PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS 1988, edited by ICOT. © ICOT, 1988

PERFORMANCE OF AND-PARALLEL EXECUTION OF LOGIC PROGRAMS ON A SHARED-MEMORY MULTIPROCESSOR*

Yow-Jian Lin† and Vipin Kumar

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

ABSTRACT

This paper presents the performance results of the implementation of an AND-parallel execution model of logic programs on a shared-memory multiprocessor. The execution model is meant for logic programs with "don't-know nondeterminism", and handles binding conflicts by dynamically detecting dependencies among literals. The model also incorporates intelligent backtracking at the clause level. Our implementation of this model is based upon the Warren Abstract Machine (WAM); hence it retains most of the efficiency of the WAM for sequential segments of logic programs. Performance results on Sequent Balance 21000 show that on suitable programs, our parallel implementation can achieve linear speedup on dozens of processors. We also present an analysis of different overheads encountered in the implementation of the execution model.

1 INTRODUCTION

The high-performance requirement of future computing systems is a major force behind the interest in parallel execution of logic programs. Many different kinds of parallelism are present in logic programs[8]. 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. Many schemes for exploiting AND-parallelism have abandoned the backtracking feature of logic programming, and thus changed the semantics by excluding "don't-know nondeterminism" (e.g., PARLOG [5], Concurrent Prolog [27], GHC [30]). These schemes also require the programmer to explicitly specify dependencies between literals (via read-only annotations or mode declarations) to deal with binding conflicts. The scheme described in this paper is meant for logic programs with "don't-know nondeterminism", and handles binding conflicts by dynamically detecting the dependencies among literals.

A number of solutions have been proposed to determine dependencies among literals of a clause. The early solution proposed by Conery and Kibler [9] 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 [10]. These schemes sacrifice the degree of parallelism to reduce the run-time overhead. In [22, 24], we presented an execution model that uses tokens associated with shared variables to do the dependency analysis dynamically. In [22, 24] we also showed that this model exploits roughly the same degree of parallelism as Conery's Model, and provides more parallelism than the schemes of Chang, et al. [4] and DeGroot [10]. Our model also performs more accurate intelligent backtracking than the ones presented by Chang, et al. [3] and Hermenegildo, et al. [17]. In [23], we presented a bit-vector implementation of (a slightly simplified version of) the token-based execution model. In this implementation tokens are represented in terms of bit-vectors, which allows dynamic detection of dependencies at the cost of a few logical operations on 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.

The goal of any parallel implementation for executing logic programs is to gain speedup over the best sequential implementation. The Warren Abstract Machine (WAM) has been recognized as the fastest and the most efficient sequential implementation for years [32]. An implementation which is significantly different from the WAM can be an order of magnitude slower. As advocated by Hermenegildo[13, 15], it is important to incorporate AND-parallelism in the WAM in such a way that most of its memory management efficiency and performance optimizations are retained. We have incorporated our bit-vector implementation in the WAM, and tested its performance on Sequent Balance 21000, a shared-memory multiprocessor. Experimental results show that, for suitable programs, our parallel implementation can achieve linear 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 Department of Computer Sciences at the University of Texas at Austin.

†New Address: Bellcore, Morristown, New Jersey 07960-1910

Hermenegildo[14, 15, 17, 16] proposed a WAM-based implementation of an extension of the execution model developed by DeGroot[10]. 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[19]) or for committed choice languages (e. g., GHC[18], PARLOG[21] and Flat Concurrent PROLOG[28]).

2 THE EXECUTION MODEL AND ITS BIT-VECTOR IMPLEMENTATION

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 [24]. Preliminary versions of forward execution and backward execution algorithms appear in [22] and [25], respectively.

Conceptually, our forward execution algorithm can be viewed as a token passing scheme. A token is created for each variable 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. Let P_i represent the i-th literal in a clause. 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).

A straightforward implementation of tokens in the WAM would interfere with 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.

Let P_i denote the i-th literal (counting from left to right) in the clause body. In the bit-vector implementation, we associate a bit-vector with each shared variable V of a clause C. The length of the bit-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) to the current binding of V. A literal P_i is considered to have the token for a shared variable V if there is no unsolved literal P_j ($j < i$) such that the j-th bit in the bit-vector of V is 1. For example, consider the following clause.

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

Suppose after the unification of p0, X and Y are non-ground 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. Conceptually, p1 has tokens for both X and Y at this moment.

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.

This bit-vector implementation of tokens makes it possible to check the executability of a literal at the expense of a small number of logical operations. It is also possible to maintain B-list for each literal as a bit-vector (if a bit is 1, then the corresponding literal is on the B-list). This representation makes manipulation of B-lists very efficient (e.g., merging two B-lists is a simple bitwise OR operation on two bit-vectors). Details of the bit-vector implementation are given in [23, 26].

Although the implementation of tokens using bit-vectors is quite efficient, independence and ground checkings of variables at the end of the execution of literals could be expensive (especially if the variables are bound to large structures). 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. We will refer to such an "optimization" as **the dependency-check optimiza-**

tion. 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 [26] for more details.

If the dependency-check optimization is not applicable, and if the variables are bound to large structures, then a simple (but approximate) method proposed by DeGroot [10] can be used. In this method all the ground terms (including ground structures and ground lists) appearing in the original program 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 a structure; or they are variables with the same address.

3 INTEGRATING THE EXECUTION MODEL IN WAM

This section discusses how our execution model is incorporated in the WAM in such a way that most of WAM's memory management efficiency and performance optimizations are retained. Many issues discussed in this section are common to the incorporation of any AND-parallel execution model in the WAM, and were previously discussed by Hermenegildo [14, 16] and Borgwardt [1, 2].

3.1 The Configuration

The execution of logic programs in the WAM is a sequence of steps manipulating data objects in the DATA space consisting of HEAP/STACK/TRAIL/PDL[32]. In a multiprocessor system the DATA space (stacks) of the WAM is distributed to all the processors. Figure 1 shows a possible "multiple-WAM" configuration on a shared memory multiprocessor (e.g., Sequent Balance 21000). In this configuration, all the processors share the same copy of compiled program in the CODE space. The DATA space is divided into m smaller portions, denoted as $\text{DATA}_1, \ldots, \text{DATA}_m$. Each DATA_i consists of its own HEAP/STACK/TRAIL/PDL and some additional areas, including a JOB area for recording the description of every job (an unsolved literal) to be picked up by idle

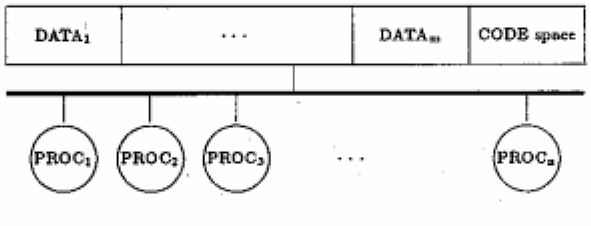

Figure 1 A Configuration of Multiple WAM

processors.[1] The number of DATA portions (m) should be greater than or equal to the number of processors (n) in the system so that at any time each processor can have exclusive use of a certain portion. Each processor is first assigned a unique DATA portion to work with, and the remaining DATA portions (if $m > n$) 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. On sequential parts of the program, the execution of a processor on any of the DATA portions is identical to WAM. Any time when a processor encounters the execution of a parallel clause, it creates a special "clause frame" on its STACK, and then adds a "job frame" in the JOB area 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.

This configuration is similar to the one presented by Hermenegildo[14]. As discussed in [14], it is important (for efficiency as well as correctness) that each DATA portion is associated with a JOB area, and that these JOB areas are maintained as stacks.

3.2 The Rule for Stealing

In the WAM, backtracking results in discarding of some computation as well as recovery of memory. In the multiple-WAM, failure in one processor can require discarding of computation in other processors, as the computation of many processors can be related to each other. To make sure that the memory-recovery techniques of the WAM remain applicable, it may be necessary to disallow the execution of certain jobs on certain DATA portions. In other words, a processor may not be allowed to execute an available job upon a DATA portion if it may interfere with memory recovery during backtracking [13, 16, 1, 2]. The importance of the steal rule (which specifies whether a job can be executed upon a DATA portion) was independently recognized by Borgwardt [1, 2] and Hermenegildo [13, 14, 16]. Hermenegildo also presented a detailed discussion (and possible solutions) of the problems (the *trapped goal* problem and the *garbage slot* problem), that may be encountered if a proper steal rule is not followed [16]. There is a strong connection between the steal rule and the chosen strategy for discarding computation during backtracking. In [26], we have categorized different discarding strategies and corresponding steal rules, and have presented the discarding strategy (and the steal rule) used in our implementation. All the steal rules use the depth-first left-to-right ordering among subgoals in the proof tree to determine whether a job can be scheduled upon a DATA portion.

[1] A detailed description of these data structures is presented in [26].

A parallel goal is considered **available** to a processor PROC$_i$ if it can be executed on the current DATA portion of PROC$_i$ according to the steal rule. Because of the restriction imposed by the steal rule, it is possible that none of the executable goals are available to a process PROC$_i$ (i.e., none of them can be executed on the current DATA portion of PROC$_i$). In this case, PROC$_i$ is forced to wait until one of the goals that is available to it becomes executable. If the number of DATA portions (m) is larger than the number of processors (n), then PROC$_i$ can also switch its current DATA portion (with another that is not currently assigned to any other processor) and see whether an executable goal is available with respect to the new DATA portion.

3.3 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 an algorithm which always generates a "greater" value dynamically for a parallel goal appearing later in the (depth-first left-to-right) ordering. One such algorithm is given in [26]. Hermenegildo presented another algorithm in [16]. 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.[2] Hence any parallel goal is always available to any processor. This optimization will be referred to as **the labeling optimization**.

3.4 Job Scheduling

For a given goal, one processor is selected to start the execution. Other processors become idle and look for available goals 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 in stealing some goal, it starts execution. After a processor P has finished executing some parallel goal, it tries to find an available 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 (i.e., if an available goal is found which is also executable), 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 simple demand-driven strategy is also used by many other researchers (e.g., [16]), as it releases the burden of a busy processor for distributing goals to other processors.

3.5 Handling Failures in Multiple WAMs

When a failure occurs, we have to choose the backtrack literal and perform rollback (i.e., discard some computation). In a parallel implementation, if the computation affected by the backtracking resides only in one processor, then the rollback can be done just as it is done in the 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. 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 data from their DATA portions (see [26]). 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 (that caused the origination of the received message) has been wiped out by some other failure. This requires the use of time-stamps (discussed in [24]) which can be quite expensive to implement.

Another possibility is to handle just 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 dozens of processors, we have chosen to implement the second alternative.

When a failure occurs in PROC$_i$, if the failed literal is not a parallel literal, then the backtracking is performed locally in PROC$_i$ as it is done in the WAM. Otherwise, the backtracking is accomplished in three phases:

Phase I: The recognition phase
PROC$_i$ raises a global flag and then waits until every other processor recognizes the flag.

Phase II: The reset-cancel phase
PROC$_i$ chooses the backtrack literal.

[2]More precisely, if a parallel goal P is deterministic (i.e., the proof tree rooted at this goal has no choice point), then all the descendant goals of P can be given the label of P.

Phase III: The clean-up phase

Each processor discards that part of the computation from its current working memory which is affected by the failure. If the number of DATA portions is larger than the number of processors in the system, then processors which finish their own clean-up early help clean up the DATA portions on the spare list. All the processors need to synchronize at the end of this phase before resuming the forward execution.

4 PERFORMANCE RESULTS

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[3]. Before the execution begins, we specify the number of processors (p) and the number of DATA portions (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.

The implementation has been tested on many programs. Each Horn-clause program is first compiled into a WAM-code program using a modified version[4] of the Berkeley PLM compiler [31]. The APEX-code program is then constructed by adding our extended instructions for parallel execution to the WAM-code program[5]. 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 1. 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 are given in Table 1. 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*' problem; MATRIX, given in [6], multiplies two 50×50 matrices; QSORT, taken from [12], executes '*quicksort*' to sort a list of 511 numbers; TAK is a program for computing the

	HANOI	MATRIX	QSORT	TAK
Quintus*	7.92	11.60	2.95	7.27
SBProlog*	32.86	97.82	—	59.34
STIME†	115.65	237.84	6.17	111.08
PTIME(1)†	116.93	242.41	6.90	112.61
PTIME(2)†	58.56	122.50	3.89	56.79
PTIME(3)†	39.19	83.29	3.17	38.02
PTIME(4)†	29.46	62.97	2.54	28.94
PTIME(5)†	23.72(4.9)	50.91(4.7)	2.38(2.6)	23.17(4.8)
PTIME(6)†	19.74	42.73	2.23	19.43
PTIME(7)†	17.06	37.15	2.13	16.86
PTIME(8)†	14.89	32.84	1.98	15.03
PTIME(9)†	13.33	29.60	1.93	13.39
PTIME(10)†	12.10(9.6)	26.92(8.8)	1.91(3.2)	12.19(9.1)
PTIME(11)†	11.01	24.71	1.87	10.96
PTIME(12)†	10.19	22.91	1.84	10.33
PTIME(13)†	9.43	21.38		9.65
PTIME(14)†	8.81	20.10		8.98
PTIME(15)†	8.31(13.9)	18.91(12.6)		8.26(13.5)
PTIME(16)†	7.82	17.98		8.03
PTIME(17)†	7.36	17.21		
PTIME(18)†	7.05	16.24		
PTIME(19)†	6.68	15.65		
PTIME(20)†	6.47(17.9)	14.97(15.9)		7.20(15.4)

	CDESIGN	IBTAK
Quintus*	0.85	7.28
SBProlog*	2.72	57.84
STIME†	5.55	109.62
PTIME(1)†	1.60(3.5)	110.66
PTIME(2)†	1.29(4.3)	55.46
PTIME(3)†	1.30	28.38
PTIME(4)†	1.31	19.16
PTIME(5)†	1.32	17.88(6.1)
PTIME(6)†	1.33	14.40
PTIME(7)†	1.10	14.06
PTIME(8)†	1.37	11.17
PTIME(9)†	1.16	10.68
PTIME(10)†	1.41	10.74(10.2)
PTIME(11)†	1.44	10.13
PTIME(12)†	1.42	9.24

* (on SUN 3/50 — 1.5 MIPS)

† (on Sequent Balance — 0.5 MIPS)

Table 1 Execution time (in seconds) of different benchmarks on Sequent Balance 21000. Speedup figures are shown in parentheses for some cases.

[3] The APEX instruction set is an extension of the WAM instruction set. The details of the extended instructions are given in [26].

[4] The main difference is that our version does not include cdr-coding. As stated in [29], in the absence of hardware support, cdr-coding does not result in an efficient implementation.

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

function '*takeuchi*(18,12,6)'[6]; CDESIGN is the circuit design program given in [12, 20]. IBTAK is a program (built using TAK) which could benefit from 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 perform the labeling optimization, i. e., we make use of the fact that they are all deterministic programs and hence generate the same label for all the parallel goals (See Section 3.3).

Next, 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 super-linear speedups.

4.1 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[7] 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 1 does not include printing time. Clearly, the APEX is able to achieve almost linear speedup on HANOI for twenty processors.

Though 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, 14]. 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 a 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 (which would have resulted in a somewhat better performance).

The execution tree of '*takeuchi*' benchmark is rather different from the other three benchmarks we just discussed. 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. Even so, the performance on this benchmark (with granularity control) is still very good. To avoid creating activities of small granularity, TAK is coded in such a way that the parallel clause is called recursively only 8 times. Any "deeper" calls are made to a sequential clause (see Appendix A.2).

4.2 Nondeterministic Programs

As pointed out by Fagin in [12], the '*circuit design*' program does not have much AND-parallelism, but has much room for performance improvement due to intelligent backtracking. Our CDESIGN results in Table 1 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. The saturation is mainly caused by the lack of AND-parallel activities.

Note that intelligent backtracking can eliminate even more redundant computation on many processors than on one processor. Consider the dependency graph of Figure 2. On one processor intelligent backtracking, upon the failure of p3, skips the choice point of p2, and goes directly to p1 (and thus saves the work in re-solving p2).

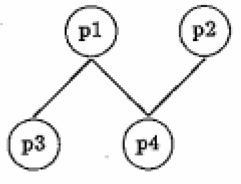

Figure 2 A dependency graph

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

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

In WAM, backtracking to p1 also wipes out the computation of p2. Hence the first execution of p2 is wasted. In parallel execution, if p3 fails even before p2 has finished execution, then p2 is interrupted and reset (thus avoiding some computation which was going to be wasted anyway). This phenomenon is illustrated by 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 the same as given in Figure 2. 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. A close inspection of this program would make it clear that the gain from intelligent backtracking on 1 processor is minimal. Hence PTIME(1) for IBTAK is slightly larger than STIME (see Table 1) because the small gain from intelligent backtracking could not overcome the small[8] overhead of parallel execution. Hence, in absence of gains due to intelligent backtracking, we should expect the speedup performance of IBTAK to be similar to that of TAK. However, IBTAK obtains better speedup than TAK, or even super-linear speedup (e. g., see PTIME(5)).

4.3 The Analysis of Overheads

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

1. The overhead due to the creation of new data objects such as clause frames, goal frames and job frames, and due to updating information in these data objects.
2. 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.
3. The overhead of polling DATA portions to steal goals.
4. The overhead due to backward execution coordination. This is due to the time spent in the recognition phase, the reset-cancel phase and the clean-up phase.

For deterministic programs, the difference between PTIME(1) and STIME is only due to the first two kinds of overhead. Note that the overhead in the first group is roughly proportional to the number of frames created for parallel execution. Since we know the number of frames created for the first three benchmarks shown in Table 1, we are able to estimate that the creation and manipulation of each frame costs about 0.3ms overhead. Note that the cost of creating and manipulating a frame is roughly equal to the cost of a logical inference in our system. we expect this equivalence to hold even on a different (faster) hardware. Clearly, no gain from parallel execution of a clause would result if the size of parallel activities is smaller than the cost of creating two frames[9]. For good performance, the size of parallel activities should be much larger than the cost of creating and manipulating these frames. This is clearly illustrated by the timing difference of columns I and III of Table 2. If parallel activities are created for each parallel clause, the overhead for creating frames becomes a significant fraction of the total execution time (e. g., compare PTIME(1) of column III in Table 2 with STIME of TAK in Table 1).

	I	II	III	IV
Dependency-Check Optimization	yes	no	yes	no
Granularity Control	yes	yes	no	no
PTIME(1)	112.61	114.23	156.96	191.05
PTIME(2)	56.79	56.93	78.66	
PTIME(3)	38.02	38.24	52.80	
PTIME(4)	28.94	29.14	40.11	
PTIME(5)	23.17	23.49	32.52	
PTIME(6)	19.43	19.49	27.28	
PTIME(7)	16.86	16.87	23.77	
PTIME(8)	15.03	15.01	21.03	
PTIME(9)	13.39	13.59	18.89	
PTIME(10)	12.19	12.21	17.70	
PTIME(11)	10.96	11.25	15.70	
PTIME(12)	10.33	10.35	15.19	
PTIME(13)	9.65	9.61	14.19	
PTIME(14)	8.98	8.98	13.37	
PTIME(15)	8.26	8.57	12.37	
PTIME(16)	8.03	8.12	12.45	
PTIME(20)	7.20	7.39		

Table 2 Execution time of the APEX on TAK with and without Dependency-Check Optimization and Granularity Control

The overhead of manipulating bit-vectors depends upon the number of parallel literals, the number of shared variables in each parallel literal, and the run-time binding conditions of these shared variables. But in many cases it can be minimized by the dependence-check optimization (see Section 2). In all the results shown in Table 1, this overhead is minimal because of such optimization. To get an estimate of this overhead, we executed TAK without incorporating dependency-check optimization. From

[8]The parallel processing overheads are small in this example because of the granularity control

[9] One clause frame and at least one job frame must be created before parallel execution of a clause could take place.

PTIME(1) in columns III and IV of Table 2 and from STIME of Table 1 for TAK, it is clear that the average overhead for dependency analysis per parallel literal is roughly the same as that of creating a parallel frame. It is also clear from Table 2 that the granularity control is very effective in reducing the effect of the dependence-check overhead. TAK is an unusual program, as each parallel literal in TAK has six variables. In a parallel goal with fewer variable, the dependence-check overhead will be proportionately smaller. Note that the dependence-check overhead could also be large if the shared variables are bound to large structures. By adopting the approximate checking procedure of DeGroot (see Section 2), we are able to keep this overhead small (at the risk of loosing some parallelism).

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. Asymptotically, the duration of the recognition phase should be proportional to p. But for small p, it is roughly equal to the time taken by a couple of logical inferences (because each processor checks the global flag after completing each logical inference). The duration of the reset-cancel phase is usually small (and does not change with p), but it depends upon the specific failure. The duration of the clean-up phase can be large, but it can even go down (as p increases) depending on how well the work of clean-up is distributed.

4.4 The Effect of the Labeling Optimization and Spare DATA portions

Note that all the APEX-code programs of the benchmarks discussed so far have incorporated the labeling optimization and the dependency-check optimization. As discussed in Section 3.3, all the parallel literals created during the exection of a deterministic program will have the same label. Hence parallelism is not restricted by the steal rule at all (see Section 3.3). To see how the steal rule can affect the performance, we executed the same 'takeuchi' program without optimizing the compiled code. In Table 3, column I gives the execution time of running 'takeuchi' program with the labeling optimization, whereas column II gives the result of running the same program without such optimization. It is not surprising to see that the performance becomes worse.[10] Also, the performance result in Column II does not improve linearly with the number of processors, as the overall performance is dependent upon how often processors get stuck due to the steal rule at run time (see Section 3.2).

	I	II	III	IV	V
Labeling Optimization	yes	no	no	no	no
Number of DATA Portions	p	p	$2 \times p$	$p + 10$	$p + 20$
PTIME(1)	114.23	113.33	113.33	113.33	113.33
PTIME(2)	56.93	67.92	58.49		
PTIME(3)	38.24	67.63	42.40		
PTIME(4)	29.14	43.76	33.25		
PTIME(5)	23.49	41.71	27.77	25.10	24.14
PTIME(6)	19.49	34.26			
PTIME(7)	16.87	34.25			
PTIME(8)	15.01	30.55			
PTIME(9)	13.59	27.60			
PTIME(10)	12.21	25.92	17.03	17.03	13.83
PTIME(11)	11.25	24.88			
PTIME(12)	10.35	22.15			
PTIME(13)	9.61	20.83			
PTIME(14)	8.98	19.73			
PTIME(15)	8.57	18.59			
PTIME(16)	8.12	17.22			
PTIME(20)	7.39	14.44			

Table 3 Execution time of the APEX on TAK with and without the labeling optimization, and for different number of DATA portions. Each execution incorporates granularity control, but does not include dependency-check optimization.

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 $s = 2 \times p$. Column IV and V show the timing of the APEX running the nonoptimized 'takeuchi' program with $s = p + 10$ and $s = p + 20$, respectively. It is clear from these results that by adding enough extra stacks, the restrictive influence of the steal rule is minimized. For example, with $p = 10$ and $s = 30$, the execution time of the nonoptimized version is 24.14 seconds, which is very close to the execution time (23.49 seconds) obtained by the optimized version.

5 CONCLUDING REMARKS

This paper presented a brief description of the implementation of an AND-parallel execution model on a shared-memory architecture, and its performance results on Sequent Balance 21000. Our experimental results have shown that it is possible to perform dynamic data-dependency analysis and intelligent backtracking without incurring excessive run-time overhead. In particular, the

[10] The only exception is PTIME(1). The reason for faster PTIME(1) in column II is that when the label representation exceeds the length limitation, the execution becomes sequential. Hence during the execution fewer parallel activities are created. In the program with deterministic optimization, the creation of parallel activities is not inhibited due to final label length because only one label is used. Note that the granularity control is being used in both cases to limit parallel activities.

technique of granularity control (i.e., spawning parallel activities only if they are large enough) was found to be very effective in diminishing the effect of the overheads. Since the implementation is WAM-based, we are able to retain the execution efficiency of the WAM for sequential segments of the execution. Granularity control, a low overhead execution model and a WAM-based implementation were all crucial to obtaining linear speedup over the sequential implementation. Our current implementation is best suited for a shared-memory multiprocessor with a small number of processors (e.g., Sequent Balance). By changing some design decisions (in job scheduling and backtracking), our implementation could be easily moved to a larger shared-memory multiprocessor (e.g., BBN Butterfly). Although we have only tested our implementation upto 20 processors, it is clear that on suitable programs with appropriate granularity control, the speedup could grow linearly for many more processors.

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. Our current implementation deals only with pure Horn-Clause logic programs. Since most practical logic programs contain non-logical constructs such as CUT, assert, retract, etc., we need to extend our implementation to handle such constructs. An approach presented by DeGroot [11] to handle side-effects in the Restricted AND-Parallelism scheme appears promising, and could be incorporated in our execution model.

ACKNOWLEDGEMENT

The authors are grateful to Manuel Hermenegildo for many useful discussions. Sequent Computer Corp. provided access to a 20-processor Sequent Balance 21000 multiprocessor.

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. Parallel interpretation of logic programs. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 163–170. ACM, October 1981.

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

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

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

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

[13] M. V. Hermenegildo. A restricted AND-parallel execution model and abstract machine for Prolog programs. Technical Report PP-104-85, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, August 1985.

[14] 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.

[15] 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.

[16] 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*, pages 556–576. The MIT Press, 1987.

[17] 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.

[18] N. Ichiyoshi, T. Miyazaki, and K. Taki. A distributed implementation of flat GHC on the multi-PSI. In *Pro-*

ceedings of the Fourth International Conference on Logic Programming, volume 1, pages 257–275. The MIT Press, May 1987. Melbourne, Australia.

[19] 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.

[20] V. Kumar and Y.-J. Lin. A data-dependency based intelligent backtracking scheme for prolog. *The Journal of Logic Programming*, 5(2), June 1988.

[21] Melissa Lam and Steve Gregory. PARLOG and ALICE: a marriage of convenience. In *Proceedings of the Fourth International Conference on Logic Programming*, volume 1, pages 294–310. The MIT Press, May 1987. Melbourne, Australia.

[22] 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.

[23] Y.-J. Lin and V. Kumar. AND-parallel execution of logic programs on a shared memory multiprocessor: A summary of results. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 1123–1141. The MIT Press, August 1988. Seattle, Washington.

[24] Y.-J. Lin and V. Kumar. An execution model for exploiting AND-parallelism in logic programs. *New Generation Computing*, 5(4):393–425, 1988.

[25] 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.

[26] Yow-Jian Lin. *A Parallel Implementation of Logic Programs*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, May 1988. Austin, Texas.

[27] E.Y. Shapiro. A subset of concurrent prolog and its interpreter. Technical Report TR-003, ICOT, Tokyo, Japan, January 1983.

[28] 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.

[29] 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.

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

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

[32] D. H. D. Warren. An abstract prolog instruction set. Technical Note 309, Artificial Intelligence Center, SRI International, October 1983.

APPENDIX

A THE LISTINGS OF BENCHMARKS

A.1 HANOI

```
% Solve 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(18,12,6)

goal :- atak(18,12,6,X,8),write(X).
atak(X,Y,Z,W,0) :- !, stak(X,Y,Z,W).
atak(X,Y,Z,W,N) :- K is N - 1, ptak(X,Y,Z,W,K).

% for sequential computation
stak(X,Y,Z,W) :- X > Y, !, WX is X-1, stak(WX,Y,Z,X1),
                 WY is Y-1, stak(WY,Z,X,Y1),
                 WZ is Z-1, stak(WZ,X,Y,Z1),
                 stak(X1,Y1,Z1,W).
stak(_,_,Z,Z).

% for parallel computation
ptak(X,Y,Z,W,N) :- X > Y, !, tak2(X,Y,Z,X1,Y1,Z1,N),
                   atak(X1,Y1,Z1,W,N).
ptak(_,_,Z,Z,_).
% the parallel clause
tak2(X,Y,Z,X1,Y1,Z1,N) :- WX is X-1, atak(WX,Y,Z,X1,N),
                          WY is Y-1, atak(WY,Z,X,Y1,N),
                          WZ is Z-1, atak(WZ,X,Y,Z1,N).
```

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) :- atak(X,Y,Z,W,8).
p1(X,Y,Z,W) :- atak(Y,X,Z,W,8).
p1(X,Y,Z,W) :- atak(Z,Y,X,W,8).
p1(X,_,_,X).

p2(2).    p2(5).    p2(8).    p3(X,Y) :- X =< Y.
```