Unrestricted And-Parallel Execution of Logic Programs with Dependency Directed Backtracking

Nikos Drakos[1]
School of Computer Studies
University of Leeds
Leeds LS2 9JT
England
nikos@uk.ac.leeds.ai

Abstract

A model of unrestricted And-parallel execution for logic programs is described, based on Dependency Directed Backtracking (DDBT) aimed at improving the efficiency of execution while remaining faithful to conventional syntax and semantics. This is achieved with maximum exploitation of parallelism, DDBT, support of opportunistic stream parallelism and potentially parallel backtracking. The same dependency information needed for DDBT is used to resolve variable binding conflicts. An algorithmic description of the behavior of processes embodying logical procedures is given. Circular dependencies arising out of the opportunistic nature of execution are removed by imposing an ordering on dependencies with an *overwriting rule.* Various aspects of the model are discussed and illustrated with examples.

1 Introduction

Logic programming has grown out of work on automatic theorem proving, and is based on the idea that logic can be used as a programming language. The widespread use of such programming languages has been hindered by what is often cited as a major disadvantage - execution can be very slow. This problem is being tackled on many fronts:

• by enhancing the control mechanism of logic programming languages with various schemes of DDBT [Bruynooghel984,Coxl985,Changl985,Kumarl986].
• by developing special purpose machines for the execution of logic programs such as the Warren Abstract Machine (WAM) [Warren 1983] and the Berkeley Programmed Logic Machine [Fagin 1987, Dobry 1984].
• by adapting the execution model of logic programming languages for use on a multiprocessing architecture [Lil986, Conery 1987, Hermenegildo l986]

The above ideas are by no means mutually exclusive. Various parallel execution models incorporating DDBT [Kumar 1986a, Dembinskil985, Borgwardtl 986], have been expressed in terms of abstract machine microinstructions [Faginl987]. But many of the parallel execution models have been criticized for expensive runtime support, forcing the computation of all solutions simultaneously, no support of "stream parallelism", etc. A common characteristic of these models is their approach to the problem of variables shared by more than on subgoals in a clause resulting in schemes of *restricted And-parallelism.*

This work focuses on the use of *intelligent* or *term-based* DDBT [Bruynooghe 1984, Drakos 1988] in the context of parallel execution. DDBT is a simple mechanism, where each object is tagged with the set of derivation steps at which it is constructed or modified. On subsequent failures, the modification histories of failing objects can form a set of possible *culprits.* Backtracking to any intermediate derivation step between the current step and the most "recent" step in the culprits would not remove the failure. Consequently, intermediate steps potentially containing choice points can be ignored.

DDBT can not only help in reducing the search space in the manner described above but also allow a higher degree of parallelism during forward execution and introduce some parallelism during backtracking. This is achieved by using the dependency information required for DDBT in sequential models to resolve *variable binding conflicts* in the parallel model.

The advantages of this approach are mainly:
• The maximum amount of And-parallelism is exploited
• No preprocessing or runtime support is required to detect goals with shared variables
• *Opportunistic* stream parallelism is supported
• Parallel backtracking is possible
• Only one solution at a time is computed
• It *is fairer* than other models with desirable termination properties [Drakos 1989]

2 Parallel Execution of Logic Programs

A *program clause* is a formula of the form A<--B_1...,B_n n>0, where B_l...,B_n are atoms or negated atoms. A is an atom called the *head* of the clause and B_l.., B_n is the *body* of the clause. The commas in the body denote conjunction. A *logic program* is a finite set of program clauses. The set of all program clauses with the same predicate p in their heads is called the *definition* of p. A *goal clause* is a like a program clause but without a head. Each B_1 (i = l,...,n) is called a *subgoal* and can be interpreted procedurally as a

[1]This work is supported by a grant from the University of Leeds

procedure call. The empty clause (denoted □) has an empty head and body.

Parallelism can be introduced in logic programming mainly in two ways. *Explicit parallelism* is associated with the addition of concurrent constructs resulting in extended logic languages, able to cope with processes and concurrency so that they are suitable for applications such as systems programming (e.g. Parlog, Concurrent Prolog and GHC). *Implicit parallelism* involves defining parallel operational semantics equivalent to the sequential semantics and without changing the syntax aiming to improve efficiency with a multiprocessor implementation. In the context of implicit parallelism, apart from *low level parallelism* (parallel unification and parallelism in the control mechanism) the two main sources of parallelism are And-parallelism and OR-parallelism.

And-parallelism is the attempt to resolve simultaneously more than one literals in the body of a program clause. The main difficulty arises from *shared variables*. Consider the logic program,

$$f(X) \leftarrow p(X), q(X). \quad p(1). \quad q(2).$$

with goal $\leftarrow f(X)$. X should be bound to a value that satisfies both p and q. But if p and q are executed in parallel, then each will produce a different binding for X - a *variable binding conflict*.

One common solution to the problem of shared variables adopted in models of *restricted* And-parallel execution is to allow only one of the literals to bind the shared variable and postpone the solution of the others until a value has been found. These models differ in their methods for detecting literal dependencies, ranging from purely static to semi-dynamic and dynamic. Once the dependencies are known (either at compile or run-time), a *literal ordering* strategy is adopted to schedule literal execution.

With *unrestricted* And-parallelism all the literals can proceed in parallel even when sharing variables. The variable bindings are globally visible and tagged with the identity numbers of the processes that have created, modified or seen them. When another literal tries to assign a different value to a bound variable, these dependency tags can be used to resolve the conflict.

In *OR-Parallelism* the main concern is a parallel definition of the *search rule* obtained from the parallel execution of the alternative procedure definitions. The model for unrestricted And-parallelism can be adapted to include OR-parallelism but this may increase the process creation overheads, reduce too much the granularity of a process, introduce binding environment overheads, increase the communication overheads, and force the exploration of all solutions simultaneously [Conery1987]. For these reasons it was decided to focus on And-parallelism.

3 Unrestricted And-Parallelism

A *process* is an instance of a procedure in execution, allocated to a distinct processor. Within each process P_i, a goal G_i is resolved with one of the clauses in the definition of the predicate for G_i. Each process proceeds independently and communicates with the others via messages.

Suppose that a process P_i has been created to resolve the goal G_i whose predicate is A_k with a clause in the definition of A_k. If the chosen clause is $A \leftarrow B_1,\ldots,B_q$ and $A_k\theta_i = A\theta_i$ (i.e. A_k and A are unifiable and θ_i is their m.g.u.) then the resolvent G_{i+1} is $\leftarrow(B_1,\ldots,B_q)\theta_i$. Each $B_j\theta_i$, $j=1,\ldots,q$ is called a *child* of P_i. Variables and variable bindings may be globally visible but protected with a *locking mechanism* for *concurrent data access* [Diel1984]. Consequently, any process may read the value of a particular variable but only one process may write or update it at a time.

During forward execution, a process is created for the top goal G_0 and for each of its children. A process terminates when its goal is unified with a unit clause. Execution terminates when all processes terminate. But in the case of failure some of the chosen clauses within each process must be redone (backtracking). Two types of failure can be identified:

• A *shallow failure* occurs when the chosen clause $A \leftarrow B_1,\ldots,B_q$ fails to resolve with A_k but there are still untried clauses in the definition of the predicate for A_k.
• A *deep failure* occurs when the chosen clause fails to resolve with A_k but there are no untried clauses left.

In the first case, another clause is chosen and execution continues. But in the second case, dependency information can be used by a deeply failing process to send appropriate messages to other process to modify their choices. More formally, let $t_1,\ldots,t_m$ be the finite sequence of terms occurring in A_k,A or in B_j, $j=1,\ldots,q$. Let $t_j' = t_j\theta i$, $j = 1,\ldots,m$. Now define the function producers from terms to ordered sets of process_id's as follows:

$$\text{producers}(t_j) = \{\ \} \text{ if } t_j \in G_0 \text{ or } t_j \in A, B_1,\ldots,B_q$$

$$\text{producers}(t_j') = \begin{cases} \text{producers}(t_j) \text{ if } t_j' = t_j,\ j = 1,\ldots,m \\ \text{producers}(t_j) \cup \{\ i\ \} \text{ otherwise} \end{cases}$$

Intuitively terms occurring in the top goal G_0 or in any subgoals in the chosen clause have no producers, otherwise if a term is modified by the substitution θ_i then the process-id of the current process is included in the producers of the term. In the latter case the process P_i is called the *producer* of the term.

When a process moves to a different alternative, it might be the case that it has produced not only "troublesome" terms but also terms that have been accepted or "consumed" by other processes. So, on undoing the variable bindings of a discarded alternative, the consumer processes must be identified and be instructed to check again their choices. More formally, consider a process P_i with goal A_k and the terms t_j' as before. Then P_i is the *consumer* of a term t_j if t_j is a non-variable term and $t_j \in A_k$. Now it is possible to define a function consumers from terms to sets of process-id's such that:

$$\text{consumers}(t_j) = \{\ \} \text{ if } t_j \in G_0$$

$$\text{consumers}(t_j') = \begin{cases} \text{consumers}(t_j) \cup \{ i \} \text{ if} \\ \quad P_i \text{ is a } \textit{consumer} \text{ of } t_j \\ \text{consumers}(t_j) \text{ otherwise} \end{cases}$$

A shallow failure may occur because the current goal has inherited bindings made by other processes or simply because it cannot be satisfied. In the latter case execution trivially proceeds, but in the former, those processes blocking the current alternative must be "remembered" so that if at a later stage one of them changes its choice, the currently rejected alternative must be re-checked. During deep failure all the alternative clauses in a process P_i have been blocked and one of the blocking processes must change its choice. Suppose that the chosen "culprit" is P_j. Then the current alternative of P_j can be thought of as being blocked collectively by the rest of the processes causing the exhaustion of all the alternatives of P_i. If one of them changes, then another alternative of P_i compatible with the current one for P_j may become available.

To summarize, there are two ways in which an alternative clause of P_i may be rejected. The first is direct as a result of unification failure. The other is indirect by participating in the exhaustion of all the alternatives of another process, and P_i being chosen as the one to retract its alternative. These notions can be captured more formally with the functions **top** and **culprits**. **top** is a function from ordered sets of process-id's to process-id's returning a chosen backtrack point (culprit). Now let P_i be the current process with goal $\leftarrow A$ and the set of clauses $C = \{ A_1, \ldots, A_n \}$ be the definition for the predicate of A. Then

$$\text{culprits}(i) = \left[\bigcup_{j=1,n} \left[\text{producers}(t_l) \cup \text{producers}(t_m) \right] \right] \cup \left[\text{culprits}(k) - \{ k \} \right]$$

such that $t_l \in A$ and $t_m \in A_j$, $A_j \in C$, t_l and t_m are terms failing to unify, i is the process-id of P_i, k is the process-id of P_k, P_k is deeply failing and **top(culprits(k))** = i.

The function **culprits** has a dual use. Apart from being used to locate backtrack points as described above, it partly determines the "dependents" of a particular process. A process P_i such that $P_j \in$ **culprits(i)** is considered as depending on P_j in the sense that if P_j changes then one (or more) of the rejected alternatives of P_i may become unifiable. But a process P_i also depends on P_j if P_j has produced variable bindings consumed by P_i, so that if P_j changes its currently active alternative then P_i must be reexamined for compatibility. More formally, a function **dependents** can be defined from process-id's to ordered sets of process-id's as follows:

$$\text{dependents}(i) = \{ j : i \in \text{culprits}(j) \text{ or } i \in \text{consumers}(t_k) \}$$

such that i,j are process-id's, t_k is a term in the goal A of process P_j, k = 1,...n, n is the number of terms in A.

Processes communicate via three kinds of messages. A KILL is sent from any process to all its children. On receiving it, a process sends a KILL message to its children and terminates. A RESET is sent from a process P_i on rejecting an alternative clause C to all process with process-id's in **dependents(i)**. On receiving a RESET message, a process P_j sends a KILL message to all its children, a RESET message to all processes in **dependents(j)** and restarts execution from its first alternative clause. A REDO is sent from a process P_i after deep failure, to the process with process-id **top(culprits(i))**. On receiving it, a process P_j sends a KILL message to its children, a RESET message to all in **dependents(j)** and moves to its next alternative clause. Consider the following example:

q(a). q(b). r(a,a). r(S,T) :– sr(S,T). sr(a,a). p(b,a). p(a,b). t(c).

The goal clause G is ←q(X), t(Z), r(Y,Y), p(X,Y). Tracing the execution of the above logic program can be simplified with the following assumptions:

- Processes get higher process-id's in a left to right order within each clause body but a process P always has a lower process-id than its own children or the children of any of the processes created for goals in the same clause body (e.g. the process-id for sr is higher than all other process-id's in Figure 1).
- In the case of shared variables a process with a lower process-id has precedence over any other in producing variable bindings (although in practice this will be random).
- The function **top** returns the highest process-id of a set of process-id's.

The execution of the above logic program with goal G is shown as a series of frames in Figure 1. Each frame contains a *forest* of trees of depth 1. Each tree represents a process with the process-id encircled to its left. The trees are rooted at the goal for the corresponding process, with each leaf being an alternative clause. The leftmost one is the currently active.

In Frame 1 P_1 to P_3 resolve their goals with their first alternative clauses. A shallow failure occurs in P_4 due to P_1 so P_4 moves to its second alternative. This is blocked by P_3, a deep failure occurs and P_3 is sent a REDO message since **top(culprits(4))**= 3 . In Frame 2 P_3 receives the REDO and tries the second alternative after RESETting its dependent P_4. P_4 restarts and shallow fails as before but succeeds with the second alternative by binding Y to b and consuming the substitution produced by P_1. Meanwhile P_3 creates the child process P_5 which fails deeply and sends a REDO to P_4. In the next frame P_4 receives the REDO, deeply fails and sends a REDO to P_3 since **top(culprits(4))** = 3. P_3 deeply fails and in turn sends a REDO to P_1 and a KILL to P_5. P_1 accepts the REDO in frame 4 and moves to its next alternative after RESETting P_3 and P_4 and the first solution is found.

4 Circular Dependencies

If the second assumption of the previous section were relaxed, then two or more processes blocking each other could give rise to circularity problems. Consider the following logic program,

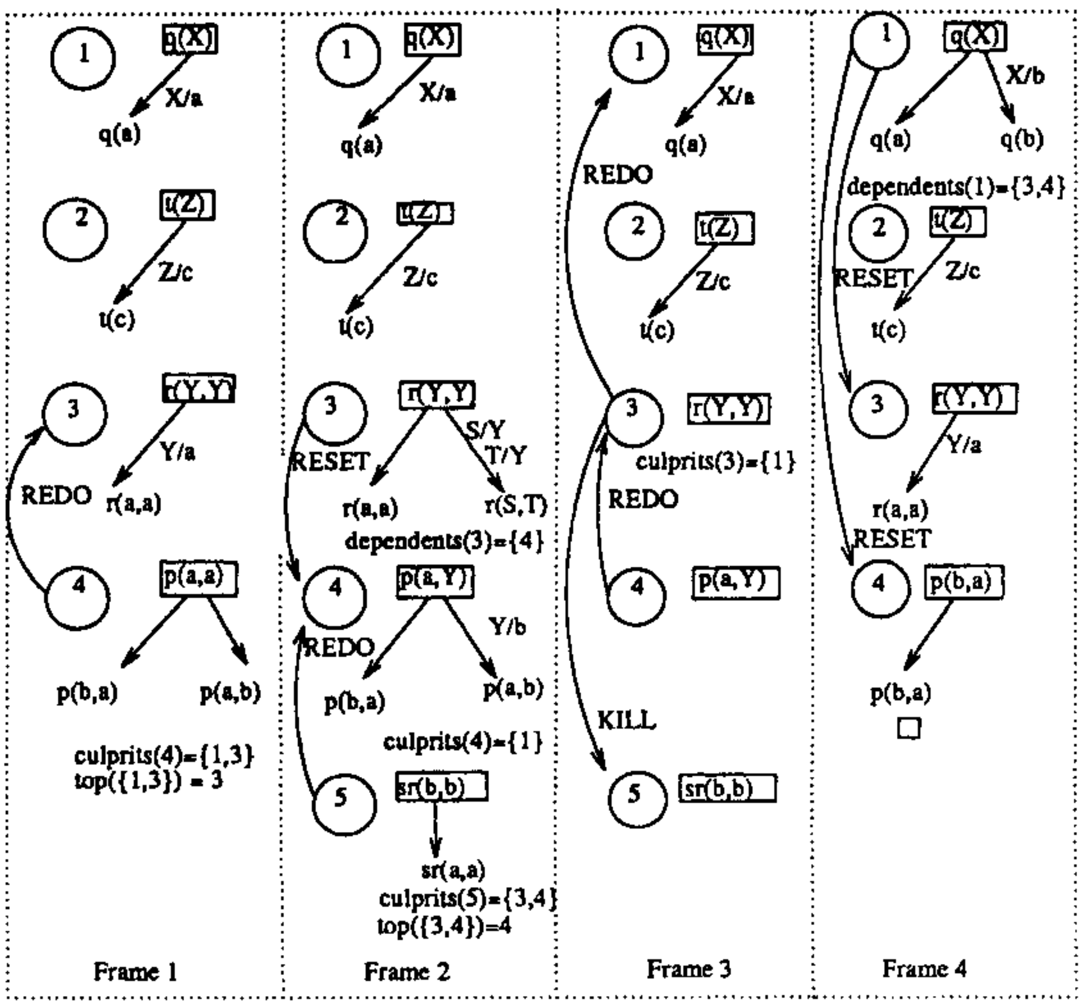

Figure 1. And-parallel execution of a logic program

a(l,2). a(l,l).b(l,2). b(2,l).

and the goal <-a(X,Y), b(Y,X). Then the following scenario is possible:

> Two processes P_1 and P_2 are created to solve the two subgoals. P_1 produces the variable binding { X/l } and P_2 produces { Y/l }. So shallow failures occur both in P_1 and P_2 which move to their second alternatives, sending a RESET message to each other since dependents(l) = {2 } and dependents(2) = { 1 }. So they both restart from their first alternative clause and the cycle may be repeated...

In a sequential model of execution either P_1 or P_2 would be the producer of both X and Y. Similarly, in parallel execution models with *variable annotations* [Pereiral986] or in models of restricted And-parallelism [Changl985,Faginl987,Hermenegildol986] this is not a problem as consumer processes always follow the completed execution of producers. Such solutions though, introduce a sequential element limiting the amount of parallelism and increasing the administration overheads.

There are three ways in which a process may depend on another process, i.e. (i) by direct failure, (ii) by indirect failure when receiving a REDO and (iii) by consuming substitutions. These make possible six different cases of direct circular dependencies. In the previous example the circularity arose out of two direct failures. It can be shown that such dependencies can be eliminated by imposing an *a posteriori* ordering with an *overwriting rule:*

> If a process P_1 receives a substitution 0 from P_j such that j > i then 8 is *overwriten* so that P| becomes the producer of 9. Also, if a subsequent failure occurs because of 0, a RESET is sent to P_j.

The overall effect of the rule is to make "older" processes the producers, thus restoring the partial order between producers and consumers imposed by the structure of the program. Consider the execution of the previous logic program under these circumstances:

> P_1 produces {X/l }, P_2 produces (Y/l } as before, but eventually P_1 overwrites Y and becomes the producer of { Y/2 }, sending a RESET to P_2. P_2 eventually accepts the RESET, tries and rejects b(l,2) and on moving to b(2,l) the first solution is found.

An algorithmic description of the behavior of a process is given in Program 2.

```
1. let the current process be $P_j$
and $G$ its goal with predicate $p$.
2. let $\{C_1,\ldots,C_n\}$ be the definition of $p$
3. for i from 1 to n do
    4. rename the variables of $C_i$ obtaining $C_i'$
    5. let $C_i'$ be $A \leftarrow B_1,\ldots,B_m$
    6. if $A$ and $G$ unify (plus overwriting) with
    m.g.u. $\theta$ then
        7. create new processes for each $B_k\theta$, $k = 1,\ldots,m$
        8. WAIT
    else
        9. undo $\theta$
        10. KILL($B_i$), $i = 1,\ldots,m$
        11. RESET($P_i$), $i \in$ dependents(j)
    endif
endfor
12. REDO(top(culprits(j)))
13. WAIT
```

Program 2. Algorithmic description process behavior

5 Opportunistic Stream Parallelism

Opportunistic stream parallelism can best be illustrated with the following logic program and the goal <-front(s(0),X,[l,2]) in the usual notation:

```
front(N,X,Z) :-append(X,Y,Z), !ength(X,N).
length([],0).
length([Q | R],s(N)):- length(R,N).
append([],X,X).
append([U | X],Y,[U | Z]):- append(X,Y,Z).
```

The predicate front(N,Y,Z) denotes the relation between a number N (in the *successor* notation i.e. 0 is 0, s(0) is 1, s(s(0)) is 2 etc.) and a list Y containing the first N elements of a list Z. One possible execution scenario up to the first solution is shown in Figure 3, assuming that enough processors are available, and that variables shared by different goals but in the same argument position are bound by processes with lower process-id's (this is just a convention making more consistent the inherently sequential, frame by frame trace of the execution and is not required by the model).

In frame 1 of Fig. 3, P_1 produces the substitution {X/[]} but a deep failure occurs in P_2 which sends a REDO back to P_1 In return, P_1 sends a RESET since P_2 depends on it. In the second frame, P_1 tries the second alternative and produces {X/[U1|X1], Ul/1} consumed by P_2. Note that P_2 starts checking the length of the list [11XI] even before XI is bound eventually by P_3 (the child of P_1). {XI/[] } is consumed by the child of P_2, P_4 and the first solution is arrived at with {X/[l]}. In the case of multiple solutions it is assumed that all active processes participate in a "failure" when a solution is arrived at A REDO can be sent to the youngest process to resume the execution towards subsequent solutions.

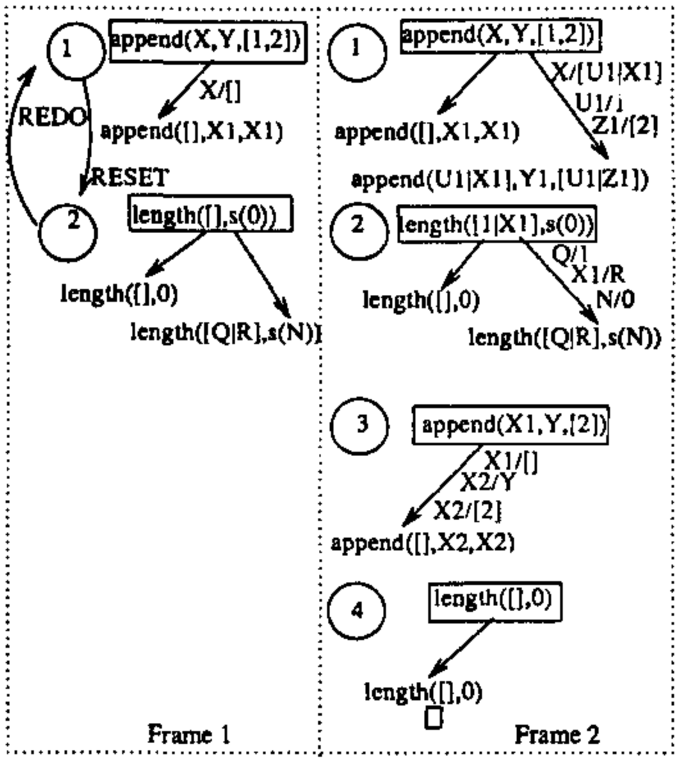

Figure 3. And-parallel execution with stream parallelism

Stream parallelism or *data flow corouting* [Clarkl981,Clarkl984,Faginl987] as it is otherwise known is an alternative control strategy where a call need not be completely evaluated before execution can proceed. In the definition of front, the first subgoal can be considered as producing successively longer lists with elements from Z, and length checking if they have N elements. Conventionally, append would have to be completely evaluated before execution can move to length, but with data flow corouting it is possible for length to start checking the length of list Y even before its elements are known. Usually this is achieved with annotations designating literals as producers and consumers and with strict protocols for interleaved execution. With opportunistic stream parallelism, as shown in the example, the producers and consumers as well as the order of execution are arbitrary. As a consequence, the full potential of stream parallelism may not be exploited but on the other hand there are no administration costs.

6 Discussion

Perhaps the most serious criticism against DDBT is the expensive representation and manipulation of the dependency sets. This cost can be reduced with a bit-integer representation so that a set of process-id's can be encoded as a single binary integer. Then set operations such as union and intersection can be performed with fast logical machine instructions [Drakosl988]. Decoding bit-integers into their constituent process-id's can be done efficiently using logarithms. The disadvantage of this encoding scheme is that very large binary numbers may be required

(2^{P+1}1 where p is the number of activated processes).

A simulation prototype of this model is being developed in order to assess performance gains and costs, with emphasis on the large bit-integer storage costs, the extra costs of overwriting in unification, message passing overheads, etc. At a later stage the design will be expressed in terms of an abstract logic machine based on the WAM (Warren Abstract Machine [Warrenl983]). The expected result is the specification of a high performance truly parallel logic machine.

Acknowledgements

I wish to thank my supervisor Richard Thomas for his support, encouragement and advice. Also thanks to Leyni.

References

[Borgwardtl986] P. Borgwardt and D.Rea, "Distributed Semi-Intelligent Backtracking for a Stack-based AND-parallel Prolog," pp. 211-222 in *Proceedings of the 3rd IEEE Symposium on Logic Programming,* Salt Lake City, Utah (1986).

[Bruynooghel984] M. Bruynooghe and L.M. Pereira, "Deduction Revision by Intelligent Backtracking," pp. 195-215 in *Implementations of Prolog,* ed. J.A. Campbell, Ellis Horwood Limited (1984).

[Chang 1985] J.-H. Chang, A.M. Despain, and D. DeGroot, "AND-Parallelism of Logic Programs Based on Static Data Dependency Analysis," pp. 218-225 in *Digest of Papers of COMPCON Spring '85,* (1985).

[Claricl984] K.L Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," *ACM Trans, on Programming Languages and Systems* Vol. 8 (1) pp. 1-49 (1984).

[Clarkl981] K.L. Clark and F. McCabe, "The Control Facilities of IC-PROLOG," pp. 122-149 in *Expert Systems in the Micro Electronic Age,* ed. D. Miche,Edinburgh University Press (1981).

[Conery l987] J.S. Conery, *Parallel Execution of Logic Programs,* Kluwer Academic Press (1987).

[Coxl985] P.T. Cox and T. Pietrzykowski, "Intelligent Backtracking in Plan Based Deduction," *IEEE Trans, on Pattern Analysis and Machine Intelligence* Vol. 7 (6) pp. 682-692(1985).

[Dembinskil985] P. Dembinski and J. Maluszynski, "AND-Parallelism with Intelligent Backtracking for Annotated Logic Programs," pp. 29-38 in *Proceedings of the 1985 Symposium on Logic Programming,* Boston (July 1985).

[Diel1984] H. Diel, "Concurrent Data Access Architecture," *Proceedings of the International Conference on Fifth Generation Computer Systems,* pp. 373-382 ICOT, (1984).

[Dobryl984] T.P. Dobiy, Y.N. Patt, and A.M. Despain, "Design Decisions Influencing the Microarchitecture for a Prolog Machine," *MICRO,* (17 Proceedings)(Oct 1984).

fDrakos!988] N. Drakos, "Reason Maintenance in Horn-Clause Programs," pp. 77-97 in *Reason Maintenance Systems and their Applications,* ed. G. Kelleher,EUis Horwood (1988).

[Drakos 1989] N. Drakos, "Multiple Use of Predicate Definitions in Horn Clause Logic Programs with DDBT and And-Parallelism," School of Computer Studies Research Report 89.4, University of Leeds (1989).

[Faginl987] B. Fagin, "A Parallel Execution Model for Prolog," CSD 87/380, University of California, Berkeley (1987).

[Hermenegildol986] M.V. Hermenegildo, *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel,* The University of Texas at Austin PhD (1986).

[Kumarl986a] V. Kumar and Y-J. Lin, "An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs," pp. 55-68 in *3rd Int. Confi on Logic Programming,* ed. E. Shapiro,Springer-Verlag, London, United Kingdom (1986).

[Kumarl986] V. Kumar and Y-J. Lin, "A Framework for Intelligent Backtracking in Logic Programs," pp. 108-123 in *6th Conf. on the Foundations of Software Technology and Theoretical Computer Science,* ed. K.V. Nori,Springer-Verlag, New Delhi, India (1986).

[Li 1986] P.P. Li and A.J. Martin, "The Sync model: A Parallel Execution Method for Logic Programming," pp. 223-234 in *Proceedings of the 3rd IEEE Symposium on Logic Programming,* Salt Lake City, Utah (1986).

[Pereiral986] L.M. Pereira, L. Monteiro, J. Cunha, and J.N. Aparicio, "Delta-Prolog: A Distributed Backtracking Extension with Events," pp. 69-83 in *3rd Int. Conf on Logic Programming,* ed. E. Shapiro,Springer-Verlag, London, United Kingdom (1986).

[Warrenl983] D.H.D. Warren, "An Abstract Prolog Instruction Set," Technical Note 309, SRI International, AI Center, Computer Science and Technology Division (1983).