AN APPROACH FOR PARALLELIZING OPS5 PRODUCTION SYSTEMS AND A FASTER MATCH, C BASED INDEXING SCHEME ON HYPERCUBE MACHINES

Rajendra K. Shrivastava, S.S. Iyengar, Doris Carver

Department of Computer Science
Southern University
Baton Rouge, LA 70813

ABSTRACT

An algorithm for reducing the amount of time spent by a production system in its MATCH phase is presented. The algorithm utilizes C language’s arithmetic operations that are not efficiently computed in standard production system languages, and also constrits the activity of the MATCH phase as is performed in OPS5 programming language. The parallelization is achieved by distributing the search amongst the nodes of a hypercube; the MATCH phase is an effort to satisfy the LHS [Left Hand Side] of a PR [Production Rule] by the contents of a WM [Working Memory]. C programming language is used for the implementation of the “Monkey and Banana” problem written in OPS5.

I. INTRODUCTION

A. Production Systems

Production Systems, an important area of research in the field of applied Artificial Intelligence, have been used to simulate human knowledge, reasoning and thinking within specific domains. Production Systems are the basis of many rule-based Expert Systems. They are data driven programs that react in a different manner for any modification to the data. The basis for production systems is the MATCH-ACT or the recognize-act cycle (2,3,11,17) where the production rules iterate over the working memory to evaluate the current status and determine the next action to be taken. A typical production system with its components and their interactions is figure 1.

This type of programming has been used to solve problems like object identification, scene recognition, computer aided design, medical diagnostics, geological exploration, robot navigation, path planning, quality control, and machine automation in assembly plants.

Production Memory represents the rule-base knowledge of the production system within a particular domain, and consists of an unordered collection of production rules. These rules are constructed as IF <condition> THEN <action>; where the conjugated set of conditions to be met are called the LHS [Left Hand Side] of a rule and the action of modifying the data-base or drawing conclusions are the RHS [Right Hand Side] of a rule. The knowledge is expected to be static over the program’s execution; however the it may be updated at certain break points within the execution cycle if the production system is capable of executing in a self learning mode.

Working Memory is the dynamic data base, representing the problem statement, on which the production rules operate. The changes are a direct consequence of the actions specified in the RHS of the selected rule. The contents of a working memory are a collection of instantiated elements which have to be compared and matched with the element templates in the LHS of a production rule.

Inference Engine is that part of a production system which embodies the programmers expertise in its decision making mechanism. The inference engine has the following four tasks to perform.

1. MATCH: This is an incremental equivalence operation specified by the LHS of a production rule on the working memory contents. It iteratively evaluates the LHS of the production rules on current working memory and forms the conflict set of production rules that have been found to be true or satisfied. The conflict set is a collection of ordered pairs of the type <Production Rule, List of elements matched by LHS>.

Figure 1: Production System Representation.
2. Conflict Resolution: On the basis of some criterion [priority, probability, recency, ...], select one production rule from the conflict set formed by task 1. The selection is based on application dependent strategy and may be any one or a combination of the following schemes:

i. Fire that rule whose pattern (LHS) is more specific than the others. The rule that has a pattern which forms the superset of other patterns.

ii. Fire that rule whose pattern match has an element that was most recently modified or created in the working memory.

iii. In a non deterministic system involving diagnostics, exploration or cognitive modeling, select that rule which has the highest probability of being true.

iv. Incorporate priorities built into the production system rules and select the rule with the highest priority.

If the conflict set is empty then halt the system.

3. Act: The production system performs the action specified by the RHS of the selected production rule. The RHS may be a conclusion/seduction or any combination of the actions like (modify, make, remove) that affect the working memory.

4. Loop: While working memory is not empty return to task 1 for the MATCH phase.

B. OPS5 Language for Production Systems

OPS5 is a language that is used widely to write production system programs, the working memory is composed of a set of elements that have been defined and instantiated with specific values. OPS5 production rules use the definition of elements and form the LHS as a combination of particular conditions where the binding is by variable names; the LHS is a template which may have a large set of possible matches during the entire life cycle of the production system.

Elements: These are the building blocks of an OPS5 production system program. Elements are similar to the Pascal language record data structure, where the name of the element corresponds to the record name and attributes of the element correspond to the fields of the record. However unlike a record, element attributes can have either a single character string or a numerical value only. Figure 2 illustrates an element named 'monkey'.

An element is the basis of data in the working memory and is bounded by the parenthesis pair as delimiters. The definition of an element is found following the string 'literalize'. Attributes are preceded by the cap character ^ and their values are either alphanumeric constants like (nil, floor), or variables, enclosed within angled brackets like <p>.

Definition of an element sets up the template for the element which may be instantiated to any value in the working memory. These elements are used to define the current working memory contents and also the conditional part of a production rule as condition elements in the LHS of a production rule.

```
(literalize monkey ; representing a memory element named monkey
 at ) ; attribute representing horizontal location of monkey
 on ) ; attribute representing vertical location of monkey
 holds ) ; attribute representing the type of object held by monkey.
```

Figure 2: OPS5 definition of an element named 'monkey'.

```
(p holds::obj-notceiL:at-monkey
 (goal ^status active ^type holds ^obj-name <o> ) ; PR named holds::obj...
 (phys-obj ^name <o> ^weight light ^at <p> ^on <> ceiling ) ; CE named phys-obj
 (monkey ^at <> <p> ) ; CE named monkey

--> (make goal ^status active ^type at ^obj-name nil ^to <p> )) ; RHS for holds::obj..
```

PR is the production rule name.
CE is a condition element name.
LHS is the collection of condition elements preceding the '->' operator.
RHS is the action specified by the production rule holds::obj-notceiL:at-monkey

Figure 3: OPS5 definition of a production rule named 'holds::obj-notceiL:at-monkey'.

490
An example for the definition of an element named 'monkey' is illustrated in figure 2, and a working memory instantiation of the element monkey, is (monkey 'at' 7-7 'holds' nil 'on floor'). The above instantiation of an element states that 'there is a monkey on the floor at a location 7-7 and it is holding nothing'. The value 7-7 is treated as a character string and not as a numerical entity. The same element may be used as a condition element part of the LHS for a production rule as in figure 3.

Production Rules: Rules form the basis for representing knowledge in production systems since they specify the action to be taken under a given set of conditions, represented in the RHS and LHS of the rule respectively. Each rule has a set of conditions that consists of particular instantiations and interacting relationship amongst the elements that forms the LHS. In OPS5 a LHS begins with the string '{p ' and that rule is referenced to by the name that follows it. In figure 3, the production rule name is 'holds::obj-notcell:at-monkey'. Interaction amongst the attributes that share a variable name or a constant value is illustrated as 'goal 'obj-name<oo> <-> phys-obj 'name<oo>' and in '{phys-obj 'at<oo> <-> monkey 'at <o> <oo>}'. The RHS of that rule states '{make goal 'status active 'type at 'obj-name nil 'to <oo>' which means that a new element of class 'goal' is added to the working memory and the MATCH-FACT phase is repeated.

C. Hypercube Architecture

The hypercube simulator used for this research is based on a multitasking kernel for C and Fortran programs that emulates the INTEL IPSC/2 parallel scientific computer (4,8,12) using message passing and shared memory features, hosted on a VAX-11/780 system with BSD 4.3 UNIX operating system.

II. COMPUTATIONAL PROBLEMS

Since a fast recognise-act cycle is the basis of productions systems, several researchers have worked on the problem of improving the MATCH phase. Some estimates have placed the time spent in iterative matching of working memory elements with the production rules in the MATCH phase at 90% (5) of the total execution life. The search mechanism has been partially improved by Forgy (5) in his RETE algorithm and further modified by Mirranzer, Gupta, Shrivastava (6,9,13,14).

Table I is a snapshot of the working memory, and it contains data elements 3,5 and 8 that MATCH the conditions in the LHS of production rule 'holds::obj-notcell:at-monkey', as is illustrated in figure 4.

<table>
<thead>
<tr>
<th>Table I: Working memory contents.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (phys-obj 'at 9-9 'name bananas 'on ceiling 'weight light)</td>
</tr>
<tr>
<td>2 (phys-obj 'at 7-7 'name couch 'on floor 'weight heavy)</td>
</tr>
<tr>
<td>3 (phys-obj 'at 3-3 'name ladder 'on floor 'weight light)</td>
</tr>
<tr>
<td>4 (phys-obj 'at 7-7 'name blanket 'weight light)</td>
</tr>
<tr>
<td>5 (monkey 'at 7-7 'holds blanket 'on couch)</td>
</tr>
<tr>
<td>6 (goal 'type holds 'status active 'obj-name bananas)</td>
</tr>
<tr>
<td>7 (goal 'type at 'status active 'obj-name ladder 'to 9-9)</td>
</tr>
<tr>
<td>8 (goal 'type holds 'status active 'obj-name ladder)</td>
</tr>
</tbody>
</table>

RETE algorithm is used to compute the conflict set without iterating over the working memory during the MATCH phase of a production system's execution. The algorithm maintains an n-ary tree like indexed network illustrated in figure 4, for the production rules, where n is the number of condition elements in the production system, and determines the number of descendants to the ROOT node in the RETE network. The RETE network has four different type of nodes, and they are:

1. Constant-test nodes: Nodes where the attributes are tested for having constant values. For example 'on = ladder' and 'type = holds'.
2. Memory-nodes: Nodes that retain the results of the match from the test node directly preceding it. For example the tokens that match the test 'status = active' are [6,7,8], corresponding to the elements in table I.
3. Two-input nodes: Nodes that test for variable bindings amongst the LHS condition elements of a production rule. The network maintains common nodes for those attributes of condition elements that are shared amongst more than one production rule as variables. The associated two-input node will have entries for a successful match of an attribute variable that is shared amongst the condition elements and satisfy all the left and right antecedents. For example checking on the variable 'name = <oo>' results in [3,8] being stored at the corresponding two-input node where condition element 8 satisfies all the left antecedent for 'CE = goal' and condition element 3 satisfies all the left antecedent for 'CE = phys-obj'. The value of the variable <oo> is 'ladder'.
4. Terminal-nodes: There is exactly one such node for each rule in the production system. A successful match for a rule is be detected by the flowing of a token to the terminal node, as is the case in figure 4, for rule 'holds::obj-notcell:at-monkey'. The RETE algorithm removes the necessity of iterating over the set of production rules by using a tree-structured sorting network which indexes onto the patterns found in the LHS of the production rules. Any change in the working memory is transformed into tokens which are distributed via the ROOT through the network. This results in two type of memory that the algorithm has to maintain. The first type a-memory, is that part of the network which retains the results from the constant-test nodes and the second type b-memory, retains results from the two-input nodes. RETE minimizes the matching phase of the elements in the elements by constructing the search network and maintaining state of partial matches in
α-memory and β-memory, wherein space is compromised in favor of speed. RETE search algorithm is implemented in the commercially available OPS5 programming language. There are two major problems in executing production systems that are written in OPS5, and they are:

1. The first deals with the time spent by the production system as a consequence of the act phase where the working memory is modified. The disadvantage is apparent during the deletion or modification of an element wherein the partial matches in the network have to be unwound, which may lead to memory contention and communication problems in a parallel environment. This unwinding feature of the RETE network is illustrated by a RHS action of the type 'if condition then store a, b in associated two input node'. This form of variable binding could potentially lead to a massive β-memory, since RETE saves all partial matches and the same condition test may have different LEFT or RIGHT antecedents. A good illustration from figure 4 is the duplication of the two-input node, 'name = <o>' for two separate RIGHT antecedent conditions involving the tests 'on = ceiling' and 'on = ceiling', but sharing the same LEFT antecedent. When the complete network is formed for all the production rules there is a large amount of redundant information that is stored in the network for determining a single successful match.

2. The second problem associated with production systems, as implemented by the RETE algorithm, is the potential for a combinatorial explosion of the β-memory. Constant test nodes, forming the α-memory, are relatively few, unlike the two-input node that makes up the β-memory. A two-input node 'name = <o>' may be logically represented as '∀a ∈ LEFT ANTECEDENT, ∀b ∈ RIGHT ANTECEDENT, if a 'name' = b 'name' then store a, b in associated two input node'. This form of variable binding could potentially lead to a massive β-memory, since RETE saves all partial matches and the same condition test may have different LEFT or RIGHT antecedents. A good illustration from figure 4 is the duplication of the two-input node, 'name = <o>' for two separate RIGHT antecedent conditions involving the tests 'on = ceiling' and 'on = ceiling', but sharing the same LEFT antecedent. When the complete network is formed for all the production rules there is a large amount of redundant information that is stored in the network for determining a single successful match.

Miranker's TREAT algorithm (9) utilizing state saving and conflict set support was proposed to implement a more efficient scheme for variable-handling in the match phase of the production systems. TREAT algorithm was implemented on the DADO2 (15,16) machine with a good degree of success for the class of production systems that have a large proportion of working memory as compared to the production memory, which may not be the case for all production systems. The limitation on TREAT'S performance was due to the fine grained architecture of the DADO and DADO2 machines, which were balanced binary tree machines consisting of microprocessors as the processing element at each node and the data being distributed at the terminal nodes of the tree to form the working memory. Subsequently it has been our effort to utilize the medium grained architecture of hypercubes (4,8,12,13) and design a more efficient MATCH algorithm using the cube as a true MIMD machine, to improve the performance of production systems.

Figure 4: Partial RETE network configuration for elements in table 1.
III. MATCH AND PARALLELIZATION ALGORITHM

The time spent in MATCH phase is directly proportional to the sizes of production memory and working memory, and is the most time consuming feature of production systems. In our proposed MATCH algorithm we have taken two approaches to reduce this phase of production systems.

a. The working memory is partitioned into homogeneous blocks of data thereby reducing the time spent by the program in a selective iterative matching of a rule's condition element to data. This partitioning is feasible in a C programming environment and not in OPS5 which treats working memory as one collection of heterogeneous elements.

b. Our new MATCH algorithm also incorporates an indexing scheme which reduces the number of condition elements that are used in a production rule for determining the conflict set.

The indexing scheme is based on contents of the current working memory for a few key elements. These elements are used by an arithmetic hashing function to generate an index value which is used in the C 'switch(index), case' construct, rather than the scalar operations of 'IP ( (elem1.attribute1 = XXX) and (elem2.attribute2 = YYY) THEN (RHS..))' as in OPS5 to determine the matchability of rules with working memory.

Figure 5 outlines the algorithm to be followed for transforming an OPS5 production system into a parallelized C program without loss of integrity. The entries of set CE (Condition Elements), [goal, monkey, phys-obj], are determined from the definition of elements in the program; size of the set CE is the same as the number of unique elements used by the production system.

```
start

START PASS I
input OPS5-PRODUCTION-SYSTEM
CE->NULL; /* the set CE of Condition Element types used in the program; */
TEMP initialized to 0; /* TEMP is a distribution table of CE in each rule. */
SCe->NULL; /* From CE and TEMP determine a set SCe of Condition Elements such that:
  a. the members of the set do not have multiple instantiations within a Production Rule.
  b. the entire set is present in most of the Production Rules. */

while not eof do
  1) add to CE, the name following ( literalize ; /* scanning the definitions of elements; */
  2) generate TEMP; /* scanning the production rules, looking for CE elements; */
end do;

3) from TEMP and CE /* generating SCe ; */
   i) if CE occurs once within most production rules add CEc to SCet ;
   ii) reduce SCet to a subset SCet which occurs in most production rules.
END PASS I

START PASS II
ATRc->NULL; /* set of attributes ATRc that need a minimum Variable Binding; */
while not eof and with SCet do
  4) within most production rules, if A is an attribute of SCet and not instantiated as an
     independent variable add A, to ATRc ;
end do;
END PASS II

START IMPLEMENTATION
V->NULL; /* set of vector values for every Production Rule using ATRc attributes; */
while not eof and with ATRc do
  5) for each firing of a production rule
     i) generate the vector value Vi prior to executing the RHS and
     ii) identify the rule it is associated with;
    6) increment i;
end do;

7) distribute \frac{V_{max}}{N} indexed rules per node;
END IMPLEMENTATION
end
```

Figure 5: The algorithm for parallelization and implementation of OPS5 production systems.
Using TEMP and CE, the second stage is an intermediate step for determining the subset $S_C$ which contains the elements that occur in a majority of production rules. This subset will subsequently be used to determine the set of indexing attributes $A_{TR_{CE}}$ and generate their vector values, at runtime.

Other researchers (10) in their attempt to improve the time complexity of the MATCH phase have proposed, rewriting of production rules with a reduction in number of condition elements of the LHS; while increasing the overall number of production rules thereby reducing the exponent value but increasing the production memory size in a time complexity of $O(P \times W^C)$. Our scheme transforms the scarier MATCH operations into an arithmetic function for evaluating the vector value used in conjunction with the switch-case construct of C. From the three condition elements (goal, monkey, physical-object) in CE a set of condition elements $S_C$ is to be selected such that they fulfill the following conditions. a) The set occurs in a significant proportion of the production rules. This enables the user to cover a large section of the production memory. Based on this criterion alone, we may select all three condition elements goal, monkey and phys-obj, as is clear in table II. b) The members of the set do not have multiple instantiations in any given rule. This ensures the deterministic nature of the decision-vector and reduces the computation required to determine the vector. Using this constraint eliminate the

<table>
<thead>
<tr>
<th>RULE #</th>
<th>CONDITION ELEMENT AND INSTANTIATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>goal:1, monkey:1, phys-obj:0</td>
</tr>
<tr>
<td>2</td>
<td>goal:1, monkey:1, phys-obj:0</td>
</tr>
<tr>
<td>3</td>
<td>goal:1, monkey:1, phys-obj:0</td>
</tr>
<tr>
<td>4</td>
<td>goal:1, monkey:1, phys-obj:1</td>
</tr>
<tr>
<td>5</td>
<td>goal:1, monkey:1, phys-obj:1</td>
</tr>
<tr>
<td>6</td>
<td>goal:1, monkey:1, phys-obj:1</td>
</tr>
<tr>
<td>7</td>
<td>goal:1, monkey:1, phys-obj:1</td>
</tr>
<tr>
<td>8</td>
<td>goal:1, monkey:1, phys-obj:0</td>
</tr>
<tr>
<td>9</td>
<td>goal:1, monkey:1, phys-obj:3</td>
</tr>
<tr>
<td>10</td>
<td>goal:1, monkey:1, phys-obj:1</td>
</tr>
<tr>
<td>11</td>
<td>goal:1, monkey:0, phys-obj:2</td>
</tr>
<tr>
<td>12</td>
<td>goal:1, monkey:1, phys-obj:1</td>
</tr>
<tr>
<td>13</td>
<td>goal:1, monkey:1, phys-obj:1</td>
</tr>
<tr>
<td>14</td>
<td>goal:1, monkey:1, phys-obj:1</td>
</tr>
<tr>
<td>15</td>
<td>goal:1, monkey:1, phys-obj:1</td>
</tr>
<tr>
<td>16</td>
<td>goal:1, monkey:1, phys-obj:1</td>
</tr>
<tr>
<td>17</td>
<td>goal:1, monkey:1, phys-obj:1</td>
</tr>
<tr>
<td>18</td>
<td>goal:1, monkey:1, phys-obj:1</td>
</tr>
<tr>
<td>19</td>
<td>goal:1, monkey:1, phys-obj:1</td>
</tr>
<tr>
<td>20</td>
<td>goal:1, monkey:0, phys-obj:1</td>
</tr>
<tr>
<td>21</td>
<td>goal:1, monkey:1, phys-obj:1</td>
</tr>
<tr>
<td>22</td>
<td>goal:1, monkey:1, phys-obj:1</td>
</tr>
<tr>
<td>23</td>
<td>goal:1, monkey:0, phys-obj:1</td>
</tr>
<tr>
<td>24</td>
<td>goal:1, monkey:1, phys-obj:0</td>
</tr>
</tbody>
</table>

Table II: TEMP, distribution of CE elements.

condition element phys-obj which ranges in occurrence from 0 to 3 in any given production rule and restrict to elements (goal, monkey) to form the set $S_C$. This set consists of the condition elements that will be considered for determining the indexing attributes $A_{TR_{CE}}$. The criterion for forming this set is stated in step 4 of algorithm in figure 5. The set is composed of those attributes that are either instantiated as constants or else instantiated as dependent variables. These attributes required for generating the indexing vector value are \{ goal 'type, goal 'object-name, monkey 'on, monkey 'holds \}. The justifications for that particular selection are provided below and are based on examples from figure 6 illustrating a small subset of production rules for Monkey and Banana problem. i. 'goal 'type' is an attribute that is initialized to a constant in all production rules. ii. 'goal 'object-name' is instantiated either as the dependent variable from the element 'phys-obj 'name << object', constant, or nil value as illustrated in the rule 'on::floor', constant. iii. 'monkey 'on' and 'monkey 'holds' are either instantiated as constants as in rule 'on::floor' of figure 6 or bound to the variable 'goal 'object-name' as in rule 'holds::obj-notceil' of figure 6. iv. 'goal 'status' was not selected since it has just one constant value active in all the production rules; it is not used as vector component but as a condition variable, controlling the while loop as shown in figure 7 for the implementation of the vector indexed MATCH in C code. v. 'monkey 'at' was not selected since it is mainly used in a small subset of production rules that have the attribute 'goal 'type' 'at'; secondly as a variable it is dependent on the attribute 'phys-obj 'name' 'at' bound by the variable 'phys-obj 'name' 'at' which was not a member of the set $S_C$. The rules 'at::monkey' and 'holds::obj-notceil' of figure 6 respectively illustrate the points at hand.

Figure 6: Some OPS5 production rules.
This phase needs the following steps to be executed after identification of ATR<sub>C</sub>, the set of indexing attributes.

1. Execute the OPS5 program using all possible test case scenarios. Keep a trace of the execution along with all relevant information, as illustrated in figure 8. The main data to be retained are the initial working memory contents at the start, the various rules that are executed and the final outcome for that particular scenario.

2. The OPS5 constants are transformed into integers for the C version of production systems. The integer values associated with the OPS5 attribute constants are selected such that

\[ \text{sizeof} \{X\} = \{|\text{NIL, ON, HOLD, AT, COCH, LGHT, ACTV, LADR, FLOR, BANS, CEIL, BLNK, HEVY, SATS}\} \]

select a set of integers {INT} such that sizeof(INT) = sizeof(X) and \( \{a,b \in \text{INT}\} \) and a+b = b+a, a*b = b*a, a>b; this is string equivalence, not equivalence of integer products. An example of a set of integers that does not satisfy the constraints stated above is \{INT - FALSE\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\} is false for \( a = 11, b = 1 \) with respect to the condition \( a>b \). A valid set of integers that may be used is \{INT\} = \{10, 21, 22, 23, 30, 40, 48, 31, 32, 33, 34, 45, 49\).

Table III: Sets INT and ATR<sub>C</sub>

<table>
<thead>
<tr>
<th>10→NIL</th>
<th>21→ON</th>
<th>22→HOLD</th>
<th>23→AT</th>
<th>30→COCH</th>
<th>40→LGHT</th>
<th>48→ACTV</th>
</tr>
</thead>
<tbody>
<tr>
<td>31→LADR</td>
<td>32→FLOR</td>
<td>33→BANS</td>
<td>34→CEIL</td>
<td>35→BLNK</td>
<td>45→HEVY</td>
<td>49→SATS</td>
</tr>
</tbody>
</table>

3. Each rule in OPS5 is translated into a simple C version. At this point no effort is made to optimize the C code except for the partitioning of the working memory on the basis of CE in table I. The only modification is the generation of the vector value as an integer variable 'index'. The index value is the last action executed by the RHS of a rule and is generated as an arithmetic expression which is then used as the switch variable in the C 'case' statement for the hypercube implementation of the production system.

\[ \text{index} = (\text{goal} \times \text{<X>} \times 10^6 + (\text{goal} \times \text{object-name} <Y>) \times 10^3 + \text{<monkey} \times \text{on} <A> \times 10^2 + \text{<monkey} \times \text{holds} <B> \) \]

If ATR<sub>C</sub> has N attributes then the \( n \) component is multiplied by \( 10^{6(n-1)} \) to form the vector where \( n > 1 \). N.

Table IV: Vector values by phase 2 in fig. 7.

<table>
<thead>
<tr>
<th>CASE</th>
<th>TYPE</th>
<th>NAME</th>
<th>ON</th>
<th>HOLD</th>
<th>VECTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ON</td>
<td>LADR</td>
<td>FLOR</td>
<td>NIL</td>
<td>21313210</td>
</tr>
<tr>
<td>2</td>
<td>ON</td>
<td>LADR</td>
<td>FLOR</td>
<td>LADR</td>
<td>21313231</td>
</tr>
<tr>
<td>3</td>
<td>ON</td>
<td>LADR</td>
<td>NIL</td>
<td>ON</td>
<td>21313310</td>
</tr>
<tr>
<td>4</td>
<td>ON</td>
<td>FLOR</td>
<td>LADR</td>
<td>BLNK</td>
<td>21323135</td>
</tr>
<tr>
<td>5</td>
<td>ON</td>
<td>FLOR</td>
<td>COCH</td>
<td>BLNK</td>
<td>21323035</td>
</tr>
<tr>
<td>6</td>
<td>ON</td>
<td>FLOR</td>
<td>LADR</td>
<td>NIL</td>
<td>21323110</td>
</tr>
<tr>
<td>7</td>
<td>ON</td>
<td>FLOR</td>
<td>COCH</td>
<td>NIL</td>
<td>21323010</td>
</tr>
<tr>
<td>8</td>
<td>ON</td>
<td>FLOR</td>
<td>NIL</td>
<td>ON</td>
<td>21323210</td>
</tr>
<tr>
<td>9</td>
<td>ON</td>
<td>FLOR</td>
<td>COCH</td>
<td>BANS</td>
<td>21323110</td>
</tr>
<tr>
<td>10</td>
<td>HOLD</td>
<td>NIL</td>
<td>FLOR</td>
<td>LADR</td>
<td>22110331</td>
</tr>
<tr>
<td>11</td>
<td>HOLD</td>
<td>NIL</td>
<td>FLOR</td>
<td>BLNK</td>
<td>22110335</td>
</tr>
<tr>
<td>12</td>
<td>HOLD</td>
<td>NIL</td>
<td>FLOR</td>
<td>NIL</td>
<td>22110310</td>
</tr>
<tr>
<td>13</td>
<td>HOLD</td>
<td>LADR</td>
<td>LADR</td>
<td>NIL</td>
<td>22313110</td>
</tr>
<tr>
<td>14</td>
<td>HOLD</td>
<td>LADR</td>
<td>FLOR</td>
<td>NIL</td>
<td>22313210</td>
</tr>
<tr>
<td>15</td>
<td>HOLD</td>
<td>LADR</td>
<td>LADR</td>
<td>BLNK</td>
<td>22313315</td>
</tr>
<tr>
<td>16</td>
<td>HOLD</td>
<td>LADR</td>
<td>COCH</td>
<td>BLNK</td>
<td>22330310</td>
</tr>
<tr>
<td>17</td>
<td>HOLD</td>
<td>BANS</td>
<td>LADR</td>
<td>NIL</td>
<td>22333210</td>
</tr>
<tr>
<td>18</td>
<td>HOLD</td>
<td>BANS</td>
<td>FLOR</td>
<td>NIL</td>
<td>22333010</td>
</tr>
<tr>
<td>19</td>
<td>HOLD</td>
<td>BANS</td>
<td>COCH</td>
<td>BLNK</td>
<td>22333315</td>
</tr>
<tr>
<td>20</td>
<td>HOLD</td>
<td>BANS</td>
<td>LADR</td>
<td>BLNK</td>
<td>22333235</td>
</tr>
<tr>
<td>21</td>
<td>HOLD</td>
<td>BANS</td>
<td>FLOR</td>
<td>LADR</td>
<td>22333231</td>
</tr>
<tr>
<td>22</td>
<td>HOLD</td>
<td>BANS</td>
<td>FLOR</td>
<td>BANS</td>
<td>22333233</td>
</tr>
<tr>
<td>23</td>
<td>HOLD</td>
<td>BANS</td>
<td>FLOR</td>
<td>BANS</td>
<td>22333233</td>
</tr>
</tbody>
</table>

Table IV: Vector values by phase 2 in fig. 7.

<table>
<thead>
<tr>
<th>CASE</th>
<th>TYPE</th>
<th>NAME</th>
<th>ON</th>
<th>HOLD</th>
<th>VECTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>AT</td>
<td>NIL</td>
<td>FLOR</td>
<td>NIL</td>
<td>23103210</td>
</tr>
<tr>
<td>25</td>
<td>AT</td>
<td>NIL</td>
<td>COCH</td>
<td>BLNK</td>
<td>23103035</td>
</tr>
<tr>
<td>26</td>
<td>AT</td>
<td>NIL</td>
<td>COCH</td>
<td>NIL</td>
<td>23103010</td>
</tr>
<tr>
<td>27</td>
<td>AT</td>
<td>LADR</td>
<td>FLOR</td>
<td>LADR</td>
<td>23313231</td>
</tr>
<tr>
<td>28</td>
<td>AT</td>
<td>LADR</td>
<td>FLOR</td>
<td>BLNK</td>
<td>23313035</td>
</tr>
<tr>
<td>29</td>
<td>AT</td>
<td>LADR</td>
<td>LADR</td>
<td>BLNK</td>
<td>23313135</td>
</tr>
<tr>
<td>30</td>
<td>AT</td>
<td>LADR</td>
<td>FLOR</td>
<td>NIL</td>
<td>23313210</td>
</tr>
<tr>
<td>31</td>
<td>AT</td>
<td>BANS</td>
<td>LADR</td>
<td>BLNK</td>
<td>23333233</td>
</tr>
<tr>
<td>32</td>
<td>AT</td>
<td>BANS</td>
<td>COCH</td>
<td>BLNK</td>
<td>23333033</td>
</tr>
<tr>
<td>33</td>
<td>AT</td>
<td>NIL</td>
<td>FLOR</td>
<td>BLNK</td>
<td>23103235</td>
</tr>
</tbody>
</table>

An index value of 22333233 would break down into the OPS5 equivalent form of these two element instances in the working memory (goal "type HOLD "object-name BANS) and (monkey "on FLOR "holds BANS). The multiplying base value \( 10^6 \) was derived by the constraint stated as \( \{\forall a, b \text{ INT} \mid (\text{base} \times a) + b = (\text{base} \times b) + a \} \). In trying to identify a base value that will retain the uniqueness of the working memory representation; simple algebra gives us a rule of thumb to determine the value of base as, base = \( 10^i \) where, \( i \) is the lowest positive integer for the condition \( 10^i > \text{maximum value of the set INT} \) to hold true. For the Monkey and Banana production system, which we have translated into C code, it is found that max(INT) is 49 from table III, since \( 100 > 49 \), the base value is 100 for this implementation.
1. ORIGINAL OPSS CODE

(p hold::obj-notceill::at-monkey
(goal "status active" type holds "obj-name <o>"
(phys-obj "name <o>" "weight light" at <p> "on <> ceiling"
(monkey "at <> <p>"
---> RHS

2. INTERMEDIATE C CODE GENERATING THE VECTOR VALUES

switch (WM_gol(GL_cur)->type) {
  case HOLS:
    for(i=0;i<OB_cur;i++)
      if(WM_gol(GL_cur)->nam == WM_obj[i]->nam) WM_gol->nam = i;
      if((WM_mon->at == WM_obj[WM_gol->nam]->at) &&
        (WM_obj[WM_gol->nam]->wght == Lght) &&
        (WM_obj[WM_gol->nam]->on == CEIL))
      { indz = WM_mon->on*100 + WM_mon->holds +
         WM_gol(GL_cur)->type*1000000 + WM_gol(GL_cur)->nam*10000;
         c0un++;
         arr[coun] = indz;
         printf("INDEX IS %d at %d in %d *",indx,mynode(coun));
         RHS }
    break;
  } ....* end of case *
  } ....* end of switch *

3. FINAL C CODE INDEXING ON THE VECTOR VALUE 'indx'

switch (indx) {
  case 22313210: {
    for(i=0;i<OB_cur;i++)
      if(WM_gol(GL_cur)->nam == WM_obj[i]->nam) WM_gol->nam = i;
      RHS;
      indx = WM_mon->on*100 + WM_mon->holds +
         WM_gol(GL_cur)->type*1000000 + WM_gol(GL_cur)->nam*10000;
      break;
  } ....* end of case *
  } ....* end of switch *

Figure 7: Phases in transforming an OPS5 production rule to indexed C code.

4. The intermediate C program is executed to generate the set of vectors corresponding to the column containing VECTOR entries in table IV. Each vector value is associated with the firing of a production rule for that particular instantiation of the working memory elements, monkey and goal. The values are used for indexing and replace the condition elements used in S_C within each production rule associated with that vector value as illustrated in figure 7.

V. CONCLUSIONS

A sequential OPS5 production system has a time complexity of the order O(W^3) to determine a successful match for a rule, where W is the average size of working memory and C is the average number of condition elements in the LHS of a production rule. Since a production memory has P rules in the production memory the time complexity is of order O(P*W^3) to evaluate all the rules and generate the conflict set. We have tried to reduce the value of P, W and C without modifying the outcome of the program. These three main goals were achieved as follows, by our algorithm, as a distributed production system on a hypercube, for the Monkey and Banana problem.

A. Working Memory

The reduction in working memory size W is due to homogeneous partitioning, on the basis of condition elements in CE. Table I as a heterogeneous structure has 8 items while as homogeneously partitioned structures has a maximum of 4 elements. The homogeneous structures are 'WM_gol(),WM_obj(),WM_mon' in figure 7, and reduces the time complexity to an order of O(P*W^3).

B. Condition Elements

The reduction in condition element size C is due to the indexing scheme as outlined in the paper, and leads to a time...
**C. Production Rules**

Let the number of production rules in the OPS5 program be \( P \), the number of vectors generated be \( V_{max} \), and the number of processors in the hypercube be \( N \). The time for all the rules to be considered would have a complexity of \( O\left(W C^2 V_{max}/N\right) \), where \( V_{max}/N \) is the time required by the hypercube system to execute the correct switch instruction for a 'case' block of that size. The only reason for our implementation to be slower than the OPS5 version would be the time to evaluate switch-case value for a block of size \( V_{max}/N \) in \( C \), is greater than the time spent in evaluating \( P \) production rules in OPS5. This might hold true in those production systems that have an extremely large variation in working memory contents and normally do not require a speedy recognize-act cycle; a set of such production systems are tutorials for mathematics, grammar, spelling and etc.

**REFERENCES**


