Performance of the Compiler-based Andorra-I System[1]

Rong Yang, Tony Beaumont, Inês Dutra, Vítor Santos Costa and David H D Warren

Department of Computer Science
University of Bristol
Bristol BS8 1TR, U.K.

Abstract

Andorra-I is an experimental parallel Prolog system based on the Basic Andorra model. This model supports both dependent and-parallelism, by executing determinate goals in parallel, and or-parallelism, stemming from the nondeterminate goals. In this paper, we present a new compiler-based version of the Andorra-I system and discuss its performance. We study the system with a wide set of real-life, non-trivial logic programming applications. As Andorra-I provides a powerful programming model, we are able to include Prolog programs, committed-choice programs, and Andorra-style programs. The results show that the system is very effective at exploiting both forms of parallelism, that it compares well with exclusively or-parallel and exclusively and-parallel systems, and that the extra complexity of the model is manageable. Basic performance compares quite well with other compiler-based systems such as SICStus and JAM.

1 Introduction

Andorra-I is an experimental parallel Prolog system based on the Basic Andorra model. The essential idea of the Basic Andorra Model is that, instead of executing goals according to the simple left-to-right selection function used in standard Prolog, we first execute all goals for which at most one clause matches (namely *determinate goals*). These goals can be executed in (and-) parallel. When there are no such determinate goals left to execute, we select a goal from the remaining goals (namely *nondeterminate goals*) to make a choicepoint as in Prolog. The different branches of the choicepoint can be explored in (or-) parallel. Therefore, the Basic Andorra Model supports both dependent and-parallelism (between the determinate goals) and or-parallelism (for the nondeterminate goals). Moreover, the parallelism can be exploited entirely transparently, and no explicit user annotations are needed.

In designing the Andorra-I system we had two goals. Firstly, we intended to investigate the model's feasibility, that is, whether the extra complexity

[1]This report appears in the Proceedings of the 10th International Conference on Logic Programming, MIT Press, June 1993

introduced by the model is manageable and whether parallelism in programs can be exploited efficiently. Secondly, we wanted to investigate whether the model is useful for a wide range of applications. To test the model's feasibility, Andorra-I is a full parallel implementation that realises the model in shared memory-machines. To investigate the model's usefulness, we experimented with the system on a wide range of applications.

We have previously presented[16, 17] the main implementation issues and early performance results of the first, interpreter-based, version of Andorra-I. Since then, a much faster, compiler-based, version of Andorra-I has been developed, and tested on a wide variety of applications.

In this paper, we focus on the compiler-based version of the Andorra-I system. We are interested in the efficiency of the new system and therefore analyse its performance for a range of interesting applications of logic programming. We investigate both the sequential performance, that is how the system compares to traditional, sequential, logic systems such as a standard Prolog implementation, and the parallel performance, that is the speedups obtained from the system and how well they compare to other parallel logic programming systems.

To study the performance of the system we use a set of real-life, non-trivial logic programming applications. We believe these examples give a representative picture of the system's range of application.

The paper is organised as follows. We first present the major concepts and components in the design of the Andorra-I system. Next, we describe in more detail the applications that form our benchmark set. We then present tests of Andorra-I on these benchmarks. We give a picture of the system's overall performance, as compared to other logic programming systems, and then concentrate on the parallel speedups, on the basic performance and on the parallel overheads. We finally summarise our work and give the main conclusions.

2 Implementation of Andorra-I

2.1 The Andorra-I Components

The Andorra-I system can be divided into three main components: the *preprocessor*, the *engine* and the *scheduler*. The preprocessor takes a source program as input and generates the code that is executed by the engine. The engine executes this code, i.e. gives all the functionality of the Basic Andorra model. The scheduler is invoked by the engine to find work when its supply of work is exhausted.

2.1.1 The Preprocessor and the Engine

The preprocessor[17] consists of four components: a mode analyser, a sequencer, a determinacy code generator and a compiler.

After preprocessing, Andorra-I programs are executed by the engine[16]. The engine in fact consists of teams of abstract agents called *workers*, where each worker normally corresponds to a physical processor. Each team, when active, is associated with a separate or-branch in the computation tree and is in one of two computation phases:

Determinate: As long as determinate goals exist in the or-branch, all such goals are candidates for immediate evaluation. This phase ends when no determinate goals are available, or when any goal fails. In the first case, the team moves to the nondeterminate phase. If a goal fails, the corresponding or-branch must be abandoned. The team will then backtrack and eventually try to find a new or-branch to explore.

Nondeterminate: If no determinate goals exist, the leftmost goal (or the particular goal specified by the user) is reduced. A choicepoint is created to represent the fact that the current or-branch has now forked into several or-branches, while the team itself will explore one of the or-branches. If other teams are available, they can be used to explore the remaining or-branches.

We follow the SRI model[21] as a way of providing support for or-parallelism, the implementation of which follows that of the Aurora or-parallel Prolog system[12]. Each team therefore has a binding array to keep the conditional bindings of variables.

To support and-parallel execution, locking of variables is introduced.

Our memory management system, necessary to support both and- and or-parallel execution, divides the stacks into several segments, one for each team (i.e. each or-branch). The segments are further divided into smaller chunks to allow workers who are working in and-parallel within a team to allocate entries independently.

We have also available a sequential version of Andorra-I in which the implementation to support parallel execution such as binding array, variable locking, stack chunking and calling to the scheduler are removed.

2.1.2 The Scheduler

During either the determinate or nondeterminate phase, each worker works privately on a piece of work (called a task). When a worker completes a task, it needs to find a new task to work on. In this case, the engine calls the scheduler, another component of Andorra-I.

There are two cases when a worker needs to find a new task. The first is in the determinate phase. In this case, as long as other members in the team are still busy, the scheduler takes an and-task (i.e. a goal) from one of the busy members to let the worker continue work in the determinate phase.

The second case is in the nondeterminate phase. As in Aurora, the search tree in Andorra-I is divided into private and public regions. In the private region, the engine backtracks to take new alternatives as in normal Prolog. When a worker has exhausted all private branches, it will ask the scheduler to find an or-task (i.e. an alternative) from the public region. Scheduling or-tasks is handled by the Bristol scheduler[2], developed for Aurora, which we have plugged directly into Andorra-I.

In addition to finding a task, the scheduler must also be able to intelligently adapt team organisation to the available parallelism. For example, if a team is "under employed", and there are fewer and-tasks than there are workers in the team, the scheduler should move some workers out of the team, either to join other busy teams, or to create a new team according to some criteria. This is the work of the top-scheduler[7].

2.2 Compiler-Based Andorra-I

Andorra-I was first developed as an interpreter-based system[16], although it did include a small emulator for the determinacy code. The compiler-based version preserves the interpreter's data-structures, whilst including a full compiler and emulator. The decision to preserve the interpreter's data-structures was based on the good performance of the interpreted version.

The system uses an abstract machine, the Andorra-I Abstract Machine, that is similar to that in the parallel PARLOG system, JAM[5], but extended with features such as backtracking and true head unification.

Three forms of unification are supported in Andorra-I: *read-only*, for executing determinacy code, *write-only*, for setting arguments in the body of a clause, and *read-write*, for normal unification. The three forms of unification are supported by different sets of instructions. For example, the clause:

```
a([a,X],f(X),Y) :- integer(Y), !.
```

is translated into the following code illustrating read-only unification:

```
wait_list  1,wait,Lfail1            % a([
read_constant  a,wait,Lfail2        %    a
read_list  wait,Lfail3              %   ,
read_x_variable  4                  %    X
read_nil  wait,Lfail4               %   ],
wait_structure  f/1,2,wait,Lfail5   %   f(
read_x_value  4,wait,Lfail6         %     X),
wait_x_variable  3,3,wait,Lfail7    %   Y) :-
bip_1  integer,3,wait,Lfail8        % integer(Y),
proceed                             % !.
```

The last two arguments in the above read-only instructions are labels which indicate the continuation of computation after the events of "suspend-on-variable" and "fail" respectively (called a wait-label and a fail-label). In this simple clause, as there are no multi-suspensions, all wait-labels are just a key word "wait" which means to stop the read-only unification immediately after a suspension record for the current variable is created. If there were a real label, the computation would continue at that point. The fail-labels in the code vary depending on what the other alternative clauses are.

Andorra-I is a pure goal-stacking system. In contrast to JAM, no environments are supported. Since creating a goal-frame can be quite expensive, Andorra-I reduces the number of goal frames it needs to create by always having a *current goal* in its registers. If the current goal reduces determinately into a single new goal, no stack frame is created for the new goal. If it reduces determinately into several goals, the stack frame for the new current goal is only partially created. Consider the small procedure:

```
a(X) :- b(X), c(X), d(X).
```

The Andorra-I compiler generates the following code:

```
create_last  d/1,3,3      % d(
put_x_value  1,1          %   X)
create  c/1               % c(
put_x_value  1,1          %   X)
create_first  b/1         % b(X)
neck  3
```

Note that goals are pushed in reverse order, and that the arguments can only be stored in a goal frame after the frame was created. Of interest also are the different flavours of the `create` instruction. The `create_last` instruction starts execution for the body of a goal. The `create_first` instruction only needs to create a new goal-frame if this is a nondeterminate reduction, and does not need to store the goal arguments, as the leftmost subgoal (in this case `b(X)`) will usually be the new current goal.

The Andorra-I abstract machine also includes several instructions to support the determinacy code. These instructions perform tests on arguments and subarguments, call builtins, and implement fast switches. Choice point creation and manipulation is supported through `try`, `retry` and `trust` instructions.

2.3 The Current Implementation

The determinacy code is created by a determinacy compiler[17], whilst a clause compiler does clause-by-clause compilation. The clause compiler includes some optimisations designed to improve register allocation, ground term representation and builtin processing. The builtin processing optimisations try to avoid creating a separate goal-frame for some builtins. The ground term optimisation simplify unification for large ground terms.

The new Andorra-I version can use the determinacy code for indexing of nondeterminate goals. The determinacy compiler maintains the clauses that are still accepted for each result of a test in the determinacy code. At runtime, the system looks at the points where it suspended in the determinacy code and tries only the clauses that were still matching at one of these points.

The current emulator is based on the original determinacy code emulator, and is compatible with the interpreter engine (this causing some overheads in the actual execution).

3 The Applications and Benchmarks

We have selected a number of substantial application programs to show the parallel performance of Andorra-I. Some of these are standard Prolog applications; others demonstrate the particular advantages of the Basic Andorra Model.

3.1 Applications

The application programs include an aircraft flight scheduling system based on work by Software Sciences for the Royal Navy, a road edge recognising algorithm from the vision group of our Department, a deductive database system for reasoning about protein structure developed at Imperial Cancer Research Fund, and a natural language question answering system from the University of Edinburgh (Chat-80), and two industrial applications from British Telecom: a workforce management system and a clustering algorithm for network management.

We now give some more details of these applications.

bt_cluster: a clustering algorithm for network management from British Telecom[4].

The program receives a set of points in a three dimensional space and groups these points into *clusters*. Basically, three points belong to the same cluster if the distance between them is smaller than a certain limit. To obtain best performance, we rewrote the original application to become a determinate-only computation. And-parallelism in this case naturally stems from running the calculations for each point in parallel. The test program uses a cluster of 400 points as input data.

bt_wms: a workforce management system from British Telecom[13].

The program generates daily job tours for the company. The target of the system is to allocate the jobs to the technicians, minimising as far as possible the number of unallocated jobs and the cost of each technician's tour. The original program was adapted to Andorra-I by flattening some guards. The input data in our benchmark are 250 jobs which are distributed in different locations and 118 technicians who are from 9 different bases. This is a sophisticated industrial application and an efficient Prolog program, although the potential parallelism is rather limited.

road_markings: a program to extract road markings from the vision research group, in the Department of Computer Science at the University of Bristol[18].

This program is adapted from a Ph.D. project to extract road markings from real world images that have been processed to identify a set of visible objects. The algorithm fits either straight lines or quadratic curves through these objects. The "best" set of curves is taken as being the road boundaries, where "best" is defined by certain heuristics such as the width between two road boundaries being constant and greater than a vehicle-width apart. This application contains and-parallelism, corresponding to parallelising the operations on each pair of objects. The testing program takes 16 objects as the input data.

chat_80_db5: an example from a natural language question-answering system Chat-80.

Chat-80 is a well-known natural language question answering system[23]. This version of Chat-80 operates on the domain of world geography. The program chat_80_db5 makes a query to the Chat-80 database. It contains or-parallelism, and it has been used as one of the or-parallel Prolog benchmarks for Aurora.

protein: an example from a deductive database system for reasoning about protein structure, developed at Imperial Cancer Research Fund[15, 11].

The protein program searches for so-called 'Greek Keys' in *Prealbumin* (human plasma) *p2ab*. This application contains or-parallelism and has been used to benchmark systems such as Aurora. We run the program in two different manners: finding all solutions and finding the first solution. The programs named **protein_all** and **protein_1st** correspond respectively to finding all the solutions and the first solution.

fly_pan: an aircraft flight scheduling system.

Fly_pan is a version of a naval flying programme generator, based on a system developed by Software Sciences and the University of Leeds for the Royal Navy, and is an example of a real-life resource allocation problem. The program allocates airborne resources (such as aircraft) whilst taking into account a number of constraints. The problem is solved by using the technique of active constraints as first implemented for Pandora[1]. In this technique, the coroutining inherent in the Andorra model is used to activate constraints as soon as possible. The program has both or-parallelism, arising from the different possible choices, and and-parallelism, arising from the parallel evaluation of different constraints. The input data used in the testing program are 11 aircraft, 36 crew members, and 10 flights to be scheduled.

3.2 Performance Benchmarks

Apart from the above applications, we also selected some smaller benchmarks in order to evaluate overall performance.

The details of these benchmarks are given below.

nrv400: naïve reverse of a list of 400 elements.

This is the only small and trivial program used in our benchmarks. The program can greatly benefit from register optimisation and it contains a lot of and-parallelism. Therefore, it has been used as a benchmark on most Prolog systems and committed choice systems.

5x4x3_puzzle: a famous puzzle which has been used as a symbolic computation benchmark implemented in many different languages including 'C', Pascal, Lisp and Prolog[8].

This is a solid-packing problem wherein a solid is filled with seven small irregularly shaped pieces. In the original puzzle, the size of solid is 5x5x5. A smaller puzzle with size 5x4x3 was developed as a parallel logic programming benchmark[19]. This is a typical generate-and-test style program. It creates a very large search tree, thus it generates a lot of or-parallelism.

warplan: a well-known general purpose plan generator[20].

This benchmark creates a plan in the blocks world to solve an 8 blocks problem.

scanner: a scanner program to reveal the content of a bitmap.

The program is an AKL[10] benchmark, based on an problem from Dewdney[6]. The program reveals the content of a box (bitmap). The input is the number of dots on each row, column, left diagonal, and right diagonal. The bitmap used in the benchmark is a picture of a star. The program contains both or-parallelism corresponding to different choices for the unknown bits (either a dot or a blank), and and-parallelism corresponding to parallel propagation of determinate bits and evaluation of constraints.

cipher: a simple substitution decoding system developed as a demonstration of Andorra-I[24].

The system reads a ciphertext and decodes it (assuming the original message is in English). The program performs a lot of letter matching, first determinately in and-parallel, then, when only nondeterminate matching is left, the program creates several choices (giving or-parallelism). An or-branch fails when the number of unmatchable words exceeds a certain limit. The algorithm takes advantage of the Basic Andorra Model to greatly reduce the search space. The length of the ciphertext used in the benchmark is 56 letters, and the maximum number of unmatchable words is set to 2.

We use the applications and benchmarks described above, giving a total of 11 programs, to test Andorra-I performance, and present the results in the next section. On the whole, we have tried to select examples from "real" programs rather than artificial benchmarks.

Program Name	Style	Parallelism
nrv400	Prolog, totally determinate	and-
bt_cluster	Prolog, totally determinate	and-
bt_wms	Prolog, totally determinate	(minor) and-
road_markings	Prolog, det.+ minor non-det.	and-
chat_80_db5	Prolog, nondeterminate	or-
5x4x3_puzzle	Prolog, nondeterminate	or-
warplan	Prolog, nondeterminate	speculative or-
protein_all	Prolog, nondeterminate	or-
protein_1st	Prolog, nondeterminate	speculative or-
fly_pan	Andorra-I, nondeterminate	and- + or-
scanner	Andorra-I, nondeterminate	and- + or-
cipher	Andorra-I, nondeterminate	and- + or-

Table 1: A LIST OF THE BENCHMARKS

Table 1 gives a summary of the applications and benchmarks. The programs marked with 'Prolog' in the style column are originally Prolog programs. The last three programs are originally written in an Andorra-I style language, i.e. taking advantage of Andorra-I's coroutining. Some of them (*scanner* and *cipher*) rely strongly on the coroutining mechanism and cannot be run in Prolog.

The first 4 programs contain only and-parallelism. The middle four benchmarks in the table contain only or-parallelism. The last three benchmarks contain both and- and or- parallelism.

4 The Performance Results

In this section, we present and discuss the performance results of Andorra-I. The first table (Table 2) shows the relative speeds of sequential and parallel versions of Andorra-I compared with other systems. Other tables (Table 3 to Table 7) give additional information.

4.1 Test Conditions

The machine used to obtain the results presented here is a Sequent Symmetry containing 12 processors. All Andorra-I results were obtained using the compiler-based version.

We used our flexible top-scheduler[7] to automatically configure teams. That is, for exclusively and-parallel benchmarks, Andorra-I automatically configures a single team; all workers are working in and-parallel. For exclusively or-parallel benchmarks, Andorra-I automatically configures 10 teams; all workers are working in or-parallel. For benchmarks with mixed parallelism, the team configuration is dynamically changed according to the kind of parallelism that is available. Thus, under the flexible top-scheduler, Andorra-I requires no manual adjustment.

The times we give are actual execution times. There are some variations of the parallel timing results between each run, mainly, we believe, due to speculative parallelism. To give the full parallel picture, we present the best and the worst parallel times for a set of 10 runs.

4.2 Overall Performance

Table 2 compares the overall performance of sequential and parallel versions of Andorra-I with SICStus and JAM, in terms of relative speed with respect to the parallel version of Andorra-I on a single processor.

The column for SICStus Prolog shows that, for the Prolog style programs, the parallel version of Andorra-I with single processor is, on average, about 3.7 times slower than SICStus Prolog. With 10 processors, Andorra-I is

prog name	SICStus 2.1	sequential Andorra-I	parallel Andorra-I			JAM
			1 p	10 p		1 p
				best	worst	
nrv400	2.17	1.26	1	8.25	8.07	1.25
bt_cluster	3.27	1.29	1	9.37	9.02	1.81
bt_wms	2.75	1.23	1	3.32	3.30	
road_markings	4.39	1.67	1	6.24	5.31	
chat_80_db5	2.24	1.26	1	7.30	6.59	
5x4x3_puzzle	6.56	1.47	1	9.66	9.56	
warplan	4.43	1.65	1	3.36	2.87	
protein_all	3.90	1.51	1	6.81	5.51	
protein_1st	3.25	1.48	1	4.20	3.71	
fly_pan	0.02	1.25	1	6.88	5.07	
scanner		1.48	1	5.47	4.41	
cipher		1.51	1	5.65	4.67	

(Relative Speed)

Table 2: The overall performance of the compiler-based system

considerably faster than SICStus Prolog on most examples. In relation to the parallel PARLOG system, JAM, Andorra-I is about 1.5 times slower due to overheads in supporting nondeterminate goals and or-parallelism.

4.2.1 Computation Size

In order to make further comparisons with Prolog, we need to discuss the important issue of computation size. Andorra-I and Prolog use different selection functions and a program might require a different number of resolutions when run on Andorra-I than when run on a Prolog system.

To illustrate this further, Table 3 lists the number of resolutions required for each program in both Prolog and Andorra-I. We count each successful reduction and built-in predicate as one resolution.

prog name	Prolog	Andorra-I	
		1 processor	10 processors
nrv400	80,612	80,612	80,612
bt_cluster	498,617	498,617	498,617
bt_wms	889,183	889,183	889,183
road_markings	23,338	23,338	45,376
chat_80_db5	6,373	6,373	6,373
5x4x3_puzzle	135,108	135,108	135,108
warplan	48,102	47,792	112,238
protein_all	125,898	106,132	117,902
protein_1st	26,334	18,846	28,914
fly_pan	7,976,002	41,019	42,319
scanner	NA	118,467	118,215
cipher	NA	10,324	11,096

Table 3: Number of resolutions in Prolog and Andorra-I

The first and second columns show the number of resolutions done by Prolog and by a single processor Andorra-I system respectively. As we would expect, for Prolog programs that are purely determinate, Andorra-I and Prolog both take same number of resolutions.

The first three nondeterminate benchmarks, *chat_80_db5*, *5x4x3_puzzle* and *warplan* also require approximately the same number of resolutions under Prolog and under Andorra-I. This is because most of the goals in these programs are nondeterminate and the execution order will be the same in both Prolog and Andorra-I.

The differences start to appear for *protein* and *fly_pan*, where Andorra-I takes a significantly smaller number of resolutions than Prolog. The program *fly_pan* was originally written for Pandora and takes good advantage of the Basic Andorra Model. Thus, running it under Prolog's depth-first left-right order is very inefficient, as nearly 200 times more resolutions are performed by Prolog. This explains why the relative speed of SICStus Prolog for this program is very slow in Table 2. In contrast to *fly_pan*, the *protein* program is written for Prolog, and when executed on Andorra-I, the number of resolutions performed is reduced by about 28%. This example demonstrates that even for some applications originally designed for Prolog, Andorra-I can automatically cut down the search space.

Another interesting issue is whether the computation size changes when we run a program in parallel. The third column in Table 3 shows the number of resolutions obtained under parallel execution using 10 processors. Obviously, for the programs which are totally determinate, there is no difference at all. Some small variations appeared in *fly_pan*, *scanner* and *cipher* and these are due to speculative and-parallelism. A large difference appeared for *protein_1st* and *warplan* and this is due to speculative or-parallelism.

The *road_marking* program is another instance of speculative or-parallelism. Although its major computation is determinate there is one nondeterminate goal allocated near the end of the execution. One of the alternatives for this nondeterminate goal leads to an infinite computation, which is pruned with a cut in the program. When Andorra-I executes the program, one single team is configured for the major computation to exploit and-parallelism. When the team reaches that nondeterminate goal, most of slaves move out of the team to work on or-tasks because there is no more and-parallelism available. There are plenty of speculative or-tasks available for these processors on the infinite branch, but this work is always pruned. That is why the number of reductions is considerably increased when we run the program with 10 processors.

4.2.2 LIPS

Logical Inferences Per Second, *LIPS*, gives a certain absolute measure of performance. In Table 4 we summarise the overall performance of Andorra-I in terms of LIPS. The number of LIPS for each system is calculated by dividing the actual number of resolutions by the actual execution time on that system. Note that on some examples Andorra-I performs fewer resolutions

prog name	SICStus 2.1	Sequential Andorra-I	Parallel Andorra-I		
			1 processor	10 processors	
				best	worst
nrv400	23.3	13.5	10.7	88.6	86.7
bt_cluster	16.3	6.5	5.0	46.9	45.1
bt_wms	13.0	5.8	4.7	15.8	15.7
road_markings	8.9	3.4	2.0	24.8	21.1
chat_80_db5	4.2	2.4	1.9	13.8	12.5
5x4x3_puzzle	8.8	2.0	1.3	13.0	12.9
warplan	7.0	2.6	1.6	12.4	10.6
protein_all	6.2	2.0	1.3	10.2	8.2
protein_1st	6.2	2.0	1.4	8.8	7.7
fly_pan	14.8	3.9	3.1	22.4	16.5
scanner	NA	5.0	3.4	18.4	14.8
cipher	NA	1.9	1.2	7.5	6.2

Table 4: Overall performance in KLIPS

than SICStus, and therefore the LIPS ratings are not directly comparable.

Table 4 shows that Andorra-I can get up to 88 KLIPS on a Symmetry machine with 10 processors. The cost per reduction varies from program to program. For the *cipher* program, the cost of each reduction is over ten times more expensive than for the cheapest program, *nrv400*. This is quite reasonable because for *nrv400* most of goals are directly executed in the argument registers without being pushed back to local stack. However, in the *cipher* program, most of goals need to suspend several times until finally being executed, hence the lower LIPS rate.

4.3 Parallel Speedups

Table 2 showed the speedups obtained using 10 processors. In general, these speedups are quite good. For those programs which have a large amount of parallelism (eg. *nrv400, bt_cluster, 5x4x3_puzzle*), we have obtained speedups which are close to 10.

The program *road_marking* would produce better speedups if we had used a larger data set. In practice the input data would be quite large but for our benchmark purposes the data set contained only 16 objects. Despite this, we believe the speedups we have obtained are quite reasonable for this size of the data set.

The program *bt_wms* shows rather small speedups. This is because, in allocating the next job, the program needs to use the data produced from allocating the current job. Thus, the major and-conjunctions are sequentialised due to the data dependency. Moreover, once certain conditions hold, a job can be allocated to a tour determinately, so removing any scope for or-parallelism. Therefore, the results for *bt_wms* are actually quite encouraging. They show that for programs which are based on a sequential algorithm, Andorra-I is still able to exploit some very fine grained parallelism. Recently, a new version of *bt_wms* with better parallelism[14] has been developed in BT, and their results show that Andorra-I can produce speedups of around 7 with 10 processors using this new version.

In order to compare the performance of Andorra-I with that of other parallel logic programming systems, we have run our benchmark programs (where possible) on both JAM and Aurora (with the current version of the Bristol scheduler). The results for the best speedups (relative to each sys-

prog name	Andorra-I	JAM	Aurora
nrv400	8.25	8.37	
bt_cluster	9.37	9.70	
chat_80_db5	7.30		7.30
5x4x3_puzzle	9.66		9.51
warplan	3.36		2.63
protein_all	6.81		9.49
protein_1st	4.20		4.10

Table 5: SPEEDUPS (10 PROCESSORS) UNDER DIFFERENT SYSTEMS

tem's single processor speed) are shown in Table 5. It shows that the or-parallel speedup of Andorra-I is comparable with Aurora, and that the and-parallel speedup of Andorra-I is comparable with JAM. This is despite the fact that Andorra-I is a more sophisticated and-or parallel system. Note that the *warplan* and *protein_1st* benchmarks contain significant or-speculative work, and that Andorra-I is able to obtain reasonable speedups through incorporating the latest version of the Bristol scheduler[3].

The last three benchmarks in Table 2 contain both and- and or-parallelism. Since we don't have an equivalent system to compare, we ran them on Andorra-I with a fixed team configuration forcing Andorra-I to exploit only one form of parallelism. Table 6 shows the results from the experiment.

prog name	And Parallel Only	Or Parallel Only	And + Or Parallel
fly_pan	6.29	1.16	6.88
scanner	5.07	2.83	5.47
cipher	2.52	5.43	5.65

Table 6: Speedups (10 processors) for and- and or-parallelism

The first column shows the speedups obtained from a single team of 10 workers, that is, speedups from and-parallelism only. The second column shows the speedups obtained from a fixed configuration of 10 teams each containing a single processor, that is, speedups from or-parallelism only. The third column shows the speedups from combined and- and or- parallelism. It demonstrates that Andorra-I can obtain speedups which are greater than those obtainable by exploiting a single type of parallelism alone.

4.4 Sequential Performance

Table 2 showed the basic performance of the parallel version of Andorra-I compared with SICStus. However, it is also worth comparing the sequential version of Andorra-I directly with SICStus Prolog and this is done in Table 7. From this we can see roughly what the overhead of supporting the Basic Andorra Model is when separated from the cost of supporting parallelism.

Table 7 shows that the sequential version of Andorra-I is about 1.7 - 4.5 times slower than SICStus. For most of the determinate benchmarks, Andorra-I is about 2.5 times slower. This overhead is mainly due to the goal frame stacking and goal suspension. The exception is *nrv400* where performance is much closer to SICStus. This is because the compiler can avoid any goal frame creation for this program.

For nondeterminate benchmarks, we would like to point out that the current method of switching between determinate and nondeterminate phases is not very efficient. The two phases are written as two different C procedures. Therefore each time we enter a new execution phase we need to reset all registers of the Andorra-I emulator. This inefficiency can easily be removed after we reorganise the code of the emulator. For this reason, the result from *5x4x3_puzzle*, which frequently creates choicepoints, is rather far behind SICStus. Other nondeterminate benchmarks show better results. This

prog name	SICStus 2.1 time in ms	Seq. Andorra-I time in ms	Ratio
nrv400	3459	5940	1.71
bt_cluster	30490	77320	2.53
bt_wms	68090	152020	2.32
road_markings	2600	6840	2.63
chat_80_db5	1500	2670	1.78
5x4x3_puzzle	15300	68280	4.46
warplan	6849	18370	2.68
protein_all	20189	52110	2.58
protein_1st	4250	9330	2.19
fly_pan	543520	10430	0.02
scanner	NA	23600	
cipher	NA	5510	

Table 7: Comparison of SICStus 2.1 and Sequential Andorra-I

is chiefly because Andorra-I's indexing scheme is more efficient than the one used by SICStus. We index on every argument while SICStus only indexes on the first argument.

We believe that after further optimisation of the emulator, a better basic performance can be obtained in the near future.

prog name	Par. Andorra-I (1 worker)		Ratio
(time in ms)	interpreter	compiler	
nrv400	33920	7510	4.52
bt_cluster	364283	99720	3.56
bt_wms	1345340	187060	7.19
road_markings	25530	11420	2.24
chat_80_db5	12130	3360	3.61
5x4x3_puzzle	214290	100390	2.13
protein_all	100390	78840	1.27
protein_1st	18420	13840	1.33
warplan	32710	30370	1.08
fly_pan	63150	13020	4.85
scanner	130460	35070	3.72
cipher	11150	8310	1.34

Table 8: Interpreted versus Compiled Andorra-I

Table 8 compares the interpreted and the compiled versions of the program set. The results show a good improvement for the determinate, and largely determinate, applications. The improvements for the largely non-determinate applications are not as substantial, but should be improved by reorganising the emulator and improving the implementation of some builtins.

4.5 Parallel Overheads

Table 2 showed how the sequential implementation of Andorra-I is faster than the parallel implementation, in other words, how much overhead we have paid to provide support for parallelism. For our benchmark suite the parallel overhead is on average about 40%, which is quite reasonable. Further investigation shows that the parallel overhead is mainly attributable to the cost of the following three aspects:

- binding arrays (or-parallel overhead): approximately 22%;
- stack chunking (and-parallel overhead): approximately 16%;
- variable locking (and-parallel overhead): approximately 4%.

These figures are more or less similar to those of the interpreter-based Andorra-I. This means that the parallel overheads have not increased after implementing the more efficient compiler-based version.

5 Conclusion

We have developed a parallel logic programming system, Andorra-I, which is, as far as we know, the first and only system integrating or-parallelism and dependent and-parallelism. The advantages of the Andorra-I system are as follows.

Firstly, from the language point of view, Andorra-I is a very powerful system. By using its preprocessor we can automatically parallelise Prolog programs. Moreover, Andorra-I can support Flat Parlog, FGHC, Flat Pandora and CHIP style CLP languages[9]. Thus, it provides a language extending Prolog with all the essential capabilities of committed-choice languages as well as some constraint solving primitives.

Secondly, the range of parallelism that we can exploit within Andorra-I is wider than that of other systems which only exploit a single form of parallelism.

Thirdly, Andorra-I shows good performance. Although the system would benefit from further optimisation, the sequential version of Andorra-I is reasonable fast: typically 2.5 times slower than SICStus Prolog (but ranging from 1.7 to 4.5 times slower), and about the same speed as JAM. This indicates that the extra complexity to support the model is manageable. The actual overheads in supporting the parallel implementation are also reasonable, about 40% on average. This means that we can exploit the parallelism contained within a program very efficiently. For the programs which have either and- or or-parallelism alone, we can obtain speedups that are as good as JAM and Aurora. For the programs which contain both and- and or-parallelism, we can obtain speedups which are greater than those obtainable by exploiting one type of parallelism alone.

We believe that Andorra-I already provides a powerful logic programming language, good performance, and a wide range of parallelism. But the system can still be improved. In the near future, we plan to optimise the system by improving memory management, by optimising the implementation of the emulator, by improving further how parallelism is scheduled in Andorra-I, and by adding extra optimisations to the compiler, such as a closer interface between the determinacy code and the individual clause code.

Andorra-I is a stepping stone towards the ultimate goal of obtaining both the best execution strategy, and the maximum parallelism for logic programs. One of the main limitations of Andorra-I is that it only considers determinate goals. Models such as the Extended Andorra Model[22] give a more general framework for the exploitation of parallelism in logic programs. We expect their implementation to benefit from our work in Andorra-I.

Acknowledgements

The development of the Andorra-I system was supported by the European Community ESPRIT Project 2471 (PEPMA). Current work on Andorra-I extensions and applications is supported by British Telecom, and we have benefitted from close collaboration with BT staff. We would like to thank all Andorra-I users, especially Dewi Munaf, for their valuable feedback.

Inês Dutra would like to thank Conselho Nacional de Desenvolvimento Científico e Tecnológico and the Federal University of Rio de Janeiro for supporting her stay at the University of Bristol. Vítor Santos Costa would like to thank the European Community ESPRIT Bursary Programme and the Universidade do Porto for supporting his stay at the University of Bristol.

References

[1] R. Bahgat. Solving resource allocation problems in pandora. Technical report, Imperial College, Department of Computing, 1990.

[2] 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*. Springer Verlag, June 1991.

[3] Anthony Beaumont and David H D Warren. Scheduling Speculative Work in Or-Parallel Prolog Systems. In *Proceedings of the tenth International Conference on Logic Programming*. MIT Press, June 1993.

[4] Barry Crabtree. Personal Communication, British Telecom Research Labs., March 1991.

[5] J. Crammond. *Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors*. PhD thesis, Heriot-Watt University, Edinburgh, May 1988. Research Report PAR 88/4, Dept. of Computing, Imperial College, London.

[6] A.K. Dewdney. *Scientific American*, page 124, September 1990.

[7] Inês Dutra. A Flexible Scheduler for the Andorra-I System. In *Proceedings of ICLP91 Pre-conference Workshop on Parallel Execution of Logic Programs*. Springer-Verlag, July 1991.

[8] Richard P. Gabriel. *Performance and evaluation of Lisp systems*. MIT Press, 1985.

[9] Steve Gregory and Rong Yang. Parallel Constraint Solving in Andorra-I. In *International Conference on Fifth Generation Computer Systems 1992*, pages 843–850. ICOT, Tokyo, Japan, June 1992.

[10] Sverker Janson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the International Logic Programming Symposium*, pages 167–186. MIT Press, October 1991.

[11] Feliks Kluźniak. Developing applications for Aurora. Technical Report TR-90-17, University of Bristol, Computer Science Department, August 1990.

[12] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[13] Dewi Munaf. Personal Communication, British Telecom Research Labs., July 1992.

[14] Dewi Munaf. Workforce management in andorra-i. Technical report, British Telecom Research Labs., March 1993.

[15] C.J. Rawlings, W.R.T. Taylor, J. Nyakairu, J. Fox, and M.J.E. Sternberg. Using Prolog to represent and reason about protein structure. In Ehud Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 536–543. Springer-Verlag, 1986.

[16] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Logic Programming: Proceedings of the 8th International Conference*. MIT Press, 1991.

[17] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting full Prolog on the Basic Andorra model. In *Logic Programming: Proceedings of the 8th International Conference*. MIT Press, 1991.

[18] Lydia Schaaser and Barry Thomas. Finding road lane boundaries for vision guided vehicle navigation. In Ichiro Masaki, editor, *Vision-Based Vehicle Guidance*. Springer-Verlag, 1992.

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

[20] David H. D. Warren. WARPLAN: a System for Generating Plans. DCL Memo 76, University of Edinburgh, June 1974.

[21] 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.

[22] David H. D. Warren. The Extended Andorra Model with Implicit Control. Presented at ICLP'90 Workshop on Parallel Logic Programming, Eilat, Israel, June 1990.

[23] David H. D. Warren and F. C. N. Pereira. An efficient easily adaptable system for interpreting natural language queries. *American Journal of Computational Linguistics*, 8(3-4):110–122, 1982.

[24] Rong Yang. Solving simple substitution ciphers in Andorra-I. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 113–128. MIT Press, June 1989.