&-Prolog and its Performance: Exploiting Independent And-Parallelism

M. V. Hermenegildo
Universidad Politécnica de Madrid (UPM)
Facultad de Informática - Campus de Montegancedo
28660-Boadilla del Monte, Madrid - SPAIN
herme@fi.upm.es *or* herme@cs.utexas.edu

K. J. Greene
MCC Deductive Computing Laboratory
3500 W. Balcones Center Dr.
Austin, Texas 78759 - USA
greene@mcc.com

Abstract

An Independent And-Parallel Prolog model and implementation, &-Prolog, are described. The description includes a summary of the system's architecture, some details of its execution model (based on the RAP-WAM model), and most importantly, its performance on sequential workstations and shared memory multiprocessors as compared with state-of-the-art Prolog systems. Speedup curves are provided for a collection of benchmark programs which demonstrate significant speed advantages over state-of the art sequential systems.

1 Introduction

The goal of &-Prolog is to provide a parallel logic programming environment in which a programmer can freely choose between concentrating solely on the conventional programming task itself or performing, in addition, the task of explicitly annotating parts of the program for parallel execution. In the former case, a parallelizing compiler uncovers the parallelism in the program. In the latter case, the compiler aids the user in the dependency analysis and related tasks. Different choices are allowed for different parts of the program. Prolog programs offer many opportunities for the exploitation of parallelism, the main ones being Or-Parallelism and And-Parallelism [3]. Several models have been proposed to take advantage of such opportunities (see, for example, [4], [15], [1], [7], [11], [20], [5], [19], [16] and their references). The

&-Prolog system as described in this paper exploits the generalized version of Independent And-Parallelism (GIAP) presented in [6] in which goals are deemed to be independent simply when no variable binding conflicts can arise and/or the complexity of the search expected by the programmer can be preserved. This includes both "strict" and "non-strict" IAP, for which theoretical results are also given in [6]. These results show that it is possible to guarantee a "no-speedown" property for programs which exploit this type of parallelism. Our system is orthogonal to, and compatible with Or-Parallel execution models and implementation techniques, as shown in [5]. However, we are concerned herein with pure And-Parallel execution.

The &-Prolog system comprises a parallelizing compiler and an execution model/run-time system. The run-time system is based on the Parallel WAM (PWAM) model, an extension of RAP-WAM [7, 8], itself an extension of the Warren Abstract Machine (WAM) [18]. Specifically, the &-Prolog system attempts to achieve the following goals:

- Develop implementation technology for the parallel execution of Prolog on multiprocessors, while retaining conventional Prolog semantics (including *don't-know non-determinism*).
- Implement and benchmark this technology and compare it to state-of-the-art sequential Prolog systems. The performance of the parallel system should be substantially better while retaining comparable resource efficiency.
- Support user-transparent exploitation of parallelism in Prolog. Though this implies that the compiler should automatically uncover parallelism in user's code, the objective also includes providing facilities permitting the user to explicitly tell the system how parts of the program are to be executed in parallel. In this case the compiler should aid the user in the dependency analysis tasks.

This paper presents the approach currently taken to reach these goals and some of the results achieved to date. Section 2 provides an overview of the &-Prolog system, including some implementation details not previously reported. Section 3 presents current performance results.

2 Overview of the &-Prolog System

Figure 1 shows the conceptual structure of the &-Prolog system. Although the compiler components are depicted in this figure as separate modules, they have been integrated into the Prolog run-time environment in the usual way. It is a complete Prolog implementation, offering full compatibility with the DECsystem-20/Quintus Prolog ("Edinburgh") standard, plus supporting the &-Prolog extensions. The user interface is the familiar one with an on-line interpreter and compiler. At the system prompt, and following

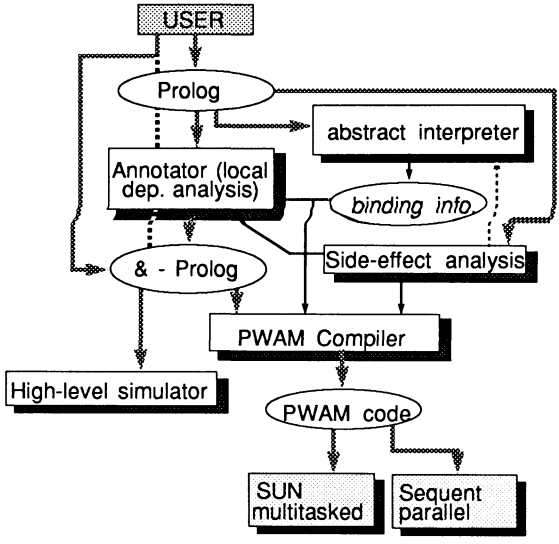

Figure 1: &-Prolog System Architecture and Performance Analysis Tools

our objective of supporting both automatic parallelism and user expressed parallelism, the user can choose to input (consult/compile) "conventional" Prolog code. In this mode users are unaware (except for the increase in performance) that they are using anything but a conventional Prolog system. Compiler switches determine whether or not such code will be parallelized and through which type of analysis. Alternatively the user can provide Prolog code which is annotated with &-Prolog constructs. This can be done for a whole file, a procedure, or a single clause, while the rest of the program can still be parallelized automatically. The compiler still checks the user supplied annotations for correctness, and provides the results of global analysis to aid in the dependency analysis task.

2.1 The &-Prolog Language

We define a new language called &-Prolog as a vehicle for expressing and implementing strict and non-strict IAP. &-Prolog is essentially Prolog, with the addition of the parallel conjunction operator "**&**" (used in place of "," (comma) when goals are to be executed concurrently) and a set of parallelism-related builtins, which includes several types of groundness and independence checks, and synchronization primitives. Combining these primitives with the normal Prolog constructs, such as "->" (if-then-else), users can conditionally trigger parallel execution of goals. &-Prolog is capable of expressing both restricted [4] and unrestricted IAP (through the use of the `wait` primitives [13]). For syntactic convenience, an addi-

tional construct is also provided: the *Conditional Graph Expression (CGE)*. A CGE has the general form (i_cond => $goal_1$ & $goal_2$ & ... & $goal_N$) where the $goal_i$ are either normal Prolog goals or other CGEs and i_cond is a condition which, if satisfied, guarantees the mutual independence of the $goal_i$s. The operational meaning of the CGE is "check i_cond; if it succeeds, execute the $goal_i$ in parallel, otherwise execute them sequentially." &-Prolog if-then-else expressions and CGEs can be nested in order to create richer execution graphs. i_cond can in principle be any &-Prolog goal but is in general either `true` ("unconditional" parallelism) or a conjunction of checks on the groundness or independence of variables appearing in the $goal_i$s. For example, the following Prolog clause

```
p(X,Y) :- q(X,Y), r(X), s(X).
```

could be written for parallel execution in &-Prolog as

```
p(X,Y):- (ground(X) -> q(X,Y) & r(X) & s(X)
                    ;  q(X,Y), (ground(X) => r(X) & s(X))).
```

2.2 &-Prolog Compiler Structure

In the compiler, input code is analyzed by four different modules as follows:

The **Annotator**, or "parallelizer", performs local *dependency* analysis on the input code. In addition, and if the appropriate option is selected, it gets information about the possible run-time substitutions ("variable bindings") at all parts in the program from the global analyzer. It also gets from the **Side-Effect Analyzer** [13] information on whether or not each non-builtin predicate and clause of the given program is *pure*, or contains or calls a *side-effect*. The annotator uses all available information to rewrite the input code for parallel execution. Its output is an annotated &-Prolog program. In addition to parallelizing unannotated Prolog programs, the annotator also checks any user-provided annotations. Some of the techniques and heuristics used in the annotator are described in [14].

The **Global Analyzer** interprets the given program over an abstract domain (specifically designed to precisely highlight dependence information) and infers information about the possible run-time substitutions at all points of the program. In addition, such binding information is also provided to the low-level PWAM compiler. The concepts and algorithms used in the global analyzer are described in [12].

The **Low-Level PWAM Compiler** (an extension of the SICStus0.5 WAM compiler [2]) produces PWAM code from a given &-Prolog program. All parts of the compiler are written in Prolog and available as a whole on-line on the &-Prolog run-time system (described in Section 2.4) making it a standalone unit.

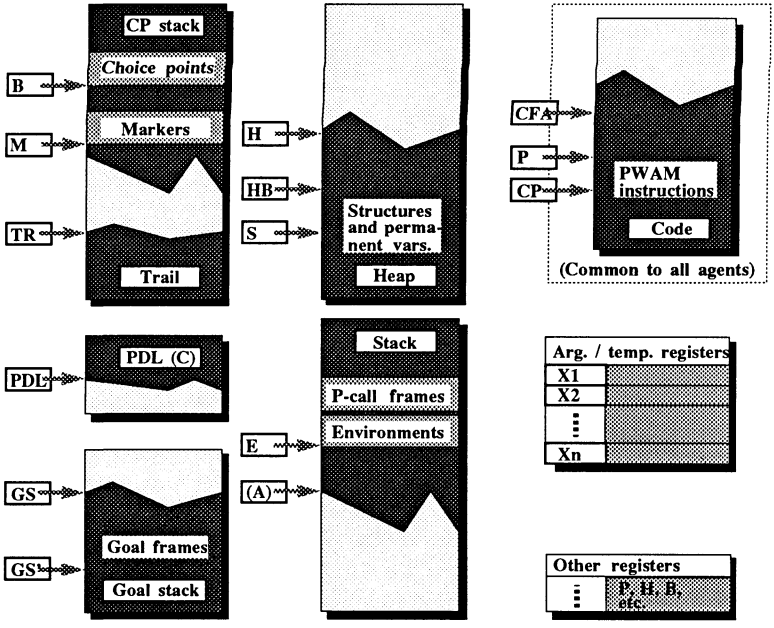

Figure 2: PWAM Storage Model: A Stack Set

2.3 PWAM Architecture

The PWAM (an evolution of the original RAP-WAM [8, 7]) is an extension of the WAM architecture [18] capable of executing logic programs in parallel as determined by &-Prolog's annotations. The &-Prolog run-time system is made up of a number of PWAMs executing concurrently (see Section 2.4). Described below are the distinguishing features of a PWAM.

As mentioned before, a fundamental design objective of the PWAM is fast sequential execution for cases where there is no available (And) parallelism. To this end, the &-Prolog semantics has been integrated naturally into the WAM storage model in the form of specialized stack frames and storage areas which are used during parallel execution. Thus the default (sequential) model is that of a standard WAM exhibiting the same high sequential performance. Special emphasis has also been given to efficiency in the management of parallelism so that most of the WAM performance and storage optimizations are still supported during parallel execution. Figure 2 shows the storage layout of a single PWAM. Each PWAM is similar to a standard WAM (with a complete set of registers and data areas, called a *Stack Set*), with the addition of a *Goal Stack* and two new types of stack frames: *P_Call Frames* and *Markers*. Goals which are ready to be executed in parallel are pushed on to the Goal Stack by the PWAM executing a P_Call

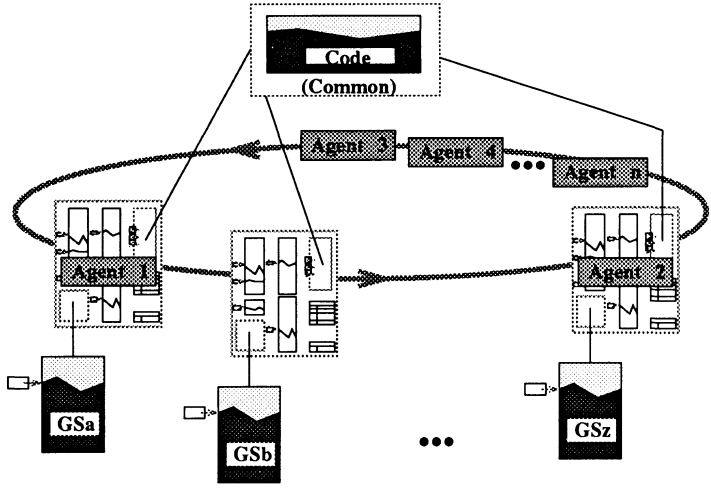

Figure 3: &-Prolog Run-time System Architecture

instruction. P_Call Frames (which are part of the environment) are used for coordinating and synchronizing the parallel execution of the goals inside a parallel call, both during forward execution and during backtracking. Markers are used to delimit *Stack Sections* (horizontal cuts through the Stack Set of a given abstract machine, corresponding to the execution of different *parallel* goals) and they implement the storage recovery mechanisms during backtracking in a similar manner to choice-points in the WAM [9]. In practice, the stack is divided into separate *Control* (Choice Point and Markers) and *Local* stacks (Environments) for reasons of locality and locking.

The instruction set of the PWAM architecture includes all WAM instructions (some of which have been modified for the PWAM) and several additional instructions related to parallel execution. The two most important are *P_Call* and *Pop_Wait*—their behavior is outlined in Section 2.4. The reader is referred to Hermenegildo [7] or Tick [17] for a more complete description of the PWAM instruction set and storage model.

2.4 &-Prolog Run-Time System

The run-time system architecture, pictured in Figure 3, is comprised of a ring of PWAM stack sets, a collection of agents, and a shared code area. The agents (processes) run programs (from the code area) on the PWAMs (PWAM stack sets), i.e. an agent reads and executes the instructions pointed to by the P pointer in a given PWAM and in doing so modifies the PWAM's state. Agents are not tied to particular PWAMs and the number of agents need not be equal to the number of PWAMs. We will use the phrase "running

PWAM" to refer to an (agent,PWAM) pair—i.e. a PWAM which is being used by an agent.

As an example of the code that a running PWAM executes, the results of compiling:

```
..., (q(X,Y) & r(X)), s(X), ...
```

is the PWAM code shown below:

```
    ... <code for head and literals which precede q(X,Y)>
    P_Call(2,L1,L2)
    Pop_Wait
    Put_Value(Y8,X0)
    Call(s/1,8)
    ... <code for literals which follow s(X)>
L1: Put_Value(Y8,X0)
    Put_Value(Y9,X1)
    Execute(q/2)
L2: Put_Value(Y8,X0)
    Execute(r/1)
```

A running PWAM executing a P_Call instruction pushes the instruction's arguments (locations in code space where the compiled literals are) onto the PWAM's Goal Stack and creates a P_Call Frame on the PWAM's Stack. The goals are then available to be executed on any PWAM (including the PWAM which pushed them). The P_Call Frame is where information regarding the status of these goals is kept. From this information it can be determined how many goals have been taken and of those taken how many have been completed. The Pop_Wait instruction is a conditional. Upon execution of this instruction, if not all of the goals (specified by the previous P_Call) have been "popped" from the goal stack, then one is popped, a Marker is written, and the goal is executed, setting the continuation to be the very same Pop_Wait instruction. If all goals have been taken and moreover they have all been completed, then execution simply continues with the next instruction (following the Pop_Wait). However, if all of the goals have been taken but not all have finished, this part of the computation must wait.

There are several scheduling strategies under study in the &-Prolog system. The strategy used in generating the performance results presented in this paper is is as follows (see, for example, [9] for alternatives). Agents which are not running PWAM code, are looking for work. They look for work by first searching the ring of PWAMs for an "idle" PWAM. A PWAM may be in one of three states: "running", "idle", or "blocked". An idle PWAM is one which is either empty or has completed the execution of a goal taken from a goal stack. If the agent finds an idle PWAM on a single traversal of the ring, he attaches to it (by marking it as "running"). If no idle PWAM is found and if there is enough memory, the agent creates a new PWAM, marks it as running, and links it into the PWAM ring. The agent

then returns to the PWAM ring to look for a goal (in the PWAMs' goal stacks) to run. The agent continues this search until a goal is found. When a goal is found, the agent removes it from the goal stack and starts to run it on the PWAM (which the agent found previously). Before starting to run the goal, the agent writes a Marker on the stack.

When an agent completes a goal, it reports success by writing into the "parent's" P_Call frame. The "parent" is the PWAM which spawned the goal (by pushing it onto its goal stack). If the goal is not the last one in the P_Call frame to complete, then the agent stays attached to its PWAM and looks for another goal to run. If, however, it is the last goal to be completed, the agent detaches from the PWAM on which the goal was run (by marking it "idle") and resumes the parent PWAM (which was blocked) and executes the continuation (the code following the Pop_Wait). Space does not permit us to discuss the memory management issues of the run-time system here, please see [9].

2.5 Shared Memory Multiprocessor Implementation

The &-Prolog system has been realized in C on both Sun and Sequent platforms. It is an extension of the SICStus-Prolog V0.5 [2] implementation. The same code runs on a Sun-3 and the Sequent Balance and Symmetry machines. Agents are UNIX processes. On startup the number of agents equals the number of processors. There is no master agent, they all run the same code. A single PWAM is created at the start, the others are created as the agents begin to look for work. System resources (CPUs) are not used when the system is idle (e.g. waiting for user input at the prompt). The interface is the standard Prolog one plus several new primitives with which the user can trace/control the execution of queries.

3 Performance Results

In this section we discuss the performance of the &-Prolog system as it is currently implemented on UNIX workstations and shared-memory multiprocessors. Our objective here is to evaluate how close &-Prolog's performance comes to our original goal of attaining speed beyond that of state-of-the-art sequential implementations, in particular, that of commercial implementations of Prolog. Despite the fact that several significant optimizations are still to be implemented, the results, as we hope to show, are quite encouraging.

3.1 Discussion of Benchmarks

The group of benchmarks used so far in the performance evaluation can be divided into two sets: the first set ("matrix-int", "matrix-float", "lmatrix-int", "qs-append", "qs-dl", "occur-no/idxng", "occur-w/idxng") are relatively

Bench	Sun3/60				Sequent Symmetry		
	Q2.2	*S0.5*	*&Pseq*	*&Ppar*	*S0.5*	*&Pseq*	*&Ppar*
matrix(50)-int	6.25	23.2	23.9	24.3	23.4	23.42	23.8
matrix(50)-float	16.1	47.4	47.7	47.87	38.7	38.8	38.8
qs(1000)-append	1.32	2.54	2.54	2.60	2.92	2.97	3.05
qs(1000)-dl-si	1.30	2.41	2.42	2.42	2.75	2.79	2.79
qs(1000)-dl-nsi	1.30	2.41	2.42	2.49	2.75	2.79	2.88
occur(50)-no/idxng	25.6	31.2	31.1	31.1	31.8	31.9	32.0
occur(50)-w/idxng	13.62	26.52	26.7	26.74	27.1	27.2	27.39
boyer(3)-si	18.7	51.64	51.64	51.64	44.1	45.5	45.5
boyer(3)-nsi	18.7	51.64	51.64	56.9	44.1	45.5	56.0
annotator(100)	0.65	0.83	0.82	0.82	0.90	0.91	0.91

Table 1: Execution Time (S): SUN3/60 vs. Sequent Symmetry, Quintus 2.2 vs. Sictus 0.5 vs. &-Prolog

small programs but with well understood granularity and ideal speedup characteristics and which have been used by us and other researchers in previous studies [17, 10, 11, 15]. Thus, they allow measurement of basic overheads and comparison with previous results. "matrix-int" is the familiar recursive matrix multiplication program. The experiments run represent the multiplication of two 50x50 matrices. "lmatrix-" is the same program but including the creation of the matrices in the timings (as used in [11]). "matrix-float" is again the same program but using floating point numbers. "qs-append" is the familiar quick-sort algorithm, sorting a 1000 element list. "qs-dl" is the difference list version. "occur-no/idxng" searches for occurrences of characters in a given list and is identical to the version used in [15]. "occur-w/idxng" is the same program, but with some argument positions permuted to improve indexing behaviour.

The second set of benchmarks ("boyer", "annotator") contains larger programs which represent more realistic benchmarks. "boyer" is the Prolog version of the boyer-moore theorem prover kernel, written by Evan Tick, proving the standard theorem used in the original LISP code. "annotate" is the annotation pass of the &-Prolog compiler, as described in section 2.2. The latter is a program with over 1800 lines and represents an interesting case of bootstrapping in the Prolog compiler tradition: the annotator annotating itself.

Independently of the size of the programs themselves, the size of data has been chosen large enough to meet several objectives: first, to produce running times large enough to be measured accurately: in the order of seconds even when running on 10 processors. This makes effects such as "warming" of the caches both in the case of the Sun3/60 and the Sequent machines secondary. Second, the data is large enough to stress the scheduling and memory management policies in the implementations, by producing a large number of processes and significant memory consumption, respectively. The sizes used strain the capacity of Quintus Prolog on the machines tested.

Bench	Number of Agents (Processors)					
	1	*2*	*4*	*6*	*8*	*10*
matrix(50)-int	23.8	11.93	5.98	3.99	3.00	2.40
matrix(50)-float	38.8	19.43	9.74	6.50	4.88	3.92
qs(1000)-append	3.05	1.59	1.2	0.81	0.66	0.64
qs(1000)-dl-si	2.79	2.79	2.79	2.79	2.79	2.79
qs(1000)-dl-nsi	2.88	1.5	0.95	0.74	0.62	0.61
occur(50)-no/idxng	32.0	16.4	8.2	5.5	4.15	3.3
occur(50)-w/idxng	27.39	13.7	7.01	4.68	3.58	2.809
boyer(3)-si	45.5	45.5	45.5	45.5	45.5	45.5
boyer(3)-nsi	56.0	29.1	15.3	10.9	9.6	8.0
annotator(100)	0.91	0.455	0.28	0.21	0.17	0.13

Table 2: Execution Time (S): &-Prolog on Sequent Symmetry, 1-10 Agents (Processors)

Bench	Q2.2	S0.5	&Pseq	&P1	&P2	&P4	&P6	&P8	&P10
matrix(50)-int	7.98	99.8	101.6	103	51.7	25.9	17.3	13.0	10.45
lmatrix(50)-int	7.99	99.9	101.6	103.2	52.13	26.62	18.1	13.84	11.28
qs(1000)-app	1.7	13.6	12.66	13.23	7.2	4.43	3.79	3.08	3.03
qs(1000)-dl-si	1.61	11.13	11.9	11.9	12.0	11.9	11.9	11.9	12.0
qs(1000)-dl-nsi	1.61	11.13	11.9	12.51	6.75	4.06	3.3	2.91	2.78
boyer(2)-si	0.70	6.15	5.83	5.83	5.83	5.84	5.83	5.85	5.86
boyer(2)-nsi	0.70	6.15	5.83	7.57	3.95	2.15	1.64	1.40	1.35
annotator(100)	0.62	3.87	4.08	4.09	2.11	1.27	0.959	0.75	0.64

Table 3: Exec. time(S): Quintus/SUN3-110 vs. Sicstus,&-Prolog/Balance

3.2 Sequential Performance: Overhead Comparison

As mentioned before, the &-Prolog run-time system, an evolution of SICStus0.5 [2], makes use of the PWAM architecture and implements it in the form of a bytecode interpreter written in the C language, augmented with macros and functions for accessing shared memory and performing locking operations. Writing the bytecode interpreter in C offers portability at a cost in uniprocessor performance with respect to systems written in assembler, such as Quintus Prolog, but it was deemed more appropriate for an experimental system. Table 1 illustrates this point. This table represents the overhead paid by remaining at the C level. In addition it also illustrates the overhead in the basic machinery of the PWAM with respect to an optimized WAM such as that underlying SICStus0.5. Comparison is made among the different systems both on a Sun3/60 and on one Sequent Symmetry processor. The Sun3/60 is chosen because it is a relatively fast CISC-based workstation with plenty of memory and cache, thus representing the same level of technology as a single Symmetry processor. They represent essentially the same sequential speed. Timing data are wallclock times in seconds on unloaded machines.

The difference in speed between Quintus2.2 and SICStus0.5 averages to around 2.4. This result is somewhat skewed by the results of integer matrix

Bench	Number of Agents (Processors)					
	1	2	4	6	8	10
matrix(50)-int	1.0	1.99	3.98	5.96	7.93	9.92
matrix(50)-float	1.0	1.99	3.98	5.97	7.95	9.90
qs(1000)-append	1.0	1.92	2.54	3.77	4.62	4.76
qs(1000)-dl-si	1.0	1.0	1.0	1.0	1.0	1.0
qs(1000)-dl-nsi	1.0	1.92	3.03	3.89	4.65	4.72
occur(50)-no/idxng	1.0	1.95	3.90	5.82	7.71	9.7
occur(50)-w/idxng	1.0	1.99	3.91	5.85	7.65	9.75
boyer(3)-si	1.0	1.0	1.0	1.0	1.0	1.0
boyer(3)-nsi	1.0	1.93	3.66	5.14	5.84	7.0
annotator(100)	1.0	1.99	3.25	4.34	5.35	7.0

Table 4: Relative Speedup: &-Prolog on Sequent Symmetry, 1-10 Agents (Processors)

multiplication, since integer operations are not as optimized in SICStus as in Quintus. If "matrix-int" is ignored the difference drops down to the expected factor of 2. SICStus0.5 and &-Prolog run the sequential versions of the programs at essentially the same speed. &-Prolog runs the parallel version of the program on 1 processor also at essentially the same speed as SICStus runs the sequential version (within 1%), despite the fact that the overheads associated with pushing all the parallel goals on to the goal stack and picking them from there are present. The largest overhead is found in "boyer" due to the small granularity and very large number of parallel goals generated.

3.3 Parallel Performance: Timings and Speedups

Table 2 presents the raw timings in seconds from the execution of the different benchmarks under &-Prolog on a 12 processor Sequent Symmetry for varying numbers of active agents. Benchmarks have been parallelized so that only unconditional parallelism (both strict and non-strict – "si"/"nsi") is exploited. Table 3 provides similar results for the Sequent Balance. Timings from the execution of SICStus0.5 on this machine and of Quintus2.2 on a slower Sun have also been included to complement the data in table 1. Memory limitations on the small Balance machine made it impossible to run the large "boyer" benchmark on any system, so a simpler query was used. The Balance is a relatively slow machine and the results are provided only for comparison with previous results on this machine (e.g. [11]). Table 4 shows the actual speedups obtained on the Sequent Symmetry (speedups on the balance are quite similar). All versions of "matrix" and "occur" show excellent speedups (raw timing figures and speedups of "occur" allow comparison with Kale's system [15]). The "annotator" and "boyer-nsi" (i.e. parallelized using "non-strict" independence) also show good speedups. This is especially encouraging since they are relatively large programs. The versions of

"qs" using difference lists and "boyer" parallelized using strict independence show no speedups, accentuating the importance of relaxing the traditional concept of independence. Speedup on "qs" is the lowest. Although it can be improved arbitrarily by increasing the size of the data, it shows the limited nature of the parallelism in the benchmark.

These speedups are encouraging and confirm previous simulation results as described in [10, 17]. However, since &-Prolog's objective is to provide performance beyond that offered by current sequential systems the data from table 4 should be correlated with that of table 1. Figures 4 through 7 represent this correlation in a graphical format. In each graph, the two horizontal lines represent the speeds of Quintus2.2 and SICStus0.5 on a Sun3/60 while the curve represents the speed of &-Prolog. All numbers are normalized to the speed of &-Prolog on a single processor Symmetry. Performance is significantly higher than that of SICStus, even if running on only two processors. Despite the sequential speed handicap of &-Prolog w.r.t. Quintus (illustrated in figure 1) substantial speedups are still obtained. Higher performance than Quintus is obtained with 2-4 processors.

4 Conclusions

We have presented an overview of &-Prolog, its implementation, and its performance on shared-memory multiprocessors and sequential workstations. The resulting system offers the familiar Prolog interface while allowing either user-transparent or user-guided parallelism. The performance results obtained so far appear very encouraging. We believe that, with similar implementation technology, &-Prolog can be an order of magnitude faster than state-of-the-art sequential systems on small parallel machines (10 processors) for programs that exhibit enough parallelism, and can guaranteed to be no slower for all programs. This is, in our opinion, of special interest in the light of the expected new generation of "desktop multiprocessors."

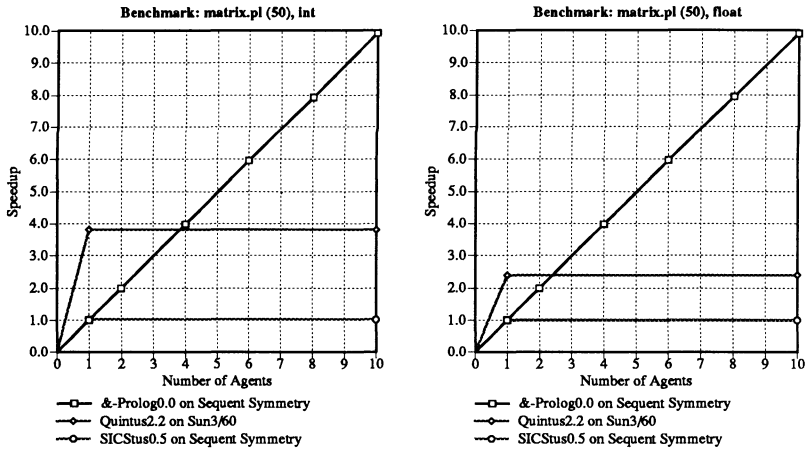

Figure 4: Speedup for matrix(50) w.r.t. Quintus on Sun3/60 and SICStus on 1 Symmetry proc

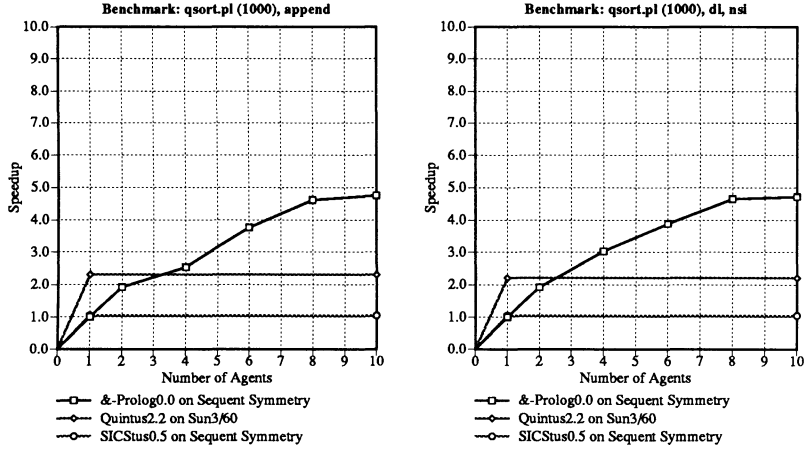

Figure 5: Speedup for qs(1000) w.r.t. Quintus on Sun3/60 and SICStus on 1 Symmetry proc

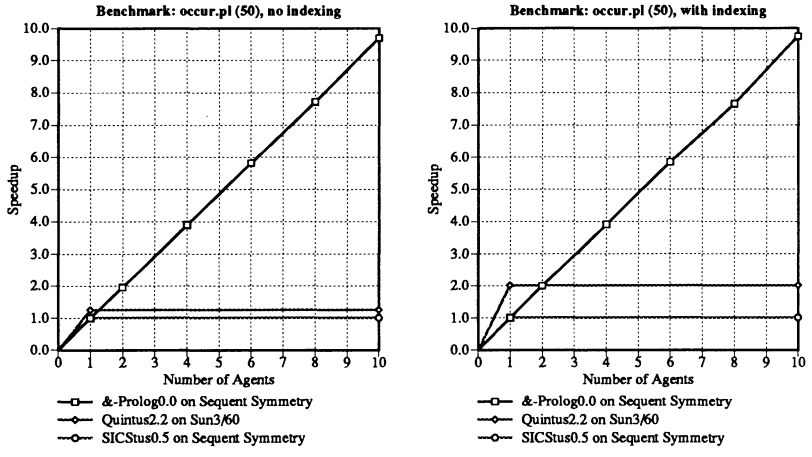

Figure 6: Speedup for occur(50) w.r.t. Quintus on Sun3/60 and SICStus on 1 Symmetry proc

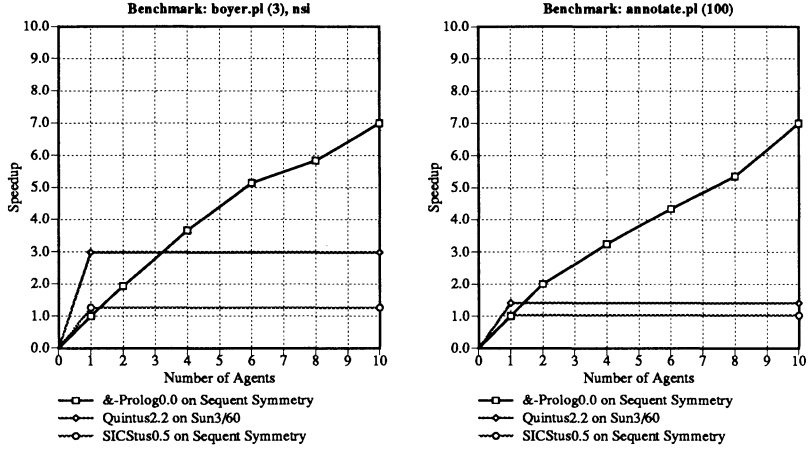

Figure 7: boyer(3) and annotator(100), w.r.t. Quintus on Sun3/60 and SICStus on 1 Symmetry proc

References

[1] P. Biswas, S. Su, and D. Yun. A scalable abstract machine model to support limited-or restricted and parallelism in logic programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 1160–1179, 1988.

[2] M. Carlsson. *Sicstus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[3] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[4] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478, Tokyo, November 1984.

[5] G. Gupta and B. Jayaraman. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *1989 North American Conference on Logic Programming*, MIT Press, October 1989.

[6] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, MIT Press, October 1989.

[7] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, U. of Texas at Austin, August 1986.

[8] M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, pages 25–40, Imperial College, Springer-Verlag, July 1986.

[9] M. V. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In *Fourth International Conference on Logic Programming*, pages 556–575, University of Melbourne, MIT Press, May 1987.

[10] M. V. Hermenegildo and E. Tick. Memory Performance of AND-Parallel Prolog on Shared-Memory Architectures. In *Proceedings of the 17th International Conference on Parallel Processing*, August 1988.

[11] Y.-J. Lin. *A Parallel Implementation of Logic Programs.* PhD thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, August 1988.

[12] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, MIT Press, October 1989.

[13] K. Muthukumar and M. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In *1989 International Conference on Logic Programming*, MIT Press, June 1989.

[14] K. Muthukumar and M. Hermenegildo. *Methods for Automatic Compile-time Parallelization of Logic Programs using Independent/Restricted And-parallelism.* Technical Report ACA-ST-233-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, March 1989.

[15] B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *1989 North American Conference on Logic Programming*, MIT Press, October 1989.

[16] P. Szeredi. Performance Analysis of the Aurora Or-Parallel Prolog System. In *1989 North American Conference on Logic Programming*, MIT Press, October 1989.

[17] E. Tick. *Memory Performance of Prolog Architectures.* Kluwer Academic Publishers, Norwell, MA 02061, 1987.

[18] D. H. D. Warren. *An Abstract Prolog Instruction Set.* Technical Report 309, SRI International, 1983.

[19] D. H. D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *International Symposium on Logic Programming*, pages 92–102, San Francisco, IEEE Computer Society, August 1987.

[20] H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *Symp. of Logic Prog.*, pages 436–448, August 1987.