# AN ABSTRACT MACHINE TO IMPLEMENT OR-AND PARALLEL PROLOG EFFICIENTLY*

J. CHASSIN DE KERGOMMEAUX AND P. ROBERT

▷ PEPSys (Parallel ECRC PROLOG System) is a research project started in 1984 in the Computer Architecture Group of the European Computer-Industry Research Centre (ECRC). Its general goals are to study and evaluate new and practicable solutions to the problems of parallel logic programming. The PEPSys Abstract Machine described in this paper was designed to allow an efficient implementation of the PEPSys computational model. Based on the WAM, it incorporates a number of novel features to support the management of the logical variable and the control of the search space of the PEPSys computational model. Both a parallel implementation on a multiprocessor and a simulation system of scalable multiprocessor architectures implement the PEPSys Abstract Machine and yield effective speedups in parallel computations. ◁

## 1. INTRODUCTION

PEPSys (Parallel ECRC PROLOG System) is a research project which started at the ECRC's Computer Architecture Group in 1984. Its objectives are to study and evaluate solutions to the problem of parallel logic programming. A language [7] and a computational model [12] were defined by mid 1986. At the beginning of 1987 an Abstract Machine tailored to this model, a compiler, and an emulator for its OR-parallel subset were designed and implemented. In the remainder of 1987 a simulator and an implementation of the PEPSys model, based on the PEPSys Abstract Machine presented here, were realized to validate the model. Current work aims at completing the simulator and the implementation, experimenting and optimizing the system with large-size PROLOG programs. In addition to the

*This paper was presented at the ICLP 88 Conference.

*Address correspondence to* J. Chassin de Kergommeaux, ECRC, Arabellastrasse 17, D-8000 München 81, West Germany.

Received December 1987; accepted May 1989.



©Elsevier Science Publishing Co., Inc., 1990
655 Avenue of the Americas, New York, NY 10010 0743-1066/90/$3.50

authors, all members of the PEPSys team have contributed to this work: Uri Baron, Bounthara Ing, Donald Peterson, Michael Ratcliffe, Jean-Claude Syre, and Harald Westphal.

The PEPSys project integrates language and model definitions. The language makes a distinction between sequential modules and parallel modules. In the sequential modules all the side effects of PROLOG are allowed, whereas they are not supported in the parallel ones. It is possible to call a parallel module from a sequential module through the predefined predicates *bagof*, *setof*, *oneof*. The language supported in the parallel modules is PROLOG with additional predicate property declarations. The property *solutions(one)* allows one to simulate the cut, not supported in PEPSys parallel modules because of its unclear semantics in parallel. Other declarations allow the programmer to indicate the sources of OR-parallelism to the compiler and the runtime system. In addition, the # operator indicates that two goals are independent (i.e. do not bind the same PROLOG variable) and can be executed in AND-parallel.

The computational model was designed to support efficient implementations of the language. It exploits OR-parallelism, independent AND-parallelism, and the combination of both. The main features of the computational model are a control of the search space to produce all solutions to the queries and a solution to the representation of the PROLOG logical variable in parallel. The basic execution mode in PEPSys is sequential, and thus the most efficient PROLOG implementation techniques may be applied. A process expresses potential parallelism which may be used by idle processors. The unexploited OR-parallelism is executed by backtracking. The unexploited AND-parallelism is executed in sequence. The solutions of AND-parallel branches are cross-produced as soon as they are computed. An original algorithm guarantees the completion of the combination of OR-AND-parallelism with sequential execution and backtracking.

Several OR-parallel processes may concurrently access and bind the same PROLOG variables. In PEPSys, a process shares the variable cells of its ancestor processes, which avoids extensive copying on process creation. Each binding is tagged with a counter containing the number of branch points created by the local process, called the OR *branch level* (OBL). All the bindings performed by a father process after the split of a son process are invalid for the son; to find out whether an ancestor process binding is valid, a process checks if the OBL of the binding is anterior to the split OBL. A hash window is associated with each process. It is used by the process to bind locally the nonlocal variables of its ancestors. The bindings in the hash windows are also stamped with the current OBL value of the local process. Hash windows are chained, and variable lookup may thus involve the exploration of chain of hash windows of ancestor processes (see Figure 1). The overhead induced by this exploration is assumed to be low because of the locality of variable references in PROLOG [9]; it is a counterpart of the cheap process creation in PEPSys. In case of a combination of OR- and AND-parallelism, processes are created to execute the cross-product of solutions of each branch. To allow these processes to explore the chains of hash windows of both the right and the left branch, a *join cell* is associated to each process executing a cross-product. This join cell contains a pointer to the hash window of the right process, a pointer to the hash window of the left process, and a pointer to their last common hash window.

The PEPSys model attempts to achieve efficiency through cooperation of several processors, each of them executing one part of the program with an efficiency close

to the best sequential PROLOG implementations. To achieve this, the PEPSys Abstract Machine has been based on an extended version of the Warren Abstract Machine [10], which is regarded today as the most efficient way of executing sequential PROLOG. The extensions were done at ECRC and are detailed in [6].

The remainder of this paper is divided in three sections: Section 2 gives an overview of the PEPSys Abstract Machine, Section 3 compares it with related work, Section 4 presents the management of multiple bindings, Section 5 presents the major extensions made to the control of the WAM in the PEPSys Abstract Machine, and Section 6 gives the first results that the simulation and the implementation of PEPSys have yielded.

## 2. OVERVIEW OF THE PEPSYS ABSTRACT MACHINE

The PEPSys Abstract Machine (PAM) is designed to be executed on each virtual or actual processor of the simulation or the implementation. Thus, each processor manages its own set of registers and data. In addition, efficient use of the multiprocessor system requires that multiprocessing facilities be provided for each abstract machine by a scheduler, allowing process creation, suspension, reactivation, and termination; execution of a portion of a program by one PAM creates potential sources of parallelism which may be used by the scheduler of an idle processor to create a new process. There are several ways to define a scheduler suitable for the PAM, but that falls beyond the scope of this paper.

The PAM incorporates all the usual features of sequential PROLOG abstract machines: tagged data objects; local, global, and trail stacks; environment and choice-point frames, allocated on the local stack; program; stacks; structure pointer; and argument registers. It includes also the instruction set operating on these data.

Compared to the sequential WAM, the PAM includes a number of novel features necessary to support the PEPSys computational model. The management of variable bindings of PEPSys implied the definition of *nonlocal* data objects, and the extension of all the *get*, *put*, and *unify* instructions dealing with variables, as well as the basic dereferencing and unification operations. For the same reason, the environment and structure pointer registers $E$ and $S$ of the PAM must be typed, the possible types being local and nonlocal. The immersion of the PEPSys control algorithm into the WAM framework yielded the definition of a large number of new control frames and instructions operating on these frames. It was found that every needed extension to the sequential control of the WAM mapped nicely onto one of the two control mechanisms of the WAM used for continuation and backtracking. Once the category of the needed extension had been determined, the extension itself was fairly straightforward.

## 3. RELATED WORK

Numerous research projects have based the implementation of their parallel logic programming model on an extension of the WAM. Among them, we can cite ANLWAM [2, 3], RAP-WAM [5], the SRI model [11], the version-vectors model [4], and the BOPLOG model [8]. Several of them [2, 11, 4] rely on a global address space and a special memory management which benefits from a shared memory system. For control issues, however, their extensions of the WAM should be similar to those proposed in the following.

The extension proposed by Hermenegildo [5] is quite different, mainly because he concentrates on restricted AND-parallelism, which does not raise the problem of variables being multiply bound.

## 4. MANAGEMENT OF VARIABLE BINDINGS

### *4.1. Management of the Variable Bindings in the PEPSys Model*

The PEPSys model [12] extends the classical three-stack-based implementation of PROLOG: *shallow bindings* are performed in the normal PROLOG stacks, while *deep bindings* are stored in *hash windows*. By shallow binding, we mean using the variable cell to store the value of the binding; deep binding means recording the binding value together with the *name* of the variable in a local data structure: the hash window. Any PEPSys process may access the stacks and hash windows of its ancestor processes. The OBL associated with each binding is used to distinguish any valid binding in the ancestor processes' stacks and hash windows. A binding performed by an ancestor process is valid for a branch of the computation tree if it was performed before the split from this process occurred. A process may bind an "ancestor's variable"; the binding is then performed in the process hash window and tagged, like any other binding, with the current OBL value of the local process (see Figure 1).

The PEPSys model makes an explicit distinction between the variables which are local and nonlocal to a process. As mentioned in [12], the PEPSys model does not rely on any particular multiprocessor architecture: contrary to [11, 2], it does not require a global address space or any particular memory allocation technique relying on shared memory, since nonlocal references include an explicit process identification used in hash window dereferencing.

### *4.2. Data Objects*

In addition to the data types used in sequential PROLOG implementations, new types are defined to distinguish explicitly the nonlocal bindings by defining an equivalent nonlocal tag for each of the usual sequential pointer types, namely: nonlocal free, nonlocal reference, nonlocal list, nonlocal structure. These nonlocal PEPSys objects also contain the identification of the process creator of the variable and the split OBL, i.e. the value of the OBL in this creator process when the computation of the current branch split (see Figure 1).

### *4.3. Binding Algorithm*

The sequential PROLOG binding operation is extended in PEPSys, to take into account nonlocal objects; these objects are *older* than any local one. When a free variable is bound to a nonlocal free variable, it becomes a nonlocal reference. A nonlocal free variable is bound in the process's hash window. When binding two nonlocal variables, they are both bound to a free variable, created locally on the global stack, to increase the locality of reference.

### *4.4. Dereferencing Algorithm*

The dereferencing algorithm implements the model defined in [12]. As in sequential PROLOG, local dereferencing follows the local chain of references. The local

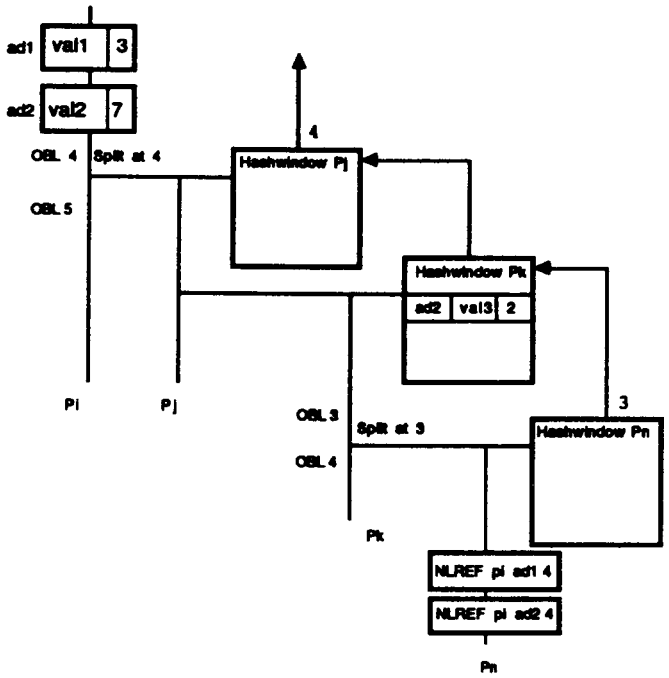



**FIGURE 1.** OBL and hash windows in PEPSys. The variable at address *ad1* was bound to *val1* when the OBL value on *Pi* was 3; the split eventually leading to process *Pn* occurred later (the OBL value on *Pi* was 4), and the shallow binding at address *ad1* is valid for process *Pn*. The variable at address *ad2* was not bound when the split occurred, and its shallow binding to *val2* 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*.

dereferencing operation stops when it reaches an object which is not a reference. The dereferencing of nonlocal variables also follows a chain of references; however, the validity of every binding encountered must be checked by comparing the OBL of the nonlocal reference against that of the actual binding.

If the dereferenced nonlocal variable is not validly bound in place, it may be validly bound in the process hash window or in hash windows of the process ancestors. A nonlocal variable is unbound only when all the relevant hash windows of the ancestors have been unsuccessfully explored. If a binding containing a local pointer is found valid by the nonlocal dereferencing operation, it must be transformed into its nonlocal counterpart before being returned (see Figure 2).

### *4.5. Extension of the Sequential Unification*

*4.5.1. General Unification.* As suggested in the preceding section, a free variable unifies with any nonlocal object; the same holds for a nonlocal variable with any PEPSys term. The unification of a nonlocal list (or structure) with a local or

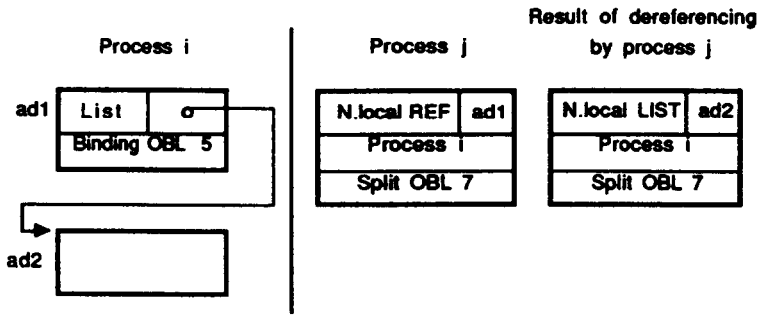
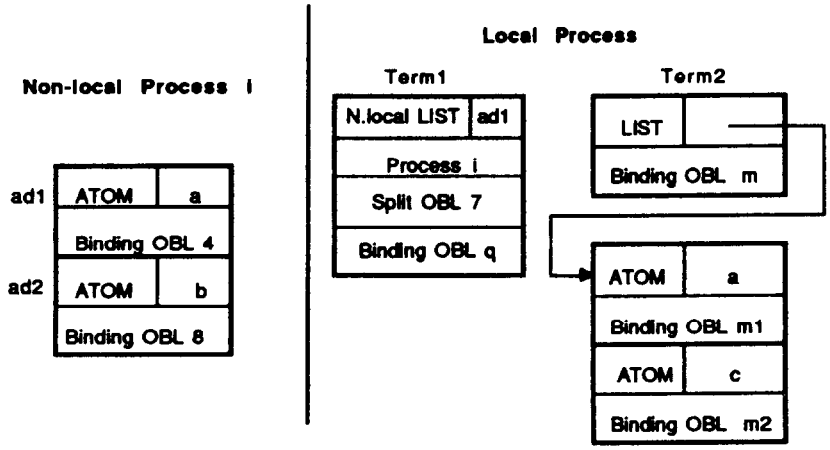



**FIGURE 2.** Nonlocal dereferencing. The nonlocal reference to *ad1* is validly bound for process *j* because the binding OBL (5) is less than the split OBL (7). Thus dereferencing of *ad1* on process *j* returns a *nonlocal list* with the same process identification *i* and split OBL (7) as the nonlocal reference.

nonlocal list (or structure) is more complex. Consider, for example, the unification of a nonlocal list and a list. As in the sequential unification, the heads and tails of the lists are recursively unified. However, the references to the head and the tail of the nonlocal list must be transformed into nonlocal references before being dereferenced; this transformation uses the process identification and the split OBL of the nonlocal list object (see Figure 3).

*4.5.2. Compiled Unification.* Unification is very often split into different instructions by the PROLOG compilers. Again, extensions to the sequential algorithms are

**FIGURE 3.** Unification of a nonlocal list to a list. The unification of *Term1* and *Term2* is performed in the local process. Before unification *Term1* is bound to [*a*|_] because the binding of the tail of the nonlocal list is not valid. The unification leads to the binding of *ad2* to the constant *c* in the hash window of the local process.



```
(C10)   parallel(Parameter, Result) :-
                create_parallelism(Parameter, Intermediate),
                use_parallelism(Intermediate, Result).
-properties([solutions(all), execution(eager),clauses(unordered)]).
(C11)   create_parallelism(Source, Object) :- ...
(C12)   create_parallelism(Source, Object) :- ...
```

**FIGURE 4.** Creation of nonlocal objects. There are two points where nonlocal references are created. The first is when a process is created to execute the clause *C12*: the arguments stored in the branch point, which are local references to the environment of the clause *C10*, are transformed into nonlocal references. The second is when the new process executes the continuation of *C12* (*use_parallelism*): the environment pointer *E* is nonlocal, allowing the *put* instructions to initialize the argument registers to nonlocal references.

needed in PEPSys, in cases involving nonlocal lists or nonlocal structures. Let us take the same example of lists unification. It is usually compiled by a *get_list* instruction which checks that the tag of the dereferenced argument is *LIST* and sets the structure pointer register *S* to the address of the head, followed by two *unify* instructions to unify the head and tail of the list, accessed through the structure pointer register *S*. In PEPSys the *get_list* instruction succeeds if dereferencing returns a nonlocal list object. In this case, the *S* register must be set to a nonlocal reference, so that the dereferencings of the head and tail of the list, in the following instructions, do the correct validity checks and transformations.

### *4.6. Creation of Nonlocal Objects*

Nonlocal objects are created when a process is started; the argument registers of the first goal to be executed by the new process have a pointer value. They are transformed into nonlocal objects. At the same time, the value of the environment register *E* must be made nonlocal. This transformation allows the actual environment of the father process to be shared by all the descendant processes. When accessed from the descendant processes, the permanent variables belonging to an environment created by an ancestor process will be transformed into nonlocal variables (see Figure 4). The same transformation is sometimes done when deallocating a nonlocal environment to guarantee that the restored environment register remains a nonlocal object.

## 5. CONTROL

### *5.1. Control in the Sequential Abstract Machine*

Unlike classical programming languages, functional or imperative (LISP or C), which provide one single high-level control mechanism—function call or subroutine call—PROLOG provides two interacting means of control: the continuation mechanism and the backtracking mechanism. The continuation mechanism is similar to

the function or subroutine call and can be statically determined at compile time, while the backtracking mechanism is dynamic and occurs on failure during unification.

*5.1.1. Continuation Mechanism.* The continuation mechanism uses two state registers—the environment pointer (register *E*) and the continuation instruction pointer (register *CP*)—and a data structure, the *environment*, which is located on the *local stack*. The environment is the activation record of a goal. It contains locations for variables and the values of the state registers *E* and *CP* corresponding to the parent goal. The environment register *E* points to the current environment, while the continuation pointer register *CP* points to the code to be executed on success of the current goal. The environment frames form an upward-linked chain.

*5.1.2. Backtracking Mechanism.* The sequential backtracking in the PEPSys abstract machine is similar to the continuation mechanism and thus slightly different from [10]. A backtrack pointer (register *B*), a backtrack instruction pointer (register *BP*), and a data structure, the *choice point*, are used. The *B* register points to the current choice point, and *BP* to the code to be executed in case of backtracking. The choice point, which is stored on the local stack, contains the information needed after a failure, to explore a new branch of the search tree.

## *5.2. Extension of the Control of the WAM in the PEPSys Abstract Machine*

The PEPSys Abstract Machine extends both the continuation and the backtrack control mechanisms of the WAM. The continuation is extended when an operation has to be executed after the success of a given goal. This is the case when the execution of the left goal of an AND-parallel conjunction succeeds. A synchronization operation with the processes executing the right branch is then necessary. This is also the case after the success of a one-solution predicate; in this case, a synchronization between the processes competing to provide the solution of the predicate is necessary so that only one of them proceeds.

In OR-parallel predicates, the backtracking is extended so that the process which created the branch point can synchronize with other processes also executing this predicate when it backtracks. Similarly, the backtracking control allows the mixing of the sequential and parallel execution of the members of a cross-product between two AND-parallel goals. The backtracking control is also used to synchronize a terminating process with its father process.

To execute an operation as a continuation, an instruction pushes a frame onto the local stack containing the parameters of the continuation operation. It inserts it into the environment chain and sets the continuation pointer (register *CP*) to the instruction executing the actual continuation operation. When executed, this instruction will access its parameters through the environment pointer (register *E*). Similarly, to execute an operation on backtracking, an instruction pushes a new frame onto the control stack containing the parameters used by the backtracking operation. It inserts it into the choice-point chain and sets the backtrack instruction pointer (register *BP*) to the instruction executing the effective backtracking operation. When this instruction is executed, it will access its parameters through the backtrack pointer (register *B*).

### 5.3. *Control of the* OR-*Parallelism*

*5.3.1. Instructions and Frames.* A process executing an OR-parallel predicate creates a branch point that may or not be exploited by idle processors. Unexploited work is executed by the sequential process, on backtracking. Some interprocess synchronization must then be provided to avoid executing the same clause twice or backtracking further and destroying shared data structures. New control instructions *par_try* and *par_retry* (see Figure 5) replace the classical sequence of *try*, *retry*, and *trust* of the WAM; these operate on a *branch_point* frame, which is a choice point extended with the data necessary for interprocess synchronization (see Figure 6).

The *par_try* instruction pushes a branch point (see Figure 6) onto the local stack. It makes it available to idle processors, and sets the backtrack instruction register to the following *par_retry* instruction. The creation of OR-parallel alternatives is restricted by the use of indexing instructions, as is the creation of choice points in the sequential WAM.

*5.3.2. Backtracking to a Branch Point.* A *par_retry* instruction is executed on backtracking by the process which created the branch point. This branch point is accessed in mutual exclusion, and, depending on the values of the numbers of started and finished clauses belonging to the OR-parallel predicate, several alternatives are possible:

*All the clauses have already started and all have finished.* The current process must go on backtracking, and the new backtrack pointers, the *B* and *BP* registers, are restored from the branch point.

*All the clauses have already started, but not all have finished.* Processes are still computing OR-parallel branches for this node of the search tree. It is not possible to backtrack further, because it would destroy data shared with these active processes. The current process suspends, and control is given back to the local scheduler, which will try to give work to the now idle processor in such a way that it can help those processes it is waiting for. The execution of the suspended process is resumed when the last child has terminated.

**FIGURE 5.** Compiling an OR-parallel predicate. The *test_or_par* instruction is generated to adapt the exploitation of parallelism to dynamic conditions. Depending on these conditions, sources of parallelism will be created (*par_try*) or not (*try*).

```
            PEPSys  Code                    Abstract    Machine  Code

-properties([solutions(all),        p/0:   test_or_par       s   eq_0
             clauses(unordered),           par_try
             execution(eager)]).           par_retry

        p :- c11.                   seq_0: try               c   11
        p :- c12.                          retry             c   12
        p :- c13.                          trust             c   13
                                     c11:  <code for clause 1>
                                     c12:  <code for clause 2>
                                     c13:  <code for clause 3>
```

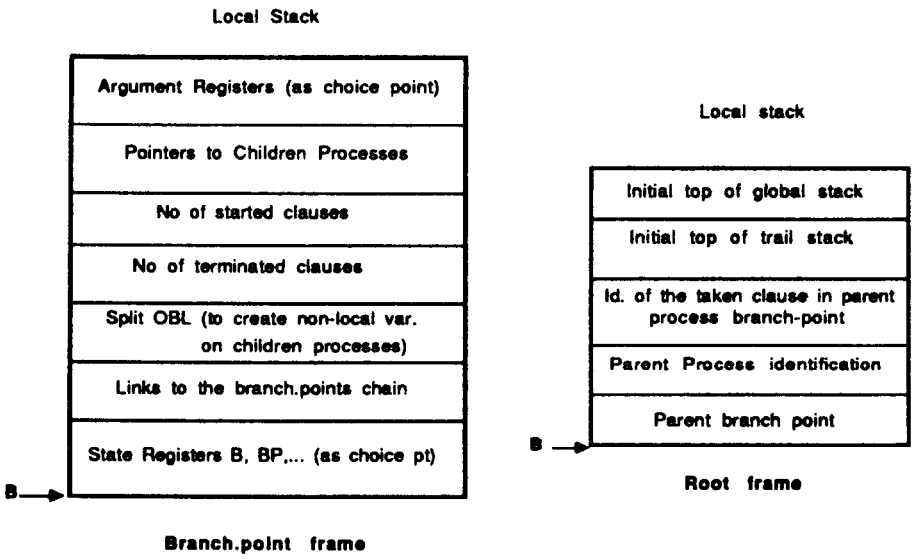



**FIGURE 6.** Control frames of the OR-parallel processes.

*All but one clause have been started, and all but one have finished.* The execution must proceed with the last remaining clause. The action taken is equivalent to the sequential *trust* instruction.

*Other cases.* The action taken is equivalent to the sequential *retry* instruction.

*5.3.3. The Birth and Death of OR-Parallel Processes.* As noted in the presentation of the language, calls to parallel PEPSys modules are encapsulated in one of the predefined predicates *bagof*, *setof*, and *oneof*. Thus each of the parallel processes is constrained to terminate by backtracking. The following actions are executed by a terminating OR process:

The parent process is informed of the termination of the process.

The scheduler of the processor is called to terminate the process.

This final backtracking of a process has been prepared at its creation. The backtrack instruction has been set to the *terminate_or_branch* instruction, and the backtrack pointer to a *rootframe*. This rootframe has been created at the first entry on the local stack, and linked to the chain of choice and branch points (see Figure 6).

## *5.4. AND-Parallelism*

*5.4.1. AND-Parallel Instructions and Frames.* The abstract machine implementation of the scheme presented in [12] for mixing independent AND-parallelism and OR-parallelism in their full generality is too complex to be presented at the same level of detail as the OR-parallelism. We will only show how a clause with AND-parallel goals is compiled (see Figure 7) and the flow of control among the instructions.

```
        PEPSys  Code                    Abstract  Machine  Code

h :-  ..., 1 # r, s,...                   h:
                                                .
                                                .
                                              fork @r
                                              call 1
                                              check
                                          @r: call r
                                              join
                                              call s
                                                .
```

**FIGURE 7.** Simplified code for an AND-parallel clause.

A process starting an AND-parallel conjunction executes a *fork* instruction which pushes a *fork point* frame onto the local stack and proceeds with the left-hand goal. The fork point is made available to the idle processors and allows them to compute the right-hand goal, while the original process computes the left-hand goal sequentially. The sequential process is synchronized, at the end of the left-hand goal, with any process executing the right-hand branch; this synchronization is executed as the continuation of the left-hand goal by the *check* instruction, which uses the fork point; this frame must thus be linked to the environment chain so that it is accessible when the continuation of the left-hand goal is executed. In case of failure of the left-hand goal, the fork frame must be removed from the list of work offered to the other processors, and the right branch must be killed, which is a form of semi intelligent backtracking. The fork point is linked to the choice- and branch-point chain, the backtrack instruction being set to the *terminate_left* instruction.

The purpose of the *check* instruction is to determine whether the right-hand branch is being computed. If it is, the current process suspends; otherwise, it starts the execution of the right-hand goal. In this case, the continuation instruction of the right-hand goal is set to the *join* instruction.

*5.4.2. Behavior of an AND-Parallel Process.* An AND-parallel process computes the right-hand goal of an AND-parallel pair of goals. The initialization of an AND-parallel process is similar to that of an OR-parallel process; the initializing process prepares its final backtracking by setting the backtrack register to the *terminate_right* instruction. It pushes a rootframe onto the local stack and links it to chain of choice and branch points. Similarly, on success of the right-hand goal, a *join* instruction has to be executed with the environment pointer referring to the nonlocal fork point; the environment pointer of the new process is thus initially set to the fork point.

The cross-product of the solutions of two AND-parallel goals is computed by the *join* instruction, which expresses potential OR-parallelism using an *extensible_branch_point* frame pushed onto the stack. The cross-product operation is similar to the pure OR parallelism: the combinations of a given right-hand solution with the left-hand solutions can be executed either by new OR-parallel processes created by idle processors, or by the process producing the right-hand solution by backtracking. The instruction *cross_product* is set as backtrack instruction by the join

instruction. It allows a right-hand process to execute one alternative of the cross-product on backtracking while synchronizing with the processes executing the other alternatives. It operates on an *extensible_branch_point* which is pushed on the stack by the join instruction and linked to the choice-point–branch-point chain. The cross-product instruction connects a pair of succeeding processes with a *join cell* (see [12]), and proceeds with the execution of the continuation of the AND-parallel conjunction of goals.

### 5.5. *One-Solution Predicates*

The PEPSys language [7] gives the programmer the possibility of declaring a predicate to be *one-solution*. A one-solution predicate means that the first—in time—parallel branch to solve the goal is allowed to proceed, while any other solutions that may be generated later are ignored.

During the execution of a one-solution predicate, any of the called goals may be OR-parallel. Then several parallel branches may succeed in solving the goal. Thus there is a need for a synchronization phase after each success. The synchronization is executed as the continuation of the one-solution predicate. A solution flag is associated with each one-solution predicate: after computing a solution to the predicate, each of the competing processes accesses this flag in mutual exclusion: the first sets the flag and proceeds; the other processes backtrack.

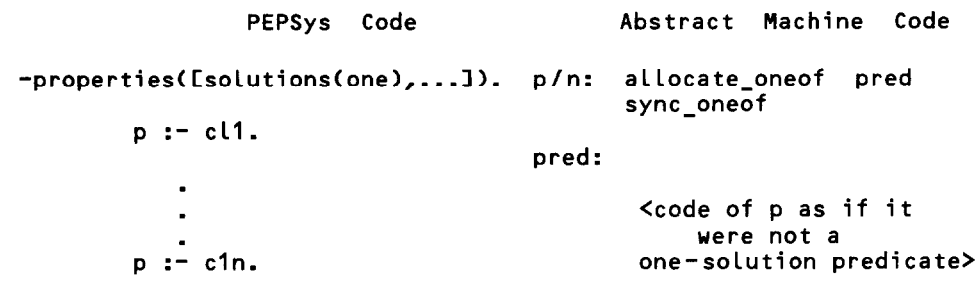
The control mechanism of the WAM is extended in the PAM to execute this synchronization action as the continuation of the one-solution predicates. This extension is provided by two instructions, *allocate_oneof* and *sync_oneof*, which operate on a *oneof-point* frame containing the synchronization flag (see Figure 8).

The *allocate_oneof* instruction pushes a *oneof-point* on the local stack, links it to the environment chain, sets the continuation instruction to *sync_oneof*, and proceeds with the multiple-solutions code of the goal. A process executing a *sync_oneof* tests the synchronization flag in the *oneof* point, and proceeds if the solution produced is the first; otherwise it backtracks.

The general algorithm makes inefficient use of the computing resources in the unsuccessful processes by forcing them to execute unnecessary backtracking. Three optimizations have been designed to minimize the useless computation:

*Backbranching.* The processes forced to backtrack follow the chain of active branch points (i.e. those used to create processes) posterior to the *oneof* point instead of executing the remaining alternatives of every choice point.

**FIGURE 8.** Compiling a one-solution predicate.



*Killing.* The successful process initiates the killing of the competing processes which do not participate in the successful solution.

*Sequential one-solution predicates.* If it is determined by the compiler that a one-solution predicate will be executed sequentially because it does not call any OR-parallel subgoal, the predicate is compiled as a sequential PROLOG predicate where cuts have been added at the end of each clause. This static analysis, which uses the PEPSys language declarations [7], has proven fairly effective.

## 6. VALIDATION AND PRELIMINARY RESULTS

The PAM is the basis of an implementation of PEPSys on a Siemens MX500 multiprocessor (equivalent to a Sequent Balance 8000 with 8 processors) and a simulator operating at the abstract instruction-set level. A PEPSys compiler, based on an existing compiler for sequential PROLOG (see [6]), is used to compile the PEPSys programs into the PAM instruction set. In both the implementation and the simulation, the instruction set is emulated.

The aims of the implementation are to show that an actual implementation of the model is possible and efficient and to experiment with large PEPSys programs. Performance figures of the implementation are very encouraging. Executing a PEPSys program on a single processor by the PEPSys emulator is about 20% slower than the execution of the equivalent PROLOG program by a sequential PROLOG emulator using equivalent implementation techniques (C language, no optimization). Effective speedup is obtained by running parallel programs on several processors, allowing the present implementation, although far from optimal (C emulator), to compete with the best sequential PROLOG implementations. Other measures confirm some hypotheses made in the definition of the model, such as the infrequent use of hash windows. Figure 9 and Table 1 shows some preliminary results obtained in the computation of four programs using only OR-parallelism, on the MX500:

*hamilton* solves the problem of finding 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.

*saltm* is the salt-and-mustard program described in [3]. The measures were made on the first version of the program. The original program was modified to remove the metacalls.

**TABLE 1.** Execution speed of PEPSys programs on the MX500. The results concerning CProlog and PEPSys refer to the NS32032 processor of the MX500 computer, while the results concerning Quintus have been measured on a Sun 3/140 workstation.

| | | Time (sec) | | | |
|---|---|---|---|---|---|
| Program | No. of inferences | CProlog | PEPSys(1) | PEPSys(7) | Quintus |
| *hamilton* | 493,644 | 722 | 285 | 44 | 21.1 |
| *mandel* | 74,232 | 162 | 43 | 7.4 | 13.2 |
| *saltm* | 22,838 | 17 | 7.5 | 1.8 | 0.5 |
| *houses* | 49,843 | 74 | 37 | 11.8 | 2.6 |

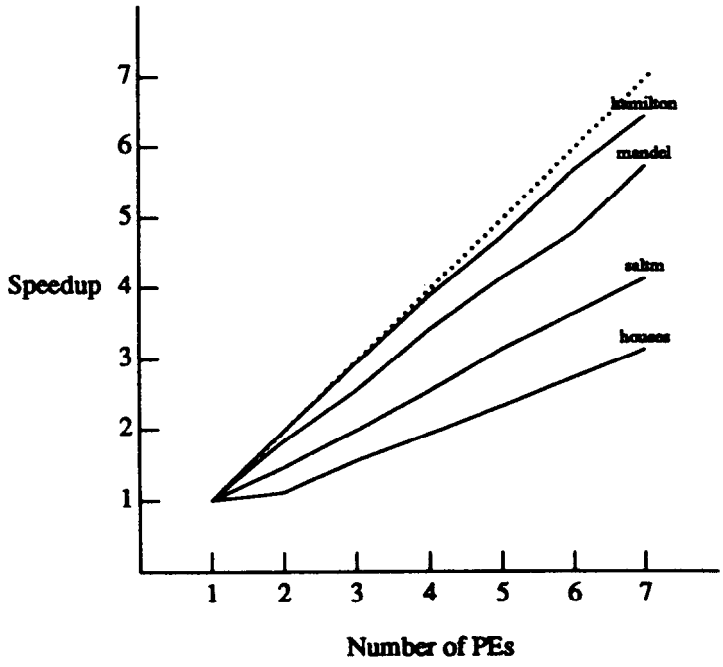



**FIGURE 9.** Speedups in the PEPSys implementation.

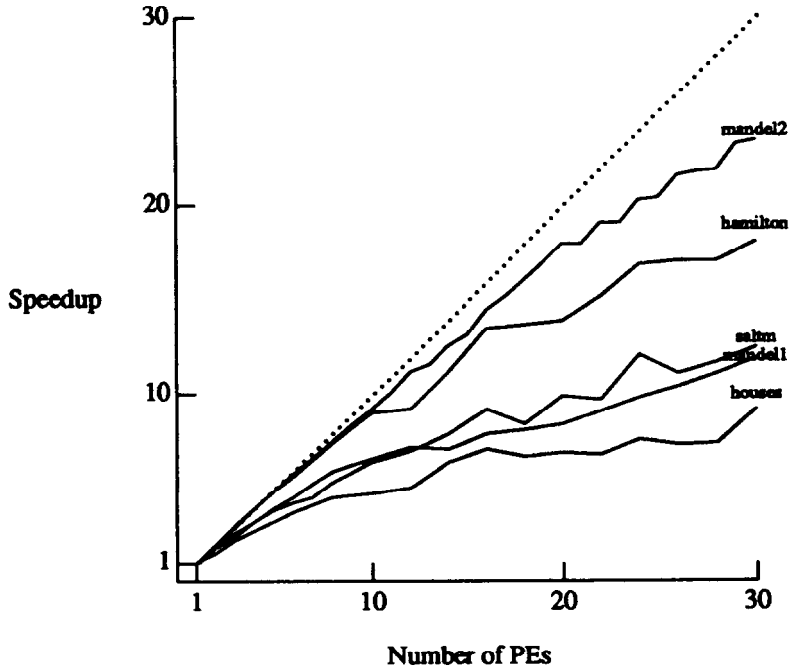



**FIGURE 10.** Speedup in PEPSys simulation of a single-cluster architecture. *mandel1* is the version of the *mandel* program used in the measures on the implementation. *mandel2* optimizes the locality of accesses to the variables at the level of the PROLOG program. The difference between the two results raises the problem of parallel-programming methodology.

*houses* gives the solution to a puzzle involving persons of different origins, living in different houses and each having a favorite pet, a favorite drink, and a favorite brand of cigarettes.

The simulator operates at the PAM instruction-set level [1]. It simulates multiprocessor architectures consisting of several interconnected clusters. Each cluster is a shared-memory, common-bus multiprocessor. Intercluster communication is performed by message passing and supported, on each cluster, by a specialized communication processor. The simulation aims at exploring the issues raised by the implementation of PEPSys on a nonglobal address space as well as estimating how the implementation results may scale up with a larger number of processors than provided by the MX500. Figure 10 shows the results obtained by the simulation on the same four programs. The speedups obtained with a small number of processors running on a single cluster are very close to the implementation results, thus validating the simulator. A number of other results have been obtained for numerous multicluster configurations, raising the problem of intercluster work allocation strategy.

## 7. CONCLUSION AND FUTURE WORK

The PEPSys Abstract Machine presented in this paper is suitable for the compilation of PEPSys programs and permits efficient implementation of the PEPSys model on a multiprocessor. The PEPSys Abstract Machine uses today's most efficient techniques for sequential PROLOG implementation. Parallelism is taken into account by extending the number of different data types, generalizing the basic dereferencing and unification operations, and extending the control management to allow interprocess synchronization.

The PEPSys Abstract Machine is the basis for the work to validate the PEPSys computational model, since it is used by an implementation of the model on a Siemens MX500 multiprocessor and by a simulation of the execution of the model on multiprocessors in a multicluster architecture. Both the implementation and the simulation have already yielded encouraging results. The performance of the PEPSys implementation running on a single processor is close to that of an efficient sequential PROLOG implementation. The MX500 implementation also achieves effective speedup through parallelism. The simulation results complement the implementation ones for larger-size multiprocessors with a more scalable architecture.

The experiments in progress have validated a number of assumptions made during the definition of the PEPSys language and model. In particular, the importance of having a good parallel-programming methodology enforced the choice made in the language definition to provide the user with declarations to indicate the best sources of parallelism. Another important assumption made in the definition of the model was also confirmed: most of the references are local to a process, and very few of them use the deep-binding mechanism in the hash windows. Of course the present results must be confirmed by more experiments on larger programs, but they are already very encouraging.

Future work in the PEPSys project includes the completion of the simulation and the implementation with the management of the full AND-parallelism. Much experimentation also remains to be done, especially on the programming methodology and the work allocation strategy. The latter appeared to be an issue for the

execution of several programs on the implantations; it is always an issue when simulating a multicluster architecture composed of a large number of processors.

The authors would like to thank the members of the PEPSys team for their valuable comments. The authors are indebted to J. Noyé and H. Benker for explaining the secrets of the WAM and their interesting suggestions. Also thanks to the members of the Gigalips project for their cooperation: D. Warren, A. Calderwood, S. Haridi, A. Ciepielewski, B. Hausman, and Ross Overbeek. Finally, thanks to the referees for their useful comments.

## REFERENCES

1. Baron, U., Ing, B., Ratcliffe, M., and Robert, P., A Distributed Architecture for the PEPSys Parallel Logic Programming System, in: *Proceedings of the International Conference on Parallel Processing '88*, Aug. 1988.
2. Butler, R., Lusk, E. L., Olson, R., and Overbeek, R. A., ANLWAM—A Parallel Implementation of the Warren Abstract Machine, Internal Report, Argonne National Lab., 1986.
3. Disz, T., Lusk, E., and Overbeek, R., Experiments with OR-parallel logic programs, in: *Proceedings of the 4th International Conference on Logic Programming*, Melbourne, May, 1987, pp. 576, 599.
4. Hausman, B., Ciepielewski, A., and Haridi, S., OR-parallel Prolog Made Efficient on Shared Memory Multiprocessor, in: *Proceedings of the 4th Symposium on Logic Programming*, Sept. 1987, pp. 69–79.
5. Hermenegildo, M., An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs, in: *Proceedings of the 3rd International Conference on Logic Programming*, London, July 1986, pp. 25–39.
6. Noyé, J., Syre, J.-C., et al., ICM3: Design and Evaluation of an Inference Crunching Machine, in: *5th International Workshop on Databse Machines*, Oct. 1987, pp. 1–14.
7. Ratcliffe, M. and Syre, J.-C., The PEPSys Parallel Logic Programming Language, in: *IJCAI*, Milano, Italy, Aug. 1987.
8. Tinker, P. and Lindstrom, G., A Performance Oriented Design for OR Parallel Logic Programming, in: *Proceedings of the 4th International Conference on Logic Programming*, Melbourne, May 1987, pp. 601–615.
9. Touati, H. and Despain, A., An Empirical Study of the Warren Abstract Machine, in: *Proceedings of the 4th Symposium on Logic Programming*, Sept. 1987, pp. 114–124.
10. Warren, D. H. D., An Abstract Prolog Instruction Set, Technical Report tn309, SRI, Oct. 1983.
11. Warren, D. H. D., The SRI Model for OR-Parallel Execution of Prolog, Abstract Design and Implementation Issues, in: *Proceedings of the 4th Symposium on Logic Programming*, Sept. 1987, pp. 46–53.
12. Westphal, H., Robert, P., Chassin, J., and Syre, J.-C., The PEPSys Model: Combining Backtracking, AND- and OR-parallelism, in: *Proceedings of the 4th Symposium on Logic Programming*, Sept. 1987, pp. 436–448.