Abstract

Information caching eliminates redundant subgoal evaluation in logic program execution by caching useful intermediate results. The idea of information caching were brought up as a logic programming practice in which users have to learn how to modify logic programs with run time assertions so that information caching can be performed as part of the logic program. We propose a model that integrates information caching into inference procedure. In addition to success results, failure information is also cached in this model to facilitate backtracking. A logic system based on this model was implemented. Results of running a set of representative logic programs on the system are reported in this paper.

1 Introduction

With the declarative style and the ability to solve problems nondeterministically, logic has been accepted as an elegant programming language [1, 2]. However, execution of logic programs is relatively slow in comparison with the execution of conventional programs due to lack of reasonably fast logic inference systems. This has been a major obstacle which keeps logic programming from being wildly used in real world. Many approaches have been proposed to speed up the execution of logic programs, which include (1) compilation [3, 4, 5], (2) exploring parallelism [6, 7, 8], (3) intelligent backtracking [9, 13, 14], and (4) information caching [16, 19]. In general, compilation reduces execution time in each step by a constant factor. Parallel execution evaluate multiple subgoals concurrently using multiple computation units. Intelligent backtracking skips unnecessary backtracking by a careful choice of backtrack point. Information caching eliminates redundant subgoal evaluation by caching useful intermediate results. Neither intelligent backtracking nor information caching changes the execution speed in each step. Instead, they reduce total amount of work. These approaches are not mutual exclusive in the sense that they can be applied to a logic program at the same time to obtain multiplicative performance improvement.

The idea of information caching were brought up as a logic programming practice [16, 19], in which users have to learn how to modify logic programs with run time assertions so that information caching operations can be performed as part of the logic program by the underlying inference mechanism. In such a scheme, the cached information is good only for one iteration of program execution and only the first answer can be obtained. In this paper, we propose a model that integrates information caching into execution procedure and extends caching scope to multiple runs of a program execution. Multiple answers can be obtained as well. Another important feature about the model is that not only success results but also failure results are cached to facilitate backtracking.

In this paper, the model and the issues pertaining to the design of such a caching mechanism are discussed. Results obtained from running some representative logic programs on an actual implementation of the model are also presented.

2 Information Caching

For certain program, information caching changes the behavior of its execution totally, and results in dramatic speed improvement. Figure 1 illustrates this point by a famous logic program which calculates the Fibonacci series. Though the program elegantly follows the mathematic definition of Fibonacci number, yet its execution is not efficient. The problem is that this recursive program calculates the same Fibonacci number again and again. That is, in order to calculate the value of fib(n), the program tries to calculate fib(n-1) and fib(n-2). However, the second subgoal, fib(n-2), will be invoked again...
their mathematical definitions. Fibonacci series and Towers of Hanoi are two typical examples. Execution of these programs without information caching incurs substantial amount of re-evaluation of some subgoals.

2 Heavily used sub-programs. This is similar to the subroutines in conventional programs. In many cases, some subroutine is used repeatedly to confirm a known status, or just to calculate some fixed values. Certainly, information caching can be applied effectively to these parts of programs.

3 Naive backtracking over non-related intermediate sub-goals. Naive backtracking forgets everything that has been done after the backtrack point. Even though the calls to the intermediate subgoals may remain the same, they have to be re-evaluated again. Intelligent backtracking schemes were introduced to deal with these cases. However, to find the cause of failure and to choose the best backtrack point incur large overhead in these schemes. Using proposed caching model, most redundant executions can be eliminated.

4 Database systems and expert systems. Dependency and recursion of this type of programs lies in data, and is hard to identify simply by examining the program structure. A program of this kind may or may not gain much from information caching when it is executed for the first time. However, the accumulated intermediate results might help speed up future runs of the program significantly.

Many cache points can be identified by analyzing the structure of the program. However, structure analysis may not be able to explore all kinds of dependency relations in the program. We have been using a learning method which examines execution history of the program and computes the frequency that a clause is called. A cache point is with a clause which is called more than once with the same binding. This learning method is more accurate than the static analysis, though it is quite straightforward. One drawback of such a learning method is that it does not apply to the first run of a program.

Data structures for storing cached data and the operations over those data structures determine most of the overhead of data caching scheme. In specific applications, the cached data can be as simple as numbers. While the cached data become more complex, the data structure and the search mechanism also become more complicated. For all ground data, a hashing table should be sufficient. The most general data structure should be
The cost to determine the generality relation between two terms is a fraction of the cost to unify two terms without occurrence check.

The proposed execution model is outlined in Algorithm 3.1. The idea behind this execution model is that, whenever a call is made, its cache table is searched first. If a success record which can unify with the input binding is found, then the most general common instance between the input binding and the stored record is returned as an answer, and the position of this record is remembered for a possible use in backtracking. If a failure record which is more or equally general than the input binding is found, the algorithm reports failure immediately; no further inference is necessary. Otherwise, if nothing in the table is useful, the execution resorts to the conventional inference procedure. Upon returning from a successful execution of a call, its result is recorded in the associated cache table, same as done for a failure result. A stack, called A-stack, is maintained to keep track of all of the alternative paths accumulated so far. That is, whenever a call is made, the pointers to the clauses which are the possible candidates, as well as the binding, are pushed onto the stack. Backtracking can be simulated directly on the A-stack.

Algorithm 3.1

Definition 3.1 (Generality Relation) A term \( \alpha \) is more general than \( \beta \), noted as \( \alpha \triangleright \beta \), if there exists non-empty substitution \( \sigma \) such that \( \sigma \alpha = \beta \). If \( \alpha \) is more general than \( \beta \), then \( \beta \) is said to be less general than \( \alpha \). Two terms are equally general, if they are identical after non-trivial variable renaming.

Non-trivial variable renaming means any two distinct variables remain distinct after the renaming. The generality relation between two bindings is defined by the consistent relation between each correspondent pair of terms. The cost to determine the generality relation between two terms is a fraction of the cost to unify two terms without occurrence check.
4 Experiments

A logic system based on proposed execution model was implemented. The objective of this implementation is mainly for the evaluation of information caching scheme. Hence its code is not optimized for net speed up. A set of representative logic programs were used to test the system. They are listed in Table 1. All of the programs, except the last one, were obtained directly from related references, therefore their source codes are omitted due to limited space.

In order to show the effects of information caching scheme, the following measures were recorded in the experiments. 

- \( \text{Calls}(C) \): total number of subgoal invocations.
- \( \text{Time}(T) \): actual run time.
- \( \text{Gain}_{\text{call}}(GC) \): the ratio of number of calls to execute the program without and with information caching. This is the theoretical gain.
- \( \text{Gain}_{\text{time}}(GT) \): the ration of actual run time without and with information caching. This is the actual gain which includes all kinds of overhead. If the value of \( GC \) or \( GT \) is greater than 1, a gain is implied. Similarly, if the value of \( GC \) or \( GT \) is less than 1, a negative gain is implied. Since the implementation of the system was not optimized for net speed up, such relative performance measures are more meaningful than absolute run time.

### 4.1 Fibonacci series, Towers of Hanoi and Ackermann

These three programs are examples of 2-linearly dependent recursion. The gain is expected high in this group. This is confirmed in Fibonacci series program. Towers of Hanoi program requires an information caching scheme which is capable of handling complex non-ground terms. The proposed caching scheme is built with such a capability. Theoretically, when caching is done against \( \text{hanoi}() \) function, the Towers of Hanoi program should obtain a \( GC \) of as much as \( 2^n \). This is not shown in the result. The reason is that \( \text{hanoi}() \) is not the only function which is critical to the program. \( \text{append}() \) is another function which also dominates the computation. Unfortunately, information caching can hardly be applied to it, because its behavior depends on the length of input lists and the length of a list is not stored explicitly in Prolog. The complexity of \( \text{append}() \) function is proportional to the sizes of its arguments, which, in this program, grow exponentially. Thus, after the execution time of \( \text{hanoi}() \) is reduced, execution time of \( \text{append}() \) is still significant. Though the dependency relation in Ackermann program is not as clear as in other programs, yet the effect of information caching does show up.

<table>
<thead>
<tr>
<th>#</th>
<th>Program</th>
<th>Comments</th>
<th>Type</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fib</td>
<td>Calculate Fibonacci series</td>
<td>A</td>
<td>[9]</td>
</tr>
<tr>
<td>4</td>
<td>queen</td>
<td>N-Queen problem</td>
<td>B</td>
<td>[12]</td>
</tr>
<tr>
<td>5</td>
<td>automata</td>
<td>A finite automata</td>
<td>B</td>
<td>[13]</td>
</tr>
<tr>
<td>6</td>
<td>flight</td>
<td>Flight scheduling</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>7</td>
<td>example A</td>
<td>Backtracking example</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

Problem types: (A) k-linearly dependent recursion (B) Heavily used sub-program (C) Naive backtracking (D) Database program

Table 1: List of test programs

```plaintext
Figure 3: Illustration of Call and BackTo

4.1 Fibonacci series, Towers of Hanoi and Ackermann

These three programs are examples of 2-linearly dependent recursion. The gain is expected high in this group. This is confirmed in Fibonacci series program. Towers of Hanoi program requires an information caching scheme which is capable of handling complex non-ground terms. The proposed caching scheme is built with such a capability. Theoretically, when caching is done against \( \text{hanoi}() \) function, the Towers of Hanoi program should obtain a \( GC \) of as much as \( 2^n \). This is not shown in the result. The reason is that \( \text{hanoi}() \) is not the only function which is critical to the program. \( \text{append}() \) is another function which also dominates the computation. Unfortunately, information caching can hardly be applied to it, because its behavior depends on the length of input lists and the length of a list is not stored explicitly in Prolog. The complexity of \( \text{append}() \) function is proportional to the sizes of its arguments, which, in this program, grow exponentially. Thus, after the execution time of \( \text{hanoi}() \) is reduced, execution time of \( \text{append}() \) is still significant. Though the dependency relation in Ackermann program is not as clear as in other programs, yet the effect of information caching does show up.

<table>
<thead>
<tr>
<th>Queue</th>
<th>Calls</th>
<th>( \text{GC} )</th>
<th>( \text{GT} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{fib} ) (N)</td>
<td>1056</td>
<td>2.166</td>
<td>20.76</td>
</tr>
<tr>
<td>( \text{hanoi} ) (N)</td>
<td>1056</td>
<td>2.166</td>
<td>20.76</td>
</tr>
<tr>
<td>( \text{hanoi} ) (N)</td>
<td>1056</td>
<td>2.166</td>
<td>20.76</td>
</tr>
</tbody>
</table>

Table 2: Illustration of Call and BackTo

4.1 Fibonacci series, Towers of Hanoi and Ackermann

These three programs are examples of 2-linearly dependent recursion. The gain is expected high in this group. This is confirmed in Fibonacci series program. Towers of Hanoi program requires an information caching scheme which is capable of handling complex non-ground terms. The proposed caching scheme is built with such a capability. Theoretically, when caching is done against \( \text{hanoi}() \) function, the Towers of Hanoi program should obtain a \( GC \) of as much as \( 2^n \). This is not shown in the result. The reason is that \( \text{hanoi}() \) is not the only function which is critical to the program. \( \text{append}() \) is another function which also dominates the computation. Unfortunately, information caching can hardly be applied to it, because its behavior depends on the length of input lists and the length of a list is not stored explicitly in Prolog. The complexity of \( \text{append}() \) function is proportional to the sizes of its arguments, which, in this program, grow exponentially. Thus, after the execution time of \( \text{hanoi}() \) is reduced, execution time of \( \text{append}() \) is still significant. Though the dependency relation in Ackermann program is not as clear as in other programs, yet the effect of information caching does show up.
4.2 N-Queen problem and Automata

These two programs are based on generate-and-test and they return multiple answers. By avoiding re-testing of the same failure cases, information caching can still be effective though it is not significant since the programs do not incur so many re-testing during its execution. According to the experiments of 5-queen problem, GC did show a minor gain, but the run time was offset by other overhead and resulted in a negative improvement. In automata program, as expected, the effectiveness of the proposed caching scheme was not observed in finding the first answer since none of the cached data was used. However, in finding subsequent answers, the effect of information caching did show up. An increased improvement was obtained as more and more answers were found. Again the actual improvement falls behind the theoretical improvement due to extra overhead.

4.3 Flight scheduling

This program exhibits an usage of information caching in a database program. Unlike recursion examples, dependency in such a program lies in relations between data items. It is indeed difficult to determine the execution behavior by just looking at the program structure. Nevertheless, information caching can effectively eliminate re-examining of those routes that had been discovered to be possible or impossible. An interesting point shown in the results is that the actual gain is larger than the theoretical gain. This is because execution of the program involves large amount of data movement which is reduced considerably when redundant path discoveries are cut by information caching.

4.4 Example-A

This is a synthesized example which shows the use of information caching in naive backtracking over non-related intermediate subgoals. It is easy to see that the first failure occurs when \( r() \) is called with an improper value of \( W \). The execution backtracks to \( s() \) and find a new value for \( V \). However, all calls will fail without a right value for \( W \). Then backtracks to \( p() \) and find a new value for \( W \). If the new value for \( W \) is still not right, all of the same calls to \( s() \) will be re-tried again before the execution find the next chance to backtrack to \( p() \) where another value for \( W \) can be obtained. The idea of intelligent backtracking is to skip unnecessary re-invocation of calls to \( s() \), and backtracks to \( p() \) directly in this case. However, current intelligent backtracking schemes incur to much overhead to find the actual cause of failure and to determine the right backtrack point. Proposed caching scheme can facilitate backtracking in certain cases. Since all the alternatives of the subgoal have been tried and the results (success or failure) are cached in the table, re-invocation to theses calls are automatically skipped. Hence, intelligent backtracking is carried out implicitly by proposed caching scheme.

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{route} & \text{length} & \text{Time} & \text{Cost} & \text{GT} \\
\hline
1 & 0.004 & 0.004 & 0.004 & 0.004 \\
2 & 0.004 & 0.004 & 0.004 & 0.004 \\
3 & 0.004 & 0.004 & 0.004 & 0.004 \\
4 & 0.004 & 0.004 & 0.004 & 0.004 \\
5 & 0.004 & 0.004 & 0.004 & 0.004 \\
6 & 0.004 & 0.004 & 0.004 & 0.004 \\
7 & 0.004 & 0.004 & 0.004 & 0.004 \\
8 & 0.004 & 0.004 & 0.004 & 0.004 \\
9 & 0.004 & 0.004 & 0.004 & 0.004 \\
10 & 0.004 & 0.004 & 0.004 & 0.004 \\
\hline
\end{array}
\]
experiment result shows a tremendous gain from this type of logic program execution.

<table>
<thead>
<tr>
<th>Query</th>
<th>Cache</th>
<th>First (and only) Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>query(X, Y, Z)</td>
<td>yes</td>
<td>183</td>
</tr>
</tbody>
</table>

5 Concluding remarks and future work

Information caching is simple yet useful to a variety of logic programs. It changes the behavior of logic program execution by removing most of the redundant execution. Hence the speed is improved and the execution space is reduced.

An execution model that caches both successful and failure information was proposed. Experiments were conducted on a system based on this model, and results were presented in this paper. Programs with k-linearly dependent recursion benefit most from caching. Usually, the execution complexity is reduced from exponential to linear. Generate-and-test type of programs can also be benefited from information caching by reducing the re-examination of tested path. In database applications, though the improvement is not as dramatic as in recursion programs, cached information accumulated from past runs can be very helpful for later runs.

In current implementation, success and failure information is cached in a table, and linear search is used for table look up. Table look up is thus the major overhead of the execution model. In the next generation, a better table maintenance mechanism with hashing and improved table organization would be implemented. The overhead should be substantially reduced then [20].

In addition to better implementation of the data caching scheme, an intelligent algorithm which analyzes the programs and inserts the caching points automatically are also under development.

References


293