Designing a High Performance Parallel Logic Programming System

M. V. Hermenegildo
R. A. Warren

Abstract: Compilation techniques such as those portrayed by the Warren Abstract Machine (**WAM**) have greatly improved the speed of execution of logic programs. The research presented herein is geared towards providing additional performance to logic programs through the use of parallelism, while preserving the conventional semantics of logic languages. Two areas to which special attention is given are the preservation of sequential performance and storage efficiency, and the use of low overhead mechanisms for controlling parallel execution. Accordingly, the techniques used for supporting parallelism are efficient extensions of those which have brought high inferencing speeds to sequential implementations. At a lower level, special attention is also given to design and simulation detail and to the architectural implications of the execution model behavior. This paper offers an overview of the basic concepts and techniques used in the parallel design, simulation tools used, and some of the results obtained to date.

Keywords: LOGIC PROGRAMMING, PARALLEL PROCESSING, RESTRICTED AND-PARALLELISM, WAM, PROLOG

1 Introduction

The sequential performance of logic programs [12] has been greatly improved since the advent of the first Prolog interpreters by the development of evaluation and compilation techniques such as those portrayed by the Warren Abstract Machine (**WAM**) [21]. Specialized architectures are also being proposed which promise further performance improvements [18] [7]. The present research concentrates on providing *additional* execution speed to logic programs (i.e. beyond that afforded by sequential implementations) through the use of parallelism, while at the same time preserving the conventional semantics of logic languages. In order to achieve this goal, and in the assumption that actual applications do have some sequential parts, special emphasis is first given to the preservation of sequential performance and storage efficiency: the techniques used for supporting parallelism are extensions to those which have brought high inferencing speeds to sequential implementations. In addition, special attention is given to the efficiency of these extensions: low overhead mechanisms are used for controlling parallel execution. Finally, special emphasis is also given to design and simulation detail. This makes it possible to quantify the overhead associated with the management of parallel execution at many levels (scheduling, communication, synchronization, resource management, cache coherence maintenance, etc.) and select the techniques and architectural organizations which provide the best performance.

The research methodology used can be described as a set of iterations over the following points:

- Analyze the opportunities for parallelism present in logic languages and select one or more which can be "reasonably" implemented.

- Design an execution model which supports these types of parallelism with the minimum possible overhead while at the same time preserving sequential inferencing speed.

- Perform a detailed simulation of the execution model under idealized architectural organization parameters. Identify the most frequent and costly operations and try to optimize them.

- Select or design a particular architectural organization and introduce into the simulation the actual parameters associated with it. Obtain feedback regarding design decisions at the higher level and revise those decisions accordingly.

- Finally, implement the model on an available multiprocessor, if determined to be suitable, or design a new one as required by the execution model.

This paper reports on some of our current efforts in addressing several of these points: section 2 presents the forms of parallelism initially chosen for exploitation (*Goal Independence AND-Parallelism* in the form of *Restricted AND-Parallelism* and *annotated, pipelined OR-Parallelism*) and the reasons behind these choices. Section 3 introduces the **RAP-WAM** execution model and describes the characteristics of this model which have been shown to be important in meeting the performance objectives mentioned at the beginning of this introduction. Section 4 then describes some of the research activities completed or under way which relate to the simulation and implementation of the model. Some of the results obtained to date are also presented. In particular, overheads in the system have been shown to be low, sequential speed close to that of sequential systems, and actual speedup attainable even with current shared memory multiprocessor technology.

2 Efficient Parallel Execution of Logic Programs

Logic Programs offer many different sources of parallelism [5]. Ideally, all these sources should be exploited simultaneously in a given system. Nevertheless, the overhead involved in the management and control of this parallelism is non-trivial and due consideration must be given to the run-time cost associated with the exploitation of any given source.

2.1 AND- and OR-Parallelism

Of the various sources of parallelism present in Logic Programs AND- and OR-parallelism appear to be the most attractive. Considerable attention is being given to the implementation of either one of these types of parallelism (or a combination of both) in many proposed parallel logic programming systems [3, 14, 2, 1, 5, 13, ...]. Our research goal is the combined use of both AND- and OR-Parallelism. Efficient techniques for implementing OR-Parallelism have been proposed or are currently under development by various groups, as reported in [22]. AND-Parallelism, although theoretically offering advantages such as being able to exploit parallelism in determinate programs, has been difficult to implement due to the overhead involved in the handling of shared variable bindings and because of its interaction with "don't know non-determinism". Most of the research reported herein refers to the implementation of Goal Independence AND-Parallelism in the form of Restricted AND-Parallelism.

2.2 Goal Independence AND-Parallelism and Restricted AND-Parallelism

Conery [5] showed how "brute force" exploitation of AND-Parallelism (i.e. the automatic scheduling of a process for every goal in the body of a clause) leads to *variable binding conflicts*. Such a conflict appears if various goals attempt to bind a shared variable to different values. One solution to this

problem is to determine one goal as the producer of the variable, and the others as consumers. In **stream AND-Parallelism** these goals all run in parallel and the value of the variable is incrementally passed ("pipelined") from the producer to the consumers. Stream AND-Parallelism is useful because it allows the description of systems of communicating processes. A drawback in stream AND-Parallelism, however, is that it is difficult to implement in the presence of non-determinism. Therefore, proposed systems which exploit this type of parallelism do not implement the conventional ("don't know") non-deterministic semantics of logic programs and implement "committed-choice" (i.e. "don't care") non-determinism instead. Such is the case in PARLOG [8], Concurrent Prolog [15], and Guarded Horn Clauses (GHC) [19]. Our interest, however, is in preserving the conventional ("don't know") semantics[1].

An alternative way of dealing with variable binding conflicts which makes support for both AND-Parallelism and "don't know" non determinism more amenable to an efficient implementation is **Goal Independence AND-Parallelism** [11]. In this form of AND-Parallelism only sets of goals which are determined to be independent (i.e. which do not share any non-ground variables) can be executed in parallel. Goal independence can be *postulated* through annotations. It can also be *determined* either statically by the compiler (perhaps guided by some information provided by the user on the type of queries that are most likely to be presented to the system [2]), completely dynamically at run-time [5], or through a combination of the above mentioned techniques. **Restricted AND-Parallelism (RAP)** [6] represents such a combination offering more opportunity for parallelism than static systems at a lower cost than dynamic ones. **RAP** combines a compile-time analysis of the clauses in the program with simple checks on variables at run-time (determining whether they are "ground" or "independent").

In this research a generalized version of **RAP** is used [11]. This version completes DeGroot's original description (by providing backward execution semantics) and overcomes the difficulty in expressing the "sufficient" conditions for independence [9] using DeGroot's conditional graph expressions. In addition, user annotations are permitted which are used to reduce (or even eliminate altogether) the number of tests generated by the compiler. Other types of AND- and OR-Parallelism are also under consideration. As an example of this generalized form of **RAP** consider the following clause:

```
child(X,Y,Z):- father(Y,X), mother(Z,X).
```

While analyzing the example above, the **RAP** compiler would find that "`father(Y,X)`" and "`mother(Z,X)`" may not be independent at run-time (not only do they share the variable `X`, but the variables `Y` and `Z` could be "linked" through unification). However, it would also find (using the independence conditions given in [9]) that these goals are guaranteed to be independent (and therefore can be executed in parallel in a straightforward way) if the clause happens to be called with the first argument (`X`) being "ground" (i.e. fully instantiated), and the other two (`Y` and `Z`) being independent (i.e. they do not "share" [11]). The result of this compile-time analysis can be encoded in a "Conditional Graph Expression" (**CGE**), and the clause rewritten as shown below.

```
child(X,Y,Z):-( ground(X), indep(Y,Z) | father(Y,X) & mother(Z,X)).
```

The declarative semantics of the clause above remains identical to that of the original clause, but the procedural semantics changes:

- Try to unify "`child(X,Y,Z)`" with the calling goal. If successful,
- Check if "`X`" is ground and if "`Y`" and "`Z`" are independent. In that case, execution of "`father(Y,X)`" and "`mother(Z,X)`" can be started in parallel.

---

[1] In fact, "committed choice" systems ultimately share this interest. For example, Codish [4] and Ueda [20] both have proposed using program transformations generated through partial evaluation in order to implement a form of "don't know" non-determinism. It is argued that the more direct approach described herein has a higher performance potential while guaranteeing that full "don't know" non-determinism will be supported.

- If the checks fail, execute "`father(Y,X)`" and "`mother(Z,X)`" sequentially.

Thus, the **CGE** embedded within the clause above can generate (depending on the result of the "checks" at run-time) two execution graphs: a sequential and a parallel one. Nesting of Conditional Graph Expressions can generate more complicated execution graphs, and the run-time system, while executing the **CGE**, will select different branches of the graphs depending on the results of the checks. The backward semantics (actions taken in case of failure) support "don't know" non-determinism and are given in [11]. Of course, if the user declares that the child rule will for example be called with its first two arguments "ground" then it is not necessary to generate any checks since "`ground(X)`" is obviously true and "`indep(Y,Z)`" is also guaranteed because Y, being "ground", cannot share any variables with Z. In this case the clause can always be executed in parallel and it would be rewritten by the compiler as

```
child(X,Y,Z):-( true | father(Y,X) & mother(Z,X)).
```

3 RAP-WAM Execution Model and Architectural Characteristics

As pointed out in the introduction, in order to achieve actual speedup an evolutionary approach is chosen at the execution model level: since logic programs, in addition to offering opportunities for parallelism, often also present code segments requiring sequential execution, great emphasis is given to the preservation of sequential performance and storage efficiency as well as to the use of low overhead mechanisms for controlling parallel execution. The approach initially taken in this design is therefore to provide the mechanisms for supporting forward and backward parallel execution of logic programs as extensions to the ones used in a high performance Prolog implementation, the **WAM** [21].

3.1 Extending the WAM for Restricted AND-Parallel Execution: the RAP-WAM execution model

An early result of the research approach described above is the **RAP-WAM** execution model [9]. This model is defined as a parallel virtual machine and described at the *abstract machine level* (in much the same way as the **WAM**). This level of description provides a concrete target for compiler design, and is suitable both for implementation on conventional multiprocessors or as a guide in the design of a specialized architecture. Figure 1 shows one of the abstract machines of the **RAP-WAM** execution model. Each of these abstract machines is similar to a standard **WAM** (with a complete set of registers and data areas) except for the addition of a "*Goal Stack*", a "*Message Buffer*", and the inclusion of new types of frames ("*Parcall Frames*" and "*markers*") in the *Stack*. Each machine has access to the *Heap*, *Stack*, and *Goal Stack* of other machines.

The instruction set of the **RAP-WAM** abstract machine includes all **WAM** instructions and several new instructions which are related to parallel execution. Figure 2 lists the instructions which currently support AND-Parallelism. Although the "check" instructions are somewhat particular to the implementation of **RAP**, the instruction set is also suitable for other AND-Parallel systems. Instructions of the form "`..._det_...`" are optimized for determinate execution. The last instructions are "pseudo instructions" which represent the actions taken upon failure and during distributed scheduling and backtracking.

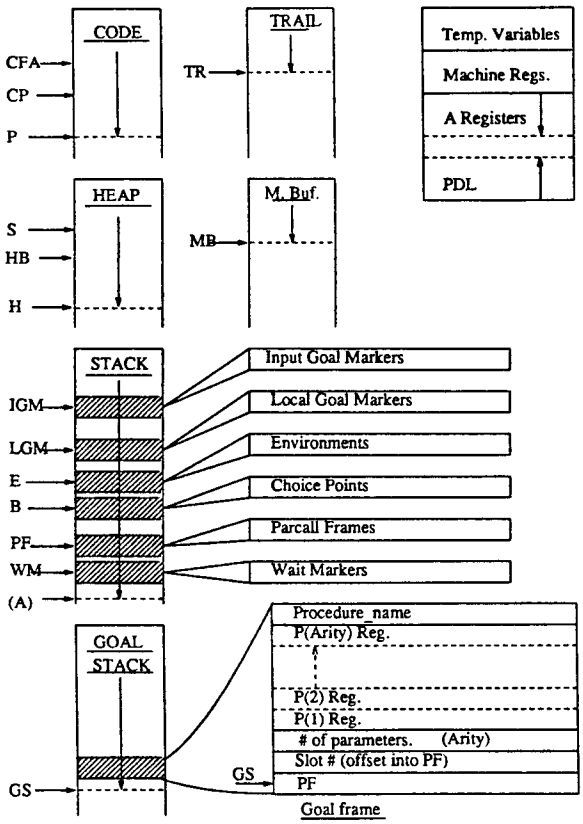

Figure 1: Data areas and registers for one abstract machine of the **RAP-WAM** model

```
check_me_else Label
check_ground Vn
check_independent Vn,Vm

proceed
idle
pop_foreign_goal
fail
kill
redo
unwind
```

```
push_call Procedure/Arity,Slot#
pop_pending_goal
allocate_pcall #_of_slots,M
check_ready Slot_#,Label
wait_on_siblings


push_det_call Procedure/Arity,Slot#
allocate_det_pcall #_of_slots,M
cut_merge
```

Figure 2: Parallel Abstract Machine Specific Instructions

3.2 Execution Model Characteristics

Space limitations make a complete description of the execution model impossible (the reader is referred to [9] and [10] for such a description). The basic characteristics of this execution model are:

- *Extended support for* **WAM** *optimizations, preservation of sequential speed and storage efficiency.* The parallel execution model still supports last call optimization, environment trimming, unification customization, clause indexing, space retrieval on backtracking and other storage and performance optimizations of the **WAM**. In particular, the total amount of storage needed by the parallel system is essentially equivalent to that of a sequential **WAM** implementation. Storage is still always retrieved from the top of all stacks during backtracking thus simplifying memory management and minimizing the frequency of invocation of garbage collection. Sequential speed is essentially equivalent to that of the **WAM**. Determinate execution is optimized.

- *Efficient support of AND-Parallelism and "don't know" non-determinism.* The model semantics ensure support for full "don't know" non-determinism during parallel execution. "Restricted" intelligent backtracking is supported as well [11]. Variable binding conflicts are detected and dealt with efficiently through the forward execution semantics of the **CGEs**. A conscious effort has been made throughout the design towards optimizing performance while reducing overhead and minimizing communication, synchronization and storage requirements. Therefore, speedup beyond **WAM** performance is attainable. This has been confirmed by the simulations [9].

- *User-transparency of control issues and distributed control.* Scheduling of goals is done dynamically at run-time in a distributed and user-transparent way [10]. A "work based" (rather than "process based") model is used which minimizes process creation. Performance degrades "softly" with resource exhaustion: code generated from conditional graph expressions will automatically run sequentially if there are no free processors available. All communication and synchronization issues are concealed within the semantics of the **CGEs**.

4 Research Activities and Preliminary Results

4.1 Measurement Tools

A series of measurement tools have been built in order to evaluate the potential performance of the execution model and the associated architectural tradeoffs [9]. The present configuration of these tools is shown in figure 3. Application programs are translated into parallel **WAM** code and fed into a **RAP-WAM** emulator together with emulation parameters such as the number of processors, type of scheduling strategy used, storage area sizes, etc. The emulator executes the program and generates (in addition to the correct result) instrumentation data such as instruction and pseudo-instruction frequencies, number of references to the different data areas, ratios of local vs. remote references for each type of object, maximum amount of storage used in each area, timings, speedups, etc. Following the methodology outlined in section 1, these measurements are made under the assumption of an idealized architectural organization with a uniform address space and no memory contention. In order to make it possible to evaluate the deviation from this ideal behavior due to the characteristics of a particular architectural organization (such as available bandwidths, cache coherence maintenance overhead, etc.) the emulator can also generate a trace file of memory references. A particular case which has been studied is that of a shared memory multiprocessor with coherent caches using a set of coherent cache simulators [17] which take the trace file data as input (figure 3).

4.2 Research Results

Relevant results obtained to date include the following [9]:

- It appears that the model can achieve *actual speedup* even if the application exhibits only low levels of parallelism.

- The overheads in the model due to the management of parallelism are low (observed to be less than 15% for a worst case). This suggests that other types of AND-parallelism with higher expected implementation overheads may achieve actual speedup using similar techniques.

- The stack memory management is very efficient and in many cases it avoids the invocation of garbage collection. Its implementation is, however, more complicated than that of other approaches, such as a simple heap (but promises better performance).

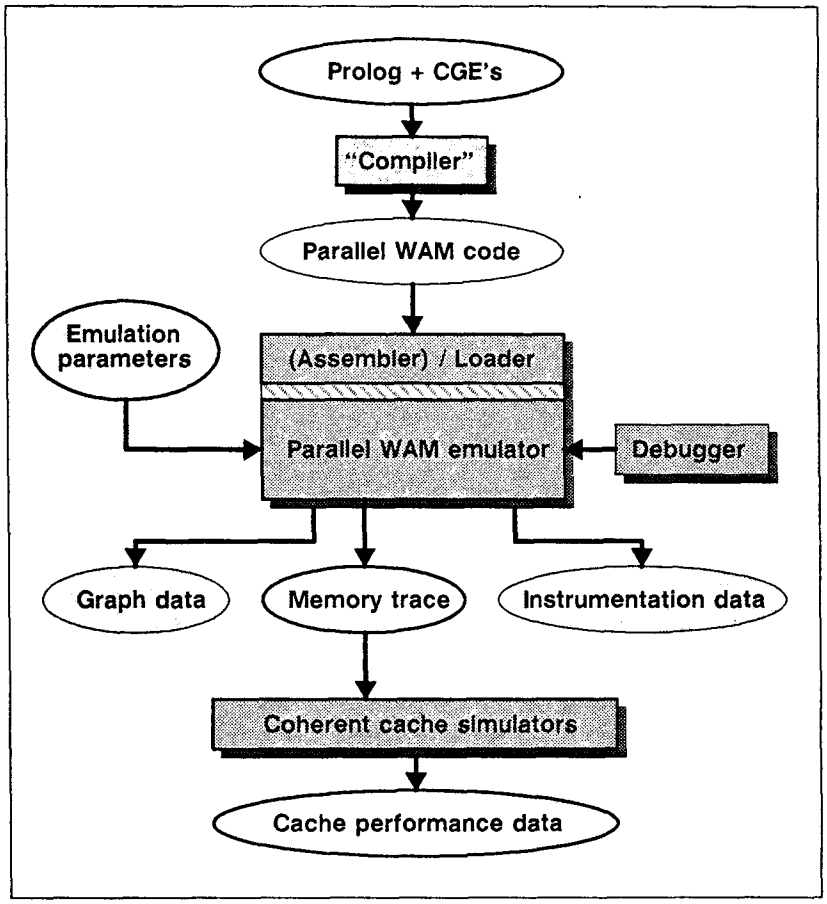

Figure 3: Simulation Tools

Additional results are being obtained from lower level simulations:

- Coherent cache simulations show a clear superiority of *write-back broadcast caches* over other types. A *hybrid broadcast cache* has been developed [17] which promises reasonable performance at a lower cost. It is conjectured that this result is due both to the high write frequency of Prolog and to the low communication overhead of the **RAP-WAM** execution model.

- The reduction in memory traffic per processor on the shared bus due to the presence of caches can be close to 80%. This makes it reasonable to predict that speeds in the order of 2 million *application* inferences per second are possible on shared memory multiprocessors built using current technology.

4.3 Areas of Current and Future Research

Some of the areas where research is proposed or currently under way are:

- *Automatic generation of* **CGE**'*s*: the "correctness" conditions proposed in [9] and a data dependency analysis of the source program (which can be aided by user annotations) are being used as the basis for the design of a tool for the automatic annotation of Prolog programs with **CGE**'s. A prototype has already successfully annotated some simple programs.

- *OR-parallelism support*: support for OR-Parallelism is being included in the model using recently proposed mechanisms for the maintenance of multiple binding contexts [22]. However, the problem of efficiently combining OR- and AND-Parallelism (beyond pipelined parallelism [16]) presents some interesting research questions.

- *Study of the memory referencing behavior of the system*: the information being obtained from this study is instrumental in understanding the architectural requirements of parallel logic programming. Current and proposed architectures and organizations are being analyzed in light of this data.

- *Support for other types of parallelism and language extensions*: specific additions or modifications to the model in order to support other types of parallelism and features not currently supported (such as *streams*, *functions*, or *goal suspension*) are under study.

5 Conclusions

An important goal of this research is the achievement of performance in logic programs beyond that of current sequential systems through the use of parallelism. One of the early research results is the **RAP-WAM** parallel execution model and the evaluation of its performance potential. This model provides precise forward and backward procedural semantics for Logic Programs annotated with **CGE**'s which can be implemented efficiently, and an abstract machine architecture which extends the optimizations present in sequential systems to a parallel environment. The results obtained so far show that Goal Independence AND-Parallelism in the form of Restricted AND-Parallelism can be efficiently implemented in the presence of "don't know" non-determinism and that significant speedups with respect to high performance sequential implementations can be obtained using the proposed execution model and current state-of-the-art multiprocessor technology.

References

[1] P. Borgwardt and D. Rea.
Distributed Semi-intelligent Backtracking for a Stack-based AND-parallel Prolog.
In *Proceedings of the 1986 Symposium on Logic Programming*, pages 211-222. IEEE Computer Society, 1986.

[2] J.-H. Chang, A. M. Despain, and D. DeGroot.
AND-parallelism of Logic Programs Based on Static Data Dependency Analysis.
In *Digest of Papers of COMPCON Spring '85*, pages 218-225. 1985.

[3] A. Ciepilewski and S. Haridi.
Control of Activities in the Or-Parallel Token Machine.
In *1984 International Symposium on Logic Programming, Atlantic City*, pages 49-58. IEEE Computer Society Press, Silver Spring, MD, February, 1984.

[4] M. Codish and E. Shapiro.
Compiling OR-Parallelism into AND-Parallelism.
In *Proceedings of the Third International Conference on Logic Programming*, pages 283-298. Springer-Verlag, 1986.

[5] J. S. Conery.
The AND/OR Process Model for Parallel Interpretation of Logic Programs.
PhD thesis, The University of California at Irvine, 1983.
Technical Report 204.

[6] Doug DeGroot.
Restricted And-Parallelism.
Int'l Conf. on Fifth Generation Computer Systems , November, 1984.

[7] T. P. Dobry, Y. N. Patt, and A. M. Despain.
Design Decisions Influencing the Microarchitecture for a Prolog Machine.
In *MICRO 17 Proceedings.* 1984.

[8] S. Gregory.
Design, Application and Implementation of a Parallel Logic Programming Language.
PhD thesis, Imperial College of Science and Technology, 1985.

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

[10] M. V. Hermenegildo.
Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs.
In *Proceedings of the Fourth International Conference on Logic Programming.* MIT Press, May, 1987.

[11] M. V. Hermenegildo and R. I. Nasr.
Efficient Management of Backtracking in AND-parallelism.
In *Proceedings of the Third International Conference on Logic Programming,* pages 40-55. Springer-Verlag, 1986.

[12] R. A. Kowalski.
Predicate Logic as a Programming Language.
Proc. IFIPS 74 , 1974.

[13] Y.-J. Lin, V. Kumar, and C. Leung.
An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs.
In *Proceedings of the Third International Conference on Logic Programming,* pages 55-69. Springer-Verlag, 1986.

[14] R. A. Overbeek, J. Gabriel, T. Lindholm, and E. L. Lusk.
Prolog on Multiprocessors.
Technical Report, Argonne National Laboratory, Argonne, Ill. 60439, 1985.

[15] E. Y. Shapiro.
A subset of Concurrent Prolog and its interpreter.
Technical Report TR-003, ICOT, January, 1983.
Tokyo.

[16] N. Tamura and Y. Kaneda.
Implementing Parallel Prolog on a Multiprocessor Machine.
In *1984 International Symposium on Logic Programming, Atlantic City,* pages 42-49. IEEE Computer Society Press, Silver Spring, MD, February, 1984.

[17] E. Tick.
Studies in Prolog Architectures.
PhD thesis, Stanford University, 1987.
In preparation.

[18] E. Tick and D. H. D. Warren.
Towards a Pipelined Prolog Processor.
In *1984 International Symposium on Logic Programming, Atlantic City*, pages 29-42. IEEE Computer Society Press, Silver Spring, MD, February, 1984.

[19] K. Ueda.
Guarded Horn Clauses.
Technical Report TR-103, ICOT, 1985.
Tokyo.

[20] K. Ueda.
Making Exhaustive Search Programs Deterministic.
In *Proceedings of the Third International Conference on Logic Programming*, pages 270-283. Springer-Verlag, 1986.

[21] D. H. D. Warren.
An Abstract Prolog Instruction Set.
Technical Note 309, SRI International, AI Center, Computer Science and Technology Division, 1983.

[22] D. H. D. Warren.
OR-Parallel Execution Models of Prolog.
In *Proceedings of TAPSOFT '87.* Springer-Verlag, 1987.