shall see, the actual implementation of copying requires quite a few more details.

The major advantage of copying is that it has a quite low overhead over the corresponding sequential system, as was shown in Muse [AK90b]. However, copying does have its problems:

- It is hard to exploit more forms of parallelism than just ORP in copying-based systems. For instance, one particularly interesting form of ANDP is *independent and-parallelism* (IAP), found in divide-and-conquer problems. Because memory management is more complex in the presence of IAP, copying is more suitable to large overheads, reducing substantially the advantages of combining IAP and ORP.
- ORP systems often have to suspend branches during execution. One reason is that a side-effect should only be executed when its execution branch is leftmost. A second reason is that in some forms of search one may be interested in exploiting more interesting branches. Suspending branches in copying-based parallel logic programming systems requires copying the branch to a separate area, and is therefore expensive.

Recent research in the combination of IAP and ORP has led to two new binding representation approaches: the SBA (Sparse Binding Array) [CSS97] and the αCOWL (Copy-On-Write design) [San99a]. The SBA is an evolution of the original Binding Array representation [War87]. In Binding Array systems, the stacks form a cactus-tree representing the search-tree, and workers expand tips of this tree. Bindings that may be different are stored locally, in 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 compare in performance 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 [RSS99b], 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 did they perform comparatively.

The paper is organised as follows. In section 2, we review in more detail the three models. Next, in section 3, we discuss their implementation, and in section 4, we present and discuss experimental results.

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, copy-on-write and sparse binding array.

2.1 Environment Copying

Environment copying was first introduced by Ali and Karlson in the Muse system [AK90a]. In this model each 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 has no work and becomes idle, it searches for a busy worker from which 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. An efficient implementation of this model requires two optimisations:

- *Incremental copying.* The overheads of copying the computation state in a sharing work operation can be reduced by copying just the parts of the execution stacks that are different among the workers involved.

Figure 1 helps to illustrate this strategy. Suppose that worker Q does not find work in its branch, and that there is a worker P with available work. Q asks P for sharing, and backtracks to the first node that is common to P, therefore becoming partially consistent with part of P. Consider that worker P decides to share its private nodes and Q copies the differences between P and Q. These differences are calculated through the information stored in the common node found by Q and in the top registers of the local, heap and trail stacks of P. To fully synchronize the computational state between the two workers, worker Q needs to install from P the bindings trailed in the copied segments that refers to variables stored in the maintained segments.

Obviously, the scheduler plays an important role here by guiding idle workers to request work from busy workers which are nearest (in terms of choice-points).

- *Tree of public or-frames.* In order to efficiently synchronize workers at sharing operations and to avoid having two workers picking the same branches, a tree of *public* (or shared) or-frames is built in a shared space. An or-frame is added to the public tree for each choice-point made public by the worker that is sharing work.

  A bottom-most scheduling strategy has been very successful with this binding model. It requires for a busy worker at a sharing operation to release all of its current private choice-points. This allows for the bottom-most choice-point be selected to maximize the amount of shared work with the requesting worker. This strategy has proven to induce coarse-grained tasks.

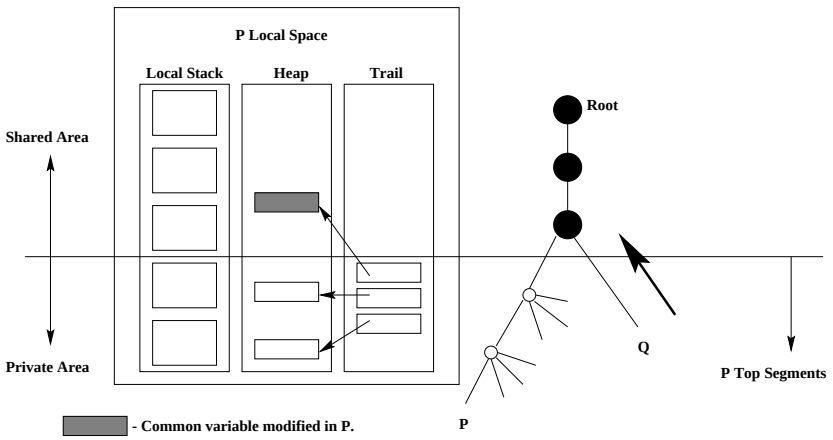

Figure 1: Some aspects of incremental copying.

2.2 Copy-On-Write

This model, named αCOWL, has been proposed by Santos Costa [San99a] with a view to overcome limitations of the environment copying model to support both ANDP and ORP. The model makes use of the copy-on-write technique that has proven effective in Operating Systems.

In the αCOWL, as in environment copying, a separate environment is maintained by each worker and a tree of public or-frames is used to synchronize sharing. The key idea of αCOWL is that: whenever a worker Q wants to share work from a different worker P, it simply *logically* copies all execution stacks from P. The insight is that although stacks will be logically copied, they should be *physically* copied only on demand. To do so, the αCOWL depends on the availability of a Copy-On-Write mechanism on the host Operating System. The αCOWL has two major advantages:

- It is independent of what we are copying, that is, we need not know what to copy, as we logically copy everything. Thus, instead of standard Prolog stacks, we may copy the environment for a constraint solver, or a set of stacks for ANDP computations. Indeed, we might not even have a Prolog system at all, and the model can be used to parallelise any system that uses a similar style of search!

- Because copying is done on demand, we do not need to worry about the overheads of copying large stacks. This is particularly a problem 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.

2.3 Sparse Binding Array

The *Sparse Binding Array* (SBA) derives from the Binding Array model. Binding arrays were originally proposed by Warren for the SRI model [War87]. 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 part of the shared space they own, and they can directly access the stacks originally created by other workers. In Binding Array based systems, workers initially do not inform the system that they have created new alternatives, and thus have exclusive access to them. This is called *private work*. At some point they may be requested to make this work available. They therefore must make the work *public*.

Most, but not all, accesses to both private and public work are read-only. The major updates to public and private work are for bindings of variables. Bindings to the public part of the tree are temptative, 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 they are also stored 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 Binding Array, workers use a private array data structure associated with each computing agent to record conditional bindings. To implement this mechanism, whenever a new variable is created, it is given a variable number uniquely identifying its entry in the binding array. The numbering of variables in the binding array is maintained by a counter that is incremented as every new variable is created. This counter is saved at every parallel choice-point so that whenever a worker attempts to execute an alternative branch it can get a copy of the counter and continue its own numbering of the variables it creates from the same.

Implementing the binding array requires a large number of changes to the original Prolog engine, and imposes an one-processor overhead of around 30% for Aurora [LBD+90], the most well-known implementation of binding arrays. Aurora base performance was substantially slower than modern Prolog systems. Comparison with the copying-based system Muse showed that Aurora had large overheads and could not get improved speedups.

The major advantage of Aurora was the fact that all work is easily available in the tree. This makes it simpler to implement sophisticated techniques to handle, say, speculative computations, often found in single-solution or leftmost-solution problems [Sin93]. On the other hand, Muse did show a good implementation of speculation [AK92]. Copying therefore became the technique of choice for parallel logic programming designers.

Interest in binding arrays was rekindled by the integration of IAP. As we have discussed, copying-based systems cannot handle IAP well [CSS97]. Moreover, IAP also requires fragmented stacks, which is one the issues that complicate the implementation of binding arrays. Unfortunately, the original binding array required strict ordering of variables, which is impossible with IAP [CSS97]. A first proposal, the paged binding array was shown to be too complex [GSC92]. The Sparse Binding Array (SBA) is a simplification of the Binding Array designed to handle IAP.

The major contribution of the SBA is that each worker has a private virtual address space that shadows the system 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. The SBA thus solves the multiple bindings problem.

Note that the shadow area inherits the structure of the shared area. This simplifies implementation, reduces sequential overheads, and allows sharing of the complex stack structure created by ANDP. On the other hand, still requires some modifications to the original Prolog engine and requires

more memory than the original Binding Array, thus increasing task-switching overhead. An initial evaluation of the SBA [SCS96] showed that porting Aurora to the SBA improved performance by a factor of 10–15% on a Sparc machine. The better performance of Aurora in task-switching allowed it to recover this overhead for larger numbers of workers in fine and medium-grained benchmarks,

3 Implementation Issues

The literature includes several comparisons of copying-based versus Binding Array based systems, and particularly of Aurora vs. Muse [BRSW91, CSS95]. 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 our three models by implementing them over the same or-parallel system: the YapOr system [RSS99b]. YapOr is derived from the Yap engine [San99b], which is one of the fast emulator-based Prolog systems currently available. Yap is two to four times faster than the sequential Aurora engine on the same hardware, and only two to three times slower than systems that generate native-code.

3.1 The YapOr

The YapOr system is based on the sequential Yap engine and was originally designed to implement copying. The main changes required to implement YapOr were in the instructions that manipulate choice-points, other changes in the initialisation code for memory allocation and worker creation, some small changes in the compiler to provide extra information for managing ORP, and 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 a shared choice-point. One instruction processes choice-points from standard predicates, and the other choice-points from sequential predicates (alternatives are exploited in order).

The scheduler is the major component of YapOr. Work is represented as a set of *or-frames* in a special shared area. A worker without work consults this area and the `GLOBAL` data-structure, which contains data on work and the status of each worker, until it finds work. If there is no work in the shared tree, the worker becomes idle and tries to share work from 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 to simplify incremental copying, the idle worker moves up in the tree before sharing.

Sharing is implemented through the algorithm in Figure 2. The idle worker waits for a sharing signal while the busy worker prepares the operation. Copying is then performed by the idle worker. The two workers then synchronise again. At the end of the day, the idle worker backtracks whereas the busy one continues.

Note that this latter component is the one dependent on copying. The engine itself communicates only with the scheduler, and does not need to know that copying is used. Further support is required for implementing suspension of work in this model, as stacks must be copied to and from a separate area in order to restart suspended work.

3.2 The αCOWL

To support sharing of work in the αCOWL, we change `p_share_work()` and `q_share_work()` to use the main function used to implement copy-on-write in Unix-style Operating Systems: 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. To start execution, the process Q is forced to backtrack to the deeper choice-point. Note that scheduling is

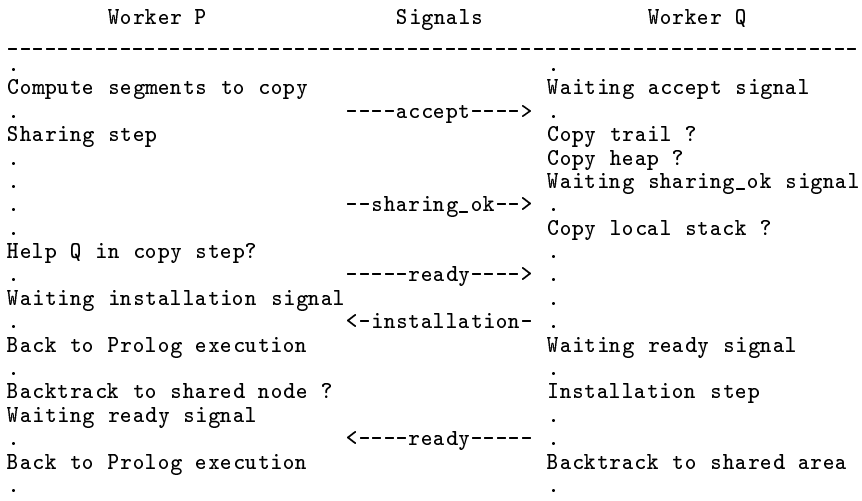

Figure 2: Sharing between workers in YapOr.

realised in exactly the same way as for the environment copying model, i.e., through the use of a public tree of or-frames in shared space.

Figure 3 shows the synchronisation steps in the αCOWL in some detail. The αCOWL also simplifies the memory management and requires an extra process to control the terminal.

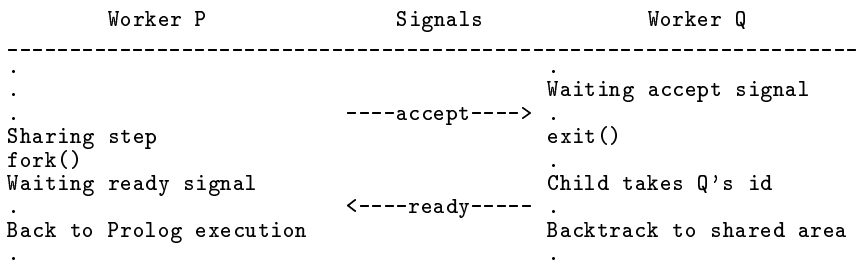

Figure 3: Sharing between workers in αCOWL.

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 The SBA

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 binding array offset as a means for calculating age, but such a counter is not immediately available. The Aurora/SBA implementation uses an age counter that records the number of choice-points above. We do not maintain such counters, and instead use the following 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 and binding 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 binding array. Binding Array based systems (with the exception of Andorra-I) thus assume unbound variables are ultimately null cells. In Aurora, a new variable is initialised as a tagged pointer to the binding array, itself null. In the SBA we do not need pointers to the binding array, 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 Binding Array based systems, conditional bindings will be moved to the SBA only *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 explained in Figure 4. Sharing involves creating an or-frame per choice-point and consulting the trail on which variables are now public.

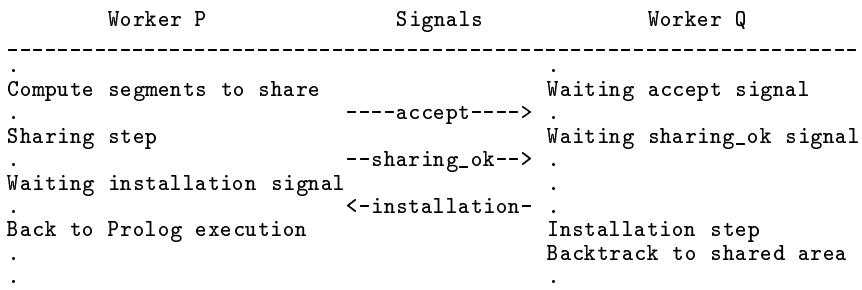

Figure 4: Sharing between workers in SBA.

4 Performance Evaluation

In order to compare the performance of these three models, we experimented the three systems in two parallel architectures: a Sun SparcCenter2000 with eight CPUs and 256MB of memory, running Solaris2.7, and a PC server with four PentiumPro CPUs and 128MB of memory, running Linux2.2.5 from standard RedHat6.0. Each CPU in the PC server runs at 200MHz, contains 256KB of cache memory, and is about four times as fast as each CPU in the SparcCenter. All systems used the same compilation flags. Due to a bug in Linux2.2.5, we could not use `mmap()` for SBA and αCOWL in the PC server. We therefore used the SYSV `shm()` routines for this purpose.

We used a standard set of all-solutions benchmarks, widely used to compare ORP logic-programming systems [Sze89]. We preferred all-solutions benchmarks because they are not susceptible to speculative execution, and our goal was to compare the models. The benchmarks includes the well-know n-queens problem, two puzzles (puzzle and cubes) from Evan Tick's book [Tic91], an hamiltonian graph problem and a naïve sorting resolution.

Table 1 shows the base running times, in seconds, for Yap Prolog, and the one worker running times, over the base runnings, for the parallel models. The overhead for YapOr confirms previous results, and is averaged over all benchmarks between 8% and 12%. The overhead for SBA is, as expected, higher but not very much so, averaged between 13% and 29%. We believe this good result stems from the optimisations discussed in the previous section. In fact, YapOr/SBA performs relatively better than Aurora compared to Muse. We believe this result supports continuing research on the SBA.

Programs	Yap Prolog		YapOr		αCOWL		SBA	
	PC	Sparc	PC	Sparc	PC	Sparc	PC	Sparc
queens12	1 (20.980)	1 (85.970)	1.09	1.05	1.07	1.08	1.09	1.14
queens10	1 (0.710)	1 (2.850)	1.10	1.07	1.07	1.08	1.10	1.16
cubes7	1 (2.545)	1 (9.505)	1.05	1.05	1.03	1.07	1.08	1.14
cubes5	1 (0.220)	1 (0.760)	1.07	1.20	1.05	1.19	1.10	1.25
puzzle	1 (2.290)	1 (9.220)	1.00	1.05	1.00	1.09	1.11	1.23
nsort	1 (35.550)	1 (145.080)	1.11	1.17	1.11	1.23	1.21	1.66
ham	1 (0.470)	1 (1.490)	1.12	1.27	1.11	1.43	1.20	1.46
Average			1.08	1.12	1.06	1.17	1.13	1.29

Table 1: One worker running times over the base sequential runnings for the parallel models.

The most surprising result was obtained for the αCOWL. The αCOWL is slower than copying for Solaris, but faster for Linux! This result is consistent across benchmarks, 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. We explain the bad results from the αCOWL in the SparcCenter due to the virtual-memory cache. We obtained similar results in other experiments, and we believed they stemmed from the virtual-memory cache used in the SparcCenter. There is a limited number of tags for address spaces, and due to forking, the αCOWL requires more tags than the original YapOr. In contrast, we suppose the good results with the αCOWL in Linux derive from the problems in the Linux `mmap()` implementation, that is being used for copying.

Table 2 and 3 show speedups relative to the one worker running time on the PC server and SparcCenter, respectively. 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 follows longer memory references and touches more cache-lines and pages.

Programs	2 workers			3 workers			4 workers		
	YapOr	αCOWL	SBA	YapOr	αCOWL	SBA	YapOr	αCOWL	SBA
queens12	2.00	1.99	2.00	3.00	2.87	2.99	4.00	3.75	3.99
queens10	1.98	1.84	1.99	2.90	1.90	2.93	3.86	2.03	3.91
cubes7	2.00	1.99	2.02	2.99	2.91	3.03	3.98	3.65	4.05
cubes5	2.00	1.89	2.02	2.97	2.52	3.04	3.95	2.18	4.02
puzzle	1.98	1.95	1.91	2.97	2.38	2.84	3.96	2.74	3.79
nsort	2.00	2.02	1.92	3.01	2.95	2.86	4.02	3.86	3.82
ham	1.99	1.88	1.98	2.95	2.70	2.94	3.90	2.50	3.85
Average	1.99	1.94	1.98	2.97	2.60	2.95	3.95	2.96	3.92

Table 2: Speedups relative to the one worker running time 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 workers and for coarse-grained applications. As granularity decreases, the overhead of the `fork()` operation becomes more costly, and system performance decreases versus other systems. As implemented, the αCOWL is therefore of interest for parallel workstations or for applications with large running times.

Programs	2 workers			4 workers			6 workers			8 workers		
	YapOr	αCOWL	SBA	YapOr	αCOWL	SBA	YapOr	αCOWL	SBA	YapOr	αCOWL	SBA
queens12	2.01	2.00	1.97	4.05	3.73	3.90	6.07	5.28	5.87	8.07	5.67	7.57
queens10	1.99	1.77	1.91	3.90	2.22	3.78	5.49	2.43	5.38	7.31	1.93	7.01
cubes7	2.00	1.95	1.97	4.00	3.71	3.66	5.94	4.83	5.28	7.84	5.15	7.14
cubes5	1.99	1.77	1.99	3.94	2.54	3.78	5.76	2.59	4.49	7.44	1.78	5.91
puzzle	1.97	1.88	1.81	3.59	2.71	3.57	5.37	3.19	5.09	7.06	3.13	6.82
nsort	2.04	2.01	2.25	3.90	3.82	4.51	5.97	5.49	6.69	7.38	6.35	7.54
ham	1.98	1.75	1.91	3.72	2.61	3.53	5.52	2.42	5.19	7.09	1.93	6.71
Average	2.00	1.88	1.97	3.87	3.05	3.82	5.73	3.75	5.43	7.46	3.71	6.96

Table 3: Speedups relative to the one worker running time on the SparcCenter.

5 Conclusions

We have discussed the performance of three models for the exploitation of ORP in logic programs. Our results show that copying has a somewhat better performance 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 since the applications which 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 simulate low-level simulation in order to better quantify how the memory footprints and miss-rates differs among models. 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 [RSS99a], useful for model-checking, and the SBA as the basis for IAP [CCG+99], which 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 projects Melodia (JNICT/PBIC/C/TIT/2495/95) and Dolphin (PRAXIS/2/2.1/TIT/1577/95).

References

[AA95] et. al. Abderrahamane Aggoun. *ECLiPSe 3.5 User Manual.* ECRC, December 1995.

[AK90a] Khayri A. M. Ali and Roland Karlsson. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, April 1990.

[AK90b] Khayri A. M. Ali and Roland Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *Proceedings of the North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[AK92] Khayri A. M. Ali and Roland Karlsson. Scheduling Speculative Work in Muse and Performance Results. *International Journal of Parallel Programming*, 21(6):449–476, December 1992. Published in Sept. 1993.

[BRSW91] Anthony Beaumont, S Muthu Raman, Péter Szeredi, and David H. D. Warren. Flexible Scheduling of OR-Parallelism in Aurora: The Bristol Scheduler. In *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 403–420. Springer Verlag, June 1991.

[CCG+99] L. F. Castro, V. Santos 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.

[CSS95] Manuel E. Correia, Fernando Silva, and Vítor Santos Costa. Aurora vs. Muse; A Performance Study of Two Or-Parallel Prolog Systems. *Computing Systems in Engineering*, 6(4/5):345–349, 1995.

[CSS97] Manuel E. Correia, Fernando Silva, and Vítor Santos Costa. The SBA: Exploiting Orthogonality in AND-OR Parallel Systems. In *ILPS97*, pages 117–131. The MIT Press, October 1997.

[CW88] Mats Carlsson and Johan Widen. SICStus Prolog User's Manual. SICS Research Report R88007B, Swedish Institute of Computer Science, October 1988.

[DSCRA98] L. Damas, V. Santos Costa, R Reis, and R. Azevedo. *YAP User's Guide and Reference Manual*, 1998. `http://www.ncc.up.pt/~vsc/Yap`.

[GJ93] Gopal Gupta and B. Jayaraman. Analysis of Or-parallel Execution Models. *ACM TOPLAS*, 15(4):659–680, 1993.

[GSC92] Gopal Gupta and Vítor Santos Costa. And-Or Parallelism in Full Prolog with Paged Binding Arrays. In *PARLE'92 Parallel Architectures and Languages Europe*, pages 617–632. Springer-Verlag, LNCS 605, June 1992.

[LBD+90] Ewing Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, David H. D. Warren, A. Calderwood, P. Szeredi, Seif Haridi, P. Brand, M. Carlsson, A. Ciepelewski, and B. Hausman. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2,3):243–271, 1990.

[RSS99a] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. Or-Parallelism within Tabling. In *1st International Workshop on Practical Aspects of Declarative Languages*, pages 137–151. Springer-Verlag, LNCS 1551, January 1999.

[RSS99b] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. YapOr: an Or-Parallel Prolog System based on Environment Copying. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence*, pages 178–192. Springer-Verlag, LNAI 1695, September 1999.

[San99a] Vítor Santos Costa. COWL: Copy-On-Write for Logic Programs. In *Proceedings of IPPS'99*. IEEE Press, May 1999.

[San99b] Vítor Santos Costa. Optimising Bytecode Emulation for Prolog. In *Proceedings of PPDP'99*. Springer-Verlag, LNCS, September 1999.

[SCS96] Vítor Santos Costa, Manuel E. Correia, and Fernando Silva. Performance of Sparse Binding Arrays for Or-Parallelism. In *Proceedings of the VIII Brazilian Symposium on Computer Architecture and High Performance Processing – SBAC-PAD*, August 1996.

[Sin93] Raéd Sindaha. Branch-Level Scheduling in Aurora: The Dharma Scheduler. In *ILPS93*, pages 403–419, 1993.

[Sze89] Péter Szeredi. Performance Analysis of the Aurora Or-parallel Prolog System. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732. MIT Press, October 1989.

[Tic91] Evan Tick. *Parallel Logic Programming*. MIT Press, 1991.

[War87] David H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog—Abstract Design and Implementation Issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.