AND-PARALLEL EXECUTION OF LOGIC PROGRAMS ON A SHARED MEMORY MULTIPROCESSOR: A SUMMARY OF RESULTS*

YOW-JIAN LIN AND VIPIN KUMAR

APRIL 1988 AI88-74

* This work was supported by Army Research Office grant #ARO DAAG29-84-K-0060 to the Artificial Intelligence Laboratory, and Office of Naval Research Grant N00014-86-K-0763 to the Computer Science Department at the University of Texas at Austin.

AND-parallel execution of Logic Programs on a Shared Memory Multiprocessor : A Summary of Results*

Yow-Jian Lin and Vipin Kumar

Artificial Intelligent Laboratory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

March, 1988

AI Lab TR88-74

Abstract

This paper presents the implementation of an AND-parallel execution model of logic programs on a shared-memory multiprocessor. The major features of the implementation are **(i)** dependency analysis between literals of a clause is done dynamically without incurring excessive run-time overhead; **(ii)** backtracking is done intelligently at the clause level without incurring any extra cost for the determination of the backtrack literal; **(iii)** the implementation is based upon the Warren Abstract Machine (WAM), hence retains most of the efficiency of the WAM for sequential segments of logic programs. Performance results on Sequent Balance 21000 show that our parallel implementation can achieve reasonable speedup on dozens of processors.

*This work was supported by Army Research Office grant # DAAG29-84-K-0060 to the Artificial Intelligence Laboratory, and Office of Naval Research Grant N00014-86-K-0763 to the computer science department at the University of Texas at Austin.

1 Introduction

Many different kinds of parallelism are present in logic programs[9]. AND-parallelism refers to executing more than one literal of a clause at the same time. Exploiting AND-parallelism is hard due to the possibilities of binding conflicts and backtracking. Among the schemes developed so far, some [34,5,37] abandon the backtracking feature of logic programming, and thus change the semantics (by excluding "don't know nondeterminism"). The scheme described in this paper is meant for logic programs with "don't know nondeterminism", and handles binding conflicts by detecting the dependency relations among literals.

A number of solutions have been proposed to determine the dependency between literals of a clause. The early solution proposed by Conery and Kibler [8] uses an ordering algorithm to determine dependencies at run time, but incurs substantial overhead. In response, other schemes were proposed by Chang, et al. [4] and DeGroot [11]. These schemes sacrifice the degree of parallelism to reduce the run-time overhead. In [28,31], we presented an execution model that uses tokens associated with shared variables to do the dependency analysis dynamically. In [28,31] we also showed that this model provides roughly the same degree of parallelism as Conery's Model, and provides more parallelism than the schemes of Chang, et al. [4] and DeGroot [11]. Our model also performs more accurate intelligent backtracking than the ones presented in [3,20]. This paper presents an implementation of (a slightly simplified version of) this execution model on Sequent Balance 21000, a shared-memory multiprocessor. The implementation is based upon the Warren Abstract Machine (WAM), hence it retains most of the efficiency of the WAM for sequential segments of programs. In this implementation, tokens are efficiently represented in terms of bit vectors. Since the information needed for intelligent backtracking is already maintained to perform dependency analysis, no extra overhead is incurred in the determination of the backtrack literal. Experimental results on Sequent Balance 21000 show that our parallel implementation can achieve reasonable speedup on dozens of processors.

Hermenegildo[17,18,20,19] proposed a WAM-based implementation of an extension of the execution model developed by DeGroot[11]. Borgwardt[1,2] proposed a stack-based implementation of the execution model developed by Chang, et al.[4]. To the best of our knowledge, our implementation is the first actual WAM-based implementation of an AND-parallel execution model on a parallel hardware. Other AND-parallel implementations are either process-based (e. g., PRISM[25]) or for committed choice languages (e. g., GHC[22], PARLOG[27] and Flat Concurrent PROLOG[35]).

2 The Execution Model

Our execution model consists of the following two algorithms: a *forward execution algorithm* which detects executable literals dynamically; and a *backward execution algorithm* which is executed when some literal fails. A detailed description of these two algorithms appears in [31]. Preliminary versions of forward execution and backward execution appear in [29] and [28], respectively.

Conceptually, our forward execution algorithm can be viewed as a token passing scheme. A

token is created for each variables that appears during the execution of each clause. Each newly created token (for a new variable V) is given to the leftmost (at the clause level) literal P which has V in its binding environment. A literal P is selected as the generator of V when it holds the token for V. A literal becomes executable when it receives tokens for all the uninstantiated variables in its current binding environment. Parallelism is exploited automatically when there are more than one executable literal in a clause.

Our backward execution algorithm performs intelligent backtracking at the clause level. Each literal P_i dynamically maintains a list of literals denoted as B-list(P_i). B-list(P_i) consists of those literals in the clause which may be able to cure the failure of P_i (if P_i fails) by producing new solutions. The literals P_k in each B-list are sorted according to the descending order of k. When a literal P_i starts execution, B-list(P_i) consists of those literals that have contributed to the bindings of the variables in the arguments of P_i. When P_i fails, P_j = head(B-list(P_i)) is selected as the backtrack literal. The tail of B-list(P_i) is also passed to P_j and merged into B-list(P_j) so that if P_j is unable to cure the failure of P_i, backtracking may be done to other literals in B-list(P_i).

After a backtrack literal P_j is chosen, we need to find an appropriate choicepoint in the proof tree rooted at P_j, and rollback the computation so that no invalid information is used in further execution. The rollback can be done in many different ways, which are discussed later in Section 4.

3 Implementing the Forward Execution Algorithm

A straightforward implementation of tokens in the WAM would require constructing linked lists to keep track of unbound variables in the binding environment of each literal. It will also require much dereferencing to check the variable bindings in order to update the lists of unbound variables. Moreover, after a failure has occurred, the rollback of computation will have to reconstruct the token lists (in addition to rewinding trail). All these together can easily ruin the efficiency of memory management in the WAM and impose substantial overhead. Since our objective is to detect the executable literals dynamically and yet efficiently, we have implemented a slightly modified version of the token passing scheme using bit vectors.

In the following discussion, a **nonground** binding is a term which contains some unbound variables, and a **ground** binding is a term without any unbound variable. Bindings of two variables X and Y are **nonground dependent** if they share an unbound variable; otherwise they are **nonground independent**.

3.1 The Bit-Vector Implementation

Let P_i denote the i-th literal (counting from left to right) in the clause body. We associate a bit vector with each shared variable V of a clause C. The length of the vector is equal to the number of literals in the clause body. The i-th bit (counting from the most significant bit) of each bit vector is 1 if P_i could contribute (or have contributed) binding to the current binding of V.[1] For example,

[1] In our implementation, we use a 32-bit word to represent every bit vector so that any clause body can consist of up to 32 literals. If a clause has more than 32 literals, then we break the clause into more than one clause each of

consider the following clause.

p0(X,Y) :- p1(X,Y), p2(X), p3(Y).

Suppose after the unification of P_0, X and Y are nonground and independent. Before the execution of the clause body begins, the bit vector of X is 110, which means that p1 and p2 can contribute to the binding of X; whereas the bit vector of Y is 101, which means that p1 and p3 can contribute to the binding of Y. A literal P_i is **executable** if for every shared variable V of P_i, there is no unsolved literal P_j ($j < i$) such that the j-th bit in the bit vector of V is 1. Clearly, if P_i is executable, then it is the generator of all the uninstantiated variables in its binding environment.

How do we check if a literal P_i is executable based on the bit vectors? For each literal P_i, we maintain a **bit-vector mask** which has 1 from bit 1 to bit $i-1$ and 0 everywhere else. For example, in the above clause, the bit-vector masks of p1, p2, p3 are 000, 100, 110, respectively. For each clause, we maintain a **finish vector** such that its j-th bit represents the execution status of P_j (0 means that the execution of P_j has succeeded). For example, just after the unification of p0, the finish vector of the above clause is 111. The executability of P_i can be determined by computing READY(V,P_i) for each shared variable V of P_i as follows:

Step 1. DD(V,P_i) $\leftarrow$ (bit vector of V) $\wedge$ (bit-vector mask of P_i)
Step 2. READY(V,P_i) $\leftarrow$ DD(V,P_i) $\wedge$ (finish vector)

If READY(V,P_i) $\neq 0$, then there is an unsolved literal P_j such that $j < i$, and the j-th bit of the bit vector V is 1. If READY(V,P_i) is 0 for every shared variable V of P_i, then P_i can be executed. In the above example, after the unification of p0, both READY(X,p2) and READY(Y,p3) are nonzero. Hence p2(X) and p3(Y) have to wait until p1(X,Y) has finished execution.

Clearly, related bit vectors need to be updated at the end of the execution of each literal to reflect the change of binding conditions. If a variable V is bound to a ground term after the execution of a literal P_i, then all the bits corresponding to P_j ($j > i$) in the bit vector of V are set to 0 (since P_j cannot contribute anything to a ground term). If two variables X and Y become dependent, then both bit vectors of X and Y are updated to be the logical OR of the original bit vector of X and that of Y. In the above example, if p1(X,Y) binds X and Y to ground terms, then after the execution of p1(X,Y) the bit vectors of both X and Y are modified to 100 (and hence both p2(X) and p3(Y) become executable). On the other hand, if p1(X,Y) makes X and Y nonground dependent, then the bit vectors of X and Y are updated to 111. In this case, only p2(X) becomes executable, and p3(Y) has to wait. Table 1 lists some possible situations and the corresponding bit vector changes. Symbols a_i and b_i can be either 0 or 1.

Note that we don't have to compute the bit vectors of different variables from scratch each time a clause is invoked (due to the unification of the head literal with a parent goal). At compile time, we compute bit vectors for each shared variable assuming that all variables in the head are nonground and independent. When the head literal is unified with the parent goal, the bit vectors of the variables in the head are modified according to Table 1.

which has less than 32 literals in its body.

Variable	Binding Condition	Bit Vector Before Update	Bit Vector After Update
X	ground	$a_1a_2a_3a_4$	a_1a_200
X	nonground independent	$a_1a_2a_3a_4$	$a_1a_2a_3a_4$
X	nonground dependent	$a_1a_2a_3a_4$	$a_1a_2a_3a_4 \vee b_1b_2b_3b_4$
Y	nonground dependent	$a_1a_2a_3a_4$	$a_1a_2a_3a_4 \vee b_1b_2b_3b_4$

Table 1 The rules for updating bit vectors after the execution of P_i (assuming the length of vectors is 4, and i = 2).

The most important advantage of this bit vector implementation is that the checking (of whether a literal is the generator of a shared variable) is simple (two bitwise AND operations, to be exact). Moreover, since at any given time the bit vector of V can be changed only by one processor (the one that just finished the execution of the generator of V), it is not necessary to lock these vectors before changing them. Also, the executability of different literals can be checked simultaneously by different processors without having to lock the vectors.

Compared with the token passing scheme, the date-dependency information in the bit vector approach is less precise in some cases. For example, consider the following clause.

p0(X,Y) :- p1(X,Y), p2(X), p3(Y), p4(X).

Suppose after the execution of p1, X is bound to f(R,W) and Y is bound to g(W,Q). Since these two bindings share a variable W, the bit vector of X and Y are updated to 1111. At this time only p2(X) can start executing. Assume that the execution of p2 binds W to a ground term c, and leaves R unbound (i.e., the binding of X becomes f(R,c)). Since the binding of X is still nonground, p2 does not change the bit vector of X. Now, even though X and Y are independent, the execution of p4 is still suspended. This happens because, in the bit vector implementation, the binding situations of the variables that are not originally in the clause (W, R and Q in this example) is not checked. This example shows that the bit vector implementation can be more conservative in detecting executable literals than the original token passing scheme. However, in many cases, this lost parallelism can be recovered using the technique described in [30].

3.2 Incorporating Information from the Programmer

It is worth pointing out that our execution model (and its bit-vector implementation) does not require program annotation. All the information needed by this scheme (such as the position of a literal in the clause, the appearance of a variable on a literal, etc.) are syntactic and can be accumulated easily during compile time. However, independence and ground checkings of variables at the end of the execution of literals could still be expensive. It is easy to incorporate the information provided by the user or by compile time analysis to reduce these checks and hence reduce the overhead. For example, if a ground binding is always imported to a shared variable X in a clause during the head unification, then no checking is necessary for any literal that accesses X. Also, if two variables are known not to become dependent any time, the independence checking between

these two variables can be omitted. Programmer can also mark those clauses that are known to result in sequential execution. We do not have to create parallel goals for such clauses. See [30] for more details.

3.3 Accumulating B-lists for Intelligent Backtracking

Recall from Section 3.1 that two masking steps are needed to check if a literal has the authority to generate bindings for a variable. If, for a literal P, the checking of a variable V succeeds in two steps, then the result of the first masking step, DD(V,P), actually represents the data dependency of P due to the variable V; i.e., P depends on every literal P_j such that j-th bit in DD(V,P) is 1. Clearly, the logical OR of DD(V,P) for each variable V in P gives us the initial B-list (represented as a bit vector) of P (if j-th bit of this vector is 1, then $P_j \in$ B-list(P)). The least significant 1 bit of a B-list vector represents the head of that B-list. The literals in B-list are automatically sorted, and two B-lists can be merged by a simple logical OR operation. This makes the implementation of clause-level backtracking presented in [29] very fast.

Consider the following clause:

p0(X,Y,Z) :- p1(X), p2(Y), p3(Y,Z), p4(X,Y,Z).

The bit-vector mask of p4(X,Y,Z) is 1110. Suppose that p0(X,Y,Z) is unified with p0(A,B,c). After the unification, the bit vector for X is 1001, for Y is 0111, and for Z is 0000. When both p1(X) and p2(Y) succeed and generate ground bindings, the bit vector for X is updated to 1000, and that of Y is updated to 0100. The finish vector becomes 0011. Since READY(X,p4), READY(Y,p4) and READY(Z,p4) are all 0, p4(X,Y,Z) becomes executable. At this moment DD(X,p4) = 1000, DD(Y,p4) = 0100, and DD(Z,p4) = 0000. Therefore the B-list vector of p4 is 1100, which means that p4(X,Y,Z) depends on only p1(X) and p2(Y), but not on p3(Y,Z) for this particular case. If p4(X,Y,Z) fails, then p2(Y) is chosen as the backtracking literal, and the remaining B-list vector (1000) is merged into the B-list vector of p2(Y) (= 0000); i.e., the new B-list vector of p2(Y) becomes 1000 (= 0000 $\vee$ 1000). Should p2(Y) fail later, p1(X) is chosen as the backtracking literal.

Note that when backtracking happens, these bit vectors need to be unwound to the value just before the execution of the backtracking literal. We use an extra bit-vector trail stack to keep the address and value of any bit vector that gets changed after the execution of a literal.

4 The Rollback of Computation

Once a literal fails and a backtrack literal P_j is chosen, the following two questions arise in any possible implementation:

- Which part of computation should be discarded?
- How to make sure that the job of discarding computation is carried out properly?

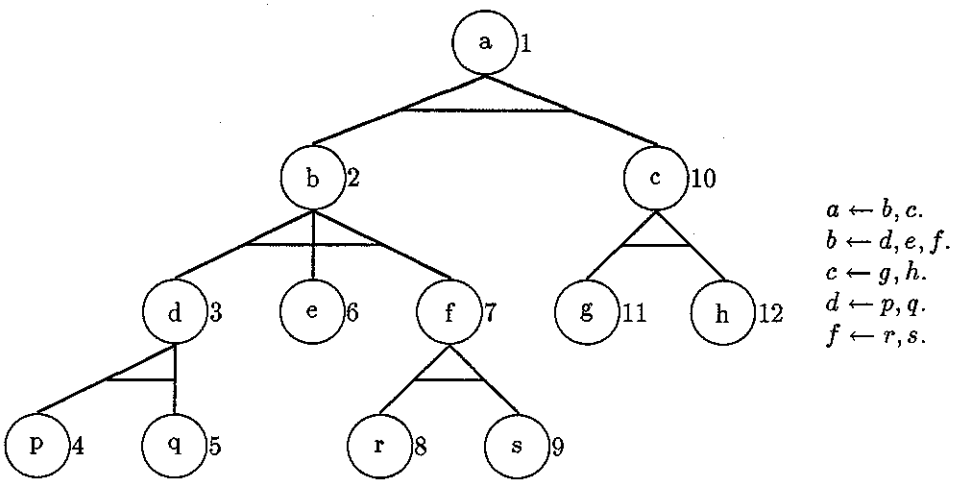


Figure 1 A proof tree

4.1 Which part of computation should be discarded?

Figure 1 represents a proof tree for a set of clauses given therein, where each goal is numbered according to the depth-first, left-to-right order. Suppose the execution of **e** failed and we choose **q** as the backtrack point. The simplest strategy is to discard every goal which is to the right of **q** (i.e., every goal whose number is greater than 5) — a **proof tree level discarding** strategy. However, if **b** and **c** can be executed independently, then we know that redoing **q** should not have any effect on the execution of **c**, **g**, and **h**. Therefore we can limit the discarding domain within the scope of **b**. Inside this discarding domain, we can do it conservatively by discarding all those goals in the scope of **b** whose number is greater than 5 — a **conservative clause level discarding**; or do it selectively by discarding only those goals which can 'possibly' be affected by the redoing of **q** — a **selective clause level discarding.**

The choice of the discarding strategy also determines what goals can be stacked on top of others. A goal G_i should be stacked on top of G_j only if backtracking to G_j requires discarding G_i. Thus a processor has to follow a **steal rule** to determine if, after finishing a sequential segment[2] of execution, it can take some goal from the goal list of any processor in the system (including that of itself). The importance of steal rule was independently recognized by Borgwardt [1,2] and Hermenegildo [17,19]. Hermenegildo also presented a detailed discussion of the problems (the trapped goal problem and the garbage slot problem) that may be encountered if a proper steal rule is not followed [19]. Of the three discarding strategies, the proof tree level strategy is identical to that presented in [19], whereas the selective clause level strategy is similar to that presented in [2].

In our implementation, we choose to perform conservative clause level discarding based upon the trade-off between saving repeated work and limiting the ability of picking up available goals. To

[2] As defined in [2], a **sequential segment** refers to a sequence of goals which a processor would have executed in sequential implementation without a break.

avoid the trapped goal problem and the garbage slot problem, we restrict our steal rule as follows:

> Suppose after the execution of a sequential segment, the last recorded goal at a processor CPU_m is G_i. CPU_m can take a goal G_j from any processor (including itself if its goal list is not empty) if
>
> 1. G_j is a right sibling of G_i or a descendant of any of those siblings; or
> 2. all the siblings of G_i have been solved successfully and G_j is a right sibling (or a descendant of any of those siblings) of the parent goal of G_i.
>
> The second criterion can be recursively applied to the ancestor of G_i. Any such G_j is then called an **available** goal for CPU_m at this time.

How restrictive this steal rule will be is another issue yet to be investigated. However, in our implementation, we can have more stacks than processors. Hence, if a processor is unable to pick up any goal due to this steal rule, then it may be able to start execution with a different stack.

4.2 The Labeling Scheme

In a sequential implementation, the depth-first, left-to-right ordering of goals in the proof tree is implicitly maintained by the physical address of each goal in the stacks. This makes the comparison of ordering between goals very efficient (i. e., just a simple address comparison). In a distributed stack implementation, since goals can be located in different stacks, physical addresses no longer reflect the ordering of goals. In order to enforce the steal rule mentioned in the previous section, we need to associate with each parallel goal (i. e., the literal in the body of a parallel clause[3]) a label. Since the proof tree is not complete until the execution terminates, we need an algorithm which can (based upon the chosen ordering criteria) always generate a "greater" value dynamically for a parallel goal appearing later in the ordering. One such algorithm is given in [30]. Hermenegildo presented another algorithm in [19]. Either algorithm is applicable to any implementation which requires dynamic labeling to determine the ordering between different goals. Note that if a program is deterministic, then all the parallel goals can be given identical labels. This means that any processor can execute any available goal.

4.3 How to discard the computation properly?

Once we have decided which part of computation should be discarded, we need to perform the rollback carefully. In the sequential implementation there is no fear of accessing invalid bindings, as (after deep backtracking has occurred) the forward execution does not resume until the rollback of computation is completed. In a parallel implementation also, if the computation affected by the backtracking resides only in one processor, then the rollback can be done just as it is done in a sequential execution. However, if the computation affected by backtracking has spread to several processors, then the rollback of computation would require the coordination of several processors.

[3] A **parallel clause** is a clause for which AND-parallel execution would be exploited.

To preserve the correctness of execution, a processor should resume forward execution only if it knows either that the rollback is completed or that any information that is created or accessed by this processor will not be canceled by the rollback of other processors (unless a new failure occurs).

One way of implementing the rollback is to send messages to the relevant processors to inform them that certain failure has occurred, and that they may have to clean up certain goals from their stacks. Note that many failures can happen at the same time. Therefore it is possible for one failure to be wiped out (before being processed completely) due to another failure. When a processor receives a message due to a failure, it has to know whether or not the failure has been wiped out by some other failure. This requires the use of time-stamps (discussed in [31]) which can be quite expensive to implement.

Another possibility is to just handle only one failure at a time, and start processing a new failure only after the previous one has been fully processed. This requires global synchronization among all the processors, which can be expensive if the number of processors is very large. Since our implementation is meant for a tightly coupled multiprocessor with a small number (dozens) of processors, we have chosen to implement the second alternative.

5 Implementation on Sequent Balance and Performance Results

5.1 The Overview

The APEX (**A**ND-**P**arallel **EX**ecution) is the implementation of our scheme on the Sequent Balance 21000 multiprocessor. This implementation, written in C, executes byte-code representation of the APEX instructions[4]. Before the execution begins, we specify the number of processors (p) and the number of DATA portions[5] (s), $s \geq p$, to be used in a particular run. We also specify the size of memory (m) for each DATA portion. Since the virtual space on Sequent Balance is only 16 Mbytes, $m \times s$ must be less than 16 Mbytes.

Associated with each DATA portion, there is a goal list for storing goals that can be picked up by any processor in the system. Each processor is first assigned a unique DATA portion to work with, and the remaining DATA portions (if $s > p$) are maintained in a spare list. Any time during the execution, there is a one-to-one mapping between the processors and the DATA portions which are not in the spare list. If a processor is unable to execute available goals on its current DATA portion, it exchanges its DATA portion with some other portion in the spare list[6].

One processor is selected to start the execution. Other processors become idle and look for available goals (using the steal rule stated in Section 4.1) from the goal list of any DATA portion in the system, including that of DATA portions in the spare list. When an idle processor succeeds

[4] The APEX instruction set is an extension of the WAM instruction set. The details regarding extended instructions are given in [30].

[5] Each DATA portion is a collection of heap, local stack, trail, goal list, etc.

[6] Another possible arrangement is to have a fixed number of DATA portions per processor. A processor can work only on its DATA portions, switching between them as necessary. This scheme (similar to the one presented by Hermenegildo in [17]) requires too many DATA portions, and does not use memory as effectively as the other scheme.

in stealing some goal, it starts execution. Any time when a processor encounters the execution of a parallel clause, it creates a special " clause frame" on its local stack, and then adds a "job frame" in the goal list of its DATA portion for every literal in the clause body (except the leftmost literal). It then continues its execution on the leftmost literal of that clause. After a processor P has finished executing some parallel goal, it tries to find an available[7] goal in the goal list of its own DATA portion. If an available goal is not present in its own DATA portion, then it starts polling the goal list of other DATA portions. If the polling is successful, then P starts executing the stolen goal. Otherwise, after polling for a certain amount of time, P exchanges its DATA portion with some portion in the spare list and resumes polling based upon the status of the new DATA portion. This steal-by-demand strategy releases the burden of a busy processor for distributing goals to other processors.

Note that precise checking of the type of a term, or checking the dependency between two terms can be expensive if the terms are large structures. To minimize the overhead, we have implemented a simple (but approximate) method proposed by DeGroot [11]. In this method all the ground terms appearing in the original program (including ground structures and ground lists) are tagged as constants so that checking the type of any such term at run time is very fast. Type-checking for other structures and lists is done conservatively (i.e., the arguments are checked only in the first level). Two nonground terms are considered dependent if any of them is a list or structure; or they are variables with the same address.

The implementation has been tested on many programs. Each Horn-clause program is first compiled into a WAM-code program using a modified version[8] of the Berkeley PLM compiler [38]. The APEX-code program is then constructed by adding our extended instructions for parallel execution to the WAM-code program[9]. Both the WAM-code and the APEX-code programs are then transformed into the byte-code representations to be executed by our implementation. The byte-code representation of the WAM-code program is run on one processor and one DATA portion to obtain the STIME shown in Table 2. This figure does not include any overhead due to parallel execution, and truly reflects the sequential execution time of the program. The byte-code representation of the APEX-code program is run on p processors and s DATA portions for different values of p and s. The timings for different programs is given in Table 2. In this table, PTIME(i) refers to the execution time of running the APEX-code program on i processors and i DATA portions. To compare the sequential speed with other implementations, we also list the time needed by Quintus PROLOG and SBProlog respectively to execute the same Horn-clause logic programs (in compiled mode) on SUN-3/50. Since SUN-3/50 is roughly three times faster than Sequent Balance, clearly the sequential execution of the APEX is competitive with SBProlog.

Among the programs tested, HANOI generates solution steps for a 15-disk '*Towers of Hanoi*'

[7] See Section 4.1 for definition.

[8] The main difference is that our version does not include cdr-coding. As stated in [36], in the absence of hardware support, cdr-coding does not result in an efficient implementation.

[9] Actually, a compiler can be developed that could generate the APEX-code programs for parallel execution directly from the Horn-clause programs.

	HANOI	MATRIX	QSORT	TAK	CDESIGN	IBTAK
Quintus*	7.92	11.60	2.95	1.64	0.85	8.10
SBProlog*	32.86	97.82	—	9.70	2.72	59.16
STIME†	115.65	237.84	6.17	19.40	5.55	114.73
PTIME(1)†	116.93	242.41	6.90	25.68	1.60	152.53
PTIME(2)†	58.56	122.50	3.89	13.08	1.29	76.88
PTIME(3)†	39.19	83.29	3.17	8.86	1.30	38.64
PTIME(4)†	29.46	62.97	2.54	6.70	1.31	30.62
PTIME(5)†	23.72	50.91	2.38	5.44	1.32	26.09
PTIME(6)†	19.74	42.73	2.23	4.58	1.33	21.00
PTIME(7)†	17.06	37.15	2.13	4.11	1.10	15.54
PTIME(8)†	14.89	32.84	1.98	3.64	1.37	16.31
PTIME(9)†	13.33	29.60	1.93	3.31	1.16	15.73
PTIME(10)†	12.10	26.92	1.91	3.02	1.41	12.60
PTIME(11)†	11.01	24.71	1.87	2.96	1.44	12.45
PTIME(12)†	10.19	22.91	1.84	2.70	1.42	11.75
PTIME(13)†	9.43	21.38				
PTIME(14)†	8.81	20.10				
PTIME(15)†	8.31	18.91				
PTIME(16)†	7.82	17.98				
PTIME(17)†	7.36	17.21				
PTIME(18)†	7.05	16.24				
PTIME(19)†	6.68	15.65				
PTIME(20)†	6.47	14.97				

* (on SUN 3/50 — 1.5 MIPS)
† (on Sequent Balance 21000 – 0.5 MIPS)

Table 2 Performance Results on Sequent Balance 21000 (all the numbers are in seconds)

problem; MATRIX, given in [6], multiplies two 50 x 50 matrices; QSORT, taken from [15], executes '*quicksort*' to sort a list of 511 numbers; TAK is a program for computing the function '*takeuchi*(15,10,5)'[10]; CDESIGN is the circuit design program given in [15,26]. IBTAK is a program (built on top of the Takeuchi function) which could take advantages of both AND-parallel execution and intelligent backtracking. The listings of HANOI, TAK and IBTAK are given in Appendix A. In each program, parallelism is exploited only on selected clauses. For the first four programs (HANOI, MATRIX, QSORT and TAK), we also make use of the fact that they are all deterministic programs and hence generate the same label for all the parallel goals (See Section 4.2).

5.2 The Overheads

The overheads due to parallel execution in the APEX can be categorized into five groups.

1. The overhead due to the creation of new data objects such as clause and job frames.
2. The overhead due to updating information in those new data objects.
3. The overhead due to manipulating bit vectors. This consists of (i) checking bit vectors to decide if a goal is executable, and (ii) updating bit vectors to reflect the change in binding conditions after a goal has finished execution.
4. The overhead of polling DATA portions to steal goals.
5. The overhead due to backward execution coordination. This consists of the synchronization overhead, the reset_cancel overhead and the clean_up overhead.

For deterministic programs, the difference between PTIME(1) and STIME is only due to the first three kinds of overhead. Note that the sum of the first two kinds of overheads is usually proportional to the number of frames created for parallel execution. Since we know the number of frames created for the first three programs in Table 2, we are able to estimate that creating and manipulating each frame costs around 0.3 ms overhead. Clearly, for good performance, the parallel activities should have granularity larger than 0.3 ms.

It is very hard to estimate the other three kinds of overheads, as they are dependent on run-time situation. The overhead of manipulating bit vectors varies from one program to another. But in many cases it can be minimized by a compile-time analysis (see Section 3.2). In the first five benchmark programs, this overhead is minimal because of such analysis. The overhead of polling increases with the number of processors and the number of DATA portions. The relationship between the backward-execution-coordination overhead and the number of processors p is not so clear-cut. When p goes up, the synchronization overhead goes up, the reset_cancel overhead remains roughly the same, and the clean_up overhead may even go down depending on how well the work of clean_up is distributed.

[10] Takeuchi function is a simple benchmark that Ikuo Takeuchi of Japan used for Lisp.

5.3 The Benchmarks

In this section we analyze the performance of the APEX on each benchmark. For deterministic programs, the APEX can only speed up the execution by executing independent literals in parallel. For nondeterministic programs, both AND-parallel execution and intelligent backtracking mechanisms of the APEX can potentially reduce the execution time, which can even result in superlinear speedups.

5.3.1 Deterministic Programs

'*Towers of Hanoi*' is a typical divide-and-conquer problem. It is suitable for AND-parallel execution, as its execution tree is well balanced. However, the parallel activities will be too fine-grain if the granularity is not controlled. To avoid creating activities of small granularity, HANOI is coded in such a way that the execution becomes sequential when problem size[11] drops below 7. The solution steps are accumulated in a tree structure and are printed at the end of computation. The timing shown in Table 2 does not include printing time. Clearly, the APEX is able to achieve almost linear speedup on HANOI for twenty processors.

Although logic programming is not particularly suited for numerical computation, we choose '*matrix multiplication*' as a benchmark simply because that it has been used by many other researchers [7,17]. MATRIX contains a long sequential segment in the beginning of the computation (for constructing a 50×50 matrix and transposing a copy of it). When the number of processors increases, so does the influence of the sequential segment over the overall performance. Although the execution tree skews to the right, it does not have large impact on the speedup. On twenty processors, the APEX can achieve a speedup of 16.

'*Quicksort*' is another divide-and-conquer problem. It differs from the '*Towers of Hanoi*' problem in the sense that it needs a long computation (which has $O(n)$ complexity, where n is the length of the list to be sorted) to split the problem into two subproblems. Since the total computation is $O(n \log n)$, the speedup can be no more than $O(\log n)$. Moreover, the splitting may result in an unbalanced execution tree. For these reasons we do not expect the APEX (or any other parallel implementation) to achieve good speedup on this problem. On the 511-element list (which was chosen to avoid the effect of unbalanced tree), the APEX is able to get roughly three times speedup on eight processors. In this case, no effort was made to avoid creating small granularity tasks.

The execution tree of '*takeuchi*' benchmark is somewhat different from the other three benchmarks discussed above. After the 'takeuchi' procedure is called at the top level, the number of parallel activities grows rapidly, and then shrinks to zero at one point before the same procedure is called recursively with different arguments. This means that processors could spend more time idling. This explains the poor performance of TAK on large number of processors.

Note that all the APEX-code programs of the benchmarks discussed so far have been optimized for deterministic computation. As discussed in Section 4.2, all the parallel literals created will have the same label. Hence parallelism is not restricted by the steal rule at all. To show how the steal

[11] The problem size is given by the number of disks to move.

	I	II	III
PTIME(1)	25.68	24.57	—
PTIME(2)	13.08	14.41	12.97
PTIME(3)	8.86	9.48	9.21
PTIME(4)	6.70	8.93	7.50
PTIME(5)	5.44	8.92	6.63
PTIME(6)	4.58	7.91	5.69
PTIME(7)	4.11	7.32	6.09
PTIME(8)	3.64	6.88	5.55
PTIME(9)	3.31	5.92	5.36
PTIME(10)	3.02	5.48	4.69
PTIME(11)	2.96	5.20	5.01
PTIME(12)	2.70	4.49	4.61

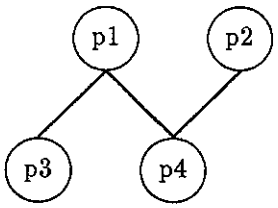
Table 3 The Effect of Code Optimization and Spare DATA Portions on Deterministic Programs

rule can affect the performance, we executed the same 'takeuchi' program without optimizing the compiled code. In Table 3, column I is the execution time of running 'takeuchi' program with determinism optimization, whereas column II is the result of running the same program without such optimization. It is not surprising to see that the performance becomes worse. Also, adding an extra processor is not guaranteed to improve the performance (e. g., compare the values of PTIME(4) and PTIME(5) in column II). Clearly, the performance is dependent upon how often processors get stuck due to the steal rule at run time.

To reduce the influence of the steal rule, our implementation permits more DATA portions than the number of processors. Column III in Table 3 shows the results of the APEX running the nonoptimized 'takeuchi' program with $s = 2 \times p$. We can see that the performance improves when the number of processors is small. But due to the polling overhead, the improvement becomes insignificant or even negative (e. g., see the value of PTIME(12) in columns II and III). We argue that the possibility of getting stuck due to the steal rule decreases when the number of processors increases. Therefore the need for extra DATA portions decreases as well.

5.3.2 Nondeterministic Programs

As pointed out by Fagin in [15], the '*circuit design*' program does not have much AND-parallelism, but has much room for performance improvement due to intelligent backtracking. Our results in Table 2 on CDESIGN verify this. In fact, despite the overheads of parallel execution, the APEX achieves more than three times speedup using just one processor (STIME/PTIME(1) > 3). This shows that even without exploiting AND-parallelism, the intelligent backtracking scheme of the APEX can improve the execution performance of nondeterministic programs. On two processors, the speedup is improved to more than four times. Beyond that, the speedup saturates. There are several possible explanations for the saturation, but we suspect that it is mainly caused by the lack of AND-parallel activities. To examine the performance of the APEX on programs with potential



(b)

```
p0(X,Y) :- p1(X), p2(Y), p3(X), p4(X,Y).
```

(a)

Conditional Graph Expression:

```
p0(X,Y) :- (ground([X,Y]) | (indep([X,Y]) | p1(X) & p2(Y))
                                & (ground(X) | p3(X) & p4(X,Y))).
```

(c)

Figure 2 An Example Clause, its Dependency Graph, and a CGE

of both AND-parallelism and intelligent backtracking, we constructed the IBTAK program (see Appendix A.3).

IBTAK contains a clause that can potentially benefit from intelligent backtracking. The data dependency graph of that clause is similar to the one shown in Figure 2b. The first two literals in the clause body call the 'takeuchi' procedure with different set of arguments to generate numbers. The last two literals test those numbers to see if they are satisfiable. IBTAK is essentially a collection of several 'takeuchi' calls. Hence, in absence of intelligent backtracking and the backward execution coordination overhead, we should expect the speedup performance of IBTAK to be similar to that of TAK. Therefore by comparing the performance results of running IBTAK and TAK, we can see how the gain from intelligent backtracking and the overhead for the backward execution coordination balance each other. From Table 2, the ratio (PTIME(1) for TAK)/(PTIME(1) for IBTAK) is about the same as (STIME for TAK)/(STIME for IBTAK). This indicates that the gain due to intelligent backtracking in IBTAK is nullified by the overhead of backtracking. While running the same program with more processors, we observe dramatic performance improvement, or even super-linear speedup (see PTIME(7)). This is due to the early detection of unsatisfactory bindings in parallel execution which in turn reduces the overhead of creating frames, updating, and cleaning up.

6 Related Works

6.1 Borgwardt's Scheme

Borgwardt [1] proposed the first stack-based execution scheme that exploits AND, OR and stream parallelism in the execution of logic programs on a shared-memory multiprocessor. The execution model for AND-parallelism in his model is the one developed by Chang, et al.[4]. In [2], Borgwardt presents a distributed implementation of the semi-intelligent backtracking scheme of Chang, et al.[3].

There are several problems with the discarding mechanism in Borgwardt's scheme[2]. He cancels only dependent forward siblings on type I backtracking. This can result in incorrect computation. For example, given data dependency graph of Figure 2(b), it is possible that the first solution generated by p1 could cause both p3 and p4 to fail. Suppose p4 fails first and initiates type I

backtracking to p2[12]. Since p3 is not a dependent forward sibling of p2, p3 is not canceled. Later when p3 fails and causes a type I backtracking to p1, canceling only p3 and p4 (as Borgwardt has proposed) without restoring the state of p2 will prevent p4 from trying the combination of the second solution from p1 and the first solution from p2.

Another problem is due to his use of partially distributed coordinating mechanism. As discussed in Section 4.1 this type of coordinating system does not work in general unless some sort of time-stamp mechanism is incorporated. Consider the execution of the clause in Figure 2(a). Suppose the dependence relations between those literals are as shown in Figure 2(b). It is possible that both p3 and p4 fail almost at the same time. Assume that the following scenario happens in sequence:

- p3 fails and asks p1 to redo.
- p1 receives the request and asks p2, p3, and p4 to rollback.
- Before p4 receives the rollback request, p4 fails and asks p2 to redo.
- p2, p3, and p4 receive and acknowledge the rollback message.
- p1 receives rollback-complete messages from all of the literals. It tells every one to resume execution.
- p2 finishes its execution and, at this time, receives the redo request from p4.

Clearly, the redo request p2 just received should have been wiped out earlier by the redo at p1. But, in the absence of time-stamps, we are unable to tell when the request was sent, and hence cannot prevent the extra redo action from happening.

6.2 Hermenegildo's Abstract Machine

In [17,18,20,19], Hermenegildo developed a WAM-based implementation of an extended version of DeGroot's RAP. In [21], Hermenegildo and Tick presented simulation results of memory referencing characteristics of RAP-WAM on a shared memory multiprocessor architecture.

A limitation of CGE (Conditional Graph Expression) is that it adds spurious dependencies between literals causing loss of concurrency. For example, the CGE of Fig 2(c) adds an extra dependency between p2 and p3. Hence, p3 cannot start execution until p2 has finished. Also, if p3 fails, it has to backtrack to p2, where a more precise choice would be p1. Any other way of writing CGE for this clause would add some other unnecessary dependency between literals. As discussed in [28] and [31], these drawbacks are not shared by our execution model.

One advantage of Hermenegildo's implementation of RAP is that a parallel goal is created only if it can be executed in parallel. For example, in the clause of Fig 2, if X and Y are dependent then in Hermenegildo's implementation p1 and p2 are executed just as in the sequential execution. In our scheme, once the clause has been tagged as a parallel clause, a parallel goal is created for every literal in the clause.

[12] Without cause-of-failure analysis, p4 cannot backtrack directly to p1.

6.3 Fagin and Despain's PPP

Fagin and Despain [14] have developed PPP as an extension of PLM [13] to incorporate AND-parallelism, OR-parallelism and intelligent backtracking. They use static data-dependency analysis scheme of Chang, et al.[4,3] to detect AND-parallelism and to perform intelligent backtracking.

Fagin and Despain present extensive simulation results of the performance of PPP on a variety of benchmark programs. In most of these programs, the performance improvement due to parallel execution is very limited. It is not clear whether the lack of good performance is due to a weak AND-parallel execution model, due to the overheads of their implementation assumed in the simulation, or due to the lack of parallelism in the programs tested.

7 Concluding Remarks

This paper presented the implementation of an AND-parallel execution model on a shared-memory architecture. Our experimental results have shown that it is possible to perform data-dependency analysis dynamically without incurring excessive run-time overhead. Since the implementation is WAM based, we are able to retain the execution efficiency of WAM for sequential segments of the execution. This was crucial to obtain performance improvement over sequential implementations.

Although our results are very encouraging, we need to perform a more thorough evaluation by testing its performance on a variety of programs. In particular, we would like to find the classes of programs that can benefit from AND-parallel execution and intelligent backtracking. We would also like to test the suitability of different steal rules and discarding strategies.

The parallel execution scheme presented in this paper deals only with pure Horn-Clause logic programs. Most practical logic programs contain non-logical constructs such as CUT, assert, retract, etc.. DeGroot has recently presented an approach to handle side-effects in the Restricted AND-Parallelism scheme [12]. This approach can be adopted to apply to our execution model.

A major drawback of WAM-based parallel implementations is that they require the target architecture to have a large shared memory equally (and rapidly) accessible to all the processors (e. g., as in Sequent Balance multiprocessor). This kind of tightly-coupled architectures can be built only for a small number of processors. Hence if parallelism is to be exploited at a large scale (several orders of magnitude), then some sort of deviation from WAM appears necessary. Process based models do not require a shared-memory multiprocessor system and can be implemented on a message passing architecture (e. g., the Hypercube [33]) which are much more scalable. Hence process oriented implementations do hold great promise and have been investigated by many researchers [10,16,32,23,39,24]. In fact, both WAM based and process based implementations can be used together in a multiprocessor that has a large number of loosely coupled clusters, each cluster having many tightly coupled processors. In such a system, a process based model can be implemented on the top level, and a WAM based model can be implemented within each cluster.

Acknowledgement

The authors would like to thank Manuel Hermenegildo for many useful discussions.

Appendix

A The Listing of Benchmarks

A.1 HANOI

```
% generate solution steps for 15-disk "towers of hanoi" problem
goal :- hanoi(15,R),write(R).

hanoi(N,R) :- move(N,left,center,right,R).

% for parallel computation
move(N,A,B,C,R) :- N < 7, !, move1(N,A,B,C,R,[ ]).
% the parallel clause
move(N,A,B,C,[R1,movedisk(A,B),R2]) :- M is N-1, move(M,A,C,B,R1), move(M,C,B,A,R2).

% for sequential computation
move1(0,_,_,_,R,R) :- !.
move1(N,A,B,C,RO,RI) :- M is N-1, move1(M,C,B,A,RT,RI), move1(M,A,C,B,RO,[movedisk(A,B)|RT]).
```

A.2 TAK

```
% compute the function takeuchi(15,10,5)
goal :- tak(15,10,5,X),write(X).

tak(X,Y,Z,W) :- X > Y, !, tak2(X,Y,Z,X1,Y1,Z1), tak(X1,Y1,Z1,W).
tak(_,_,Z,Z).

% the parallel clause
tak2(X,Y,Z,X1,Y1,Z1) :- tak1(X,Y,Z,X1), tak1(Y,Z,X,Y1), tak1(Z,X,Y,Z1).

tak1(X,Y,Z,W) :- X1 is X-1, tak(X1,Y,Z,W).
```

A.3 IBTAK

```
% the test program for both AND-parallel execution and intelligent backtracking
goal :- p(5,10,15,W), write(W).

% The main clause for intelligent backtracking.
p(X,Y,Z,W) :- p1(X,Y,Z,K), p1(Z,Y,X,W), p2(K), p3(K,W).

p1(X,Y,Z,W) :- tak(X,Y,Z,W).
p1(X,Y,Z,W) :- tak(Y,X,Z,W).
p1(X,Y,Z,W) :- tak(Z,Y,X,W).
p1(X,_,_,X).

% The procedure for computing "tak" is the same as that in TAK.

p2(2).    p2(5).    p2(8).

p3(X,Y) :- X =< Y.
```

References

[1] P. Borgwardt. Parallel prolog using stack segments on shared-memory multiprocessors. In *Proceedings of International Symposium on Logic Programming*, pages 2–11, IEEE Computer Society Press, February 1984. Atlantic City.

[2] P. Borgwardt and D. Rea. Distributed semi-intelligent backtracking for a stack-based AND-parallel prolog. In *Proceedings of the Third Symposium on Logic Programming*, pages 211–222, IEEE Computer Society, September 1986. Salt Lake City, Utah.

[3] J.-H. Chang and A. M. Despain. Semi-intelligent backtracking of prolog based on a static data dependency analysis. In *Proceedings of IEEE Symposium on Logic Programming*, pages 43–70, August 1985.

[4] J.-H. Chang, A. M. Despain, and D. Degroot. AND-parallelism of logic programs based on static data dependency analysis. In *Proceedings of the 30th IEEE Computer Society International Conference*, pages 218–226, February 1985.

[5] K. L. Clark and S. Gregory. *PARLOG: A Parallel Logic Programming Language.* Technical Report, Department of Computing, Imperial College of Science and Technology, London, May 1983.

[6] J. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, 1987.

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

[8] J. S. Conery and D. F. Kibler. AND parallelism and nondeterminism in logic programs. *New Generation Computing*, 3(1):43–70, 1985.

[9] J. S. Conery and D. F. Kibler. Parallel interpretation of logic programs. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 163–170, ACM, October 1981.

[10] John S. Conery. Binding environments for parallel logic programs in nonshared memory multiprocessors. In *Proceedings of the Fourth Symposium on Logic Programming*, pages 457–467, 1987.

[11] D. DeGroot. Restricted AND-parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 471–478, Tokyo, 1984.

[12] D. DeGroot. Restricted AND-parallelism and side effects. In *Proceedings of the Fourth Symposium on Logic Programming*, pages 80–89, 1987. San Francisco, CA.

[13] T.P. Dobry, Alvin M. Despain, and Y.N. Patt. Performance studies of a prolog machine architectue. In *12th International Symposium on Computer Architecture*, June 1985.

[14] Barry S. Fagin and Alvin M. Despain. Performance studies of a parallel prolog architecture. In *14th International Symposium on Computer Architecture*, 1987.

[15] Barry Steven Fagin. *A Parallel Execution Model for Prolog.* PhD thesis, University of California, berkeley, November 1987.

[16] A. Goto and S. Uchida. *Toward a High Performance Parallel Inference Machine—The Intermediate Stage Plan of PIM.* Technical Report TR-201, ICOT, Tokyo, Japan, September 1986.

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

[18] M. V. Hermenegildo. An abstract machine for restricted AND-parallel execution of logic programs. In *Proceedings of the Third International Conference on Logic Programming*, pages 25–40, Springer-Verlag, 1986.

[19] M. V. Hermenegildo. Relating goal-scheduling, precedence, and memory management in AND-parallel execution of logic programs. In *Proceedings of the FourthInternational Conference on Logic Programming*, pages 556–576, The MIT Press, 1987.

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

[21] Manuel Hermenegildo and Even Tick. *Performance Evaluation of the RAP-WAM Restricted AND-Parallel Architecture on Shared Memory Multiprocessors.* Technical Report PP-085-87, MCC, 1987.

[22] N. Ichiyoshi, T. Miyazaki, and K. Taki. A distributed implementation of flat GHC on the multi-PSI. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 257–275, The MIT Press, May 1987. Melbourne, Australia.

[23] L. V. Kale. *Parallel Architectures for Problem Solving.* PhD thesis, Computer Science Department, SUNY at Stony Brook, December 1985.

[24] L. V. Kale. The REDUCE-OR process model for parallel evaluation of logic programs. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 616–632, The MIT Press, May 1987. Melbourne, Australia.

[25] S. Kasif, M. Kohli, and J. Minker. *PRISM: A Parallel Inference System for Problem Solving.* Technical Report, Computer Science Department, University of Maryland, February 1983.

[26] V. Kumar and Y.-J. Lin. An intelligent backtracking scheme for prolog. In *Proceedings of the Fourth Symposium on Logic Programming*, pages 406–414, September 1987. San Francisco, CA.

[27] Melissa Lam and Steve Gregory. PARLOG and ALICE: a marriage of convenience. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 294–310, The MIT Press, May 1987. Melbourne, Australia.

[28] Y.-J. Lin and V. Kumar. A parallel execution scheme for exploiting AND-parallelism of logic programs. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 972–975, August 1986. St. Charles, Illinois.

[29] 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 55–68, June 1986. London, England.

[30] Yow-Jian Lin. *A Parallel Implementation of Logic Programs.* PhD thesis, Department of Computer Sciences, The University of Texas at Austin, in preparation. Austin, Texas.

[31] Yow-Jian Lin and Vipin Kumar. An execution model for exploiting AND-parallelism in logic programs. *New Generation Computing*, 5(4):393–425, 1988.

[32] M. Sato, H. Shimizu, A. Matsumoto, K. Rokusawa, and A. Goto. KL1 execution model for PIM cluster with shared memory. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 338–355, The MIT Press, May 1987. Melbourne, Australia.

[33] C.L. Seitz. The cosmic cube. *Communication of ACM*, 28(1):22–33, 1985.

[34] E.Y. Shapiro. *A Subset of Concurrent Prolog and its Interpreter.* Technical Report TR-003, ICOT, Tokyo, Japan, January 1983.

[35] S. Taylor, S. Safra, and E. Shapiro. *A Parallel Implementation of Flat Concurrent Prolog.* Technical Report, Department of Computer Science, The Weizmann Institute of Science, Rehovot, 1986.

[36] Hervé Touati and Alvin Despain. An empirical study of the Warren abstract machine. In *Proceedings of the Fourth Symposium on Logic Programming*, pages 114–124, 1987. San Francisco, CA.

[37] K. Ueda. *Guarded Horn Clauses.* PhD thesis, University of Tokyo, March 1986.

[38] P. van Roy. *A Prolog Compiler for the PLM.* Technical Report UCB/CSD 84/203, Computer Science Division (EECS), University of Berkeley, 1984.

[39] Nam S. Woo and R. Sharma. An AND/OR parallel execution system for logic program evaluation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 162–165, 1987.