PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

THE PARALLEL ECRC PROLOG SYSTEM PEPSys: AN OVERVIEW AND EVALUATION RESULTS

Uri Baron, Jacques Chassin de Kergommeaux, Max Hailperin*
Michael Ratcliffe, Philippe Robert**
Jean-Claude Syre, Harald Westphal***

ECRC, Arabellastr. 17, 8000 Muenchen 81, West Germany

E-mail: {uri, chassin, michael, jclaude}@ecrcvax.uucp

ABSTRACT

PEPSys (Parallel ECRC Prolog System) is a research project aimed at the design and evaluation of a multiprocessor system for large scale parallel logic applications. Following an initial study period of parallel logic paradigms and Prolog applications, a parallel language, a computational model and then an abstract machine and compiler were defined. The PEPSys system was implemented on an 8-PE, shared memory multiprocessor system and a cluster based architecture with up to 100 PEs was simulated.

This paper discusses the different topics attacked by the project, from high level language features to the evaluation of results. It compares the initial results with those from other projects and suggests several possible improvements.

1 INTRODUCTION

PEPSys is a research project in the Computer Architecture Group of ECRC, aimed at the design and evaluation of a multiprocessor system for large scale parallel logic applications. Starting with an analysis of Prolog applications and existing parallel logic models, a team of 6 people defined a parallel language and a parallel computational model, in 1984 and 1985. An abstract machine and compiler were defined in 1986. During 1986 and 1987, the PEPSys system was implemented in two complementary ways: firstly on a commercial multiprocessor (MX500 from Siemens, equivalent to the Sequent Balance 8000), and secondly as a simulation model for a cluster-based multiprocessing architecture.

There exist two main classes of parallel logic programming systems:

- The first uses the "Committed Choice" paradigm to select clauses, and input/output dependencies between goals. Typical and well known examples of such systems are Concurrent Prolog (Shapiro, 1986), Parlog (Clark and Gregory, 1986), and GHC (Ueda, 1985). They exploit "don't care" non-determinism, and stream AND-parallelism, realized by synchronizing a set of processes producing and consuming data. In their initial definition, these languages abandon the declarative style of logic programming while retaining a similar syntax, and their semantics (at least originally) were known to suffer from ambiguities. Their expressive power is, in principle, limited to "one-solution" systems, and fits such applications as systems programming and simulation systems. Their "flat" versions (Ichiyoshi et al., 1987, Mierowski et al., 1985, Foster and Taylor, 1987), developed to have clearer semantics and more efficient execution models, have been implemented. The addition of pure OR-parallelism which is not impossible, is not quite consistent with their basic principles.
- The second is based on the "pure" parallelisms that Prolog exhibits. They exploit either OR-parallelism or AND-parallelism; some systems allow both. These systems are "all-solutions" systems, and do not have any concept of synchronization (at least at the level of the semantics). They keep the declarative semantics of Prolog, and thus are simple and consistent extensions of this language. Their application fields include knowledge base systems and decision support systems. Among the numerous research teams attacking these issues are the Argonne Labs (Disz et al., 1987), the Swedish Institute of Computer Science (Ciepielewski et al., 1987), the University of Manchester (Warren, 1987, Brand, 1988), the University of Maryland (Giuliano et al., 1987), the University of Kobe (Matsuda et al., 1987), and ECRC.

It is our belief at ECRC that most Artificial Intelligence applications (incomplete criteria search, exhaustive search, deductive data bases, decision support systems in general) require both the "all-solutions" and the non-determinism features of Prolog.

Discussion of the various approaches is beyond the scope of this paper but the interested reader may refer to (Syre and Westphal, 1985), (Takeuchi and Furukawa, 1986), for more details. PEPSys itself has several unique characteristics:

- It offers an integrated solution: PEPSys consists of a high-level parallel logic language, a parallel computational model, an abstract machine, an implementation on a commercial multiprocessor and a simulator to evaluate cluster-based multiprocessor systems. The implementation and simulation are supported by comprehensive debugging tools, graphical analysis facilities, and application programs.
- It handles both OR-parallelism and independent AND-parallelism, under explicit user control, in an "all solutions" approach. These parallelisms can be nested arbitrarily combined with sequential processing as required.
- The issues of efficiency and practicability have been constant guidelines at all stages of the project. The abstract machine was conceived so that sequential processing would exploit state-of-the-art sequential Prolog technology. In the same vein, considerable importance was attached to the overhead incurred by the

*Now at Stanford University, USA
**Now at Verilog, Toulouse, France
***Now at SoftLab, Munich, West Germany

parallel features of the system at run time, minimizing it wherever possible.

The paper is organized as follows: Section 2 introduces the main features of the high level language with an example. Section 3 presents the parallel computational model while Section 4 briefly describes the execution model (abstract machine) derived for this model. Section 5 describes the implementation of PEPSys on a Siemens MX500 multiprocessor system, and Section 6 deals with the simulation tools allowing the evaluation of different multiprocessor architectures. Section 7 is a general discussion of the PEPSys system and includes comparisons with other similar systems.

2 THE PEPSys LANGUAGE

2.1 Is a language necessary?

A careful study of non trivial programs written in normal Prolog, and an investigation of possible parallel applications led to the following conclusions:

- There are parts of a program which must be kept sequential.
- OR parallelism: many predicates composed of several clauses are semantically changed when turned into a parallel form without care and Prolog programmers often use the sequential control offered by its operational semantics. There is information on parallelism which can be easily expressed by the programmer at source level (some predicates, looking trivially parallel, may just generate more overhead than speedup)
- AND parallelism: Automatic methods and algorithms to detect AND parallel goals in a clause are not yet satisfactory, and often lead to conservative cases for static dependency algorithms or a significant overhead at run time for dynamic algorithms.
- Moreover, from application programs, and our current experience, we realize that the "true" and "interesting" parallelisms in a program are often localized to a few predicates belonging to the higher levels of the search trees generated by queries.

These considerations led us to define an extension of Prolog, allowing the expression of OR-parallelism and independent AND-parallelism (goals having either no uninstantiated shared variables or which do not instantiate any shared uninstantiated variables to different values). It provides all solutions to a query (the non determinism is a fundamental feature, given the target application areas). It keeps the declarative semantics of Prolog intact and in particular, does not resort to the I/O dependencies expressed as modes (Parlog), variable annotations (Concurrent Prolog), or even implicit data flow (GHC).Finally, it allows the programmer to control both the nature and the amount of parallelism and is flexible enough to be a good vehicle for experiments and further extensions.

2.2 Overview of the language

Modularity: The PEPSys language provides modules, which is an easy and clean way to clearly separate the sequential and parallel parts of a program. In a sequential module, the user is in the familiar Sequential Prolog environment, and uses its operational semantics, as well as "dirty" predicates to optimize the program, to express input output operations, and special control.

AND-Parallelism: An explicit operator ("#" replacing the ",") denotes the independent execution of two goals within a clause. Thus responsibility is given to the programmer is justified to keep the parallelism under control; computing two subproblems in parallel, or recursively computing a list of data are common and useful types of AND-parallelism. An interactive compiler including automatic parallelism detection would also be of great value to discover less common cases.

OR-Parallelism: In a parallel module, all clauses of a predicate must be gathered together and headed by a Property Declaration of the following form:

-properties([<Sol-prop>, <Clause-prop>,<Exec-prop>]).

The property <Sol-prop> indicates whether the predicate should deliver all its solutions, or only one (default is "all"). The property <Clause-prop> specifies if the ordering of clauses (i.e. of the solutions delivered by the predicate) is semantically significant or not (default is "ordered"). The property <Exec-prop> refers to the eager or lazy way of obtaining the solutions as the most desirable scheme (default is "lazy"). The first two properties affect the semantics of the predicates, while the third is only advice to the run time system.

2.3 The language in one example

It is difficult to describe here all the capabilities of the language (see (Ratcliffe and Syre, 1987) for a full description), but the example given in Figure 1 (being, in addition, a very popular Prolog program!) will show some interesting aspects. Only the parallel module is shown here; the sequential module simply calls the *get_solutions* predicate within a *bagof* and lists the solutions.

```
/*    PEPSys n-queens Programme      (c) copyright ECRC GmbH    */

/* Parallel Module */
?-export( [get_solutions/2] ).      % Export entry point

-properties( [] ).
get_solutions(Board_size, Soln) :- solve(Board_size, [], Soln).

% Accumulate the positions of occupied squares
-properties( [solutions(all),clauses(ordered),execution(lazy)] ).
solve(Bs, [square(Bs, Y) | L], [square(Bs, Y) | L]).
solve(Board_size, Initial, Final) :-
      newsquare(Initial, Next, Board_size),
      solve(Board_size, [Next | Initial], Final).

% Generate legal positions for next queens
-properties( [solutions(all),clauses(ordered),execution(lazy)] ).
newsquare([square(I, J) | Rest], square(X, Y), BoardSize) :-
      I < BoardSize, X is I + 1, snint(Y, BoardSize),
      not(threatened(I, J, X, Y)) # safe(X, Y, Rest).
newsquare([], square(1, X), BoardSize) :- snint(X, BoardSize).

% Generate all possible positions for the next queen
-properties( [solutions(all),clauses(unordered),execution(eager)] ).
snint( N, NPlusOneOrMore ) :-
      M is NPlusOneOrMore - 1, M > 0, snint( N, M ).
snint( X, X ).

% Check if queens on squares (I, J) and (X, Y) threaten each other
-properties( [solutions(one),clauses(unordered),execution(lazy)] ).
threatened(I, J, I, Y).
threatened(I, J, X, J).
threatened(I, J, X, Y) :- U is I - J, U is X - Y.
threatened(I, J, X, Y) :- U is I + J, U is X + Y.

% Checks whether square(X, Y) is threatened by any existing queens
-properties( [solutions(one),clauses(ordered),execution(lazy)] ).
safe(X, Y, []).
safe(X, Y, [square(I, J) | L]) :-
```

not(threatened(I, J, X, Y)) # safe(X, Y, L).

Figure 1: The "N-queens" Program in PEPSys

Consider the property declarations defined by the programmer:

solve/3: all solutions are obviously required, but making the predicate parallel cannot give any speedup.

newsquare/3 : all solutions are required, but only one of the clauses may succeed at a time, the other producing an immediate failure, hence the *ordered* and *lazy* attributes.

snint/2: this small predicate is the heart of good parallelism generation since it allows the computation tree to be split into large parts, thus providing coarse grain parallelism.

threatened/4 : this is a typical case of independent clauses, where the small granularity of the parallel tasks cannot compensate for the overhead for making them parallel.

safe/3 : this expresses a vector operation by means of recursion, which is, in principle, good for parallelism. The "#" operator indicates an AND-parallel node. However, the user could have chosen to make it serial, because there is already some coarse grain parallelism (from the *snint* predicate), and the parallelism generated by *safe* is of a comparatively finer granularity.

2.4 Static evaluation of the language

The language was tested at an early stage of the project in order to evaluate its capabilities. An abstract interpreter/evaluator ran programs on an ideal machine under unrealistic hardware assumptions, but allowing an evaluation of the usefulness of properties declarations, and how much parallelism the programs could in principle provide.

After two years of experience with the language, we can make the following comments:

- A user-controlled parallelism is always better than no control. Moreover, the pure parallelism of PEPSys has a clear semantics similar to that of conventional Prolog.
- AND and OR parallelisms are both worthwhile, there is no reason to sacrifice one of them.
- The property declarations are tied to predicate definitions and not to predicate calls. This is a well known problem illustrated by Parlog (predicate level modes) and Concurrent Prolog or GHC (call level synchronization). It is sometimes desirable to be able to call a parallel predicate with different properties from the original ones.
- The total lack of dependence between parallel computations makes the programs clear and manageable. However some dependence may be desirable in some cases (e.g. when one branch gives results making all other branches useless) for efficiency reasons.

The latter two points have been overcome in a second version of the language, by the introduction of a *with(property)* operator and an *asynchronous data base* (Ratcliffe, 1988). A lemma may be generated by one branch and used by another, allowing a loose synchronization and control among several parallel computations. While not implemented in the current system, an evaluation by a different interpreter showed that such a feature is valuable, if not absolutely necessary, for efficiency reasons.

3 THE PEPSys PARALLEL COMPUTATIONAL MODEL

The PEPSys computational model (Westphal et al., 1987) was designed to support efficient implementation of the PEPSys language. A scheme was devised for the parallel execution of coarse-grained, sequential processes running at speeds close to ordinary sequential Prolog execution. Controlling the search space to produce all (desired) solutions and managing variable bindings in a parallel environment are the two main issues addressed in the model. The model's main features are:

- Support for OR-parallelism, Independent AND-parallelism and a combination of both, together with sequential backtracking.
- Shallow binding with an explicit time-stamp mechanism.
- The ability to convert a sequential computation to a parallel one retroactively at a very low cost.

3.1 Management of Variable Bindings

The model distinguishes between *shallow-bindings*, performed in the normal Prolog stacks, and *deep-bindings*, stored in *Hash-windows*. Any PEPSys process can access the stacks and Hash-windows of its ancestor processes. The time-stamp *Or-Branch-Level* (OBL), associated with each binding permits it to distinguish the relevant bindings from the others in the ancestor processes' stacks and Hash-windows, an example is given in Figure 2. All the bindings performed on "ancestor variables" of a process are made in the process's Hash-window and time-stamped, as any other binding, with the current OBL.

Figure 3 shows a case with AND-parallelism with full OR-parallelism. If both sides of an AND-parallel split contain branch-points there can be multiple solutions for both branches. In this case the *cross product* of the left-hand and right-hand solution sets has to be formed. A member of this cross product is a process that has one process of both the left-hand and right-hand sides as ancestors. Access to the bindings of these ancestors is handled by *join-cells*, an extension to the hash-window scheme.

Processes can be identified by hash-windows; each process has exactly one current hash-window. Thus in order to pair two processes, a join-cell contains two pointers: one to the hash-window of the left-hand process and one to the hash-window of the right-hand process. A join-cell also contains a third pointer - the *last-common-hash-window* pointer to the hash-window that was current at the time of the AND-parallel split. Looking up a variable binding from a goal after the AND-parallel join works as follows: the linear chain of hash-windows is followed in the usual way until a join-cell is reached. Now a branch becomes necessary. First the right-hand process is searched by following the join-cell's right-hand-side hash-window chain. When the last-common-hash-window is encountered control bounces back to the join-cell and the left-branch is searched. The scheme also works in the nested case.

3.2 Control

The efficient generation of the complete set of solutions to a query is the major concern in controlling the search space at execution time.

OR-parallelism: A process executing an OR-parallel predicate will create a *Branch Point*. This Branch Point has two functions: (1) to co-ordinate work taken by idle processors and (2) to enable normal sequential backtracking execution.

All processors have a list of local branch points with unprocessed alternative clauses. When a processor becomes idle it searches these lists for a task and takes the first one it finds. Almost nothing in the parent branch has to be changed. With this mechanism it is possible to generate just as many processes as are needed to sustain the system. Thus the overhead for parallelism (e.g. process creation, deep binding) occurs only when useful parallel work is done.

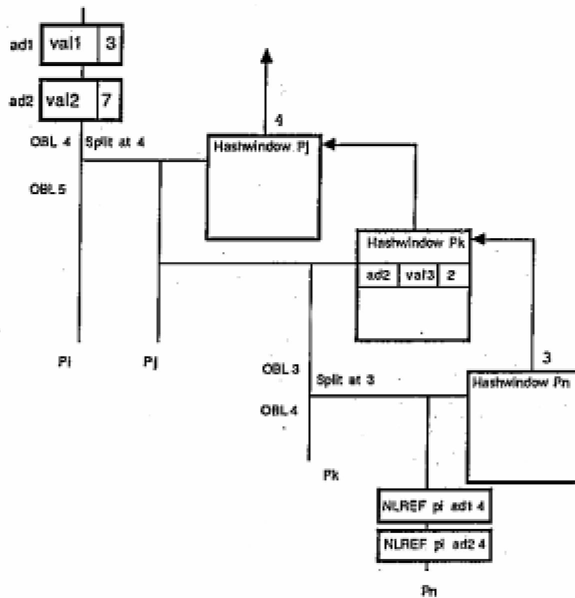

Figure 2: *OBL and Hash-windows in PEPSys*

The variable at address *ad1* was bound to val1 when the OBL value on Pi was 3; the split eventually leading to process Pn occurred later (the OBL value on Pi was 4) and the shallow binding at address ad1 is valid for process Pn. Shallow binding at address *ad2* was not bound when the split occurred and is not valid for Pj, Pk and Pn. However the variable was bound to val3 in the Hash-window of Pk when the OBL in Pk was 2, that is before the creation of Pn which occurred when the OBL on Pk was 3. The deep-binding of *ad2* in the hash-window of Pk is thus valid for process Pn.

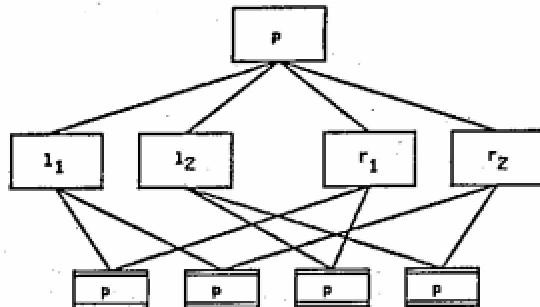

Figure 3: *Cross product by join-cells*

Clause p consists of the AND-parallel goals l and r with two solutions each. The join-cells are marked by double horizontal bars and their last-common-hash-window.

If a branch fails, it backtracks to the last goal with multiple clauses and executes the next alternative if one exists. If it is a branch point and there are no alternatives available, it has to wait for all sibling processes to terminate before it can backtrack over the branch point. This is necessary because the spawned processes - although not allowed to write in the father's data areas - can read them to find shared global variable bindings. All branches eventually fail and backtrack, thereby ensuring that the entire search tree is explored.

AND-parallelism: AND-parallel process-creation is performed using the same mechanism as in OR-parallelism. Right-hand goals are taken by idle processors on demand. Upon termination the AND-parallel branches are *joined*. If both sides of an AND-parallel split contain branch-points, the cross-product of the left-hand and right-hand solution sets is formed incrementally as the solutions arrive: whenever a left-hand solution arrives, it is immediately paired with all existing right-hand solutions and vice versa. For all these pairs OR-Parallel processes are started if processors are or become available.

OR-parallelism, AND-parallelism, and sequential execution combined: The basic scheme for forming the cross-product: gathering the left-hand solutions and the right-hand solutions in *solution-lists* and eagerly pairing them, relies on the fact that all solutions to each side are computed incrementally, and then co-exist at the same time in memory to be paired with newly arriving solutions to the other side.

If sequential backtracking execution is allowed within the AND-parallel branches, solutions could disappear: backtracking might delete the bindings of the solution and replace them with a newly computed alternative. This problem occurs only if AND-parallelism, OR-parallelism, and sequential execution are combined. To solve this problem, the treatment of the two AND-parallel branches is not symmetric. OR-parallel solutions to the left-hand sides are gathered into solution-lists and paired as soon as possible with the solutions to the right-hand side. Assume that one solution to the right-hand side is paired with all existing solutions to the left-hand side, and that all the processes for these pairs have finished (i.e. they have delivered a solution to the initial query or have failed), this right-hand branch can only continue by backtracking into the right-hand side of the AND to compute the next (sequential) right-hand solution. Before doing this the left-hand solution-list is *frozen* in its current state. This means that newly arriving left-hand solutions (e.g. those produced by still active OR-parallel branches of the left-hand side) will not be entered into the left-hand solutions list and will not be paired with the current right-hand solutions. Rather they are entered in a different left-hand solutions list, in which the next generation of left-hand solutions is gathered.

After this freezing of the current left-hand solutions list, backtracking proceeds into the right-hand side of the AND, thereby deleting one right-hand solution set from memory. Repeated backtracking will now gradually compute the whole set of right-hand solutions, which are paired with the frozen subset of left-hand solutions (called the current left generation). When the right-hand side can produce no more solutions, either by OR-parallelism or by backtracking, the current left-hand generation has been paired with all possible right-hand solutions. Now all processes of the current left-hand generation may themselves backtrack to continue producing the next generation of left-hand solutions. The right-hand side is then recomputed. When the left-hand side cannot produce any more generations, the complete cross-product has been formed.

Only when the left-hand side finally fails, i.e. backtracking has produced all generations of solutions to the left-hand side, and each has been combined with all solutions to the right-hand side, does backtracking continue back to the last choice-point or branch-point preceding the parallel AND. Before the backtracking proceeds above the parallel AND, the right-hand side should terminate or be killed. After its death, backtracking proceeds beyond the AND-parallel split.

4 THE PEPSys ABSTRACT MACHINE

4.1 Introduction

The PEPSys Abstract Machine was designed such that one Abstract Machine is mapped onto each processor and executes part of the PEPSys program with an efficiency close to the WAM (Warren, 1983). The PEPSys Abstract Machine (Chassin et al, 1988) is an extension of the WAM supporting the two main features of the PEPSys computational model: management of variable bindings in parallel and combination of OR- AND- parallel execution with sequential execution and backtracking in an all-solutions Prolog system. Like the WAM, the PEPSys Abstract Machine is composed of the three classical Prolog stacks (local, global and trail), several control and stack pointer registers, and a large number of argument registers. The data objects stored in the registers and the stacks are tagged. The machine executes a set of instructions including indexing instructions (usually referred to as *switch*), compiled unification instructions (*put, get, unify*) and control instructions (*call, proceed, try, retry*). Many of these instructions use the basic *dereference* and *unification* operations.

4.2 Management of bindings

One of the main features of the PEPSys computational model is the explicit distinction between local and *non-local* variables and the systematic tagging of bindings with an OBL, used to check the validity of non-local variables' bindings. The data objects used in the PEPSys Abstract Machine use two tags: one tag defining the type of the object, as in the WAM, the other tag being the OBL value when the binding was performed. In addition to the WAM tagged data objects, new *non-local* data objects were defined. These are the *non-local* counterparts of some of the WAM tagged data objects: *non-local free, non-local reference, non-local list, non-local structure*. The non-local data objects also contain the explicit identification of their creator process as well as a split-OBL, used if the variable is bound, to check the validity of the binding.

The **dereferencing** algorithm had to be extended in order to cope with the two sorts of bindings used by the PEPSys model: *shallow bindings* in the stacks and *deep bindings* in hash-windows, the validity of both depending on their OBL tag when the binding was performed by an ancestor process. When it points to a non-local environment, the environment pointer register E must be a non-local data object, in order to allow a correct dereferencing of the non-local permanent variables of the Prolog stack. The **unification** algorithm was also extended. The general unification must deal with several more data objects and uses two different ways of binding variables: local variables are bound in the stacks while non-local ones are bound in hash-windows. The WAM instructions used to compile unification had to be modified to process non-local variables. In particular, the structure pointer register S, used to access compound objects, may have the *non-local reference* type, to allow correct dereferencing of non-local sublists or substructures.

4.3 Control

The two means of control in Prolog have been extended to allow inter-process synchronization. The first one is the continuation mechanism, similar to a subroutine call, used at the end of the execution of a goal to call the next goal of the clause. The second is the backtracking mechanism, without equivalent in imperative languages whereby a previous state of the computation is restored.

The continuation mechanism is invoked when an operation has to be performed after the success of a given goal: at the end of the execution of one of the branches of an AND-parallel conjunction, or after executing a *one solution* predicate, in order to allow only one of the successful processes to proceed. The backtracking mechanism has been extended to synchronize processes executing an OR-parallel predicate, to mix sequential and parallel execution in a cross-product and to synchronize a terminating process with its father process. These extensions have been done by extending the WAM with new control frames and instructions.

5 IMPLEMENTATION OF PEPSys

The implementation of PEPSys on a commercial multiprocessor (Chassin et al., 1988) was considered an important milestone in the PEPSys project because it was the only way to test the actual behavior of the computational model and experiment with the problems raised by implementing and debugging in parallel. Furthermore, an implementation allows large programs to be executed and actual measures of the efficiency of the model to be obtained.

For simplicity, it was decided to implement PEPSys on a shared memory multiprocessor, although the model does not require a global address space. The Siemens MX500 is a Sequent Balance 8000 built under license, where the number of processors has been limited to 8. As a compromise between ease of debugging and efficiency, the implementation was coded in C, the abstract machine instructions being emulated. To run a program on a given number of processors, less than or equal to 8, the implementation creates the same number of UNIX processes. In the following, we will assume that each UNIX process executes on a processor and will refer to them as processors.

5.1 The PEPSys emulator

The emulator is executed by each processor and its code is shared by the processors. The PEPSys program is stored in two data areas, a code array and a dictionary, which are shared by the processors. The local-global stack area is also shared: each processor has private write access on a fixed size portion of the area while it has read access on the whole stack area. The classical local and global stacks of each processor are implemented in the portion of the stack area privately allocated (write access) to the processor. The trail stacks are private to each processor.

The data objects are stored in two 32 bits words. The first contains the classical tag-value pair of Prolog implementations while the second contains the binding OBL and, in the case of non-local objects, the split-OBL and the processor-process identifications. The data, E and S registers have the same format as the stack data objects.

Hash-windows are allocated in the global stacks of the processors. Each Hash-window entry is composed of two data

objects: the first is the non-local reference bound, the second being the value it is bound to. The entries in the Hash-windows are normally trailed.

Local dereferencing is the same as classical sequential Prolog. When it returns a non-local reference, a non-local dereferencing procedure is called which may, if the variable is unbound for the process, call a recursive hash-window search procedure.

5.2 Process management and work allocation strategy

In each Abstract Machine, parts of the PEPSys program are executed as several processes. Only one process is active at a given time, the others being suspended. When a process cannot backtrack because its stack is being used by other processes, it suspends and a new process is created. To limit the creation of "black-holes" in the stacks, work is obtained prefrentially from the computational subtree of processes using the stack of the suspended process. If no work is available in this subtree, the complete computation tree is explored. Memory locking is used to take a piece of work but not whilst simply searching. Each process is defined by a *root-frame* and a *hash-window*, both placed on the top of the global stack of the processor.

5.3 Debugging and Instant Replay

To debug the parallel system two different tools were used: the DYNIX symbolic debugger *pdbx* and a custom-built debugger at the level of the Abstract Machine instructions. However, as real parallel systems are nondeterministic, it is very difficult to reproduce identical executions. Bugs may appear rarely and disappear as soon as an attempt to use the debugging tools is made. To alleviate this problem, a simple implementation of the Instant Replay mechanism (Leblanc and Mellor-Crummey, 1987) was done to make deterministic runs of programs possible. The basic idea is to record a minimal amount of information during a parallel execution and to use this recorded information to make deterministic executions possible. Replayed executions ensure that the synchronization events are executed in the same order. Debugging using Instant Replay requires the recording of an execution producing an intermittent bug, which is usually possible by running a large number of executions in recording mode. Then it is possible to use the other debugging tools during replayed executions, which is then deterministic with regard to the erroneous initial execution. The PEPSys implementation of the Instant Replay is efficient, since recording is only 3% slower than normal execution.

5.4 Measures

The measures were performed as single user of the MX500, with the migration and the swapping of the UNIX processes executing the PEPSys program disabled. Several types of measures are made using the implementation. The first type gives the "raw" efficiency, the second allows the checking of some assumptions basic to the PEPSys model and the last type estimates the various sources of overhead. In the following, the most important results are given for a subset of the programs used to benchmark the implementation. These programs are:

- **hamilton:** finds a closed path through a graph such that all the nodes of the graph are visited once.
- **mandel:** computes a mandelbrot set of 300 points.
- **queens:** computes all solutions to the 8-queens problem
- **saltm:** salt-and-mustard program described in (Disz et al., 1987).
- **tina:** tour generator program. This program was written at ECRC, using the PEPSys language.

Raw efficiency: Figure 4 shows the efficiency of the PEPSys implementation compared to commonly used sequential Prolog implementations: interpreted CProlog, running on the MX500 computer and compiled Quintus Prolog, running on a Sun 3/50 (16 MHz) since Quintus is not available on the MX500. The Siemens MX500 Elementary Processor is an NS32032 running at 10 MHz, giving an equivalent of 0.7 VAX780 MIPS. The ratio of the efficiency of the NS32032 processor to a 68020 running at 16 MHz is usually estimated to be 3. The ratio of the efficiency of a WAM emulator written in C (such as PEPSys) to the efficiency of an assembly code WAM emulator (such as Quintus) can be estimated to be more than 2.

The classical *naive-reverse* program runs at 3 Klips on a single PE. Of greater significance for a parallel system is the efficiency of parallel programs running on the implementation (naive-reverse is a sequential program in PEPSys) and the measure of the *speedup*, the ratio of the sequential execution time to the parallel execution time. Using the correction factors given above, it appears that the PEPSys implementation provides effective speedups over efficient sequential implementations. Figure 5 is a graph of the speedups provided by parallelism, from the same programs.

Figure 4: Raw efficiency of some benchmark programs

Program	CProlog	PEPSys(1)	PEPSys(8)	Quintus (Sun 3/50)
Hamilton	722 s.	281 s.	39.3 s.	21.1 s.
Mandel	162 s.	41.5 s.	6.3 s.	13.2 s.
Queens	173 s.	63.7 s.	9.2 s.	3.47 s.
Saltm	17.0 s.	7.5 s.	1.6 s.	0.5 s.
Tina	319 s.	111.0 s.	16.5 s.	9.5 s.

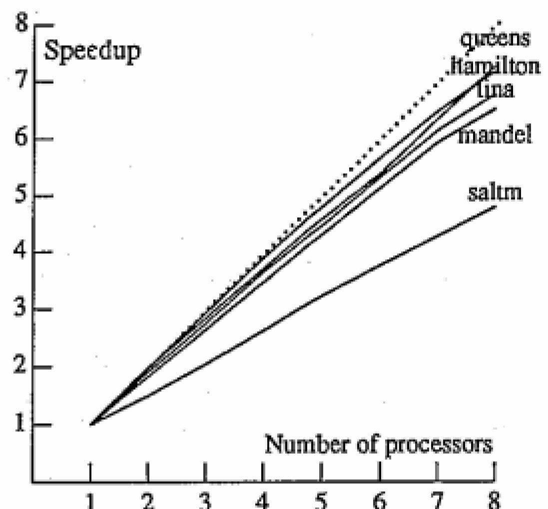

Figure 5: Speedups of the PEPSys implementation

Use of Hash-windows: One of the main assumptions made

in the definition of the PEPSys computational model was that, because of the locality of references in Prolog, the hash-window use would be infrequent enough to introduce only a low overhead in parallel executions. Several measures of hash-window usage have been made: the percentage of dereferencing operations involving a hash-window search, the longest hash-window chain explored in a dereferencing operation, the number of hash-window accessed during a computation and finally an estimate of the overhead introduced by the use of hash-windows. The main results are summarized in Figure 6 and on Table 7: they confirm that the use of hash-windows remains infrequent and even if the hash-window chain explored is long (longest value so far is 20), most of the hash-window accesses occur in the local one. The overhead introduced by hash-window dereferencing remains always low.

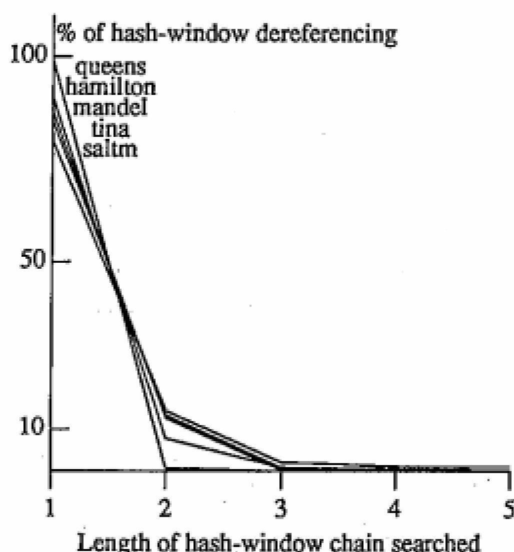

Figure 6: Use of hash-windows in the PEPSys implementation

This graph gives the percentage of hash-window searches involving the exploration of 1 (local hash-window) to 5 hash-windows during the execution of the benchmarks on 8 PEs.

Figure 7: Hash-window Dereferencing Overhead

This table gives the percentage of the execution time spent in hash-window dereferencing for configurations ranging from 2 to 8 PEs.

Program	2	4	8
Hamilton	0.07%	0.07%	0.5%
Mandel	7%	5%	4%
Queens	0.5%	0.8%	0.9%
Saltm	7%	9%	10%
Tina	0.3%	0.9%	1.3%

Processors activities: The aim of this measure is to find out the percentage of the computation time spent searching for work compared to the processing of the PEPSys program. The results depend a lot on the program, the overhead being significant in the programs providing finer grained parallelism.

Size of the processes: This is the measure of the number of inferences executed by each process during a parallel computation. Although a large number of processes have a very short life, most of the work is performed by a small number of processes. There is a strong correlation between the granularity of the parallelism and the speedup, the parallel computations where the average lifetime of the processes is short (less than 20 inferences) providing a bad speedup. A better load-balancing strategy could increase the average granularity of the processes.

6 SIMULATION

The purpose of simulation was twofold: to verify the feasibility of implementing the PEPSys model on a multi-processor machine and to experiment with novel schemes for *efficient*, cost-effective implementations. A flexible simulation environment was considered to be of paramount importance and the results were correlated with those of the implementation project.

6.1 The Simulated Architecture

Several successful attempts have been made to implement OR-parallel computational models on limited-resource, shared memory machines (Chassin et al., 1988, Disz et al., 1987). A natural extension was to view such an architecture as a *cluster* in a multi-cluster machine. Figure 8 depicts the abstract view of this multi-cluster architecture. By adding more clusters, the number of PEs (Processing Elements) has been increased and the notion of non-equidistant communication paths between PEs introduced. When PE*i* of cluster *j* wishes to access a variable on PE*k* on cluster *l*, it induces two levels of communication: intra-cluster and inter-cluster. By executing several large branches of a program's search-tree on single clusters with a limited number of PEs and shared-memory, the locality of computation found in many Prolog programs is exploited while minimizing costly (remote) communication.

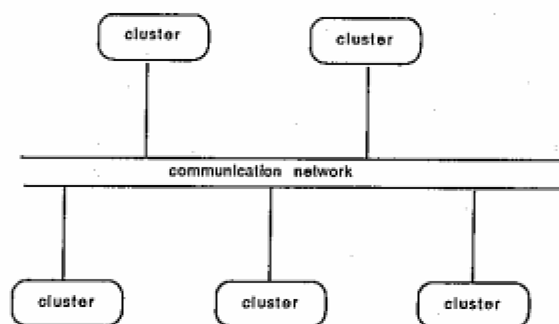

Figure 8: PEPSys Multi-Cluster Architecture.

The problem of communication between clusters was attacked on two levels: adding additional hardware and using sophisticated methods to reduce such communication. Each cluster was augmented with a Cluster Processor (CP) whose primary function is to handle *inter-cluster communication*. The novelty of this CP is that it has different levels of 'intelligence' corresponding to the requirements of a particular system. At the lowest level the CP is the hardware interface to the inter-cluster network with some buffering capabilities. A more 'intelligent' CP is a custom-made hardware unit, with several specialized concurrent sub-units, performing different functions concurrently. These functions are: servicing remote

dereferencing requests, managing local load-balancing, processing aborting, servicing PE requests and local bookkeeping. The CPs communicate with each other over a common bus via message passing.

The block diagram of a cluster processor (CP) is shown in Figure 9. The CP has four message queues managed by hardware as FIFOs. The size of these buffers is small as only a few messages are expected to be in a queue at any given time. Buffer overflow is handled by directing excess messages to the CP's private memory. The CP has a small private memory used for cluster bookkeeping, as a temporary scratchpad and for buffer overflow areas. It has a fairly large set of dedicated registers in addition to a set of general purpose registers. The dedicated registers are used for fast access to CP data tables and counters.

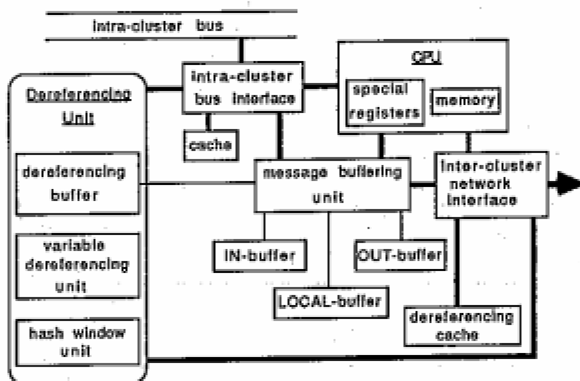

Figure 9: A Cluster Processor Block Diagram.

6.2 The Simulation System

Lisp was chosen as the implementation language for the PEPSys, event-driven simulation system. This provided a powerful programming environment, debugging facilities and allowed very fast code development. The complete system comprises about 1200 function definitions or about 17000 lines of code.

An event corresponds roughly to the execution of a single abstract-machine instruction by a single PE. More complex operations, e.g. deferencing, are broken up into several pseudo instructions.

The complete system also includes timing options, stepwise execution, checkpointing execution and limited run-time statistics gathering. Further analysis of trace information is provided by a comprehensive suite of tools, including graphics facilities.

6.3 Simulation Results

A multitude of configurations with the number of clusters and PEs ranging between one to ten were simulated. For a single cluster, architectures up to 30 PEs were simulated. *No* optimisations of any kind were included in these architectures, i.e. the CP was an ordinary processor running at the same speed as a PE, it had *no* parallel sub-units and management of its message buffers was done in software. The load-balancing scheme employed was simple and no restrictions were imposed on suspended PEs to request work from their (remote) children. None of the important optimisations to PE dereferencing or local work management were done.

The graph in Fig. 10 shows the speedup obtained for the Hamiltonian Path program using a maximum of 20 processors on several configurations. From these results it is clear that the overhead incurred by adding additional clusters while being tolerable, (cf. the speedup for 20 PEs on a single cluster is 13.9 and the speedup on 20 PEs on 2 clusters - 11.5), becomes considerable as the number of clusters increases. We are currently working on new strategies to reduce this overhead.

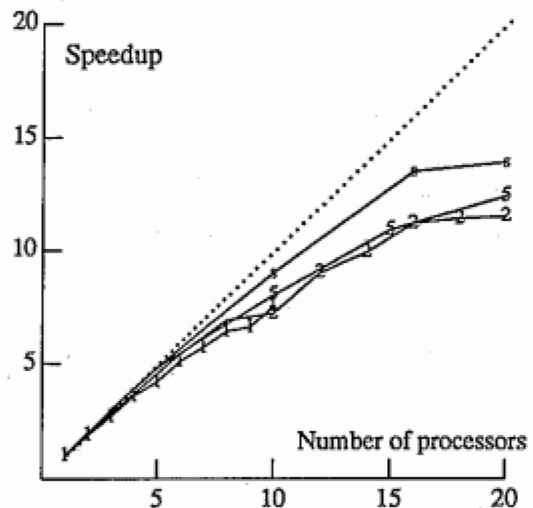

Figure 10: Performance of Hamilton Path program.

The three factors mentioned below account for the less-than optimal speedups:

- simple load-balancing
- no optimisations were performed - the CP should execute faster than PEs and should have special hardware.
- the test-programs must be large enough with respect to the amount of sustainable parallelism they exhibit

Due to the process-oriented nature of the PEPSys computational model, the inter-cluster bus does not cause a communication bottleneck. The results of monitoring the inter-cluster bus for a 100 PEs on 10 clusters shows an average bus utilisation time of well under 20% per time interval. The distribution of communication (in the form of message passing) between any two clusters in a configuration is almost uniform, excluding the cluster initiating the computation which always has a heavier load. By dividing the work at the highest possible level in the search tree, this extra overhead can be eliminated.

Comments: Several straightforward optimisations which must be made to the model have become apparent (process creation, derefencing) and our ideas of special hardware (CP, derefencing unit) are more lucid. Of prime importance is the intra- and inter- cluster load-balancing between PEs. Experiments with different scheduling strategies has already resulted in a notable increase in performance. More details on load-balancing can be found in (Baron et al., 1988).

7 DISCUSSION AND RELATED WORK

Efforts to harness logic programming for parallel processing are prolific. Several models similar to PEPSys have been defined: ANLWAM (Disz et al., 1987, Disz et al., 1987), the RAP-WAM (Hermenegildo and Nasr, 1986), the SRI model

(Warren, 1987), the Versions Vectors model (Ciepielewski et al., 1987) and the BOPLOG model (Tinker and Lindstrom, 1987) and implemented on multi-processor, shared memory machines. All these models except (Hermenegildo and Nasr, 1986) provide for OR-parallel execution and some of them (Brand, 1988) are currently being extended to allow AND-parallel execution and a combination of AND and OR parallelism. The initial design of the PEPSys model permits the deliberate combination of both OR-parallelism and Independent AND-parallelism with sequential backtracking. Thus in PEPSys more parallelism can be exploited in the execution of a broader class of Prolog programs. The price paid in providing such mechanisms in PEPSys is the complexity of the model. Special cases have to be taken into account for efficiency (e.g. the left and right hand side of an AND-parallel node executing on the same PE) making implementation difficult and debugging tedious. Furthermore, there is much room for optimisation in the original model: when an AND-parallel goal is deterministic or sequential (see (Chassin et al, 1988)), the general combination mechanisms need not be invoked and the implementation of such cases is reduced to almost the same level of difficulty as handling 'pure' OR or AND parallel goals.

Another factor contributing to the model's complexity is the management of variable bindings where deep bindings, done in hash-windows, are distinguished from shallow ones made in the normal stacks. Access to variable bindings in the dereferencing operations may be a complex operation, involving a search through a potentially unbounded chain of hash-windows. On the other hand, variable management in the SRI model is far simpler, where variable access, binding and unbinding are fast, constant-time operations. One of the SRI model's main attractions is its simplicity, both in control and variable binding management, but it and some of the other models mentioned above support OR-parallel execution only, which naturally simplifies matters somewhat. Moreover, task creation in the SRI model demands an expensive binding installation and untrailing operation, non-existent in PEPSys.

The scheduling of work amongst processors is crucial for efficiency. Results from both the simulation and implementation reinforce the need for a sophisticated scheduling policy enforcing locality while maintaining the desired process grain-size. However, the importance of a good scheduling policy is less in PEPSys than in the SRI model, where the overhead of task creation is not fixed, but depends on the "proximity" of the task in the computational tree.

While the ANLWAM, SRI and BOPLOG models are specialised for shared memory, multi-processor architectures and their implementations exploit this fact (these models use a global addressing scheme; they do not need to tag bindings with the owner process identity, as the physical address of a variable is used instead.), the PEPSys model does not rely on any particular memory system, and in this sense is more general. It is not clear how the above-mentioned models can be scaled up to execute on a many-processor, distributed architecture.

One of the explicit design goals of the SRI model was to make dereferencing a constant time operation. In PEPSys, hash-window chains can be of arbitrary length and therefore a cause for concern. While such concern is justifiable, the results from both implementation and simulation on shared-memory and non-shared memory architectures, show that the occurrence of long hash-window chains is infrequent and furthermore the overhead of hash-window dereferencing is minimal.

8 CONCLUSION AND FUTURE WORK

In this overview, the different aspects and results of the Parallel ECRC Prolog System have been presented. The initial objectives were to study all facets, and to provide an efficient and user friendly system at all possible levels. The first objective has been achieved by studying Prolog applications, defining a high level Prolog-like language incorporating mechanisms to express parallelism, and defining a computational model whose implementation retains the optimized WAM based execution. The actual implementation of PEPSys on the MX500 system has brought an invaluable experience in developing real scale applications, tools for debugging and evaluation. The simulation tools extended this experience to less conventional architectures. These programs were supported by other secondary tools such as a compiler, tracing and low level debugging facilities, and numerous tools to extract and exploit the results of experiments.

The second objective, has been a major concern during all design and development phases of PEPSys. The PEPSys language offers some desirable facilities while preserving the Prolog programming methodology where appropriate. That user controlled OR-parallelism and independent AND-parallelism fit very important application areas was confirmed by application evaluations. Having both OR and AND parallelism made the model more complex, and perhaps more difficult to implement efficiently than other more restricted approaches. Current results indicate that a non-optimal implementation provides effective speedups over the most efficient sequential Prolog implementations when executing large size problems.

Current work includes completing the current MX500 implementation and simulator as well as performing numerous experiments in order to evaluate and refine all the components of the PEPSys project. Future work will join the PEPSys project with its companion project in the Computer Architecture Group, the KCM Project, implementing a high speed Prolog/Lisp co-processor, in order to provide a high performance Multiprocessor Logic system.

ACKNOWLEDGMENTS

Other contributors to the PEPSys project are: Bounthara Ing, Donald Peterson, and Wolfgang Rapp. Bruno Poterie and Jacques Noyé contributed their knowledge of Prolog compilation techniques. ECRC's Technical Group has provided much valuable support throughout this project. The exchange of benchmark programs, graphical tools, evaluation results and technical information with the Gigalips Group (Manchester University, Argonne Labs, SICS, MCC) has been very fruitful.

References

[Baron et al., 1988]
U. Baron, B. Ing, M. Ratcliffe, P. Robert.
A Distributed Architecture for the PEPSys Parallel Logic Programming System.
In *Proceedings of the 1988 International Conference on Parallel Processing*. Chicago, August, 1988.

[Brand, 1988]
P. Brand, S. Haridi, D.H.D. Warren.
Andorra Prolog- The Language and Application in Distributed Simulation.
FGCS'88.
1988

[Chassin et al, 1988]
J. Chassin de Kergommeaux and P. Robert.
An Abstract Machine to implement efficiently OR-AND parallel Prolog.
To appear in Journal of Logic Programming.
1988

[Chassin et al., 1988]
J. Chassin, J.C. Syre, H. Westphal.
Implementation of a Parallel Prolog System on a Commercial Multiprocessor.
In *European Conference on Artificial Intelligence*, pages 278, 283. Munich, August, 1988.

[Ciepielewski et al., 1987]
A. Ciepielewski, B. Hausman, S. Haridi.
Or-parallel Prolog Made Efficient on Shared Memory Multiprocessors.
In *Symposium on Logic Programming*. San Francisco, August, 1987.

[Clark and Gregory, 1986]
K. Clark and S. Gregory.
PARLOG: Parallel Programming in Logic.
acm Transactions on Programming Languages and Systems 8(1):1-49, January, 1986.

[Disz et al., 1987]
T. Disz, E. Lusk, R. Overbeek.
Experiments with OR-parallel logic programs.
In *Proceedings*, pages 576-599. The International Conference on Logic Programming, Melbourne, Australia, May, 1987.

[Foster and Taylor, 1987]
Ian Foster and Stephen Taylor.
Flat Parlog: A Basis for Comparison.
Technical Report, Imperial College London and the Weizmann Institute, March, 1987.

[Giuliano et al., 1987]
Mark E. Giuliano, Madhur Kohli, Jack Minker, Deepak Sherlekar, Arcot Rajasekar.
Experiments with Parallel Logic Programming in PRISM.
Technical Report UMIACS-TR-87-36, CS-TR-1887, University of Maryland, July, 1987.

[Hermenegildo and Nasr, 1986]
M. Hermenegildo and R.I. Nasr.
Efficient management of backtracking in AND-parallelism.
In Ehud Shapiro (editor), *Proceedings*, pages 40-54. Third International Conference on Logic Programming, London, July, 1986.

[Ichiyoshi et al., 1987]
N. Ichiyoshi, T. Miyazaki, K. Taki.
A distributed implementation of Flat GHC on the Multi-PSI.
In *Proceedings, Melbourne, Australia*, pages 257-274. The International Conference on Logic Programming, May, 1987.

[Leblanc and Mellor-Crummey, 1987]
T. Leblanc, J. Mellor-Crummey.
Debugging Parallel Programs with Instant Replay.
IEEE Transactions on Computers C-36(4):471-481, April, 1987.

[Matsuda et al., 1987]
H. Matsuda, M. Kohata, T.Masuo, Y. Kaneda, S. Maekawa.
Implementing Parallel Prolog on Multiprocessor System PARK.
In M. Kitsuregawa (editor), *5th International Workshop on Data Base Machines.* , Karuizawa, Japan, October, 1987.

[Mierowski et al., 1985]
C. Mierowski, S. Taylor, E. Shapiro, J. Levy and M. Safra.
The design and implementation of Flat Concurrent Prolog.
Technical Report CS85-09, Weizmann Institute, Rehovot, 1985.

[Ratcliffe, 1988]
Michael Ratcliffe.
Two Extensions to the PEPSys Logic Programming Language.
Internal Technical Report pepsys/28, ECRC, June, 1988.

[Ratcliffe and Syre, 1987]
M. Ratcliffe, J.C. Syre.
The PEPSys Parallel Logic Programming Language.
In *Proceedings.* IJCAI, Milano, Italy, August, 1987.

[Shapiro, 1986]
Ehud Shapiro.
Concurrent Prolog: A Progress Report.
IEEE Computer 19(8):44-58, August, 1986.

[Syre and Westphal, 1985]
Jean Claude Syre and Harald Westphal.
A Review of Parallel Models for Prolog.
Technical Report CA-07, ECRC, June, 1985.

[Takeuchi and Furukawa, 1986]
Akikazu Takeuchi and Koichi Furukawa.
Parallel Logic Programming Languages.
In Ehud Shapiro (editor), *Proceedings*, pages 242-254. Third International Conference on Logic Programming, London, July, 1986.

[Tinker and Lindstrom, 1987]
P. Tinker, G. Lindstrom.
A performance oriented design for OR parallel logic programming.
In *Proceedings*, pages 601-615. The International Conference on Logic Programming, Melbourne, Australia, May, 1987.

[Ueda, 1985]
Kazunori Ueda.
Guarded Horn Clauses.
Technical Report TR-103, ICOT, June, 1985.

[Warren, 1983]
David H. D. Warren.
An abstract prolog instruction set.
Technical Report tn309, SRI, October, 1983.

[Warren, 1987]
D.H.D. Warren.
The SRI Model for Or-Parallel Execution of Prolog. Abstract Design and Implementation Issues.
In The IEEE Computer Society Press (editor), *Proceedings - 1987 Symposium on Logic Programming*, pages 46-53. IEEE, September, 1987.

[Westphal et al., 1987]
H. Westphal, P. Robert, J. Chassin, J.-C. Syre.
The PEPSys Model: Combining Backtracking, AND- and OR-parallelism.
In The IEEE Computer Society Press (editor), *Proceedings - 1987 Symposium on Logic Programming*, pages 436-448. IEEE, September, 1987.