Knowledge Base Compilation

F. D. Highland
C. T. Iwaskiw
AI Technology Center
IBM Systems Integration Division
18100 Frederick Pike
Gaithersburg, MD 20879

Abstract

The effective use of knowledge based systems technology to solve complex and real-time problems in embedded system environments requires that the performance of these systems be maximized for conventional processor architectures. This paper presents a technique for the complete compilation of knowledge bases directly into procedural code. The technique is based on determining what actions an interpretive inference engine would take with the specific knowledge base and generating only the code needed to perform those actions. This eliminates the overhead of interpreting a representation of the knowledge base and significantly improves performance. The performance gains produced by this approach will be examined using examples from the High Performance [imbedded Reasoning (HiPER) System, a prototype tool which utilizes this technology.

1.0 Introduction

While knowledge based system technologies have shown promise in many useful application areas, their use in some applications has been limited by the performance of existing implementations. This is particularly true in real-time monitoring and control applications which must reason about large amounts of data in very short periods of time. It is also true in embedded applications in which the knowledge based system must share the resources of the computer with a number of other applications. One solution to the performance problem is to develop specialized hardware to support the inferencing process. However, this solution is undesirable for many embedded applications because of the added complexity of additional hardware and problems with accessing data on existing systems.

A major performance problem for knowledge based systems on conventional hardware is the inherent interpretive nature of their implementations. Even when the inference engine is implemented in a compiled language, the general purpose design of an application independent inference engine requires the interpretation of data structures which represent the knowledge base. This process of interpretation is inefficient in comparison to fully compiled implementations. By restricting the inferencing techniques available to decision trees or other single paradigm reasoning strategies, compilation of the knowledge base can be performed. However, this limits the knowledge engineers ability to represent and solve some classes of problems.

This paper will present a technique that can be used to convert knowledge bases directly into compilable high level language code providing high performance using existing and proven reasoning algorithms. Interpretive approaches to implementing backward and forward chaining reasoning will first be examined to identify areas where performance improvements could be made. Then a design for representing knowledge bases in a program form and the generation of this form from the rules of a knowledge base will be presented. Finally, the implementation of this technique in the High Performance Embedded Reasoning (HiPER) System prototype will be discussed and results obtained from this prototype compared with those from other tools.

2.0 Interpretive Approaches to Reasoning

Most knowledge based systems represent knowledge internally as collections of data structures consisting of nodes and links between the nodes. The nodes represent the rules, the tests within the rules, the actions to be taken by the rules, and the facts on which reasoning is performed. The links represent the relationships between facts and rule tests, the ways in which the results of tests are combined into rule conditions, and the references made to these tests and conditions from other rules in the knowledge base. The nodes are usually organized into trees or networks which are augmented with lists of references and current data values.

Inference engines implement the reasoning process by traversing the networks that represent the knowledge base. At each node, the inference engine examines the contents of the node to determine what data is involved and what tests need to be performed on that data (test evaluation), which nodes to process next based on the results of those tests (test result propagation), and what actions to perform when a rule becomes true (performance of rule actions). Based on the results of this process the inference engine follows one or more links to the other nodes.

As an example of this, consider a simple backward chaining system which organizes its rules as a set of goal trees, each containing one or more rules that provide conditions and fact values concluding a single goal. The rules and the goal tree that would result from them are represented in Figure 1. The reasoning process for such a system would begin by selecting a goal node and interpreting that node and each of its successors in order to depth-first traverse the goal tree. This process would continue until a node is encountered that requires a fact that is not currently known. The value for the fact is would then acquired by interpreting information in the fact node that indicates how to obtain the value. When a value is obtained, test evaluation occurs by interpreting information in test nodes that reference that fact. The results of these tests are then propagated up the goal tree by reversing the traversal process and interpreting each node to determine how the results of previous nodes are to be combined and passed to the next node in the tree. In addition to the interpretation that must be performed for test evaluation and result propagation, the tree traversal process must also be interpretive because connections are only known to the nodes which represent the trees. This form of backward chaining reasoning mechanism is similar to that of EMYCIN [Van Melle *el al,* 1981], PIE (Burns *et al,* 1986], example systems proposed by Winston and Horn [1984], and many other systems.

Forward chaining reasoning algorithms operate in a similar manner except that the depth-first traversal to find a fact to evaluate is not performed. Instead, the system must search for facts that match rule conditions. Typically, this involves propagation of facts to the rules and evaluation of rule tests based on those facts. This again involves interpretation of the test nodes to evaluate the tests with respect to a set of facts. After a test is evaluated the results are propagated up the rule tree by interpreting the nodes at each step to determine how the results of previous nodes are to be processed, combined, and passed to the next node in the tree. This form of forward chaining reasoning mechanism is similar to that of example systems proposed by [Winston and Horn, 1984] and has been highly optimized in OPS5[Forgy, 1982].

The interpretive traversal of these trees or networks and the determination of the steps required to process each node degrades the performance of the reasoning process. At each node, the reasoning system must determine which of a possible set of steps are required and then perform those steps using information available at the node. Once the steps required to process a node have been performed, a determination of which node(s) to process next must be made. The performance of knowledge based systems could be greatly improved if the interpretation processes were eliminated and the equivalent steps were performed by concise, compiled code.

3.0 Compilation of Reasoning

The basic approach to knowledge base compilation is to determine the steps that would be taken by an interpretive inference engine during the reasoning process and generate only the code needed to perform those steps. Through analysis of the knowledge base, the set of possible steps required to process each node can be significantly reduced, often to a single step. This can then be converted to concise, efficient, procedural code to implement the reasoning process. Similarly, the steps that must succeed this processing can also be identified so that code can be generated. Using this approach, any interpretive inferencing system can be converted to a compiled system which avoids the interpretive steps.

The technique presented here for rule compilation addresses the generation of code for test evaluation, test result propagation, and performance of rule actions. These activities are common to forward chaining, backward chaining, and mixed chaining (combined forward and backward chaining) reasoning processes. Since the evaluation and propagation steps are the most frequently used operations in reasoning, their compilation will produce the most benefit in overall performance. Methods to improve the performance of selection of facts to evaluate for backward chaining reasoning arc not specifically addressed but the same techniques may be used to improve this process as well.

In order to perform rule compilation, the rules must first be converted into a network representation with nodes for each rule component and links indicating the relationships of rule components to other rule components and facts. This is usually the same network that is used by the interpretive inference engine during execution of the knowledge base as in Figure 1. In addition, a mechanism must be developed to compile the traversal of the network for the propagation of results in order to preserve the dynamic nature of the knowledge based systems. This mechanism will be presented below.

The compilation technique first partitions the network of rules into a set of rule tree subnetworks. Although the subnetworks are considered to be trees, they need not be proper trees and may contain references to other subnetworks within the tree. The criteria used for this partitioning is arbitrary with respect to the compilation approach. Partitioning the networks on a rule basis provides an easily determined criteria that is adaptable to incremental compilation techniques. To maximize performance, the criteria could select subnetworks that minimize references outside of the subnetwork in order to reduce procedure call overhead.

Next, each rule tree is converted into a rule procedure in which the nodes are represented as parts of a case structure within a while loop controlled by a case index. Each part of the case structure consists of a segment of code that performs the function of a node in the rule network generated from the knowledge base. Each of the code segments conditionally passes control to its parent node in the tree through manipulation of the

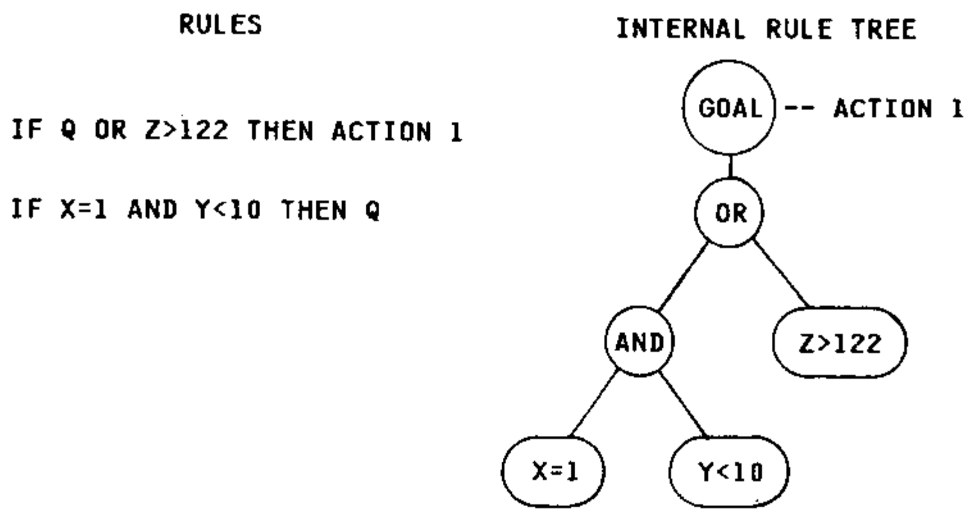


Figure 1. Example Rules and their Internal Representation

case index. Code segments may also pass other information, such as data bindings or certainty factors, to their parents. The use of a tree structure guarantees that only one parent exists for each node in the rule tree for the nominal case eliminating the need for multiway branching and stack mechanism to maintain branch information. If the invocation of parents outside of the rule tree is required, they can be accessed by procedural invocation of the rule procedures that represent them. A stack may be required in situations where multiple data values must be passed to the parents individually. This need may also be eliminated by passing the data values as a list of values rather than as separate items. The use of the while loop around the case structure permits propagation (execution of a series of code segments) to continue as far as possible on each invocation of the rule procedure. It also allows any node in the rule tree to be invoked on demand for additional flexibility in node evaluation (for example, when one part of an element of a structure of data changes). The code for the terminal parent node in this structure must contain logic to identify a rule as concluded'. This can consist of placing an entry in a conflict set to be selected for activation by a conflict resolution mechanism or direct execution of the rules action.

Finally, data distribution procedures are produced to activate the rule trees based on changes to data items in the knowledge base or other need to evaluate the rule. Through analysis of the knowledge base, a list of rule trees and the nodes within those trees can be developed for each data item. From these lists, procedural code can be generated to invoke the appropriate node of each rule tree when the data item is changed.

Using this implementation, the process of evaluation and propagation is started by the data distribution procedures when a data item is updated. The distribution procedures pass an index value for the node to be evaluated to the rule procedures case structure. This causes selection and evaluation of the desired node. Depending on the results of the evaluation, the index is reset to cither the index value of the parent node to be evaluated or to the exit index. The process continues until the rule is concluded or all subconclusions that can be derived from the currently available information are exhausted.

A simplified example of this process is given in Figure 2. Figure 2a shows a rule that forms a simple knowledge base. Figure 2b shows the internally generated rule tree resulting from this rule. Figure 2c shows a simplified version of the code resulting from this knowledge base. Whenever the values of X or Y are changed, the DISTRIBUTE ? procedures invoke the rule tree procedure RULE1 with the appropriate index value to cause evaluation of the tests. The results then propagate up the tree as far as possible. Details of the code to evaluate the AND condition are dependent on the particular reasoning algorithm being implemented. It could contain logic to test certainty factors in an evidential reasoning approach like FMYCIN [Van Melle *el al.,* 1981] or it could test for the existence of all inputs and combine them into a binding token in a RETE pattern matching approach [Forgy, 1982). The result of this evaluation is used to either pass control to the next node-segment or to terminate processing by making the appropriate setting of the index value. Code to manage and pass data values to parent nodes is also reasoning algorithm specific and has been omitted from this example for simplicity.

4.0 Discussion

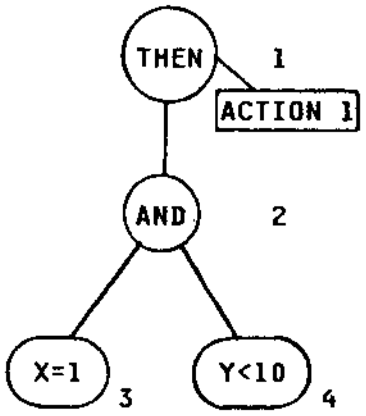
This technique generates a set of rule processing and data distribution procedures that are specific to the knowledge base. It combines the specific rules of the knowledge base with the inferencing logic to produce a unique code module. Hence, the inference engine is no longer physically separated from the rules and can be optimally merged with the rule code. Despite the lack of a physically distinct inference engine, the functionality of the conventional approach (separate inference engine and knowledge base) has been pre-

KNOWLEDGE BASE

```
IF X=1 AND Y<10
 THEN ACTION 1
```

A

RULE TREE



B

GENERATED CODE

```
PROC RULE1(I);
  WHILE (I<>0);
    CASE (I) OF
    1: /* Action node */
       ACTION 1
       I = 0;
    2: /* AND node */
       IF both sons true
       THEN
         I = 1;
       ELSE
         I = 0;
    3: /* Test node */
       IF X=1 THEN
         I = 2;
       ELSE
         I = 0;
    4: /* Test node */
       IF Y<10
         I = 2;
       ELSE
         I = 0;
    END;
  END;
END;

PROC DISTRIBUTE_X;
  RULE1(3);
END;

PROC DISTRIBUTE_Y;
  RULE1(4);
END;
```

C

Note: I = Case/Node Index

Figure 2. The Conversion of Rules into Code

served since the resulting modules take the same actions that the inference engine would have taken.

The processing performance gains obtained by the compilation technique depend heavily on the complexity of rule language, amount of interpretation done at execution time by the inference engine, and the underlying algorithms used. The performance gains occur at the points at which selections must be made between alternative paths or processing steps and at points where search or indirect data access must occur. In general, knowledge representation languages which offer a richer set of features will encounter more alternate paths during processing and can be expected to experience greater performance gains.

The implementation of rules and the inference engine as procedural code can provide additional efficiencies of implementation beyond the removal of the interpretive overhead. Data and knowledge base control blocks can be represented as program variables and referenced directly rather than using indirection, linked lists, or other dynamic structures that require search or more complicated traversal. The data required at each step in the processing can be determined during code generation and its use, representation, and access highly optimized. Expressions used in rule condition tests and action statements can be implemented efficiently in procedural code rather than as complex interpretive structures. Interpretive inference engines cannot take advantage of these efficiencies because they must be generalized for all possible knowledge bases and structures.

The conversion of knowledge bases into code also improves the embeddability of the resulting system. The compiled knowledge base is compatible with any environment that the language that is generated is compatible with. By using standard compiler building techniques, the generated language can also be easily retargeted.

The code generated by the compilation technique will potentially require more storage than the corresponding data structures used by interpretive techniques. This is because the generalized logic used by an interpretive approach is customized and repeated in each node. However, since the code that is generated represents only a small set of the possible actions that could be taken on a node and can be highly optimized, the amount of code generated for each rule will be significantly smaller than that of an interpretive inference engine. This type of trade-off (more storage for lower processing requirements) is typical of processing optimization techniques. Since the code is generated from the same representation that is used by the interpretive approach, the amount of storage required will be directly proportional to the amount of storage required by the interpretive form of the knowledge base and combinatorial explosions resulting from rule and data interactions will not occur. The size of the generated code can be further reduced through the proper use of utility procedures to perform logic operations that are invariant across knowledge bases.

An alternate approach to knowledge base compilation is to represent the rule network as a set of decision trees which can then be represented by a series of nested IF statements. This approach has been used in systems such as Rulemaster. 1 lowever, this approach effectively fixes the order of evaluation of the parameters involved (based on their values) and does not provide access to individual nodes on demand or allow for the complexities of pattern matching. Since access to individual nodes is not provided, this approach does not support techniques such as dynamic rule ordering based on confidence or forward chaining reasoning.

Another alternative is partial compilation of the knowledge base. In this approach the conditions and actions of rules are compiled into code but the inference network remains as an interpreted data structure. This technique has been used bv KnowledgeTool [IBM, 1987], and ETC |Drastal e*t'al.,* 1986). This approach only improves the performance of condition evaluation and action execution and does not address the issue of rule result distribution which is central to the inferencing process.

5.0 Implementation of HiPER

The High Performance Embedded Reasoning (HiPER) System is a prototype knowledge based system development tool that utilizes this knowledge base compilation technology. It consists of a Knowledge Base Compiler for converting a text form of the knowledge base into PL/1 or C code and a Development Environment to allow the user to create and analyze knowledge bases.

The underlying reasoning mechanism of HiPER is the forward chaining YES/RETE algorithm |Schor *et al,* 1986| as used in KnowledgeTool [IBM, 1987] and ECLPS [IBM, 1988). The HiPER Knowledge Base Compiler parses a text form of the knowledge base into a network representation. This representation is then converted into a RETE network, which is then traversed to produce procedural code. This code, along with run-time utilities which provide knowledge base independent functions and environment interface software to adapt the knowledge base to different operating environments, is then linked into a single load module for execution.

A number of additional features and enhancements have been added to the basic YES/RETE algorithm used in HiPER. These include:

Breadth-first RETE Token Processing: Other implementations of RETE algorithm process tokens containing data updates in a depth first manner. The breadth-first modification allows more processing to be performed at each node, cutting down on processing overhead when many objects are matched. It also reduces the need for a stack to hold queued tokens thereby simplifying the code.

Integrated Frames: The simple (flat frame) working memory element representation used in most OPS derivative languages has been extended to allow frames with inheritance. To resolve the problem of integrating an access oriented frame system with update oriented RETE processing, an update oriented frame system is implemented by augmenting the data distribution procedure with franr inheritance processing.

Backward Chaining: In order to provide additional inferencing capability the RETE processing is augmented with logic to acquire data (goals) and maintain the set of rules that need to be fired in order to attain the goals. This provides a form of backward chaining reasoning.

6.0 Results

In order to assess the relative performance gains of the compilation approach against other approaches, factors such as the computer hardware, operating environment, implementation language, and underlying reasoning algorithms must be taken into account. Eor most knowledge based system development tools these factors are so different that derivation of any meaningful comparisons would be difficult. However IBM's KnowledgeTool provides a good point of comparison because it is implemented in the same high level language as one of the versions of HiPER (PL/I) and uses the same underlying reasoning algorithm. It should be noted, however, that KnowledgeTool is not a completely interpretive implementation but uses a partially compiled approach that compiles rule tests and actions into PL/I code and interprets the RETE processing. The performance advantage provided by the fully compiled approach over a fully interpretive system would be more substantial.

Two benchmarks were executed using HiPER and KnowledgeTool Version 1.1. One was a version of the Monkey and Bananas problem that has been used by NASA to compare the performance of other expert system building tools [NASA, 1986]. The other was a version of the Maintenance Operations Center Automation (MOCA) prototype application written for a major airline. The second application represents a real-world problem with a large amount of data that performs a difficult scheduling problem. Both applications were run with the same rules on the same processor (IBM 3090-200) and utilized timing routines that measured the virtual CPU time from the firing of the first rule to the firing of the last rule in the benchmark scenario. The results of these benchmarks are shown in Table 1 and Table 2. The virtual CPU number indicate a performance improvement of between 3 and 11 times for the compiled approach over the hybrid approach used in KnowledgeTool. More storage is required for the compiled approach but these data points indicate the storage requirement to be only on the order of 75% greater than the hybrid approach. This seems to be an acceptable trade-off with respect to the performance gains possible.

Measure	HiPER	KnowledgeTool
Object Count	34	34
Virtual CPU Time	0.032	0.099
Code Size (Bytes)	78K	48K
Bytes/Object	2.3K	1.4K

Table 1. Comparative Results for Monkey and Bananas Benchmark

Measure	HiPER	KnowledgeTool
Object Count	163	163
Virtual CPU Time	1.49	16.96
Code Size (Bytes)	497K	265K
Bytes/Object	3.0K	1.6K

Table 2. Comparative Results for MOCA Benchmark

7.0 Summary and Conclusions

A technique for the full compilation of knowledge bases has been presented. This technique provides a performance improvement over the conventional interpretive approach that varies depending on the complexity and features of the rule language. This technique is superior to decision tree generation approaches because it allows direct access to each node in the resulting rule networks supporting forward chaining reasoning and allowing the use of techniques such as dynamic rule ordering. It also goes beyond approaches which compile only the condition tests and action statements of rules because it also compiles the inferencing mechanism to provide increased performance and embeddability of the resulting applications.

The relative advantages of this approach have been analyzed by comparison of benchmarks coded with the HiPER system, a prototype knowledge based system tool that uses this approach, with similar systems. The use of this compilation technique in the HiPER System prototype has demonstrated that significant performance gains are possible over partially compiled techniques for only a modest increase in memory requirements.

Acknowledgments

The authors would like to thank Hugh Gallivan, Jim Tani, Renee Rochester, and Chuck Grady for their contributions in the implementation of the HiPER System prototype.

References

[Drastal *et aL,* 1986] Drastal, G., DuBois, T., McAndrews, L., Raatz, S. and Straguzzi, N. Economy in Expert System Construction: The AEGIS Combat System Maintenance Advisor. *SPIE Applications of Artificial Intelligence Conference.* 1986.

[Eorgy, 1982] Forgy, C.L. Retc: A Fast Algorithm for the Many Pattern/Many Object Match Problem. *Artificial Intelligence* 1982.

[Burns *et aL* 1986| Burns, N.A., Ashford, T.J., Iwaskiw, C.T., Starbird, R.I\ and Flagg, R.L. The Portable Inference Engine: Fitting Significant Expertise into Small Svstems. *IBM Systems Journal,* Vol. 25, No. 2. 1986. '

[IBM, 1987] *IBM KnowledgeTool Users Guide and Reference.* IBM Corporation, SI120-9251. 1987.

[IBM, 1988] *Enhanced Common LISP Production System User's Guide and Reference.* IBM Corporation, SC38-7016. 1988.

[NASA, 1986] Timing Tests of Expert System Building Tools. U.S. Government Memorandum FM7(86-51). NASA Eyndon B. Johnson Space (Center. 1986.

[Schor *et al,* 1986] Schor, M, Daly, T., Lee, U.S., and Tibbitts, B.R. Advances in RETE Pattern Matching. *Proceedings of AAA 1-86.* pp. 226-232. 1986.

[Vein Melle *et al,* 1981] Van Melle, W., Scott, A.C., Bennet, J.S., Peairs, M.A. *The EMYCIN Manual.* Heuristic Programming Project Report HPP-81-16. Stanford University. 1981.

[Winston and Horn, 1984J Winston, P.M. and Horn, B.K.P. *LISP.* Addison Wesley, Reading Mass. 1984