An-Najah Univ. J. Res., Vol. 13 (1999) 37-56

Run-Time Elimination of Dead-Rules in Forward-Chaining Rule-Based Programs

إزالة القواعد بعد انتهاء تنفيذها في برامج القواعد المنتجة

Wael Mustafa

وائل مصطفى

Computer Science Department, An-Najah N. Univ., Nablus, Palestine.

Received: (29/8/1998), Accepted: (4/7/1999)/

Abstract

This paper presents an optimization method to improve execution time of forward-chaining rule-based programs. The improvement is achieved by deleting rules that finish firing during run-time. The conditions of the deleted rules are not matched against working memory in later execution cycles and hence, the execution time is reduced. Information obtained from control and data-flow analyses is utilized to determine when rules finish firing during run-time. Since rules are deleted during run-time only after they finish firing, the optimization does not change the semantics of the source program. The optimization method can be a final step to other optimization methods. The results of applying the optimization to three CLIPS rule-based programs are presented. These results show significant improvement when the source program contains rules that require significant matching time and finish execution early during run-time.

تقدم هذه الورقة طريقة لتقليل زمن تنفيذ برامج القواعد المنتجة عن طريق حذف القواعد بعد انتهاء تنفيذها مباشرة مما يؤدي إلى عدم مقارنة شروط القواعد المحذوفة مع محتويات الذاكرة العاملة خلال دورات تنفيذ البرنامج المتبقية. وتبين هذه الورقة كيفية معرفة متى انتهاء تنفيذ القواعد باستخدام طرق تحليل سير المعلومات والمسيطرة بين قواعد البرنامج. وتعرض الورقة نتائج حذف القواعد بعد انتهاء تنفيذها لثلاثة برامج بلغة CLIPS.

1. Introduction

Human expertise in various domains has been successfully written in rule-based languages. This is due to the similarity between the rule construct and the manner in which humans naturally express their expertise. However,

rule-based programs execution involves matching rules conditions repeatedly against a dynamic set of facts that represent the state of the problem being solved. This often results in excessive computational time.

Several works have been done to improve the execution time of forward-chaining rule-based programs. The RETE algorithm (Forgy, 1982) reduces the time needed to match rules conditions by utilizing previous matching results and matching only the changed facts after each rule execution. Other efficient matching algorithms have also been developed (Miranker, 1987; Ishida, 1994; Kimura et al, 1995; Lee et al, 1997). The work of (Lopez et al, 1998) includes techniques to improve performance by restructuring program rules.

Existing methods for detection and deletion of dead, or nonreachable, rules in rule-based programs handle only rules that do not fire at all or apply only to monotonic backward-chaining languages (e.g., Prolog) (Stachowitz et al, 1987; Bellman et al, 1988; Chander et al, 1997; Murrell et al, 1997; Zlatareva, 1997; Levy et al, 1998). This paper presents an optimization to dynamically remove rules that become dead during run-time in forward-chaining rule-based programs. The source program is transformed into a semantically equivalent program that contains code to delete rules during run-time just after they finish firing. Conditions of deleted rules are not matched during the rest of execution. This reduces the matching time for the program. The optimization maintains the semantics of the original program since only rules that finish firing are deleted during run-time.

The optimization process is illustrated in Figure 1. First, Control-Flow Analysis (CFA) is applied to the source program. CFA produces a Control-Flow Graph (CFG) that describes all possible execution paths between program rules. Data-flow analysis is then applied to the CFG producing live-rule information that is used to determine when rules finish firing during run-time. The optimizer uses both the CFG and live-rule information to produce an optimized program that contains code to remove rules after they finish firing during run-time.

The rest of the paper is organized in the following manner. Section two briefly reviews rule-based languages and data-flow analysis. Section three

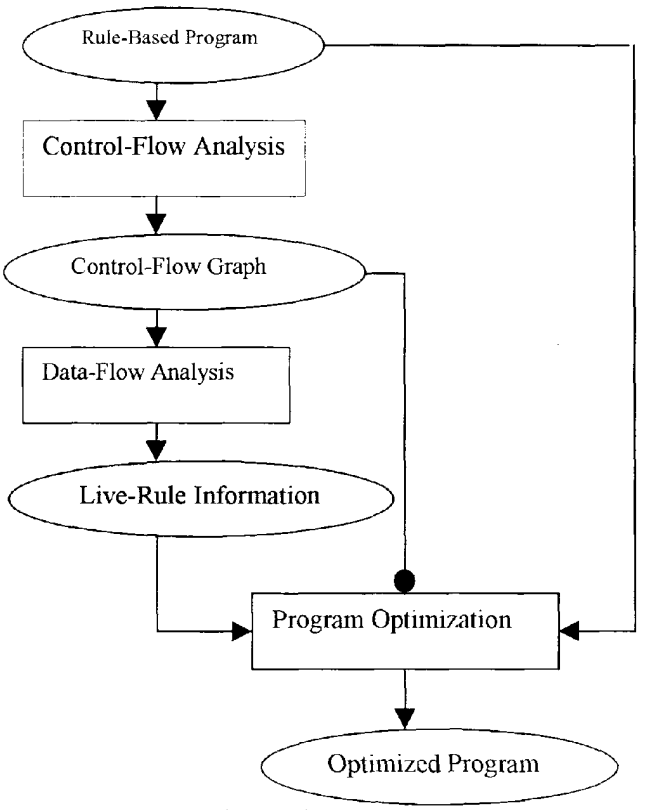

Figure 1. Diagram of the optimization process

shows how to use data-flow analysis to determine when rules finish firing. Section four presents the optimization algorithm. Section five presents the results of optimizing three example programs. Section six presents the conclusion.

2. Background

Forward-chaining rule-based languages are a family of programming languages that are mainly used for implementing expert systems. CLIPS (COSMIC, 1989) is an example of such a language that has been used for real-world projects in various areas ranging from government to business and industry.

A forward-chaining rule-based program consists of three components: Working Memory (WM), a set of rules, and an interpreter. The WM contains the dynamic knowledge state of the problem being solved. A *rule* is a condition-action pair. A *condition* is a list of patterns that test the contents of WM. An *action* is a list of operations that mainly modify the WM. The *interpreter* executes rules whose conditions are satisfied by the current WM. The operation of the interpreter is usually known as the *recognize-act cycle*, which consists of three phases: Find all rules with satisfied conditions, choose one of these rules, and then execute, or fire, the chosen rule.

```
; Exchange loops on itself swapping any two
; consecutive elements if they are not ordered.
(defrule exchange
  ?f1<-(element (val ?v1) (index ?i1))
  ?f2<-(element (val ?v2) (index ?i2))
  (test (= ?i2 (+ ?i1 1)))
  (test (> ?v1 ?v2))
  =>
  (modify ?f1 (val ?v2))
  (modify ?f2 (val ?v1)))

; Switch fires only once. It sets the array index to
; 0 and it adds a fact to enable rule print.
(defrule switch
  (declare (salience -10))
  =>
  (assert (i 0))
  (assert (step print)))
```

```
; Print loops over itself printing and
; deleting the elements of the array.
(defrule print
  (step print)
  ?f1<-(i ?i)
  ?f2<-(element (val ?v)
          (index ?i))
  =>
  (printout t ?v crlf)
  (retract ?f1)
  (assert (i (+ ?i 1)))
  (retract ?f2))
```

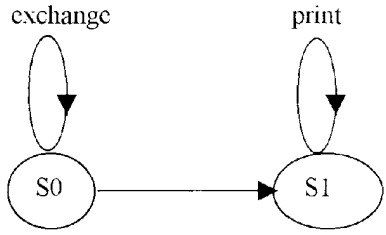

Figure 2. Code and control-flow graph for program to sort an array in ascending order

An example of a forward-chaining rule-based program is shown in Figure 2. This program sorts an array in an ascending order. The program executes as follows. First, rule **exchange** loops over itself swapping

unordered, adjacent elements until the array becomes sorted. When the array becomes sorted, rule **switch** fires setting the array index to 0 and adding a fact to satisfy the condition of rule **print**. **Print** then loops over itself printing and removing the elements of the array. The *salience* declaration in rule **switch** specifies rule priority. The rule with the highest salience value is always chosen for execution when there are multiple rules with satisfied conditions. A rule that does not include a salience declaration is given the default salience 0.

Control-Flow Analysis (CFA) of forward-chaining rule-based programs is a process that extracts control-flow from a source program. CFA produces a directed graph that describes the control-flow between program rules. This directed graph is called a *Control-Flow Graph* (CFG) (Omer, 1993) or a *Restricted Flow Graph* (O'neal, 1993). We use the former name in this paper. A CFG consists of a set of nodes and a set of labeled edges. *Nodes* represent abstracted WM states and labeled *edges* represent valid transitions between abstracted WM states. An edge is a list of three things: an originating state, a rule name, and a destination state.

As an example, Figure 2 shows a CFG for the program that sorts an array. This CFG indicates that **exchange** fires zero or more times leaving the current WM state s_0 unchanged. **Switch** then fires exactly once changing the current WM state to s_1. At s_1, rule **print** executes zero or more times.

Data-Flow Analysis (DFA) produces useful information about procedural programs. DFA works by propagating information through a control-flow graph of a procedural program. Such a graph consists of a set of nodes that represent statements and a set of edges that describe control-flow between statements. A *data-flow problem* is to find information about a certain program entity at each node in the control-flow graph. Virtually all data-flow problems can be modeled and solved in a uniform way using the concept of Monotone Data-flow System. A *Monotone Data-flow System* (MDS) is a tuple D=(L,$\wedge$,TRANS,G,F), where:

1. The pair (L, $\wedge$) is a bounded semilattice with top and bottom elements. L elements are sets of information items. The *meet* ($\wedge$) is a binary

operation that describes how to combine information of two execution paths in the control-flow graph.

2. TRANS is a set of monotonic functions that describe the effect of nodes in N on data-flow information.
3. G is a control-flow graph of a procedural program with a set of nodes N, and a set of edges E.
4. F: N→TRANS is a total function.

An MDS can be solved using the *General Iterative Algorithm* (GIA) (Kildall, 1973). The GIA is efficient in both time and space. It first initializes data-flow information at each node to be the top element in the lattice (L, $\wedge$). Then, it iteratively propagates data-flow information through the control-flow graph until the propagation yields no new information at any node. The direction of information propagation can be bottom-up or top-down. This depends on the problem being solved. The GIA for bottom-up data-flow problems is as follows.

Algorithm: The General Iterative Algorithm (GIA) for bottom-up data-flow problems

Input: An MDS D=(L,$\wedge$,TRANS,G,F) with G=(N,E).

Output: INF: N→L, a total function.

Method: INF is computed for each node by successive approximations.

begin

for every node $n \in N$ **do** INF(n):= *top*;

while changes to any INF(n) occur **do**

for every $n \in N$ **do**

$$INF(n)=TRANS_n(\bigwedge_{s \text{ is a successor of } n} INF(s))$$

end

The GIA always terminates and produces conservative information (Kam et al, 1975). Conservative information takes into account all possible

execution paths, and can be safely used to perform optimizing transformations on the source program. A more detailed discussion of data-flow analysis of procedural programs can be found in (Aho et al, 1988).

3. Live-Rule Analysis

The information to determine when rules finish firing during run-time is obtained by applying data-flow analysis to the CFG of the forward-chaining rule-based program. In this section, we first show how to define the problem of obtaining such information as a bottom-up data-flow problem, then we apply the GIA to this problem.

Given a forward-chaining rule-based program CFG, G, with a set of states S and a set of edges E, we define rule r as *live* at state $s \in S$ if there is an edge $e \in E$ labeled r leaving state s or leaving any of s successors. We also define rule r as *dead* at state s if r is not live at s. For example, in the CFG of Figure 2, rule **exchange** is live at state s_0 and dead at s_1. The *live-rule problem* is to find the set of live rules at each state $s \in S$.

The live-rule problem is a bottom-up data-flow problem that can be modeled as an MDS. L is the set of all subsets of program rules. The meet operation is union since a rule is live at a state if it is live at any of its successors. The bottom element in the semilattice $(L,\cup)$ is the set that contains all program rules and the top element is the empty set. $TRANS_s$ defines the set of live rules at state s as follows. Let X be the set of live rules at all successors of s. $TRANS_s(X)$, the set of live rules at state s, is X union the set of rules that can fire at state s. Applying the GIA to the live-rule problem results in the following algorithm.

Algorithm: The GIA Applied to the live-rule problem

Input: G=(S,E), a CFG of a forward-chaining rule-based program.

Output: live(s), for each state $s \in S$.

Method:

begin

for every state $s \in S$ **do** INF(s):=Φ;

while changes to any INF(s) occur **do**

for every state $s \in S$ **do**

$$INF(s) := [\bigcup_{t \text{ is a successor of } s} INF(t)] \cup \{r \mid r \text{ can fire at } s\};$$

for every state $s \in S$ **do** live(s):=INF(s);

end

As an example, applying the algorithm above to the CFG in Figure 2 yields the following. The set of live rules at s_0 is {**exchange, print, switch**} and the set of live rules at s_1 is {**print**}.

4. Run-time Deletion of Dead Rules

This section presents an algorithm to obtain a forward-chaining rule-based program in which rules are removed when they finish firing during run-time. The algorithm utilizes both CFG of the program and live-rule information to determine when rules finish firing during run-time.

A CFG contains all possible WM states that can be reached during run-time. At any particular time during execution the program is at one of the CFG states. A rule firing might or might not change the CFG state. In the optimized program, each time a rule firing changes the current CFG state, the rules that become dead at the new state are removed. This is accomplished by adding operations to every rule that changes the current state in the CFG to remove the rules that become dead at the new state. For example, the rule **switch** in the CFG of Figure 2 changes the current state from s_0 to s_1. **Switch** is modified in the optimized program to remove the rules that become dead at s_1 which are **exchange** and **switch**.

Only rules that change the current state in the CFG are modified in the optimized program. The rest of rules stay the same. Rules that change more than one state in the CFG cannot be modified directly to delete different dead rules in different firings. However, Based on the author's personal study to a large number of CFGs, rules that change the current state in the CFG are often control rules whose purpose is to force execution to proceed

in stages. At each stage only rules related to that stage are able to fire. Stages correspond to CFG states. In the WM, the current stage is represented by a certain fact. A control rule that changes the current state from s_i to s_j modifies the WM to contain a fact representing the new stage s_j instead of the fact representing s_i. A separate control rule is needed to change the current stage from s_i to s_j since this rule needs to test for the fact representing state s_i and to replace it with the fact representing state s_j. This rule will fire only when execution moves from s_i to s_j and will not occur elsewhere in the CFG.

The optimizing algorithm presented below does not handle rules that change more than one CFG state. The algorithm consists of two steps. First, the set of rules that become dead at each state in the CFG is found using live-rule information. Then, rules that change the current state in the CFG are transformed to remove rules that become dead at the new state. The transformation maintains the semantics of the source program since rules are deleted only after they finish firing.

Algorithm: Program Transformation to Delete Dead-Rules During Run-Time

Input: A forward-chaining rule-based program P, control-flow graph, G=(S,E), of P, and for each state $s \in S$ the set of rules live at state s, live(s). There is no rule in P that changes more than one state in S.

Output: A rule-based program in which the rules that become dead at a state $s \in S$ are removed during run-time when execution reaches s.

Method:

begin

for each state $u \in S$, find

$$D(u) := [\bigcup_{t \text{ is a predecessor of } u} \text{live}(t)] - \text{live}(u);$$

for each rule $r \in P$ that changes a state $t \in S$ to another state $u \in S$ **do**

for each rule $q \in D[u]$ **do**

Add an operation to rule r that deletes q.

end

5. Examples

The execution time improvement obtained from the optimization depends mainly on the matching time of the deleted rules. Significant timesaving results from deleting rules with conditions that require excessive matching time. In addition, deleting a rule early during execution is more timesaving since the condition of the deleted rule will not be matched against WM in more execution cycles.

The results of applying the optimization to three example CLIPS programs are presented below. The running times were obtained using the CLIPS command (*watch statistics*) on a 486 PC. Similar trend in execution time improvement was obtained on other machines. The deletion of rules during run-time was implemented in CLIPS using the *undefrule* action. This action deletes the specified rule and also removes related conditions from the RETE matching network.

Array Sorting

This program and its CFG were introduced above. The CLIPS code and the CFG are shown in Figure 2. The optimization does not reduce the time needed to sort the array because all rules are still live while the rule **exchange** fires (see Figure 2). However, the rules **exchange** and **switch** become dead at state s_1. The two rules are removed by the action of **exchange** in the optimized program, which is shown in Figure 3. This reduces the time needed to print the sorted array. The timesaving for this program depends on the number of firings of **print**, which is equal to the number of elements in the array.

```
; Exchange loops on itself swapping any two
; consecutive elements if they are not ordered.
(defrule exchange
 ?f1<-(element (val ?v1) (index ?i1))
 ?f2<-(element (val ?v2) (index ?i2))
 (test (= ?i2 (+ ?i1 1)))
 (test (> ?v1 ?v2))
 =>
```

```
; Print loops over itself printing and
; deleting the elements of the array.
(defrule print
 (step print)
 ?f1<-(i ?i)
 ?f2<-(element (val ?v)
        (index ?i))
 =>
```

Figure 3. Optimized sort program

```
(modify ?f1 (val ?v2))
(modify ?f2 (val ?v1)))

; Switch fires only once. It sets the array index to
; 0 and it adds a fact to enable rule print.
(defrule switch
 (declare (salience -10))
 =>
 (assert (dead-rule exchange))
 (assert (dead-rule switch))
 (assert (i 0))
 (assert (step print)))

(printout t ?v crlf)
(retract ?f1)
(assert (i (+ ?i 1)))
(retract ?f2))

; Remove rules that become dead during
; execution.
(defrule remove-dead-rules
(declare (salience 10))
?f<-(dead-rule ?rule-name)
=>
(undefrule ?rule-name)
(retract ?f))
```

Figure 3. Contd.

Since the improvement in execution time results only from faster execution of **print**, it is sufficient to consider already sorted arrays. The results of executing the source and optimized programs on already sorted lists are shown in Figure 4. On the average the execution time is reduced by 6%.

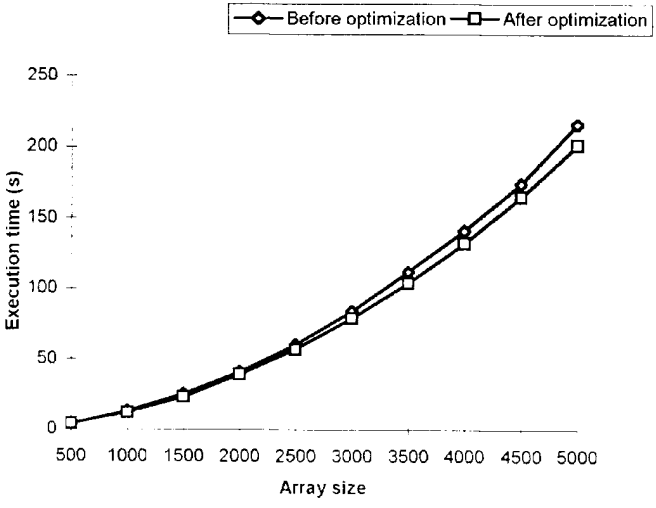

Figure 4. Execution time in seconds for the program of Fig. 2 before and after optimization

Set Operation

This program and its CFG are shown in Figure 5. The program computes the set operation $a=a\cap b\cap c$. The program executes as follows. First, rule **find** loops over itself finding the elements of the three sets intersection. When all intersection elements are found, **switch1** fires adding the fact (*step delete*). This fact makes the condition of rule **remove** satisfied. **Remove** then loops over itself deleting elements of set a. When all a elements are deleted, **switch2** fires adding the fact (*step put*) to WM. This fact makes the condition of rule **put** satisfied. **Put** then loops over itself copying all intersection elements to set a.

```
; Find determines all elements in the intersection.
(defrule find
      (a ?x)
      (b ?x)
      (c ?x)
      =>
      (assert (abc ?x)))

; Switch1 fires only when all elements in the
; intersection are found. It adds a fact to enable
; rule remove.

(defrule switch1
      (declare (salience -10))
      =>
      (assert (step delete)))

; Remove deletes all elements from set a.
(defrule remove
      (step delete)
      ?f<-(a ?x)
      =>
      (retract ?f))
```

```
; Swith2 fires only when all of set a
; elements are removed. It adds a fact to
; enable rule put.
(defrule switch2
      (declare (salience -10))
      ?f<-(step delete)
      =>
      (retract ?f)
      (assert (step add)))

; Put adds all intersection elements to set a.
(defrule put
      (step add)
      (abc ?x)
      =>
      (assert (a ?x)))
```

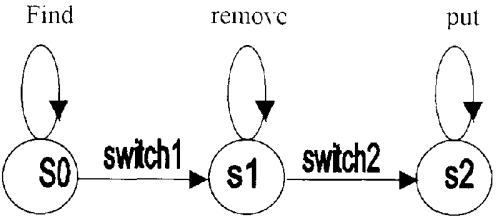

Figure 5. Code and control-flow graph for program to compute the set operation $a=a\cap b\cap c$.

In this program the rules **find** and **switch1** become dead at s_1 (see CFG in Figure 5). The rules **remove** and **switch2** become dead at s_2. Deleting the dead rules at s_1 and s_2 in the optimized program (Figure 6) reduces the time needed to execute the rules **remove** and **put**. The total timesaving depends on the number of firings of these two rules, which depends on the sizes of the three sets and the size of their intersection. More timesaving are obtained for larger sets and larger intersection size. Figure 7 shows the execution time for cases in which the three sets have the same size and their intersection size is half of the sets size. On the average the execution time was reduced by about 86%. This significant improvement is due to the fact that rule **find** which requires significant matching time is deleted early during execution of the optimized program.

```
; Find determines all elements in the intersection.
(defrule find
     (a ?x)
     (b ?x)
     (c ?x)
     =>
     (assert (abc ?x)))

; Switch1 fires only when all elements in the
; intersection are found. It adds a fact to enable
; rule remove.

(defrule switch1
     (declare (salience -10))
     =>
     (assert (dead-rule find))
     (assert (dead-rule switch1))
     (assert (step delete)))

; Remove deletes all elements from set a.
(defrule remove
     (step delete)
     ?f<-(a ?x)
     =>
     (retract ?f))
```

```
; Swith2 fires only when all of set a
; elements are removed. It adds a fact to
; enable rule put.
(defrule switch2
     (declare (salience -10))
     ?f<-(step delete)
     =>
     (assert (dead-rule remove))
     (assert (dead-rule switch2))
     (retract ?f)
     (assert (step add)))

; Put adds all intersection elements to set a.
(defrule put
     (step add)
     (abc ?x)
     =>
     (assert (a ?x)))

; Remove rules that become dead during
; execution.
(defrule remove-dead-rules
  (declare (salience 10))
  ?f<-(dead-rule ?rule-name)
  =>
  (undefrule ?rule-name)
  (retract ?f))
```

Figure 6. Optimized program to compute the set operation $a=a\cap b\cap c$.

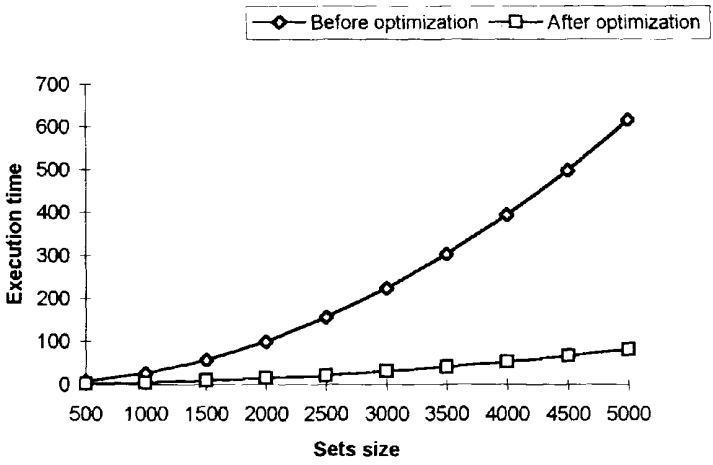

Figure 7. Execution time in seconds or the program of Fig. 5 before and after optimization. The sizes of the sets a, b, and c are equal to sets size. The size o the three sets intersection is half of sets size

Linked List Processing

This program and its CFG are shown in Figure 8. The program counts the number of elements in a linked list that are less than the average of all elements in the list.

```
; Initialize sum and count to 0 and set the current node
; to the first node.
(defrule initialize
  (first-node ?n)
  (not (current-node ?))
  =>
  (assert (count 0))
  (assert (sum 0))
  (assert (current-node ?n)))
```

```
; Increase fires when the value in the current
; node is below average. It adds 1 to below-count
; and advances the current node to the next node.
(defrule increase
  ?f1 <- (current-node ?n&~nil)
  (node (num ?n) (val ?v) (next ?x))
  (avg ?avg)
  (test (< ?v ?avg))
  ?f2<-(below-count ?c)
```

Figure 8. Code and control-flow graph for program to count the number of values less than the average of all values in a linked list

```
; Sum-and-count loops over itself counting and
; summing all node values
(defrule sum-and-count
  ?f1 <-(sum ?s)
  ?f2 <-(count ?c)
  ?f3 <-(current-node ?n&:(neq ?n nil))
  (node (num ?n) (val ?v) (next ?x))
  (not (avg ?))
  =>
  (retract ?f1)
  (assert (sum (+ ?s ?v)))
  (retract ?f2)
  (assert (count (+ ?c 1)))
  (retract ?f3)
  (assert (current-node ?x)))

; Compute fires when the end of the list is reached. It
; calculates the average, resets the current node to the
; first node and ; initializes below-count to 0.
(defrule compute
  (sum ?s)
  (count ?c)
  ?f <-(current-node nil)
  (first-node ?n)
  (not (avg ?))
  =>
  (retract ?f)
  (assert (current-node ?n))
  (assert (below-count 0))
  (assert (avg (/ ?s ?c))))
```

```
  >
  (retract ?f1)
  (assert (current-node ?x))
  (retract ?f2)
  (assert (below-count (+ ?c 1))))

; Just-advance fires when the value in the current
; node is not below average. Its firing changes the
; current node to the next node.
(defrule just-advance
  ?f <-(current-node ?n&:(neq ?n nil))
  (node (num ?n) (val ?v) (next ?x))
  (avg ?avg)
  (test (>= ?v ?avg))
  =>
  (retract ?f)
  (assert (current-node ?x)))
```

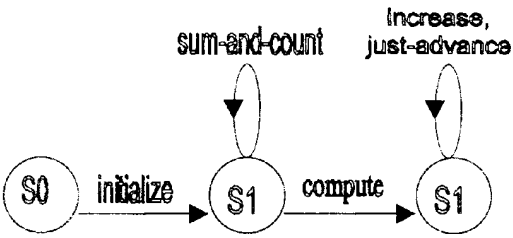

Figure 8. Contd.

The program executes as follows. First, rule **initialize** fires setting the *current-node* to the first node in the linked list and initializing *count* and *sum* to zero. Rule **sum-and-count** then loops over itself counting and summing all the values in the list. When the end of the list is reached, rule **compute** fires calculating the average, resetting *current-node* to the first node, and setting the below average count, *below-count*, to 0. The program then passes through the list again firing either **increase** or **just-advance** at each node. Rule **increase** fires when the value of *current-node* is below average. Its firing increases *below-count* value by 1 and changes the current node to the next node. Rule **just-advance** fires when *current-node* value is greater than or equal to the average. Its firing changes *current-node* to the next node.

Rule **initialize** in this program becomes dead at state s_1 (see CFG in Figure 8). The rules **compute** and **sum-and-count** become dead at state s_2. Hence, the execution of **sum-and-count, increase,** and **just-advance** takes less time in the optimized program, which is shown in Figure 9. The total timesaving depends on the total number of firings of **sum-and-count, increase** and **just-advance** which depends on the size of the linked list. Figure 10 shows the execution time of original and optimized programs as linked list size increases. On the average, the execution time was reduced by about 16%.

```
; Initialize sum and count to 0 and set the current node
; to the first node.
(defrule initialize
  (first-node ?n)
  (not (current-node ?))
  =>
  (assert (dead-rule initialize))
  (assert (count 0))
  (assert (sum 0))
  (assert (current-node ?n)))


; Sum-and-count loops over itself counting and
; summing all node values
(defrule sum-and-count
  ?f1<-(sum ?s)
  ?f2<-(count ?c)
  ?f3<-(current-node ?n&~nill)
  (node (num ?n) (val ?v) (next ?x))
  (not (avg ?))
  =>
  (retract ?f1)
  (assert (sum (+ ?s ?v)))
  (retract ?f2)
  (assert (count (+ ?c 1)))
  (retract ?f 3)
  (assert (current-node ?x)))
; Compute fires when the end of the list is reached. It
; calculates the average, resets the current node to the
; first node and initializes below-count to 0.
(defrule compute
(sum ?s)
(count ?c)
?f<-(current-node nill)
```

```
; Increase fires when the value in the current
; node is below average. It adds 1 to below-count
; and advances ; the current node to the next node.
(defrule increase
  ?f1<- (current-node ?n&~nill)
  (node (num ?n) (val ?v) (next ?x))
  (avg ?avg)
  (test (< ?v ?avg))
  ?f2<-(below-count ?c)
  =>
  (retract ?f1)
  (assert (current-node ?x))
  (retract ?f2)
  (assert (below-count (+ ?c 1))))

; Just-advance fires when the value in the current
; node is not below average. Its firing changes the
; current node to the next node.
(defrule just-advance
  ?f<-(current-node ?n&~nill)
  (node (num ?n) (val ?v) (next ?x))
  (avg ?avg)
  (test (>= ?v ?avg))
  =>
  (retract ?f)
  (assert (current-node ?x)))
; Remove rules that become dead during
; execution.
(defrule remove-dead-rules
(declare (salience 10))
?f<-(dead-rule ?rule-name)
=>
(undefrule ?rule-name)
```

Figure 9. Optimized program to count the number of values less than the average of all values in a linked list.

```
(first-node ?n)                                    (retract ?f))
(not (avg ?))
=>
(assert (dead-rule sum-and-count))
(assert (dead-rule compute))
(retract ?f)
(assert (current-node ?n))
(assert (below-count 0))
(assert (avg (- ?s ?c)))))
```

Figure 9. contd.

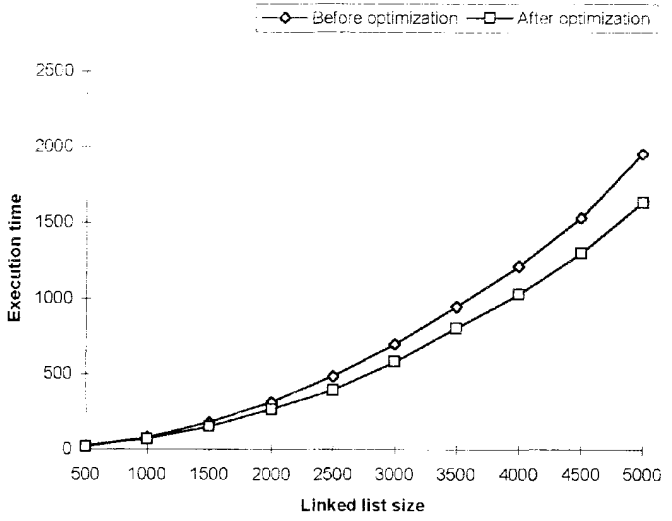

Figure 10. Execution time in seconds for the program of Figure 8 before and after optimization.

6. Conclusion

We have presented a program optimization for forward-chaining rule-based programs. This optimization results in programs in which rules are dynamically removed when they finish firing. The conditions of the deleted rules are not matched against WM in later execution cycles. This reduces

the total execution time of the program. The optimization is safe since rules are deleted only when they finish firing.

Control-flow analysis of forward-chaining rule-based programs and data-flow analysis is used to determine when rules finish firing during-run time. First, control-flow analysis is applied to the source program producing a control-flow graph. This graph describes all possible sequences in which the rules may fire. The nodes of the graph represent working memory states. The program is always at one of these states during run-time. Data-flow analysis is used to determine the set of rules that finish firing at each state in the control-flow graph.

The optimization was applied to three CLIPS programs and the results were presented. These results show that there is a significant improvement in execution time for programs that contain rules that require extensive matching time and finish firing early during execution. Further work is needed to test the optimization on larger, more complex rule-based programs. Work is also needed to explore the results of applying the optimization to programs written in rule-based languages that do not use the RETE algorithm.

References

1. Aho, Sethi, and Ullman. "Compilers: Principles, Techniques, and Tools", Addison-Wesley, Reading, MA, 1988.

2. Bellman, and Walter. Analyzing and correcting knowledge-based systems requires explicit models. *Proceedings of AAAI-88 Workshop on Validation and Testing of Knowledge-based Systems*, Minneapolis, Minn., (1988).

3. Chander, Shinghal, and Radhakrishnan. Using Goals to Design and Verify Rule-Bases. *Decision Support Systems*, **21**, (1988), 281-305.

4. COSMIC. CLIPS Reference Manual, Version 4.3, Artificial Intelligence Section. Lyndon B. Johnson Space Center, COSMIC, 382 E. Broad St., Athens, GA, 30602, (1989).

5. Forgy. Rete: A Fast Algorithm For The Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, **19**, (1982), 17-37.

6. Ishida An Optimization Algorithm for Production Systems. *IEEE Transactions On Knowledge And Data Engineering*, **6**, (1982), 549-558.

7. Kam, and Ullman. *Monotone Data-Flow Analysis Frameworks. Technical Report*, No. **169**, Dept. of Electrical Engineering, Princeton University, New Jersy, USA, (1975).

8. Kildall. A Unified Approach to Global Program Optimization. *1 st POPL*, Boston, MA, (1973), 194-206.

9. Kimura, Kobayashi, Sumiyoshi, and Takebe. A Condition Matching Algorithm for High-Cost Rules in Production Systems: Effectiveness Measurements. *Systems and Computers in Japan*, **26**, (1995), 52-63.

10. Lee, and Kimcheng. Reducing Match Time Variance in Production Systems with HAL. *Proceedings of the Sixth International Conference on Information and Knowledge Management*, (1997), 309-316.

11. Levy, and Rousset. Verification of Knowledge Bases Based on Containment Checking. *Artificial Intelligence*, **101**, No. 2, (1998), 227-250.

12. Lopez, and Kamel. Reorganizing Knowledge to Improve Performance. *IEEE Trans. Knowl. Data Eng.*, **10**, No. 1, (1998), 190-201.

13. Miranker. TREAT: A Better Match Algorithm for AI Production Systems. *Proc. Sixth National Conference on Artificial Intelligence (AAAI)*, (1987), 42-47.

14. Murrell, and Plant. A Survey of tools for the Validation and Verification of Knowledge-Based Systems: 1985-1995. *Decision Support Systems*, **21**, (1997), 307-323.

15. Omer. Control Flow Analysis of Rule-Based Programs. Ph.D. Thesis, University of Houston, Houston, Texas, USA, (1993).

16. O'neal. Comprehending Rule-based Programs: A Graphical Oriented Approach. *International Journal of Man-Machine Studies*, **39**, (1993), 147-175.

17. Stachowitz, Combs, and Chang. Validation of knowledge-based systems. *Proceedings of Second AIAA/NASA/USAF Symposium on Automation, Robotics and Advanced Computing for the National Space Program*, Arlington, Va, (1987).

18. Zlatareva. Verification of Non-Monotonic Knowledge Bases. *Decision Support Systems*, **21**, (1997), 253-261.