New Generation Computing, Vol.9, Num. 3-4, pages 233–257, Springer-Verlag, 1991.

The &-Prolog system: Exploiting Independent And-Parallelism[1]

M. V. Hermenegildo[2]
Universidad Politécnica de Madrid (UPM)
Facultad de Informática
28660-Boadilla del Monte, Madrid - SPAIN
herme@fi.upm.es *or* herme@cs.utexas.edu

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

Abstract

The &-Prolog system, a practical implementation of a parallel execution model for Prolog exploiting strict and non-strict independent and-parallelism, is described. Both automatic and manual parallelization of programs is supported. This description includes a summary of the system's language and architecture, some details of its execution model (based on the RAP-WAM model), and data on its performance on sequential workstations and shared memory multiprocessors, which is compared to that of current Prolog systems. The results to date show significant speed advantages over state-of-the-art sequential systems.

Keywords: Parallelism, Logic Programming, Prolog, Automatic Parallelization, Performance Analysis

1 Introduction

There are many types of practical parallel processors currently available in the marketplace, and multiprocessor systems are expected to be the norm in the very near future. However, the amount of software that can exploit the performance potential of these machines is still very small. This is largely due to the difficulty in mapping the inherent parallelism in problems on to different multiprocessor organizations. There are in general at least two ways in which such a mapping can be performed: it can be done explicitly in the program by the user if the language includes parallel constructs, or it can be automatically uncovered by a "parallelizing" compiler from a program that has no explicit parallelism in it. Both approaches have their merits and drawbacks. A parallelizing compiler makes it possible to tap the performance potential of a multiprocessor without burdening the programmer. However, the capabilities of current parallelizing compilers are relatively limited, especially in the context of conventional programming languages. Parallelism explicitly expressed

[1] A preliminary version of this paper appeared in the Proceedings of the 1990 International Conference on Logic Programming, MIT Press.

[2] The work reported in this paper was performed in part at MCC's DC Lab. This author was supported in part by ESPRIT project 2471 "PEPMA" and CICYT project 361.208.

by the programmer using specialized constructs can be used when the programmer has a clear understanding of how the parallelism in the problem can be exploited. However, this adds in general an additional dimension of complication to the already complicated and bug-prone task of programming.[3] In reality, although experienced users may often have a correct intuition on which of the parts of a problem (and which of the parts of the associated program) can be solved in parallel, the task of correctly determining the *dependencies* among those parts and the sequencing and synchronization needed to correctly reflect such dependencies is proving to be very difficult and error-prone. This was recently also pointed out by Karp [18] who states that "the problem with manual parallelization is that much of the work needed is too hard for people to do. For instance, only compilers can be trusted to do the dependency analysis needed to parallelize programs on shared-memory systems."

Therefore, the best programming environment would appear to be one in which the programmer can freely choose between only concentrating on the conventional programming task itself (letting a parallelizing compiler uncover the parallelism in the resulting program) or, alternatively, performing also the task of explicitly annotating parts of the program for parallel execution. In the latter case, the compiler should be able to aid the user in the dependency analysis and related tasks. Ideally, different choices should be allowed for different parts of the program.

Karp also points out the lack of good parallelizing compilers, and predicts that the technology is still several years away. One reason for this is that the programming languages that are conventionally parallelized have a complex imperative semantics which makes compiler analysis difficult and forces users to employ control mechanisms that hide the parallelism in the problem. It is very difficult to develop parallelization algorithms for such languages that are both effective and amenable to proof. On the other hand, declarative languages and, in particular, logic programming languages, require far less explication of control (thus preserving much more of the parallelism in the problem). In addition, their semantics makes them comparatively more amenable to compile-time analysis and program parallelization. In other words, such programs preserve more the intrinsic parallelism in the problem, make it easier to extract in an automatic fashion, and allow the techniques being used to be proved correct. It is our thesis that through advanced compiler techniques, such as abstract interpretation [6], automation of parallelization is indeed feasible for languages that have a declarative foundation. Furthermore, we believe that the development of these techniques will in addition further our understanding of how to parallelize other programming paradigms.

In view of the above, the objective in the development of &-Prolog is to gain insight into new concepts and techniques of general applicability in program analysis and performance improvement of declarative, knowledge-based systems such as those under the general umbrella of computational logic. The main path being explored towards this objective is the combined use of automatic parallelism, user-specified parallelism, and advanced compilation techniques such as abstract interpretation. More concretely, 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, with different choices allowed for different parts of the program. 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. It should be noted

[3] In fact, the progress from systems which require from the programmer explicit creation and mapping of processes to a particular processor interconnection topology and extensive granularity control, to systems which don't require at least some of these tasks appears to be a leap forward comparable to the appearance of the concept of virtual memory (compared to overlays) or even high-level languages (compared to programming in machine code). Of course, in the same way as there is sometimes a case for assembler programming in parts of a program which are particularly performance sensitive, there will be some cases in which complete explicit control of parallelism is indicated.

that the motivation is not based on a "dusty-deck" approach to Prolog: it is not so much a question of parallelizing existing programs as of relieving the programmer of a burden while writing new ones. In summary, our goal is to:

- develop compile-time and run-time implementation technology for the parallel execution of Prolog on multiprocessors, while retaining conventional Prolog semantics (including *don't-know non-determinism*);
- formally prove such technology correct;
- 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.

This paper presents the approach currently taken to reach these goals and some of the results achieved to date. Because of length limitations sometimes only a brief explanation of some of the issues involved will be given, but in those cases references will be provided to the relevant literature. For the same reasons a certain familiarity with Prolog, Independent And-Parallelism, and the related implementation techniques is necessarily assumed. Section 2 introduces Independent And-Parallelism. Section 3 provides an overview of the &-Prolog system, including some implementation details not previously reported. Section 4 presents current performance results.

2 Independent And-Parallelism

Prolog programs offer many opportunities for the exploitation of parallelism, the main ones being Or-Parallelism and And-Parallelism [5]. Several models have been proposed to take advantage of such opportunities (see, for example, [8], [24], [2], [12], [19], [32], [10], [30], [27] and their references). The &-Prolog system as described in this paper exploits the *generalized* version of Independent And-Parallelism (IAP) presented in [16] 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. Two goals are strictly independent if they do not share any variables. This "run-time" condition can in general be encoded at compile-time as a conjunction of checks on the groundness and independence of program variables. For example, the goals corresponding to the literals "`p(X,Y), q(Y,Z)`" can be guaranteed to have no variables in common (and therefore be run in parallel) if in their run-time instantiations `Y` is bound to a ground term and the terms to which `X` and `Z` are bound don't share any variables.

The concept of non-strict independence extends the number of goals which can be executed in parallel in a given program by allowing the parallel execution of a restricted class of goals which share variables: those in which, despite this sharing, no "left-to-right" search-space restriction is performed. For example, the goals corresponding to the program literals "`p(X), q(X)`" can safely be run in parallel provided `p/1` leaves `X` free after it executes.

Theoretical results, as well as conditions for parallelization, for both both "strict" and "non-strict" IAP are given in [16]. In that paper, in addition to formal definitions of goal independence, a parallel resolution procedure for independent goals is presented. Also, results on the correctness and efficiency of IAP execution are derived. These results can be summarized as follows:

- (correctness and completeness) if only independent goals are executed in parallel the solutions obtained are the same as those produced by standard sequential execution.[4]

[4]The finite failure set can be larger, this being understood as a desirable characteristic.

- in the absence of failure, parallel execution doesn't generate additional work (with respect to sequential execution) while actual execution time is reduced.
- in the case of failure the parallel execution is guaranteed to be no slower than the sequential execution.

The results hold for strictly independent goals, and for non-strictly independent goals as well, provided a trivial rewriting of such goals is done [16].

It should be noted that an important objective in the &-Prolog system is to make it possible to also support or-parallelism. This can be done as shown in [10, 9, 11]. The issue of combining and- and or-parallelism in an implementation is the subject of much current research [24, 3, 25]. This paper, however, concentrates on the description of the pure IAP system.

3 Overview of the &-Prolog System

The &-Prolog system comprises a parallelizing compiler aimed at uncovering IAP and an execution model/run-time system aimed at exploiting such parallelism. The run-time system is based on the Parallel WAM (PWAM) model, an extension of RAP-WAM [12, 13], itself an extension of the Warren Abstract Machine (WAM) [29]. 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 language extensions, which will be described in section 3.1. The user interface is the familiar one with an on-line interpreter and compiler. At the system prompt, and following 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 (implemented as "prolog flags") 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. There is also an on-line visualization system (based on the X-windows standard) which provides a graphical representation of the parallel execution and has proven itself quite useful in debugging and performance tuning.

3.1 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)[5] and a set of parallelism-related builtins, which includes several types of groundness and independence checks, and synchronization primitives.[6] Combining these primitives with the normal Prolog constructs, such as "`->`" (if-then-else), users can conditionally trigger parallel execution of goals. &-Prolog

[5]The backward operational semantics (backtracking) of the "&" construct is conceptually equivalent to standard backtracking except that dependency information is used to economically perform a limited form of intelligent backtracking. See [15, 12] for details.

[6]There is also a set of builtins for controlling the use of resources (number of processors, memory, etc.) which automatically take appropriate values and that the user need not be concerned with.

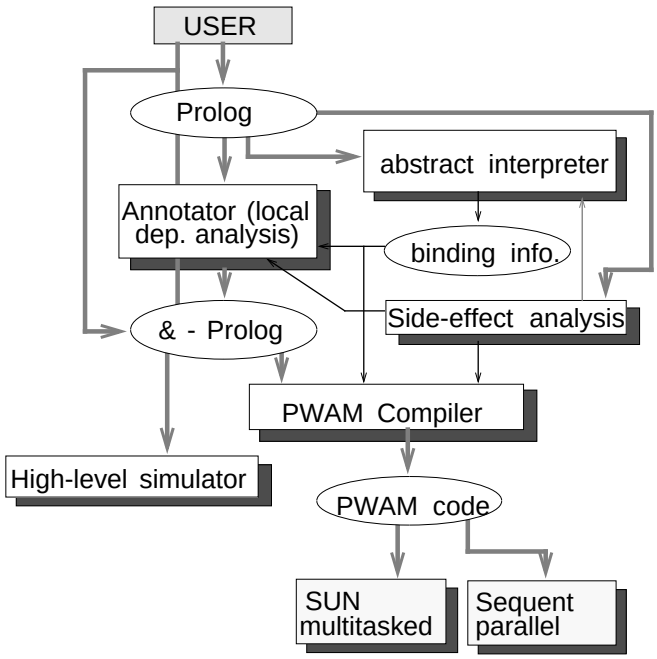

Figure 1: &-Prolog System Architecture and Performance Analysis Tools

is capable of expressing both restricted [8] and unrestricted IAP (through the use of the `wait` primitives [20]). For syntactic convenience, an additional 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 CGE can be viewed simply as syntactic sugar for the Prolog conditional construct:

(*i_cond* `->` $goal_1$ `&` $goal_2$ `&` ... `&` $goal_N$
`;` $goal_1$ `,` $goal_2$ `,` ... `,` $goal_N$)

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) ).
```

or, with the same meaning as the CGE above, as

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

or, and still within &-Prolog, as

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

It is important to note that the annotated code (and also the compiled bytecode , if needed) can be saved into a file, in addition or as an alternative to being executed in parallel on the host multiprocessor. This serves several purposes: the parallelized code can be used as the input for a series of high- and low-level simulators (AND/OR-SIM [26] and PLOPS/CACHE [12, 28]). Also, the user can thus see the output of the parallelization stages and monitor the compiler if desired.

3.2 &-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** (described below). It also gets from the **Side-Effect Analyzer** (also described below) 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. The **Low-Level PWAM Compiler** then translates this code to PWAM instructions for execution on the run-time system. As an example, consider the following clause, part of a matrix multiplication relation:

```
matvecmul([Vect1|TVect],Vect2,[Res|TRes]):-
        vecmul(Vect1,Vect2,Res),
        matvecmul(TVect,Vect2,TRes).
```

The output of the annotator may be, for example, as follows (this annotation assumes no knowledge being received from the global analyzer):

```
matvecmul([Vect1|TVect],Vect2,[Res|TRes]):-
        ( ground([Vect2,Vect1,TVect]),indep(Res,TRes) =>
                vecmul(Vect1,Vect2,Res) &
                matvecmul(TVect,Vect2,TRes) ).
```

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 [21].

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. For example, given the same clause as in the example above,

```
matvecmul([Vect1|TVect],Vect2,[Res|TRes]):-
        vecmul(Vect1,Vect2,Res),
        matvecmul(TVect,Vect2,TRes).
```

the analyzer might infer, in view of the rest of the program, that the first two arguments are always ground when the clause is called, and that the third argument is a free variable. This information, which is obviously not obtainable from local clause-at-a-time analysis alone, is sent to the annotator, as mentioned before, which would then generate the much more efficient annotation below:

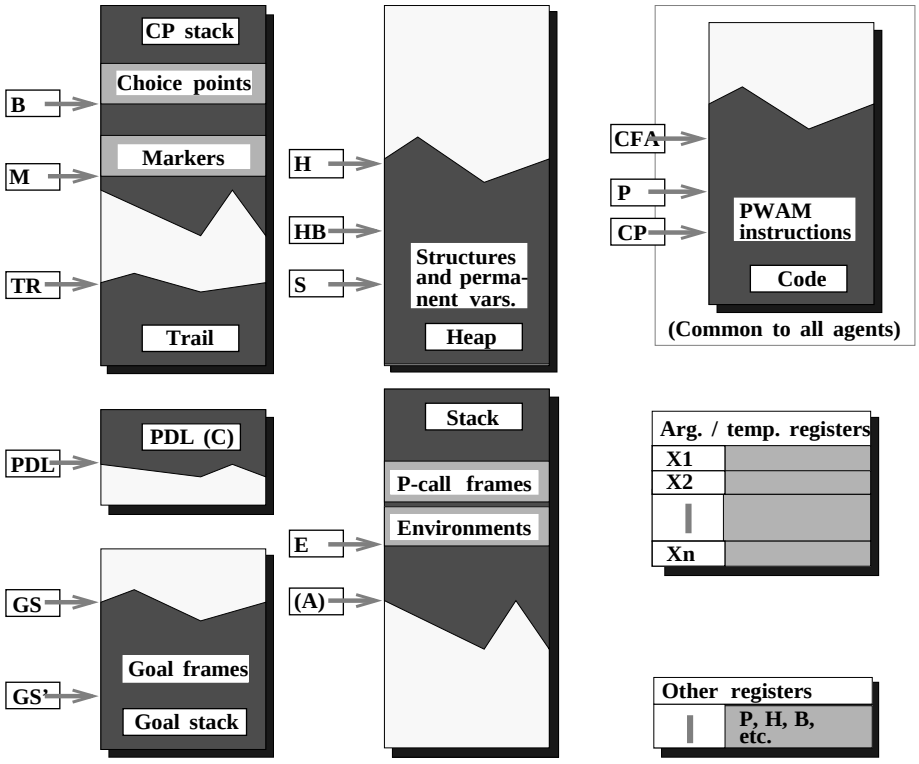

Figure 2: PWAM Storage Model: A Stack Set

```
matvecmul([Vect1|TVect],Vect2,[Res|TRes]):-
        vecmul(Vect1,Vect2,Res) &
        matvecmul(TVect,Vect2,TRes).
```

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 [23, 22]. Some of the results of our experiments in abstract interpretation are reported in [31] and in [23].

The side-effect analyzer annotates each non-builtin predicate and clause of the given program as *pure*, or as containing or calling a *side-effect*. This information is used by the annotator to introduce semaphores into the clauses, if necessary, in order to correctly sequence such side-effects. The techniques used for sequencing side-effects at the &-Prolog level and at the abstract machine level are presented in [20].

The low-level PWAM Compiler (an extension of the SICStus0.5 WAM compiler [4]) 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 3.4) making it a standalone unit.

3.3 PWAM Architecture

As stated above, the &-Prolog code is translated into PWAM (Parallel Warren Abstract Machine) instructions by the low-level compiler for actual execution on the PWAM abstract machine. The PWAM (an evolution of the original RAP-WAM [13, 12]) is an extension of the WAM architecture [29] 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

Frame Type	Location	In WAM?	Need lock?	Locality
Envts./Control	Stack	Yes	No	Local
Envts./Perm. Vars.	Stack	Yes	No	Global
Choice Points	Stack	Yes	No	Local
Heap	Heap	Yes	No	Global
Trail entries	Trail	Yes	No	Local
PDL entries	PDL	Yes	No	Local
P_Call F./Local	Stack	No	No	Local
P_Call F./Global	Stack	No	No	Global
P_Call F./Wait, Sched	Stack	No	Yes	Global
Markers	Stack	No	No	Local
Goal Frames	Goal Stack	No	Yes	Global

Table 1: Characteristics of PWAM Storage Objects

3.4). 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 a PWAM executing a `p_call` instruction. P_Call Frames (which are part of the environment, occupying a number of reserved "Y" registers) 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 of parallel goals in a similar manner to choice-points for sequential goals [14]. In practice, the stack is divided into separate *Control* (Choice Point and Markers) and *Local* stacks (Environments) for reasons of locality and locking. Table 1 summarizes the types of objects allocated in these areas and their locality.

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 3.4. Table 2 lists (in simplified form[7]) the additional instructions which currently support AND-parallelism and the WAM instructions which are modified in PWAM. Control instructions are responsible for scheduling and synchronization of parallel goals. Control instructions of the form `..._det_...` are optimized for determinate execution and they provide significant performance advantage when no backtracking is needed. Instructions like `pop_foreign_goal`, `idle`, `redo`, etc. are *pseudo-*

[7]Since the PWAM is based on the SICStus WAM the actual instructions are actually considerably more numerous and specialized, containing direct encodings of argument registers into opcodes and similar optimizations proper of an optimized WAM. The instructions presented here are for the purposes of illustration and comparison with the RAP-WAM. Space limitations make a complete description of the execution model impossible.

Control Instructions	Control Instructions (Det. Exec.)
pcall [# goals], [label list]	det_pcall [# goals], [label list]
pop_wait	det_pop_wait
Modified WAM Instructions	**"Check" Instructions**
proceed	check_me_else Label
fail	check_ground Vn
	check_independent Vn,Vm
Pseudo-Instructions	
idle	redo
pop_foreign_goal	unwind
kill	

Table 2: Parallel Abstract Machine-Specific Instructions

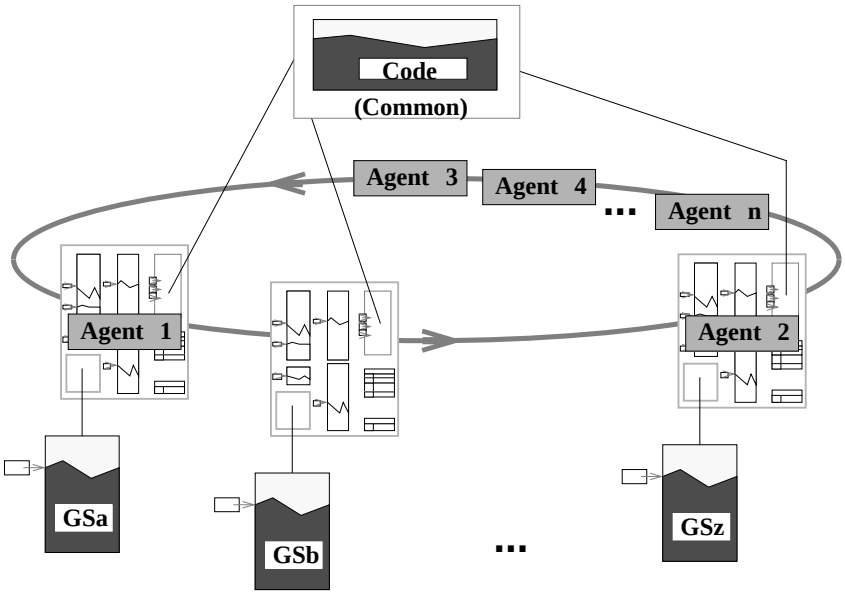

Figure 3: &-Prolog Run-time System Architecture

instructions which represent the actions taken upon failure and during distributed scheduling and backtracking. The reader is referred to Hermenegildo [12] or Tick [28] for a more complete description of the PWAM/RAP-WAM instruction set and storage model.

3.4 &-Prolog Run-Time System

The run-time system architecture, pictured in Figure 3, comprises 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 result 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 `p_call` instruction followed by a `pop_wait` instruction is generated (see previous example) from a phrase of the form `(p & ... & q)`—i.e. a request that the literals separated by `&`s be executed concurrently.

A running PWAM executing a `p_call` instruction pushes the instruction's arguments (locations in code space where the compiled literals are) on to the PWAM's Goal Stack and creates a P_Call Frame on the PWAM's Stack (in the current environment[8]). 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 as follows (see, for example, [14] 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 it attaches to that PWAM (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 on to 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

[8]Note that while the PWAM uses conventional *environment sharing* for sequential goals, it uses a rather interesting combination of *goal stacking* and environment sharing for parallel goals: goal descriptors are stacked in the Goal Stack, but their associated storage is in shared environments in the Stack.

Bench	Sun3/60			
	Q2.2	*S0.5*	*&Pseq*	*&Ppar*
matrix(50)-int	6.25	23.2	23.9	24.3
matrix(50)-float	16.1	47.4	47.7	47.87
qs(1000)-append	1.32	2.54	2.54	2.60
qs(1000)-dl-si	1.30	2.41	2.42	2.42
qs(1000)-dl-nsi	1.30	2.41	2.42	2.49
occur(50)-no/idxng	25.6	31.2	31.1	31.1
occur(50)-w/idxng	13.62	26.52	26.7	26.74
boyer(3)-si	18.7	51.64	51.64	51.64
boyer(3)-nsi	18.7	51.64	51.64	56.9
annotator(100)	0.65	0.83	0.82	0.82

Table 3: Execution Time (S): on SUN3/60, Quintus 2.2 vs. Sictus 0.5 vs. &-Prolog Seq. vs. &-Prolog Par. (1 Proc.)

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; we refer the reader instead to [14].

3.5 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 [4] implementation.[9] 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.

4 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.

4.1 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",

[9] The extensions to SICStus represented a 10% code increase –2000 additional lines of "C"– in the emulator, a 6% increase –300 additional lines of Prolog– in the Library, and a 300% increase in the compiler –6000 additional lines of Prolog.

	Sequent Symmetry		
Bench	*S0.5*	*&Pseq*	*&Ppar (1 proc.)*
matrix(50)-int	23.4	23.42	23.8
matrix(50)-float	38.7	38.8	38.8
qs(1000)-append	2.92	2.97	3.05
qs(1000)-dl-si	2.75	2.79	2.79
qs(1000)-dl-nsi	2.75	2.79	2.88
occur(50)-no/idxng	31.8	31.9	32.0
occur(50)-w/idxng	27.1	27.2	27.39
fib(22)	8.55	8.59	11.22
fib(22)-gran12	8.55	8.59	8.7
hanoi(14)	15.3	15.5	16
orsim(cp2)	3.14	3.15	3.17
rem_disj	1.33	1.33	1.35
boyer(3)-si	44.1	45.5	45.5
boyer(3)-nsi	44.1	45.5	56.0
annotator(100)	0.90	0.91	0.91

Table 4: Execution Time (S): Sequent Symmetry, Sictus 0.5 vs. &-Prolog Seq. vs. &-Prolog Par. (1 Proc.)

"occur-w/idxng", "fib", "fib-gran", "hanoi") are relatively small programs but with well understood granularity and ideal speedup characteristics and which have been used by us and other researchers in previous studies [28, 17, 19, 24]. 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 [19]). "matrix-float" is again the same program but using floating point numbers. "qs-append" is the familiar quicksort 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 [24]. "occur-w/idxng" is the same program, but with some argument positions permuted to improve indexing behaviour. "fib" generates the first 22 Fibonnaci numbers, "fib-gran" is the same program but with task granularity control code added, as in [7]. "hanoi" is the towers of Hanoi program operating on 14 disks.

The second set of benchmarks ("orsim", "rem_disj", "boyer", "annotator") contains substantially larger programs which can arguably be considered realistic and representative benchmarks. "orsim" is the or-parallel simulator of [26], quite a large program itself, simulating a relatively small program. "rem_disj" is the front-end of the &-Prolog compiler. "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 3.2. 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
fib(22)	11.22	6.03	3.15	2.15	1.77	1.48
fib(22)-gran12	8.7	4.45	2.32	1.52	1.17	.98
hanoi(14)	16	8.68	5.05	4.2	3.71	3.66
orsim(cp2)	3.17	1.79	1.11	.85	.73	.66
rem_disj	1.35	.74	.38	.25	.19	.16
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 5: Execution Time (S): &-Prolog on Sequent Symmetry, 1-10 Agents (Processors)

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

Table 6: Execution time (S): Sicstus vs. &-Prolog on Balance

4.2 Sequential Performance: Overhead Comparison

As mentioned before, the &-Prolog run-time system, an evolution of SICStus0.5 [4], 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. Tables 3 and 4 illustrate this point. These tables represent the overhead paid by remaining at the C level. In addition they also illustrate 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 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

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
fib(22)	.76	1.42	2.73	3.99	4.86	5.8
fib(22)-gran/12	.98	1.93	3.70	5.65	7.34	8.76
hanoi(14)	1.0	1.84	3.17	3.81	4.31	4.38
orsim(cp2)	1.0	1.76	2.84	3.71	4.32	4.77
rem_disj	1.0	1.82	3.55	5.4	7.1	8.43
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 7: Relative Speedup: &-Prolog on Sequent Symmetry, 1-10 Agents (Processors)

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, 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. A somewhat larger overhead is found in "boyer" and "fib" due to the extremely small granularity and very large number of parallel goals generated in these programs.

4.3 Parallel Performance: Timings and Speedups

Table 5 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 6 provides similar results for the Sequent Balance. Timings from the execution of SICStus0.5 on this machine have also been included to complement the data in tables 4 and 3. 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. [19]). Table 7 shows the actual speedups obtained on the Sequent Symmetry (speedups on the Balance are normally better). All versions of "matrix" and "occur" show excellent speedups (raw timing figures and speedups of "occur" allow comparison with Kale's system [24]). The "annotator", "rem_disj" compiler front end, and "boyer-nsi" (i.e. parallelized using "non-strict" independence) also show quite good speedups. This is especially encouraging since they are 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 somewhat low. Although it can be improved arbitrarily by increasing the size of the data, it shows the limited nature of the parallelism in the benchmark. Speedup on "fib" can be greatly improved by using granularity control, but it is quite encouraging that &-Prolog still manages to get a speedup of 5.8 on such a low granularity benchmark with no granularity control at all. The or-parallel simulator offers quite reasonable speedup, specially considering that it is simulating a program with a relatively small search space (this program was chosen to allow the direct comparison with the simulation results of [26] given below). In this simulator the or-parallelism in the simulated

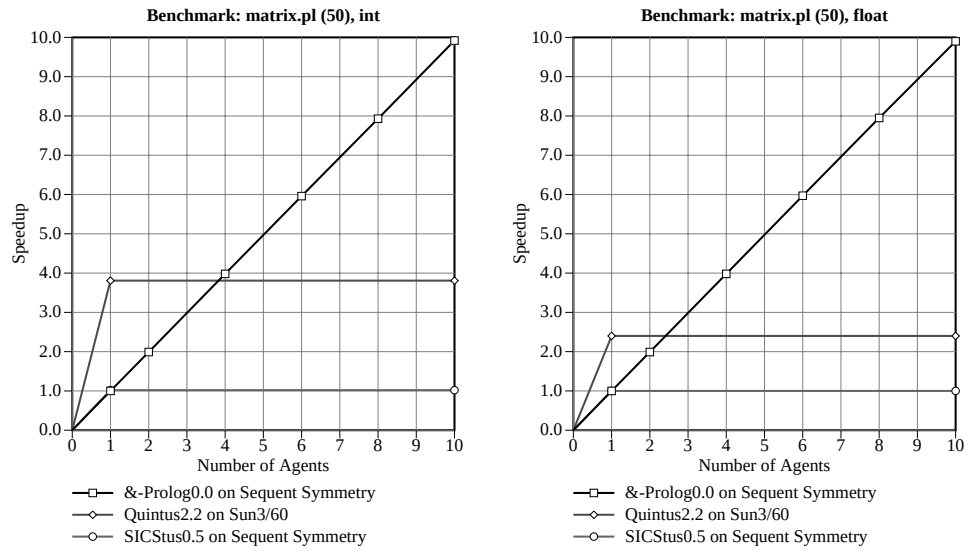

Figure 4: Speedup for matrix(50) w.r.t. Quintus on Sun3/60 and SICStus on 1 Symmetry proc

program translates into and-parallelism in the simulator. Arbitrary speedups can be obtained by running simulations of programs with larger search spaces.

These speedups are encouraging and confirm previous low-level simulation results as described in [17, 28]. In order to separate the effect of the amount of parallelism intrinsic to the benchmark from that of implementation overheads it is interesting to compare the speedups obtained with those predicted by high-level simulations. In the following we use the results presented in [26] comparing the ideal speedup figures given by the simulator described therein with &-Prolog for the "boyer" and "orsim" benchmarks (the "orsim" benchmark is compared for two sizes of programs).

	boyer_nsi(2)				orsim(sp1)				orsim(sp2)			
#	Time	Act.	Pre.(0)	Pre.(8)	Time	Act.	Pre.(0)	Pre.(8)	Time	Act.	Pre.(0)	Pre.(8)
1	8.49	1×	1×	1.00×	3.05	1×	1×	1.00×	12.42	1×	1×	1.00×
2	4.51	1.88×	1.97×	1.94×	2.81	1.09×	1.14×	1.14×	6.82	1.82×	1.87×	1.87×
3	3.13	2.71×	2.91×	2.77×	2.81	1.09×	1.14×	1.14×	4.78	2.60×	2.69×	2.67×
4	2.50	3.40×	3.76×	3.49×	2.82	1.08×	1.14×	1.14×	3.99	3.11×	3.38×	3.33×
5	2.11	4.50×	4.54×	3.94×	2.82	1.08×	1.14×	1.14×	3.23	3.84×	3.76×	3.72×

Table 8: Comparison of actual and predicted speedup for a Sequent Balance

	boyer_nsi(2)				orsim(sp1)				orsim(sp2)			
#	Time	Act.	Pre.(0)	Pre.(8)	Time	Act.	Pre.(0)	Pre.(8)	Time	Act.	Pre.(0)	Pre.(8)
1	1.85	1×	1×	1.00×	0.71	1×	1×	1.00×	3.13	1×	1×	1.00×
3	0.78	1.73×	2.91×	2.77×	0.69	1.03×	1.14×	1.14×	1.29	2.43×	2.69×	2.67×
5	0.58	3.19×	4.54×	3.94×	0.68	1.04×	1.14×	1.14×	4.78	2.60×	3.76×	3.72×
7	0.49	3.78×	5.91×	4.88×	0.70	1.01×	1.14×	1.14×	3.23	3.84×	5.36×	5.16×

Table 9: Comparison of actual and predicted speedup for a Sequent Symmetry

Tables 8 and 9 show the comparison of results from the simulator with that of &-Prolog running on a Sequent Balance and a Sequent Symmetry respectively. The benchmarks selected for comparison have relatively low maximum ideal speedups (between 1.14 and 12.77), as this would allow divergences from linear speedups to show up clearly with the limited number of processors used in the comparison. The columns in the tables have the following meaning:

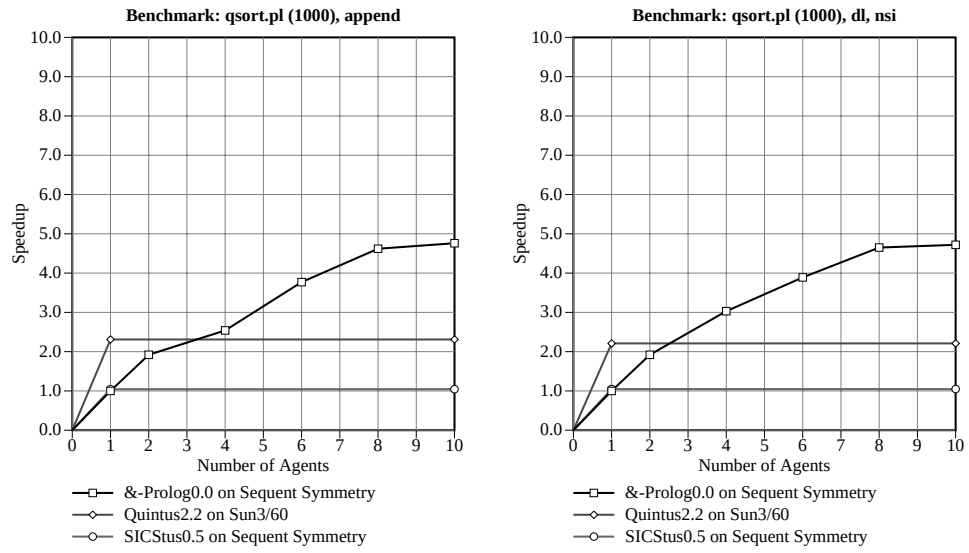

Figure 5: Speedup for qs(1000) w.r.t. Quintus on Sun3/60 and SICStus on 1 Symmetry proc

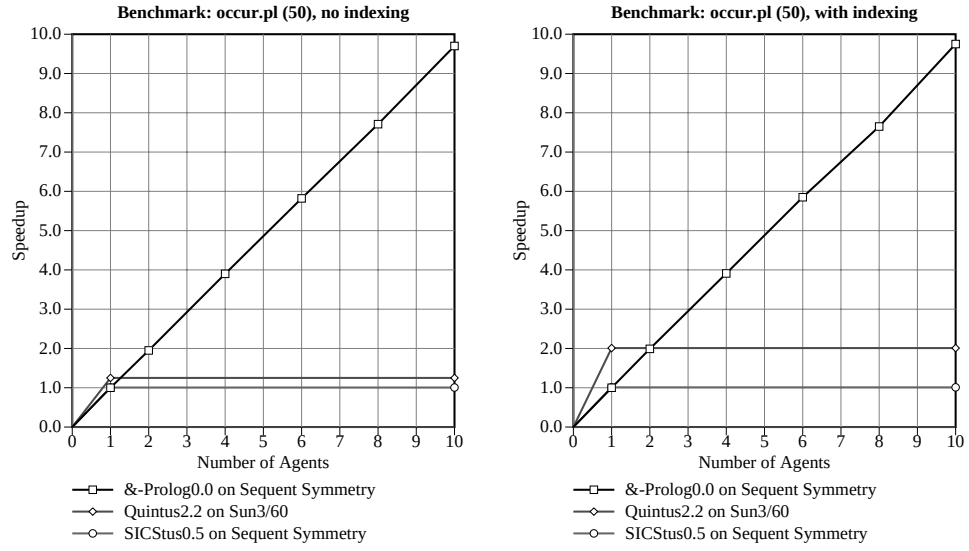

Figure 6: Speedup for occur(50) w.r.t. Quintus on Sun3/60 and SICStus on 1 Symmetry proc

\# Number of agents

Time Time in seconds to execute the program on &-Prolog. This is the fastest of 5 timings – the fastest time instead of the average time is chosen because we are interested in comparing what &-Prolog is capable of with respect to the ideal speedup.

Act. The actual speedup of &-Prolog over the execution time on 1 agent.

Pre.(0) Speedup predicted by simulator, assuming 0 units of overhead.

Pre.(8) Speedup predicted by simulator, assuming 8 units of overhead per task (4 each at start and end of task).

The agreement between the simulator's result and actual speedups on a Balance is excellent.

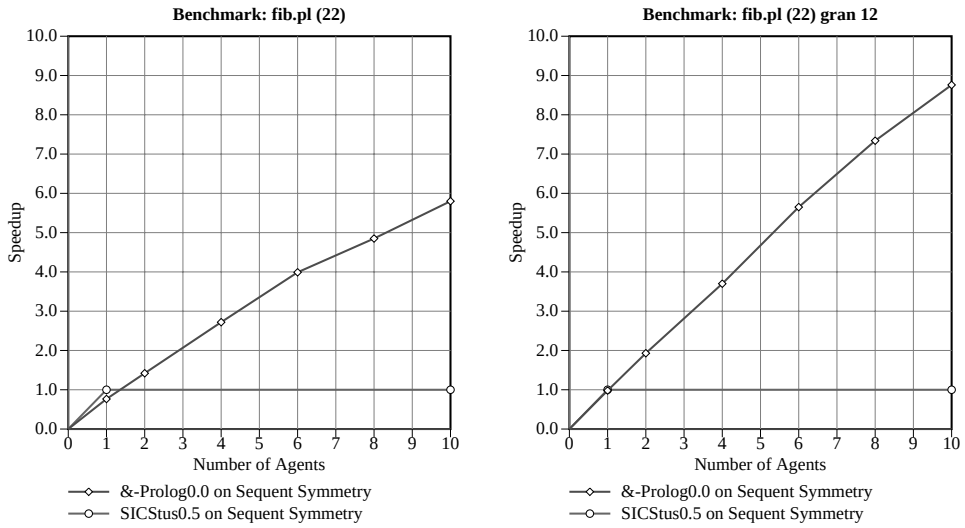

Figure 7: Speedup for fib(22) w.r.t. SICStus on 1 Symmetry proc

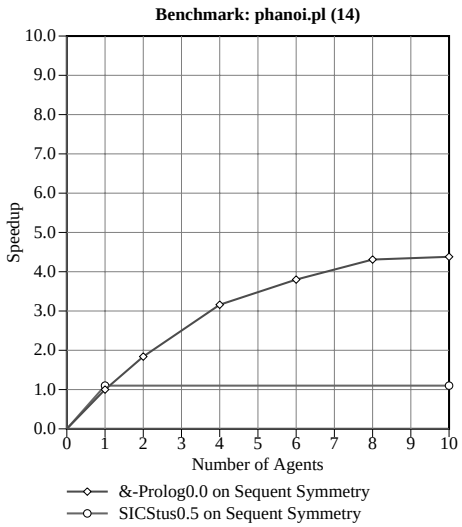

Figure 8: Speedup for hanoi(14) w.r.t. SICStus on 1 Symmetry proc

The speedup for a Symmetry is not as good as that for a Balance. We believe that the reason for this is partly that the faster processors create more bus contention on the Symmetry, which has essentially the same bus as the Balance, and partly that the &-Prolog system was originally developed on a Balance and has never been specially tuned for the Symmetry. The difference in speedup suggests that the results obtained on the Symmetry could still be improved by further fine-tuning the implementation.

Since &-Prolog's objective is to provide performance beyond that offered by current sequential systems the data from table 7 should be correlated with that of tables 3 and 4. Figures 4 through 10 represent this correlation in a graphical format. In each graph, the two horizontal lines represent the speeds of Quintus2.2 (for some benchmarks) 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

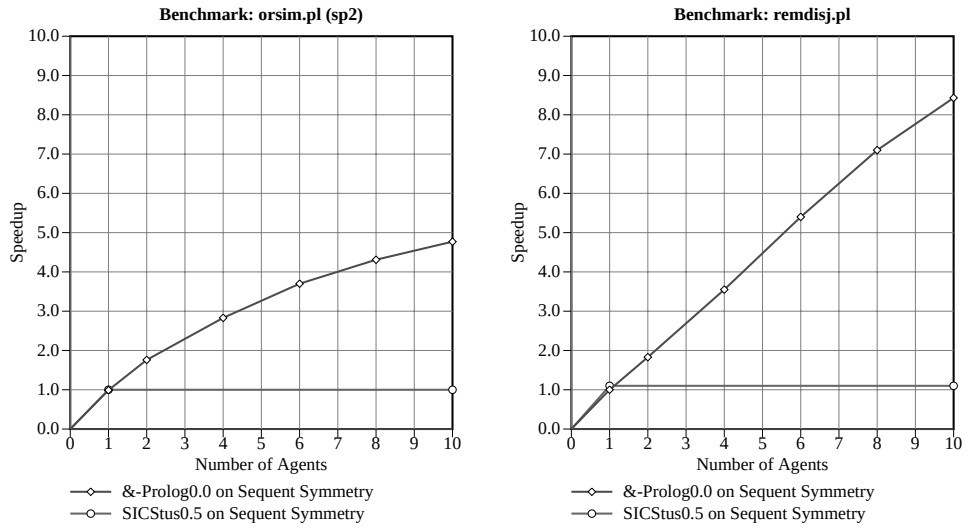

Figure 9: Speedup for orsim and rem_disj w.r.t. SICStus on 1 Symmetry proc

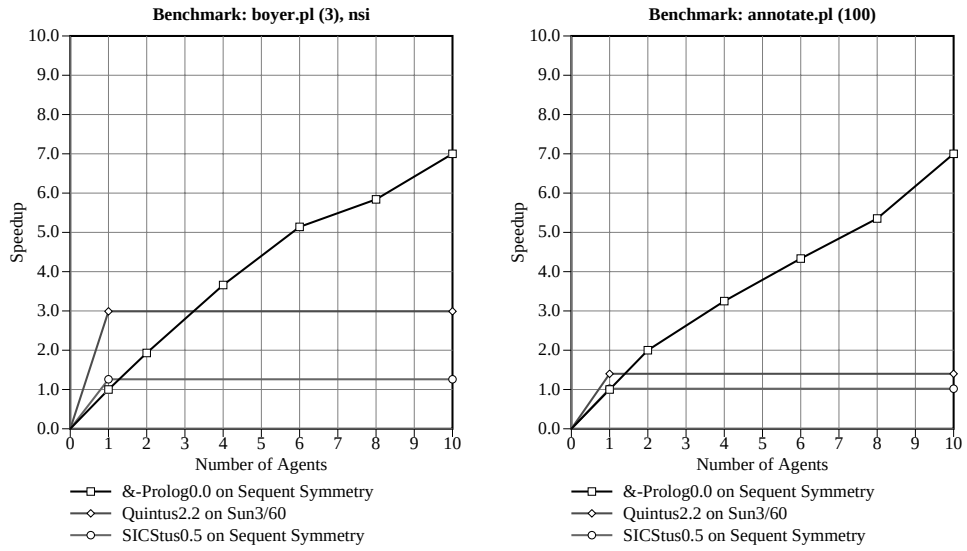

Figure 10: boyer(3) and annotator(100), w.r.t. Quintus on Sun3/60 and SICStus on 1 Symmetry proc

processor Symmetry. Performance is significantly higher than that of SICStus, even if running on only two processors. Despite the sequential speed handicap of &-Prolog with respect to Quintus (illustrated in Table 3) substantial speedups are still obtained. Higher performance than Quintus is obtained with 2-4 processors.

5 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 over an order of magnitude faster than state-of-the-art sequential systems on small parallel machines (10-20 processors) for programs that exhibit enough parallelism, and can be guaranteed to perform no slower for all programs. This is, in our opinion, of special interest in the light of the new generation of "desktop multiprocessors." Issues which deserve further study are scheduling and memory management tradeoffs, a more detailed performance evaluation, an analysis of the influence of granularity control [7], improved parallelization algorithms (specially for non-strict IAP), combination with Or-parallelism (a combination with a BC-like model [1] is currently under way [9]), application of the compilation and implementation techniques to other systems [11, 9, 11, 25, 24, 10], etc.

6 Acknowledgements

The authors would like to thank the Swedish Institute for Computer Science (SICS) for allowing the use of SICStus-Prolog V0.5 as the starting point for the implementation of &-Prolog. We would also like to thank K. Muthukumar, Roger Nasr, Francesca Rossi, Kish Shen, Francisco Bueno, Maria Jose Garcia de la Banda, Manuel Carro, and Richard Warren for their important contributions to the development of &-Prolog.

References

[1] K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[2] 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, Seattle,Washington, 1988.

[3] P. Brand, S. Haridi, and D.H.D. Warren. Andorra Prolog—The Language and Application in Distributed Simulation. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

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

[5] 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.

[6] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[7] S.K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

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

[9] G. Gupta and M. Hermenegildo. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *ICLP '91 Workshop on Parallel Execution of Logic Programs*, number 569 in LNCS, pages 146–159. Springer-Verlag, December 1991.

[10] G. Gupta and B. Jayaraman. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 332–349. MIT Press, October 1989.

[11] G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: Integrating Dependent And-, Independent And-, and Or-parallelism. In *1991 International Logic Programming Symposium*, pages 152–166. MIT Press, October 1991.

[12] M. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[13] M. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.

[14] M. 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.

[15] M. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, number 225 in LNCS, pages 40–55. Imperial College, Springer-Verlag, July 1986.

[16] 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*, pages 369–390. MIT Press, October 1989.

[17] M. Hermenegildo and E. Tick. Memory Referencing Characteristics and Caching Performance of AND-Parallel Prolog on Shared-Memory Architectures. *New Generation Computing*, 7(1):37–58, October 1989.

[18] A.H. Karp and R.C. Babb. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software*, September 1988.

[19] 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.

[20] K. Muthukumar and M. Hermenegildo. Complete and Efficient Methods for Supporting Side Effects in Independent/Restricted And-parallelism. In *1989 International Conference on Logic Programming*, pages 80–101. MIT Press, June 1989.

[21] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Int'l. Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[22] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[23] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[24] B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 313–331. MIT Press, October 1989.

[25] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *1991 International Conference on Logic Programming*. MIT Press, June 1991.

[26] K. Shen and M. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *International Logic Programming Symposium*, pages 135–151. MIT Press, October 1991.

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

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

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

[30] 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.

[31] R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

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