1997

Exploiting And-Or Parallelism in Prolog: The OASys Computational Model and Abstract Architecture

I. P. Vlachavas

Report Number:
97-033

EXPLOITING AND-OR PARALLELISM IN PROLOG: THE OASYS COMPUTATIONAL MODEL AND ABSTRACT ARCHITECTURE

I. P. Vlachavas

CSD-TR 97-033
June 1997

Exploiting And-Or Parallelism in Prolog: The OASys Computational Model and Abstract Architecture

I. P. Vlachavas[1]

Department of Informatics, Aristotle University of Thessaloniki, 54006 Thessaloniki, Greece

E-mail: vlahavas@csd.auth.gr, URL: http://www.csd.auth.gr/~plk/

Abstract - Different forms of parallelism have been extensively investigated over the last few years in logic programs and a number of systems have been proposed. OASys is an experimental parallel Prolog system that exploits and-or-parallelism and comprises a computational model, a compiler, an abstract machine and an emulator. OASys computational model combines the two types of parallelism considering each alternative path as a totally independent computation which consists of a conjunction of determinate subgoals. It is based on distributed scheduling and supports recomputation of paths as well as stack copying. The system features modular design, high distribution and minimal inter-processor communication. This paper presents briefly the computational model and describes the abstract machine discussing data representation, memory organization, instruction set, operation and synchronization. Finally performance results obtained by the single-processor implementation and the multiple-processor emulation are discussed.

Keywords: *Prolog, Abstract Machine, And-Or-Parallelism, Stack Copying, Recomputation*

1. INTRODUCTION

As parallel processing technology has reached a stage where multiprocessors are being marketed as workstation machines, considerable effort has been devoted to the exploitation of parallelism in logic programs and the development of parallel Prolog systems for multiprocessor

---

[1]Currently on leave at: Department of Computer Sciences, Purdue University, West Lafayette, IN 47907

vlachava@cs.purdue.edu

architectures. The Prolog execution model is sequential and is oriented towards Von Neumann architectures. The computation proceeds by the successive applications of a resolution mechanism combined with a top down, left to right search strategy. The search space can be normally represented by an and-or tree, where and-nodes represent the body goals of a clause and or-nodes represent the alternative clauses matching a body goal.

The motivation for parallel evaluation of logic languages comes from attempting to exploit concurrently these and- and or-nodes of the search tree. The two main types of parallelism is or-parallelism and and-parallelism. The former is exploited when alternatives clauses matching a goal are evaluated in parallel while the later is exploited by solving the goals of a clause in parallel. The second type is more complicated because of inter-dependencies among the different goals. Recently there has been considerable interest in exploiting both and- and or-parallelism in logic programs.

OASys (Or/And System) is a different approach to implementing full and-or parallelism while retaining the full Prolog semantics. It efficiently implements or-parallelism by viewing the search space as many independent and deterministic computations which rarely need to communicate. In addition, this feature provides the ability of exploiting the deterministic paths of the proof in an and-parallel way.

Concerning the parallel Prolog implementations, most of the research has focused to shared memory machines, mainly due to the ease of the runtime environment implementation as well as the low communication cost. From the other hand, message passing machines are scalable and large scale multiprocessors are commercially available.

OASys aims towards an architecture in which the processing elements performing the or-parallel computation, possess their own address space while other simple processing units are assigned with and-parallel computation and share the same address space. Communication is limited only to scheduling operations, i.e. allocation of work to available processing elements. The later is accomplished in two different ways: recomputing the alternative path or copying the computation state.

This paper introduces the OASys computational model [Vlahavas et al., 1996] and describes the design of an abstract machine that realizes this model and allows efficient compiled parallel execution of logic programs on multiprocessor architectures. The abstract machine, is abstract in the sense that certain details of design are left to the implementation stage. One of the main objectives of this research was to develop a general system for execution of logic programs that runs on a variety of parallel machines.

In the rest of the paper, section 2 outlines the main approaches for parallel execution of logic programming. Section 3 presents briefly the proposed execution model. Section 4 describes the abstract architecture in terms of data areas, registers and instruction set. Section 5 illustrates some performance results while section 6 concludes the paper and outlines the directions for further research. Appendix contains an example of a compiled Prolog program.

2. RELATED WORK

A number of approaches have already been proposed for parallel execution of logic programming languages [Gupta, 1994], but the bulk of research has dealt with either or-parallelism or and-parallelism.

In principle, or-parallelism is easy to implement since different paths of the or-parallel tree are independent of each other. However, in practice, implementation of or-parallelism is difficult due to sharing of variables between paths. The main problem in implementing or-parallelism is the efficient representation of multiple environments for storing the multiple bindings of the same variable produced by different paths. A number of methods have been proposed for environment representation. We can distinguish three main approaches:

- Shared environment, where processors share the same address space while the binding method records the different bindings to an appropriately defined data structure [Ciepielewski et al., 1983, Warren, 1984, Lusk et al., 1990, Delgado-Rannauro et al., 1991].
- Environment copying, where processors are independent and have their own copy of the environment in which they are currently working on [Ali and Karlsson, 1992, Gupta et al., 1994].
- Recomputation, where processors view of the search space as totally independent computations without having to communicate with others at all [Clocksin, 1987, Araujo and Ruz, 1994, Mudambi and Schimpf, 1994, Gupta and Hermenegildo, 1993].

The main problem in implementing and-parallelism is the handling of common variables between subgoals. The main approaches to implement this kind of parallelism have been the following:

- Independent and-parallelism, where processors work in parallel only when the runtime bindings of the variables of two or more subgoals are such that are independent of one another [Hermenegildo and Green, 1990, Lin and Kumar, 1991, Pontelli et al., 1995].

- Dependent and-parallelism, where processors work in parallel until one of them accesses the common variable applying then a priority scheme [Shen, 1992] or work in a producer-consumer way, communicating through the common variables.

This later approach, called also Stream And-parallelism, is adopted by the Committed Choice Non-Deterministic languages [Clark and Gregory, 1986, Ueda, 1986] and exploits and-or-parallelism through explicit language syntax and semantics.

The exploitation of and-or-parallelism in a single framework, is difficult to implement due to the overheads introduced by both types of parallelism. In practice, it is more efficient to combine or- with Independent and-parallelism [Baron et al., 1988, Araujo and Ruz, 1994, Gupta et al., 1994].

Andorra-I [Warren, 1990] supports both dependent and-parallelism, by executing first the determinate goals in parallel, and or-parallelism, stemming from the non determinate goals. That is when there is no determinate computation available, the leftmost goal is reduced in an or-parallel way.

3. OASYS COMPUTATIONAL MODEL

OASys approach imposes a different and much simpler computational model aiming towards higher performance. It exploits both or-parallelism and dependent and-parallelism respectively considering the search space as independent or-parallel branches consisting of deterministic conjunctions. This, in contrast to Andorra model, allows computation to proceed freely without any need for performing determinacy checking and suspending any non-determinate goals.

The search space in Prolog, can be normally represented by an and-or tree, where and-nodes represent the body goals of a clause and or-nodes represent the alternative clauses matching a body goal. In OASys, the search space can be represented by a tree where the nodes, called U-nodes, denote the unifications between the calling predicates and the heads of the corresponding procedures. A link between two U-nodes indicates a sequence of procedure calls as it is specified by the program. Figure 1 shows a Prolog program and its corresponding search tree as it is represented in OASys .

In Prolog, a node in the execution tree has already produced the variable bindings through unification. In OASys, unifications in the U-nodes are in progress while new U-nodes are

continuously produced, i.e. the U-nodes of the same path are executed concurrently. This has the effect that a path containing U-nodes may be speculative, i.e. some of the U-nodes generated may not have existed in the corresponding Prolog execution.

OASys supports or-parallelism by searching simultaneously different paths of the search tree. It also supports and-parallelism by executing in parallel the U-nodes belonging to the same path. Execution of a path terminates either when all links are successful (a solution is found), or one link is unsuccessful (failure in conjunction), or unification in an U-node fails (mismatch). A link is called successful if the unifications performed in the two U-nodes produce consistent bindings, or unsuccessful otherwise.

4. THE OASys ABSTRACT MACHINE

OASys computational model adopts quite naturally to a hybrid multiprocessor in which parts of the address space are shared among subsets of processors, as for example in a system containing multiple shared-memory multiprocessors connected by a message passing local network (figure 2).

The OASys abstract machine is a distributed (local-memory) multiprocessor system, employing a group of Processing Elements (PEs) each one of which executes a different path of the search tree (or-parallelism). Every PE could be a natural host for a team of processors sharing a common address space executing in parallel the U-nodes belonging to the same path (and-parallelism).

OASys executes compiled code. The source Prolog program is compiled in two codes, namely the main code which represents the definitions of clauses and the Compiled Clauses Dependency (CCD) code produced by the abstract interpretation of the Prolog program. During the startup phase, both of them are distributed to all PE memories. The machine's main features are:

- Efficient and-or-parallel execution of Prolog.
- Modular design.
- Simple mechanism for binding shared variables in and-parallelism.
- Simple synchronization mechanism.
- High distribution, i.e. no shared resources.
- Limited inter-PE communication.

- Efficient decentralized scheduling.

In the following sections an overview of the PE design, the execution scheme and the various components of the abstract machine, are presented.

4.1 The OASys Processing Element Overview

The main principles of PE design are the use of multiple functional units that operate concurrently and the partitioning of memory to increase the memory bandwidth (figure 3). The units have specialized hardware that enables them to execute a particular set of tasks efficiently. In order to reduce execution time and balance the processor workload, a special attention has been given in the design of the indexing mechanism and the work-scheduling algorithm. These two operations take place concurrently with the execution of the U-nodes in a PE in a producer-consumer way, communicating through a common work-pool.

Each PE passes through three phases of operation: preprocessing, scheduling and execution. A PE starts generating choices by executing the CCD code (preprocessing phase) and passes them to the scheduling unit which decides whether available work could be shared with other PEs (scheduling phase). The PE's engine works concurrently by consuming the choices passed from the scheduling unit (execution phase). The sequence of choices are used as directives in order to construct the sequence of U-nodes which in turn are supplied to the engine and assigned to special processing units (And Processing Units or APUs) which work in parallel.

So each PE consists of five main parts: a preprocessor for indexing operations, a scheduler for scheduling and communication operations, an engine for the execution of U-nodes, nine memory modules and an interconnection network (figure 3). The units of a PE exchange data using the PE Communication bus while they communicate with the memory modules via an interconnection network. In the following, theses parts are described analytically.

4.1.1 The Preprocessing Unit

This unit executes the CCD code generating a sequence of choice points along a path which are then passed to the scheduling unit. The CCD code is an abstract interpretation of the Prolog program and maps each Prolog clause to its actual address in main code together with the index table entry for every body subgoal.

For each procedure in the program there exists an index table called Procedure Index Table (PIT). The entries of a PIT contain pointers to the clauses of a procedure as well as tags. A tag specifies the data type of the first argument of the head predicate of a clause (variable, functional term, list, atom, integer, real and nil).

A call to a predicate is represented by a PIT address together with the type of its first argument. Execution of the call is actually a search in the corresponding PIT to find a clause with a matching data type. If only one matches, the call is deterministic. Otherwise we have a choice point and one of the alternative addresses is sent to the engine whereas the fate of the others is decided by the scheduler.

Each Prolog clause: *Hi: - B1, B2, ..., Bn* is represented in the CCD code notation as:
C_i : T1(ArgType1), T2(ArgType2), ..., Tn(ArgTypeN), return.
where C_i is the address of the clause in the CCD code used to derive the actual address of the clause in the main code, T_j is the address of the index table of B_j, and $ArgType_j$ is the data type of the first argument of B_j. "Return", denotes the end of the clause.

The preprocessor's execution mechanism, initially compares the ArgType (tag) of the subgoal's first argument with the tag attached in PIT's location Ti and then gives access to the corresponding clause.

The Procedure Index Table and the CCD code reside in a distinct memory area (the CCD area) of every PE. The CCD code for the N queens program together with the corresponding PITs are given in the appendix.

4.1.2 The Scheduling Unit

This unit implements the scheduling strategy of the OASys and communicates with the other PEs. The different paths of the search tree may be distributed to idle PEs. This unit decides whether it will assign the alternative paths to other PEs (or-parallel execution) or keep them for its own engine (sequential execution).

Each scheduling unit maintains a table of the PEs with available work and can be found in one of the following four states:

i) It has available work (unexplored paths):

It informs all the other PEs and continues executing its current path.

ii) It is asked, by another PE, to give work:

- It is locked against other requests,

- it simulates failure until the nearest to the root alternative (this decreases the amount of information transmitted),
- it transmits the appropriate information,
- if there is no more available work, it informs the other PEs accordingly in order to update their tables,
- it unlocks and continues executing its current path.

iii) It is asked, by its corresponding Engine, to give work. This situation arises when the Engine completes the execution of its current path either successfully or not.

- It is locked against other requests,
- it backtracks to the most recent choice point, if any.
- if there is no more available work, it informs the other PEs accordingly in order to update their tables,
- it unlocks and continues executing its new path.

iv) Its work is exhausted:

It is declared as idle, consults its table and requests work from another PE (scheduling unit). If the other scheduling unit is locked, it requests from another one.

The current implementation supports two possible ways of work distribution which is determined at compile time:

- Send the links followed so far from the root to the node and force the other PE to recompute that specific path and continue execution with one alternative.
- Copy the computation state of that node and let the other PE to continue execution with one alternative. Each PE has an identical but independent logical address space so that stack portions can be copied without relocating any pointers.

Besides the above information, the scheduling unit sends in both cases a CCD code address associated to the specific node, to be used by the preprocessing unit as a starting address to produce the choices for the rest of the path.

4.1.3 The Engine

The engine consists of a work pool, a control unit and a number of special processing units (APUs) that operate in parallel sharing a common memory.

The work pool accumulates choices supplied by the scheduling unit. The control unit executes the machine instructions (main code), constructs U-nodes taking directives from the

work pool, and distributes them to the APUs. A U-node is represented by a tuple of the form [Gi, Ci] where Gi is a subgoal and Ci is the head of a clause which has the same name and arity with Gi.

The APUs receive the addresses of a subgoal and the head of the corresponding clause and perform the unification operation efficiently. Every APU is equipped with an argument prefetching unit (not shown in the figure) which fetches and buffers arguments from the memory while the corresponding APU unifies the previous arguments.

The instruction set is described later in this section while an example is given in the Appendix.

4.1.4 Memory Organization

Since Prolog execution is memory intensive, a high bandwidth access to memory is required. With parallel operation of multiple units this requirement is even greater. In order to achieve as much parallelism of the units operation as possible, the address space of each PE is partitioned into nine distinct segments. Each segment contains only one kind of objects and there is no sharing among them. The segments are mapped into nine separate memory modules that can be accessed in parallel. The memory modules of a particular PE are shown in figure 3.

The Program area contains the compiled Prolog program (main code) with a symbol table containing all constants, functors and predicate names. The Head and Goal areas store the arguments of the head and the body of the clauses respectively. The Environment stack is used for building environments. The Heap (or Copy stack) stores the structures created by unification. The Trail stack contains state information of goals still to be executed, i.e. information about the search tree. The Backtracking stack contains state information to be restored upon backtracking and the Reset stack contains the addresses of the variables that must be unbound during backtracking.

4.2 The OASys Processing Element Architecture

4.2.1 Data Types

OASys supports 16 data types of data arguments which result in a better indexing of clauses and reduces the unification time. They are grouped in four categories: variable, constant, functional term and list (Table 1).

Variable data types are used to represent Prolog variables and are distinguished in six kinds. Void variable (VVAR) is a variable with a single occurrence in the clause. Temporary variable is any variable that is not void and occurs only in the head of the clause or in a single goal in the body. TFVAR and TBVAR denote the first and a subsequent occurrence of a temporary variable respectively. Skelet variable (SVAR) is any variable that is not void or a temporary variable. During a procedure call a number of cells are allocated in the Environment stack which correspond to the skelet variables of the clause and are declared as free variables (FVARs). BVAR is a FVAR or TFVAR which, during unification, has been bound to another Prolog term.

Constant data types represent the Prolog constants, i.e. Atom, Integer, Real and Nil. Functional Term and List data types are used to represent the Prolog structures. They are distinguished in three categories: i) Ground (GFTERM, GLIST), if they contain no variables, ii) Source (SFTERM, SLIST), if they are non ground and reside in the Program area and iii) Copy (CFTERM, CLIST), if they are non ground and reside in the Copy stack (Heap) This distinction gives the possibility of different structure handling. The Ground structures are represented in a structure sharing manner while the rest in structure copying.

4.2.2 The Register Set

The set of PE registers can be divided, according to their use, in three groups. There are registers to control the computation flow, stack pointers to access the different memory areas and a set of registers to handle the process of unification and parameter passing in APUs (Table 2).

The Program Counter (PC) points to the next instruction to be executed in the program area. The TE, TT, TH, TB and TR registers, point to the top of the environment stack, trail stack, heap, backtracking stack and reset stack respectively. The stack pointers, except TB, and the CA and CE registers represent the machine state that must be saved into the backtracking stack in the case of a choice point.

The Call element address (CA) points to the beginning address of the goal arguments in the goal area. The Call environment (CE) and the Head environment (HE) point to the environments of the clauses to which the goal and the head belong.

The contents of these three registers together with the address of the head arguments of a clause (incorporated in the syntax of the UNIFY instruction) are sent to an APU (where are stored into local registers) to perform the unification of a goal and the head of the corresponding clause.

4.2.3 The Instruction Set

The instruction set is broadly based on the APIM [Vlahavas and Kefalas, 1993]. It includes only one unify instruction, it offers a simplified compilation process and allows the parallel operation of the PE's various units. The instruction set reflects specialized operations that perform clause control, indexing and construction of U-nodes. It also provides a mechanism to implement arithmetic and other built-in operations.

The OASys instruction set differs from other implementations in that is smallest and of higher level and mainly in that it allows the parallel execution of three distinct operations, namely the clause control, the indexing and the unify operation.

The table driven indexing scheme of clauses, facilitates the representation of the search tree allowing the sharing of paths among different PEs (or-parallelism), permits the parallel operation of the preprocessing unit with the other units of a PE and requires only two kinds of simple indexing instructions, one for the deterministic goals and one for the non deterministic ones.

The data flow design philosophy of the APUs, permits their efficient parallel operation (and-parallelism), leads to a simplified architecture design and requests a single unify instruction for the implementation of unification.

The instructions are grouped in four categories, i.e. procedural, indexing, the unify and miscellaneous instructions. They are listed, (except miscellaneous) in Table 3.

The procedural instructions deal with clause control and environment management. These are:

- *CRENV N* : This is the first instruction of every clause having N skelet variables. It allocates space in the environment stack for the skelet variables of the clause, updating the contents of the HE register..

- *PROCEED* : This instruction terminates a fact and transfers control to the next goal that remain to be executed. If the goal executed was the last one of the clause (instruction EXECL or TRYL), the rewind operation takes place (see REWIND instruction).
- *WIND* : This instruction terminates the head of every clause that is not a fact. It saves state information of the suspended goals into the Trail stack and updates the contents of CE register copying the contents of the HE register into it.
- *REWIND* : This instruction terminates the final goal in the body of a clause. It is used to retrieve the state (from the Trail stack) of the goal which is the continuation of the currently executed goal. It can be omitted if the previous instruction is the EXECL or the TRYL instruction.
- *FAIL* : This instruction corresponds to a goal "FAIL" in the body of a clause. It causes the termination of the path execution and signals back to the scheduling unit requesting new work.
- *ESCAPE N, Gi* : This instruction provides a mechanism to support operating system calls and built-in operations that cannot be realized with the existing OASys instruction set. These operations are implemented in a low level instruction set (of the host machine) and are invoked by the ESCAPE instruction via the address N of the corresponding routine. Gi is the address of the arguments to which the operation will take place.

The indexing instructions limit the amount of search required to solve a query for a given program implementing the indexing mechanism. In the description that follows, Ci and Gi denote a pointer into the Program area and Goal area. respectively.

- *EXEC Ci, Gi* : This instruction represents a deterministic goal. It sets the Program counter to point to the address (Ci) of a clause with head predicate name and first argument that match the goal's predicate name and it's first argument. Gi is the address of the goal's first argument.
- *EXECL Ci, Gi* : This instruction represents a deterministic goal which is the last one in the body of a clause. It's operation is identical to the above described with the difference that it affects the operation of the PROCEED instruction.
- *TRY Gi* : This instruction represents a non deterministic goal. It stores the address of his first argument (Gi) into the CA register, it fetches an address from the work pool and stores it into the Program Counter. This address points a clause with head predicate name and first argument that match the goal's predicate name and it's first argument. The instruction checks also the work pool for alternative addresses (this is actually an indication sent by the scheduling unit) and if exist, it saves the current machine state into the Backtracking stack.

- *TRYL Gi* : This instruction represents a non deterministic goal which is the last one in the body of a clause. It's operation is identical to the above described with the difference that it affects the operation of the PROCEED instruction.

The Unify instruction, "*UNIFY N, Hi*" activates an idle APU to perform the unify operation. It sends to the APU the number and the address of the head arguments (N and Hi), the address of the goal arguments (contents of the CA register) and the addresses of the environments of the clauses to which the goal and the head belong (contents of the CE and HE registers).

Finally, the miscellaneous instructions include arithmetic and logic operations, as well as instructions that control the I/O devices (input and output data).

An example of a compiled Prolog program is given in the Appendix.

4.2.4 Operation

During the startup phase the compiled Prolog program, i.e. the main code, the CCD code and the procedure index table, are distributed to all PEs. A PE starts operate after the activation of its scheduling unit. The flow of control between the three main units of a PE, i.e. the preprocessing unit, the scheduling unit and the engine, is depicted in figure 4, while the operation of each unit is presented in the following.

Scheduling Unit : This unit has a three fold operation. It acts as an interface between a given PE and the rest of the OASys, it schedules the available work and manages the other units of the PE. It accepts as input a node address and either computation state describing that node (copying approach) or a list of nodes representing the links followed so far from the root to that node (recomputation approach). It then activates the preprocessing unit by sending the node address and either sends the computation state to the engine (in case of copying) or keeps for itself the list of nodes (in case of recomputation). The scheduling unit maintains a stack of nodes (node stack) which contains the executed nodes and alternative nodes (choices) not yet executed.

Preprocessing Unit : This unit accepts as input a starting (node) address and starts executing the CCD code. It produces a sequence of choice points along a path and supplies them to the scheduling unit.

Engine : This unit is activated by the scheduling unit, accepting a sequence of choice points which are accumulated in a work pool. It consists of a control unit and a number of And Processing Units (APUs).

The Control Unit : This unit executes the machine instructions and generates the U-nodes. It is microprogrammed because of the complex nature of the instruction set and operates in parallel with the other units. It executes the machine instructions sequentially up to the point where a UNIFY instruction is encountered. Then, it constructs a U-node and sends that to an idle APU. The Control unit continues its operation executing the next instructions, up to the point that a UNIFY instruction is encountered again.

In case that the Control unit encounters an instruction corresponding to a particular built-in operation or an arithmetic/logic operation, it halts awaiting termination of all the APUs and then continues. In the case where a unify operation fails, the corresponding APU interrupts the Control unit operation causing the termination of the path execution. Upon failure or successful termination, the Control unit signals back to the Scheduling unit.

The APU : The APU is a hardware unit that performs the unification operation efficiently. It is equipped with an argument prefetching unit that fetches the call and head arguments from the memory while the APU unifies the previous arguments. Inputs to the APU are five terms (addresses and data) supplied from the UNIFY instruction.

The APU is microprogrammed and contains a jump table followed by a number of microroutines. The jump table is used to decode the tag values of two input terms generating the address of the unify-microroutine that is to perform the required operation. When an APU terminates its operation signals to the Control unit and is placed in a wait state until a new activation occurs from the Control unit.

Synchronization : The parallel execution of APUs requires synchronized access to unbound variables which may also be accessed by other APUs. This does not add considerable additional complexity since, according to the execution algorithm, unbound variables may be bound in any order, either left to right as in Prolog, or right to left. The mechanism used for the variable bindings is a lock-test-and-set operation. According to this, whenever an APU tries to bind a variable, initially locks the variable and then it tests if the variable is still unbound. If so the APU proceeds setting a value. Otherwise, the variable binding is retrieved and a consistency check takes place.

Heap writing operations are also synchronized by setting a lock to the starting address of the space available for writing. An APU cannot allocate space on the heap until an already heap-writing operation by another APU has been completed. However, access operations are free up to the locked address.

5. PERFORMANCE MEASUREMENTS

In order to test the actual behavior of the OASys computational model and measure its efficiency, an experimental system has been developed in ANSI C under UNIX, emulating the abstract machine instruction set and several example programs were executed. This software system comprises two versions, the single-processor version and the multiple-processor version. The former, is actually an implementation of a sequential version of OASys, including one PE with one APU without scheduling mechanism.

The later is an emulation of the proposed system that used to verify the feasibility of implementing the OASys model on a multiprocessor machine and to estimate its performance. To run a program on a given number of PEs each one of which comprises a preprocessing unit, a scheduling unit and a given number of APUs, the implementation creates the same number of UNIX processes. This type of implementation adapts quite natural to the proposed abstract architecture; it offers a simple way of units' representation, an accurate method of timings estimation and almost a straightforward port to a real parallel machine.

Table 4 compares the measured performance of the sequential version of OASys and the estimated performance of the parallel version comprising 1 PE and 1 APU, with other Prolog systems (Eclipse, C-Prolog and Andorra-I Prolog). All timings were made on a uniprocessor SUN (Sparc Classic) and the benchmarks considered are:

- nrev140: naive reverse of a list of 140 elements,
- merge200: mergesort of a list of 200 elements,
- map: the map colouring problem,
- zebra: who owns the zebra problem,
- queens8: the 8-queens problem,
- hamilton: the search for hamilton paths in a graph.

We see that the sequential version of OASys is comparable in speed to C-Prolog while it is, in some cases 6, times slower than Eclipse. The runtimes of Andorra-I were obtained without preprocessing for determinacy checking. The performance of sequential Andorra-I with preprocessing of programs [Yang et al., 1993] is actually much better and reaches that of Eclipse.

The performance is actually worst than that of some sequential products and this is because the objective was to demonstrate the correctness and investigate the feasibility of implementation of the proposed computational model instead of optimising its sequential performance. It is

strongly believed that current work in applying various optimisation techniques in the coding level concerning the operation of a single PE (sequential implementation) will result in improving the overall OASys performance thus achieving better actual timings of at least an order of magnitude compared to sequential Prolog implementations, as it is shown in the speedup tables.

In order to test how the system exploits or-parallelism, and-parallelism and the combination of both types, we ran the same programs varying either the number of APUs or the number of PEs or both of them.

The results presented in the following concern only the copying approach. This not affects the derived conclusions because the two approaches proved to be quite similar with a small number of PEs, giving a precedence to the copying approach as the number of PEs increases. Although the data transfer in recomputation is approximately 4-7 times less than in copying, the above results are justified by the fact that the transfer time in the copying approach is much less than the re-execution time in the recomputation approach.

Figure 5 shows the speedup obtained by exploiting or-parallelism when running the benchmark programs with 1 to 10 PEs, each PE containing only one APU. As expected the speedup is high, since the programs considered can exploit or-parallelism and increases linearly due to small overhead. In comparison, the same figure shows the speedup obtained by the Parallel Eclipse running on the same uniprocessor machine for the hamilton program increasing the number of PEs (workers in eclipse terminology).

Figure 6 shows the speedup obtained by exploiting and-parallelism when the number of the APUs is increased from 1 (sequential execution) to 10 in a single PE. As it seems, some of the programs (e.g. nrev140, merge200) present good speedup since they are purely deterministic while others (e.g. queens8, map) present low speedup as the number of APUs increases to 4 and then remains constant. This is because the search spaces of these programs have short paths and therefore they don't exploit much and-parallelism.

However for all programs we do not expect better performance in execution time if the number of APUs is greater than 10. This is due to the fact that the preprocessor's total operation time is only 8%-15% of the total execution time of an engine with 1 APU. As the number of APUs increases the above percentage increases and the times become roughly equal.

Figure 7 shows the behavior of the system exploiting and-or-parallelism. This is demonstrated executing the queens8 program varying both the number of PEs (from 1 to 10) and the number of APUs (from 1 to 6) in each PE. We can see that speedups obtained when exploiting

both and-or-parallelism are greater that the speedups obtained from either kind of parallelism alone.

6. CONCLUSIONS AND FUTURE WORK

An abstract machine for the and-or-parallel execution of logic programs which implements the OASys computational model was presented.

The computational model supports and-or-parallelism considering the execution as distinct alternative independent computations consisting of deterministic conjunctions. The OASys abstract machine is a distributed multiprocessor system, employing a group of Processing Elements (PEs) each one of which executes a different path of the search tree (or-parallelism). Every PE consists of a team of dedicated processors sharing a common address space executing in parallel the goals belonging to the same path (and-parallelism).

The machine executes compiled code and features modular design, easily expandable. It is highly distributed with limited inter-processor communication and efficient decentralized scheduling mechanism.

A prototype software system, comprising a compiler and an interpreter of the abstract machine's instruction set, has been developed and several example programs were executed and compared with other Prolog systems.

As OASys is a prototype system, there are many issues that need further exploration. Current work focuses on improving the scheduling mechanism to decide, at run time, to allocate work using one of the two methods depending on the estimated overhead. An increase of the performance is expected, since the overhead from communication in copying together with the overhead from re-execution time in recomputation will be further optimized.

In addition, current work includes the implementation of the parallel version of OASys in two phases. During the first phase (which has started) OASys will be implemented in a network of workstations while in the second phase, it will be implemented on a real parallel machine. This is relatively straightforward because of the already developed process based emulation. Since most of the overheads taken into account in the estimation of speedups have been overestimated, the final system is expected to have at least equivalent (or better) performance.

Acknowledgments

I would like to thank D. Xochellis, D. Benis and C. Berberidis for their help in the prototype and the collection of performance data. Many thanks to Rong Yang for the help with Andorra-I. I also thank the anonymous referees for the reviewing of the manuscript and their comments and suggestions which improved this work.

REFERENCES

Ali, K. and Karlsson, R., Scheduling Speculative Work in MUSE and Performance Results, Int. J. of Parallel Programming, 21 (6), 449-476 (1992).

Araujo, L. and Ruz, J.J., PDP: Prolog Distributed Processor for Independent AND/OR Parallel Execution of Prolog, Proceedings of the 11th ICLP (P.Van Hentenryk ed.), MIT Press (1994).

Baron, U., Kergommeaux, J.C., Hailperin, M., Ratcliffe, M., Robert, P., Syre, J. and Westphal, H., The parallel ECRC Prolog system PEPSys: An overview and evaluation results, Future Generation Computer Systems '88 Conference, 841-849, Tokyo (1988).

Ciepielewski, A. and Haridi, S., A formal model for OR-Parallel Execution of Logic Programs, Proc. in Format Processing, IFIP , 299-305 (1983).

Clark, K.L. and Gregory, S., Parlog: a parallel programming in logic, ACM Transactions for Languages and Systems, 8 (1), 1-49 (1986).

Clocksin, W.F., Principles of the DelPhi parallel inference machine, The Computer Journal, 30 (5), 386-392 (1987).

Delgado-Rannauro, Sergio, Dorochevsky, M, Schuerman, K, Veron, A and Xu, J, A Shared Environment Parallel Logic Programming System on Distributed Memory Architectures, Proc. Of 2nd European Distributed Memory Computing Conference, Munich, April 91.

Gupta, G. and Hermenegildo, M., And-Or Parallel Prolog: A Recomputation Based Approach, New Generation Computing, 11, 297-321 (1993).

Gupta, G., Hermenegildo, M., Pontelli, E., Costa, V.S., ACE: And/Or-Parallel Copying-Based Execution of Logic Programs, Proc. of the 11th ICLP (P.Van Hentenryk ed.), MIT Press (1994).

Gupta, G., Multiprocessor Execution of Logic Programs, Kluwer Academic Publishers, 1994.

Hermenegildo, M. and Green, K., &-Prolog and Its Performance: Exploiting Independent And-Parallelism. Intern, Conference on Logic Programming ICLP 90, MIT press, 253 - 268 (1990).

Lin, Y. and Kumar, V., AND parallel execution of Logic Programs on a Shared- Memory Multiprocessor, The J. of Logic programming, 10, 155 - 178 (1991).

Lusk, E., Butler, R., Disz, T., Olson, R., Overbeek, R., Stevens, R., Warren, D.H.D., Calderwood, A., Szeredi, P., Haridi, S., Brand, P., Carlsson, M., Ciepielewski, A. and Hausman, B., The Aurora OR-parallel Prolog system, New Generation Computing, 7 (2,3) 243-271 (1990).

Mudambi, S. and Schimpf, J., Parallel CLP on Heterogeneous Networks, Proceedings of the 11th ICLP (P.Van Hentenryk ed.), MIT Press (1994).

Pontelli, E., Gupta, G. and Hermenegildo, M., &ACE: A High-Performance Parallel Prolog System, In Proc. IPPS'95, IEEE Computer Society (1995).

Shen, K., Exploiting Dependent And-Parallelism in Prolog: the Dynamic Dependent And-parallel Scheme (DDAS), Proceedings of the 9th JICSLP (Krzysztof Apt ed.), MIT Press (1992).

Ueda, K., Introduction to Guarded Horn Clauses. ICOT Research Center, TR-209, Tokyo (1986).

Vlahavas, I. and Kefalas, P., The AND/OR Parallel Prolog Machine APIM: Execution Model and Abstract Design, The Journal of Programming Languages 1, 245-261 (1993).

Vlahavas, I., Kefalas, P. and Halatsis, C., OASys: An AND/OR Parallel Logic Programming System, submitted for publication, available as a Technical Report (1996).

Warren, D.H.D., The extended Andorra model with implicit control. International Conference on Logic Programming '90, Workshop on Parallel Logic Programming, Israel (1990).

Warren, D.S., Efficient Prolog Memory Management for Flexible Control Strategy, Int. Symp. on Logic Programming, 198-202 (1984).

Yang, R., Beaumont, T., Santos Costa, V. and Warren, D.H.D., Performance of the Compiler-based Andorra-I System, Proc. of the 10th Intern. Conference in Logic Programming (D.S. Warren ed.), MIT Press, 150-166 (1993).

Appendix

Consider the following Prolog program for N queens:

```
queens([],R,R).                                          % C1
queens([H|T],R,P):-                                      % C2
        delete([A,[H|T],L),                                  % C2.1
        safe(R,A,1),                                         % C2.2
        queens(L,[A|R],P).                                   % C2.3

delete(X,[X|Y],Y).                                       % C3
delete(X,[Y|Z],[Y|W]):-                                  % C4
        delete(X,Z,W).                                       % C4.1

safe([],_,_).                                            % C5
safe([H|T],U,D):-                                        % C6
        X1 is H-U,                                           % C6.1
        X1 =\=D,                                             % C6.2
        X2 is U-H,                                           % C6.3
        X2 =\=D,                                             % C6.4
        D1 is D+1,                                           % C6.5
        safe(T,U,D1).                                        % C6.6
```

Given the query for 4 queens:

```
?- queens([1,2,3,4],K,M).                                % Query
```

the CCD code and the Procedure Index Tables follow while the main code is listed in the next page:

T1:	nil	C1
{queens}	list	C2

T2:	var	C3
{delete}	var	C4

T3:	nil	C5
{safe}	list	C6

```
% CCD Code
C1 :  RETURN
C2 :  CALL    T2, var
      CALL    T3, var
      LCALL   T1, var
C3 :  RETURN
C4 :  LCALL   T2, var
C5 :  RETURN
C6 :  LCALL   T3, var
Q  :  CALL    T1, list
      RETURN
```

```
% Main Code

C1:  UNIFY   3, H1      % Head of clause C1:
     PROCEED            %  queens([],R,R).

H1:  NIL                % Head arguments of C1
     TFVAR   1
     TBVAR   1

C2:  CRENV   6          % Head of clause C2:
     UNIFY   3, H2      % queens([HIT],R,P):-
     WIND
     TRY     G1         % C2.1: delete([A,[HIT],L),
     TRY     G2         % C2.2: safe(R,A,1),
     TRYL    G3         % C2.3: queens(L,[AIR],P).

H2:  SLIST              % Head arguments of C2
     SVAR    1
     SVAR    2
     SVAR    3
     SVAR    4

G1:  SVAR    5          % Goal arguments of C2.1
     SLIST
     SVAR    1
     SVAR    2
     SVAR    6

G2:  SVAR    3          % Goal arguments of C2.2
     SVAR    5
     INTEGER 1

G3:  SVAR    6          % Goal arguments of C2.3
     SLIST
     SVAR    5
     SVAR    3
     SVAR    4

C3:  UNIFY   3, H3      % Head of clause C3:
     PROCEED            % delete(X,[XIY],Y).

H3:  TFVAR   1          % Head arguments of C3
     SLIST
     TBVAR   1
     TFVAR   2
     TBVAR   2

C4:  CRENV   3          % Head of clause C4:
     UNIFY   3, H4      % delete(X,[YIZ],[YIW]):-
     WIND
     TRYL    G4         % C4.1: delete(X,Z,W).

H4:  SVAR    1          % Head arguments of C4
     SLIST
     TFVAR   1
     SVAR    2
     SLIST
     TBVAR   1
     SVAR    3

G4:  SVAR    1          % Goal arguments of C4.1
     SVAR    2
     SVAR    3

C5:  UNIFY   3, H5      % Head of clause C5:
     PROCEED            % safe([],_,_).

H5:  NIL                % Head arguments of C5
     VVAR
     VVAR

C6:  CRENV   7          % Head of clause C6:
     UNIFY   3, H6      %  safe([HIT],U,D):-
     WIND
     ESCAPE  4, G5      % C6.1: X1 is H-U
     ESCAPE  5, G6      % C6.2: X1 =/= D
     ESCAPE  4, G7      % C6.3: X2 is U-H
     ESCAPE  5, G8      % C6.4: X2 =/= D
     ESCAPE  3, G9      % C6.5: D1 is D+1
     TRYL    G10        % C6.6: safe(T,U,D1).

H6:  SLIST              % Head argum. of C6
     SVAR    1
     SVAR    2
     SVAR    3
     SVAR    4
G5:  SVAR    1          % Goal arguments of C6.1
     SVAR    3
     SVAR    5
G6:  SVAR    5          % Goal arguments of C6.2
     SVAR    4
G7:  SVAR    3          % Goal arguments of C6.3
     SVAR    1
     SVAR    6
G8:  SVAR    6          % Goal arguments of C6.4
     SVAR    4
G9:  SVAR    4          % Goal arguments of C6.5
     INTEGER 1
     SVAR    7
G10: SVAR    2          % Goal arguments of C6.6
     SVAR    3
     SVAR    7

Q:   WIND               % Code of Query:
     EXECL C2, G11      % ?- queens([1,2,3,4],K,M).
     ESCAPE 0, 0

G11: GLIST              % Goal arguments of Query
     INTEGER 1
     GLIST
     INTEGER 2
     GLIST
     INTEGER 3
     GLIST
     INTEGER 4
     NIL
     NIL
     VVAR
```

Figure and Table Captions

Figure 1: Representation of the search tree in OASys

Figure 2: Overview of the OASys architecture

Figure 3: Configuration of the Processing Element (PE)

Figure 4 : Control flow in a PE

Figure 5: Or-parallel speedups

Figure 6: And-parallel speedups

Figure 7: And-or-parallel speedups

Table 1: Data types

Table 2: The PE register set

Table 3: The OASys instruction set

Table 4: Runtimes (in seconds) for OASys and other Prolog systems

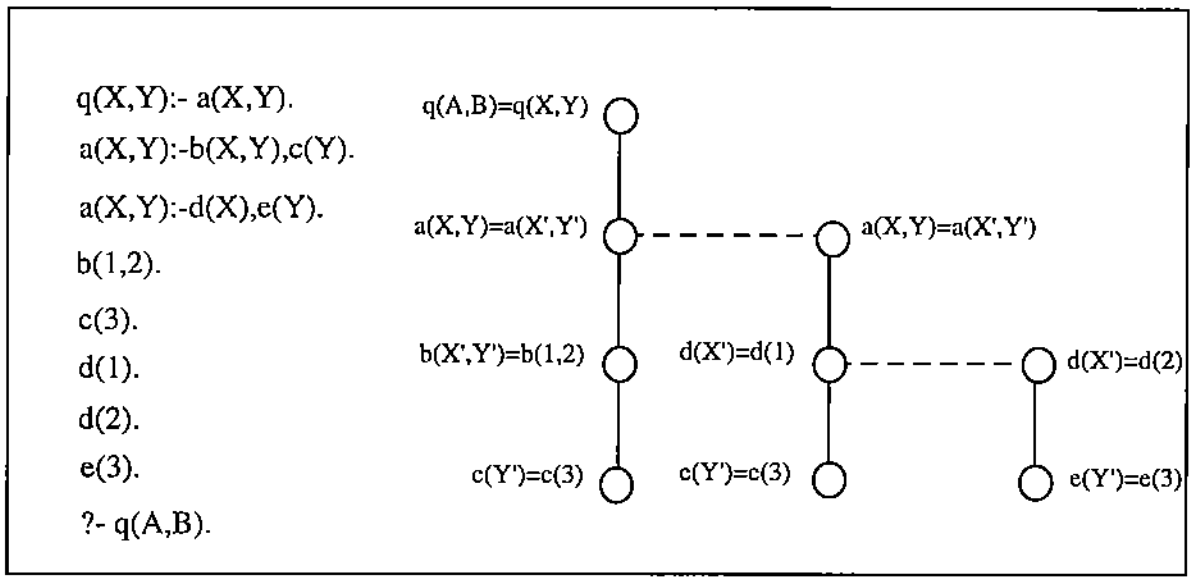

q(X,Y):- a(X,Y).
a(X,Y):-b(X,Y),c(Y).
a(X,Y):-d(X),e(Y).
b(1,2).
c(3).
d(1).
d(2).
e(3).
?- q(A,B).
q(A,B)=q(X,Y)
a(X,Y)=a(X',Y')
a(X,Y)=a(X',Y')
b(X',Y')=b(1,2)
d(X')=d(1)
d(X')=d(2)
c(Y')=c(3)
c(Y')=c(3)
e(Y')=e(3)

Figure 1

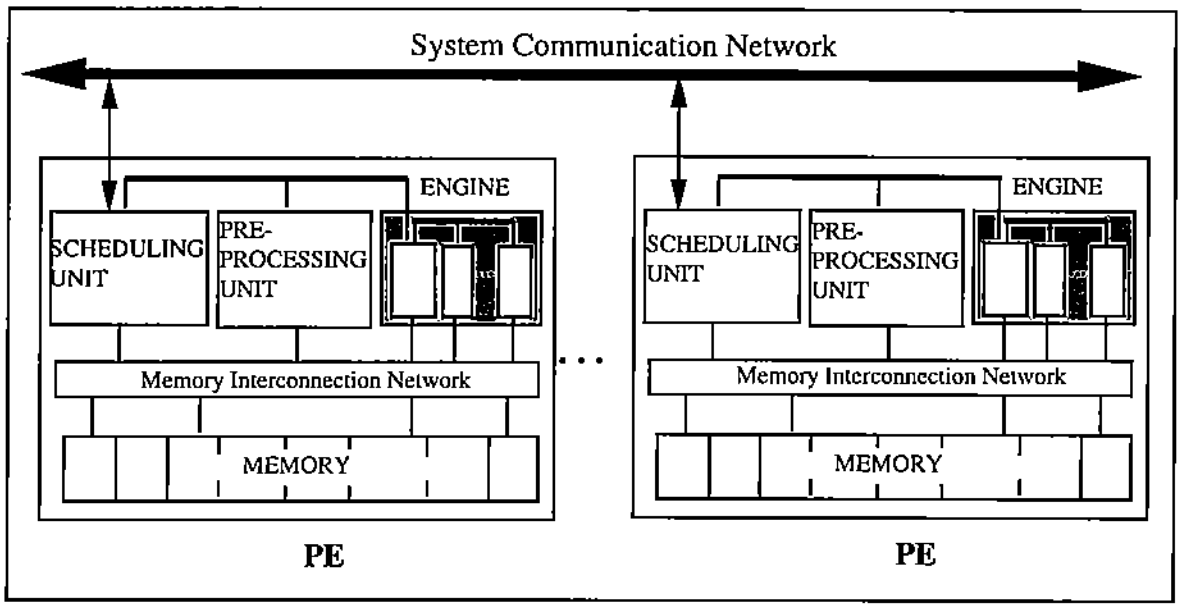

System Communication Network
ENGINE
SCHEDULING UNIT
PRE-PROCESSING UNIT
Memory Interconnection Network
MEMORY
PE
ENGINE
SCHEDULING UNIT
PRE-PROCESSING UNIT
Memory Interconnection Network
MEMORY
PE

Figure 2

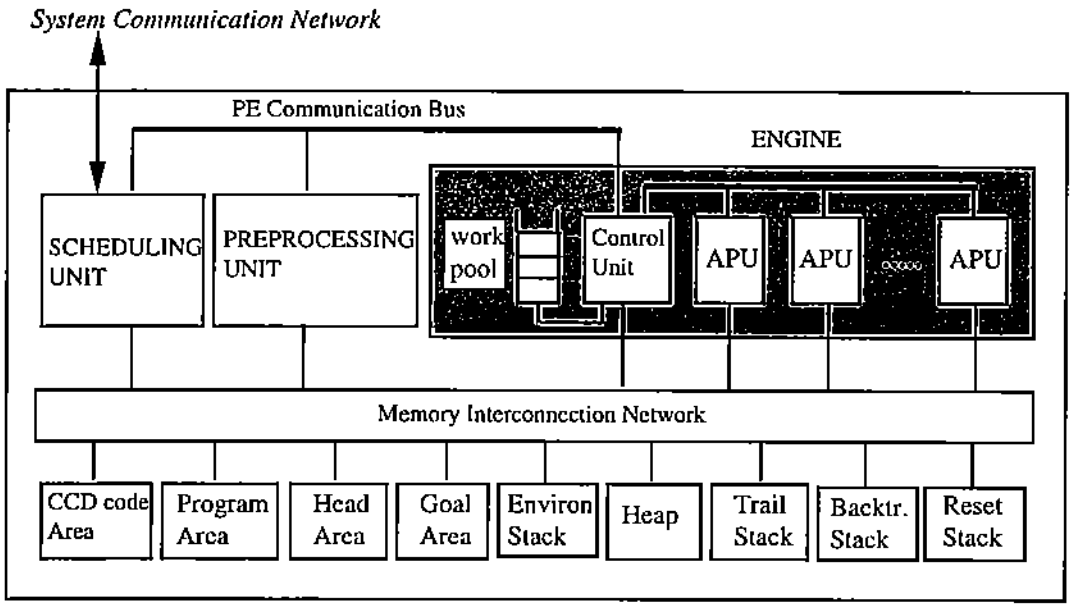
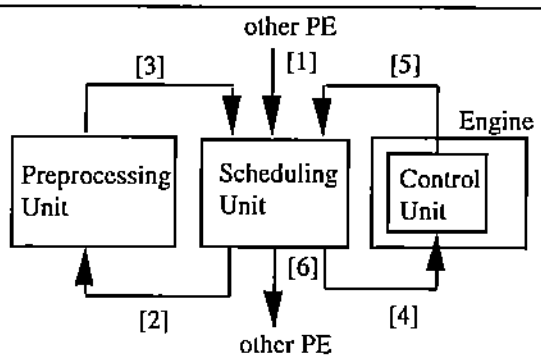

System Communication Network
PE Communication Bus
ENGINE
SCHEDULING UNIT
PREPROCESSING UNIT
work pool
Control Unit
APU
APU
APU
Memory Interconnection Network
CCD code Area
Program Area
Head Area
Goal Area
Environ Stack
Heap
Trail Stack
Backtr. Stack
Reset Stack

Figure 3



[1, 6] : Node address and Computation state or List of nodes

[2] : Node Address

[3] : Choices

[4] : Choices or Computation state

[5] : Computation State

Figure 4

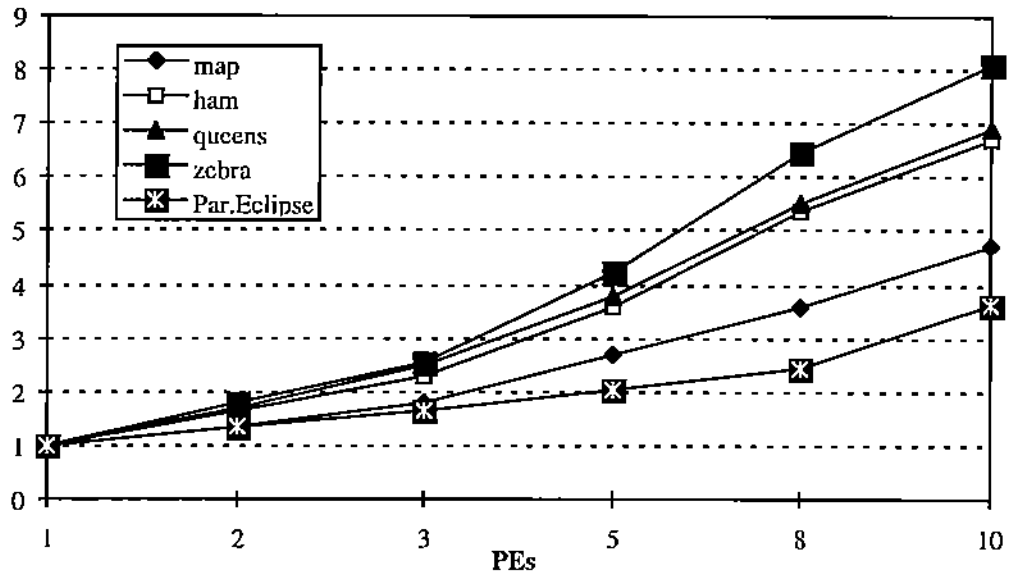


Figure 5

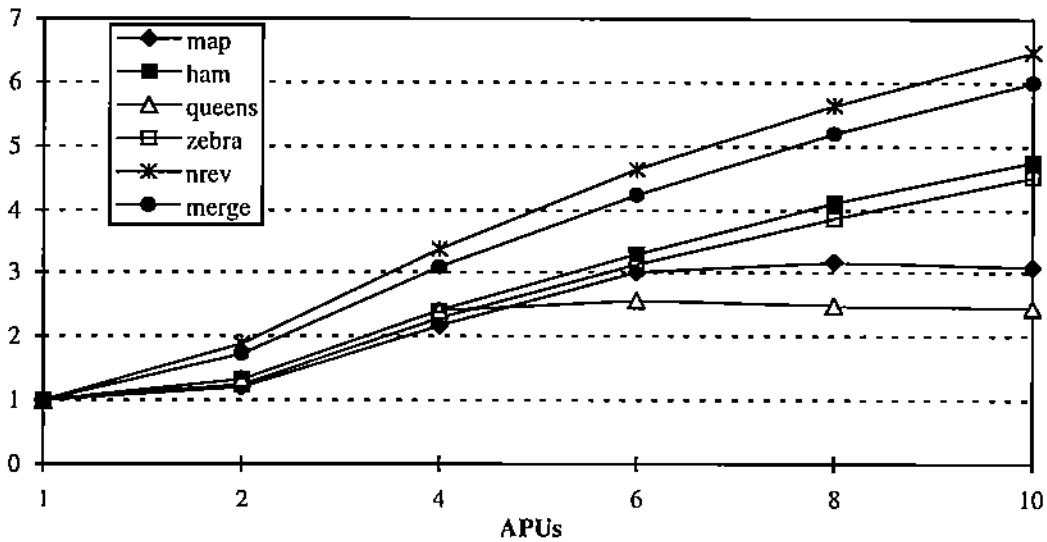


Figure 6

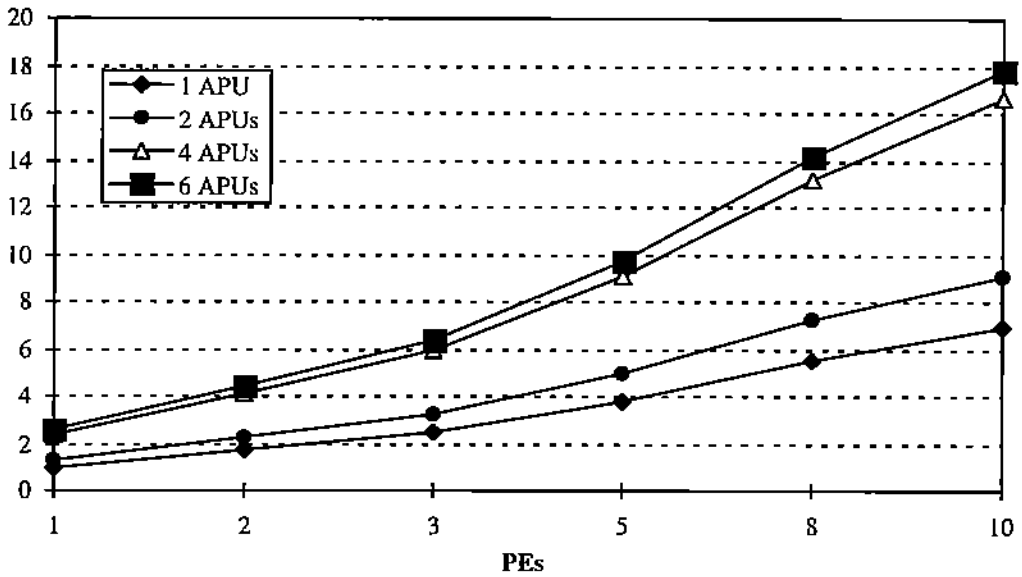


Figure 7

Variable	**Constant**
Void Variable (VVAR)	Atom
Skelet Variable (SVAR)	Integer
Free Variable (FVAR)	Real
Bound Variable (BVAR)	Nil
Temp. Free Var. (TFVAR)	
Temp. Bound Var. (TBVAR)	
Functional Term	**List**
Ground Term (GFTERM)	Ground List
Source Term (SFTERM)	(GLIST)
Copy Term (CFTERM)	Source List (SLIST)
	Copy List (CLIST)

Table 1

Name	Register
PC	Program counter
TE	Top of the environment stack
TT	Top of the trail stack
TH	Top of the heap
TB	Top of the backtracking stack
TR	Top of the reset stack
CA	Call element address
CE	Call environment
HE	Head environment

Table 2

Procedural	Indexing	Unify
CRENV N	EXEC Ci, Gi	UNIFY N, Hi
PROCEED	EXECL Ci, Gi	
WIND	TRY Gi	
REWIND	TRYL Gi	
FAIL		
ESCAPE N, Gi		

Table 3

	Sequential OASys	Parallel OASys	Eclipse	C-Prolog	Sequential Andorra -I
queens8	4.1	5.3	3.5	6.5	17.2
hamilton	55.5	61.3	11.0	32.3	229.5
map	1.8	2.5	0.3	1.4	9.3
zebra	153.6	197.2	21.8	58.0	326.5
merge200	4.3	6.1	1.1	2.1	16.0
nrev140	6.6	7.0	0.6	2.4	18.9

Table 4