PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

Performance Evaluation of the Multiple Root Node Approach to the Rete Pattern Matcher for Production Systems†

Andrew Sohn
Department of Computer and Information Science
New Jersey Institute of Technology
Newark, NJ 07102, sohn@cis.njit.edu

Jean-Luc Gaudiot
Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-2563, gaudiot@usc.edu

Abstract- Much effort has been expanded on special architectures and algorithms dedicated to efficient processing of the pattern matching step of production systems. In this paper, we investigate the possible improvement on the Rete pattern matcher for production systems. Inefficiencies in the Rete match algorithm have been identified, based on which we introduce a pattern matcher with *multiple root nodes*. A complete implementation of the multiple root node-based production system interpreter is presented to investigate its relative *algorithmic* behavior over the Rete-based Ops5 production system interpreter. Benchmark production system programs are executed (*not* simulated) on a sequential machine Sun 4/490 by using both interpreters and various experimental results are presented. Our investigation indicates that the multiple root node-based production system interpreter would give a maximum of up to 6-fold improvement over the Lisp implementation of the Rete-based Ops5 for the match step.

1 Introduction

The importance of production systems in artificial intelligence (AI) has been repeatedly demonstrated by a large number of expert systems. As the number and size of expert systems grow, there has however been an emerging obstacle in the processing of such an important AI application: the large match time. In rule-based production systems, for example, it is often the case that the rules and the knowledge base needed to represent a particular production system would be on the order of hundreds to thousands. It is thus known that applying a simple matching algorithm to production systems would yield intolerable delays. The need for faster execution of production systems has spurred research in both the software [2,3,7,8] and hardware domains [6,11].

In the software domain, the Rete state-saving match algorithm has been developed for fast pattern matching in production systems [2]. The motivation behind developing the Rete algorithm was based on the observation, called *temporal redundancy*, which states that there is little change in database between cycles. By storing the previous match results and using them at later time, matching time can be reduced [1].

Inefficiencies in the state-saving Rete algorithm were identified, based on which the non-state-saving Treat match algorithm was developed [10]. The motivation behind developing the Treat algorithm was McDermott's conjecture, stating that the retesting cost will be less than the cost of maintaining the network of sufficient tests [9].

† This work is supported in part by the NSF under grant No. CCR-9013965.

In this paper, we further identify the inefficiencies of the Rete algorithm, based on which we introduce a pattern matcher with *Multiple Root Nodes* (MRN). Section 2 gives a brief introduction to production systems and the Rete match algorithm. Section 3 explicates the inefficiencies of the Rete matcher. A Lisp implementation of the MRN-based production system interpreter is then presented along with the distinctive features of its implementation.

Section 4 presents benchmark production system programs and experimental results on both the Rete-based OPS5 interpreter and the MRN-based interpreter. Various statistics gathered both at compile time and runtime are presented as well. Performance evaluation on the two interpreters are made in Section 5 in terms of number of comparison operations and execution time. The last section concludes this paper.

2 Background

2.1 Production systems

A production system as shown in Figure 1 consists of a *production memory* (PM), a *working memory* (WM), and an *inference engine* (IE). PM (or rulebase) is composed entirely of conditional statements called productions (or rules). These productions perform some predefined actions when all the necessary conditions are satisfied. The left-hand side (LHS) is the condition part of a production rule, while the right-hand side (RHS) is the action part. LHS consists of one to many elements, called *condition elements* (CEs) while RHS consists of one to many actions.

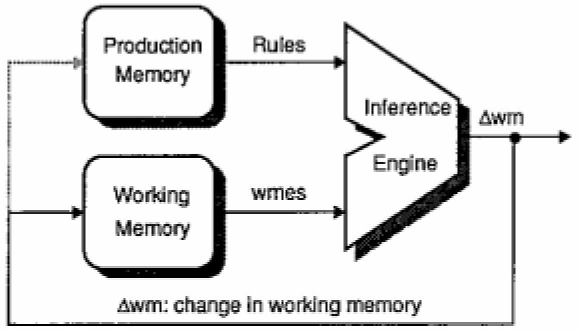

Figure 1: An architecture of production systems

The productions operate on WM which is a database of assertions called *working memory elements* (wmes). Both condition elements and wmes have a list of elements, called *attribute-value pairs* (avps). The value to an attribute can be either *constant* or *variable* for CEs and can be constant only for wmes. A simple production system with one rule is shown in Figure 2. The inference engine executes an inference cycle which consists of the following three steps:

- *Pattern Matching*: The LHSs of all the production rules are matched against the current wmes to determine the set of satisfied productions.
- *Conflict Resolution*: If the set of satisfied productions is non-empty, one rule is selected. Otherwise, the execution cycle simply halts.
- *Rule Firing*: The actions specified in the RHS of the selected production are performed.

The above three steps are also known as Match-Recognize-Act, or MRA. The inference engine will halt the production system either when there are no satisfied productions or a user stops.

2.2 The Rete match algorithm

The Rete match algorithm is a highly efficient approach used in the matching of objects in production systems [2]. The simplest possible matching algorithm would consist in going through all the rules and wmes one by one to find match(es). The Rete algorithm, however, does not iterate over the wmes to match all the rules. Instead, it constructs a condition dependency network like shown in Figure 2, saves in the network results from previous cycles, and utilizes them at a later time.

```
Production Memory                     Working Memory
Rule1:                                wme1:  [(p 1) (q 2) (r •)]
[(c X) (d Y)]          ;CE1           wme2:  [(r =) (d +)]
[(b Y)]                ;CE2           wme3:  [(c •) (d +)]
[(p 1) (q 2) (r X)]    ;CE3           wme4:  [(b 3)]
→                                     wme5:  [(b +)]
[Remove (b Y)]         ;Action 1      wme6:  [(p 1) (q 3) (r 7)]
```

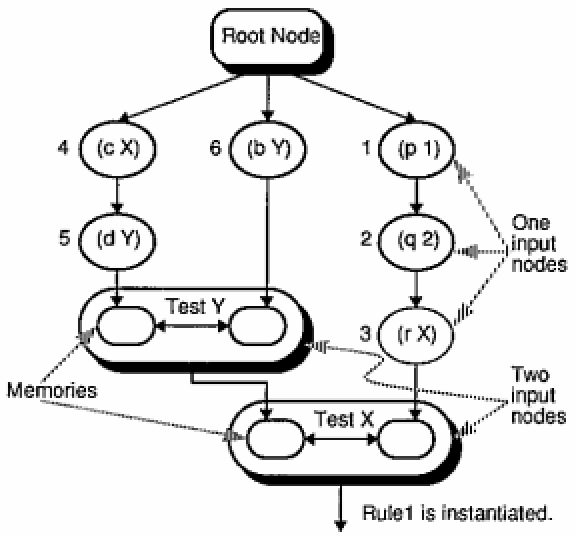

Figure 2: A Rete network for Rule 1.

Given a set of rules a network is built which contains information extracted from the LHSs of the rules. Figure 2 depicts a network for Rule 1, with the following nodes:

- *Root Node* (RN) distributes incoming tokens (or wmes) to sequences of children nodes, called one-input nodes.
- *One-Input Nodes* (OIN) test intra-element features contained in a condition element, i.e., compare the value of the incoming wmes to some preset values in the condition element. For example, CE1 of Rule1 contains 2 intra-element features and therefore 2 OINs are needed to test them. The test result of the one-input nodes are propagated to nodes, called two-input nodes.
- *Two-Input Nodes* (TIN) are designed to test inter-condition features contained in two or more condition elements. The variable *X*, which appeared in both CE1 and CE3, must be bound to the same value for rule instantiation. Attached to the TINs are left- and right memories in which wmes matched through OINs are saved. The result from two-input nodes, when successful, are passed to nodes, called terminal nodes.
- *Terminal Nodes* (TN) represent instantiations of rules Conflict resolution strategies are invoked to select and fire a rule.

There are other variations to the nodes listed above. Given the above network, the Rete algorithm performs pattern matching and we shall not go into detail. See [1,2,5] for more details.

3 The MRN Matcher and Its Implementation

The multiple root node based interpreter is presented along with its Lisp implementation.

3.1 The MRN Matcher

The Rete algorithm described earlier presents two apparent bottlenecks: one in the root node and the other in two-input nodes, as illustrated in Figure 3. Tokens coming into the root node will pile up on the input arc of the root node since there is one and only one root node which can distribute tokens one at a time to all CEs. For the network shown in

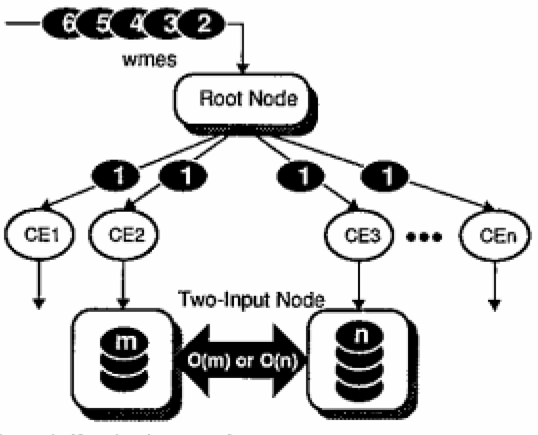

Figure 3: Two bottlenecks of the Rete. (1) piling up of wmes on an arc of the root node, resulting in a sequential distribution of wmes to all CEs one at a time. (2) $O(n)$ or $O(m)$ comparisons in TINs.

Figure 3 where there are n condition elements, the root node will have to make nx distributions to the network when x wmes are present on the input arc of the root node.

The second inefficiency can also be seen on Figure 3. Assuming that m tokens are stored in the left memory of the two-input node and a token is matched on the right input. The arrival of this last token will trigger the invocation of m comparisons with the wmes received and stored in the left memory. Should the situation have been reversed and n tokens be in the right memory, a token on the left side would provoke n comparisons. The internal workings of this two-input node are therefore purely sequential. In order to avoid wasting time in searching the entire memory, an effective allocation of two-input nodes and one-input nodes should be devised. In this paper, we will limit ourselves to the first bottleneck. Discussions on the second bottleneck can be found in[4,5].

The first bottleneck described above can be resolved by introducing *multiple* root nodes (MRN) in the network, as depicted in Figure 4. This introduction of multiple root node is based on the observation that a wme that has n AVPs never matches a CE that has m AVPs where $n<m$. For example, a wme, [(a 1) (b 2)], cannot match a CE, [(a X) (b Y) (c Z)], where X, Y, Z are variable, since the wme is missing the third AVP (c Z). However, a wme [(a 1) (b 2) (c 3) (d 4)] can match the CE. One should note here that the above observation is based on algorithmic behavior, not Ops5 syntactic behavior.

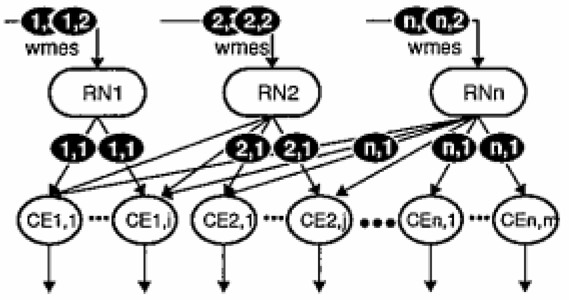

Figure 4: An MRN network. RNn distributes wmes to CEs under RN1 through RNn. A wme (i,j) refers to a wme with i AVPs, where j signifies its arrival order. The MRN network also demonstrates a parallel distribution of wmes, where n RNs can simultaneously distribute n different wmes to the network.

Constructing an MRN network is straightforward. All LHSs are split into condition elements (CEs). All CEs are grouped based on the number of AVPs in a CE, i.e., a CE with n AVPs belongs to a group n. Associated with each group is a root node which distributes a set of wmes to a particular group of CEs of the MRN network. For example, RN2 of Figure 4 distributes wmes with 2 AVPs to those CEs, where each CE has not more than 2 AVPs.

Suppose that the network has n groups, each of which has equally m CEs, i.e., the total number of CEs is nm. Assuming that the number of wmes generated in each production cycle is constant, i.e, k, then the original Rete network will need nmk distribution. Assuming that k wmes are equally distributed over the n groups, i.e., k/n wmes per each group, the MRN network will only need $(1+2+...+n)mk/n$ distributions. For an even distribution of wmes over groups, the MRN matcher is guaranteed to yield 2-fold improvement over the Rete network. Next section will substantiate our prediction. In the mean time, we shall present the Lisp implementation of the MRN-based matcher.

3.2 Characteristics of the MRN implementation

The MRN-based production system has been completely implemented in Common Lisp from scratch. A complete listing of Lisp codes can be found in [12]. Its functionality is 100% up to the Rete-based OPS5. The main features of the MRN implementation are:

- ❑ Free of global variables, except a single one which traces the number of wmes generated during the lifetime of a particular production system program,
- ❑ Over 90% of the functions written in tail-recursion, and
- ❑ A simple data structure using defstruct of Lisp.

A major reason to avoid using global variables is in that the program should be easily ported to various multiprocessor environments without having to change much of its source codes. By not using global variables, the potential communication and synchronization overhead between processes would be reduced when ported to a multiprocessor environment. Furthermore, encapsulating the scope of variables within a function would allow us to analyze the data dependency, if any, between functions, thereby resulting in easy program partitioning. The ultimate goal of parallel processing, extracting and exploiting more potential parallelism from given codes, would then become within a reachable distance. To substantiate this claim, the MRN-based production system interpreter has been implemented in a data-flow language SISAL (Streams and Iteration in a Single Assignment Language) and is currently being ported to multiprocessors, including shared memory multiprocessors such as Cray!

Much effort has been spent on writing the program in tail recursion. One reason to do so was also due partly to the portability to various multiprocessor environments. When functions are written in tail recursion, it can be much easier to understand its behavior since the program tracing is automatic. This easiness in understanding of the behavior of a program will directly translate into an easy conversion to iterations. Those vectorizing compilers or parallelizing compilers can be readily used to convert the Lisp programs into a language suitable for vector or multiprocessors.

The third feature, a simple data structure defstruct, would not necessarily be considered a good feature. The main reason employing defstruct is that it will simplify the implementation process due to its structuredness. This structured approach will shield the data dependency between data, i.e., dynamically changing memories in the network. However, this dynamic data structure consumes more memory space than other data structures such as lists. There is certainly a trade-off between the runtime memory space and the easiness in programming and debugging. Due to the space constraints, we shall not illustrate implementation details. Complete implementation details can be found in [12].

4 Experimental Results

Benchmark production system programs are presented along with the surface characteristics measured at compile time. Both the Rete-based OPS5 and MRN-based interpreters were executed on Sun 4/490 to measure their *algorithmic* performance. Statistics collected at runtime are: the execution time of a match step, the number of comparison operations for one-input nodes, and the distribution of wmes. All the measurements are done against production cycle numbers.

4.1 Surface characteristics of benchmark programs

The five programs chosen for performance analysis are commonly used ones, as seen from Table 1. Note that the size of production systems is not central to its performance evaluation. Indeed, Gupta has commented that (1) we should not expect smaller production systems (in terms of number of productions) to run faster than larger ones, and (2) there is no reason to expect that larger production systems will necessarily exhibit more speedup from parallelism [5]. The programs used in this study are:

- ❑ *Brick Sorting*, to pick a brick from a pool and place them in ascending or descending order,
- ❑ *Monkey and Banana* (MAB), for a monkey to grab a banana hanging from the ceiling,
- ❑ *N Monkeys and M Bananas* (NMAB), the MAB with *n* monkeys and *m* bananas,
- ❑ *Waltz Labeling*, a labeling algorithm developed in computer vision, and
- ❑ *N-Queen*, a classical problem which places *n* queens on *n*x*n* board.

PS	a	b	c	d	e	f	g	h	i
Brick	7	16	15	2336	2	4	4	20	60
MAB	25	70	43	8409	59	5	14	16	58
5MAB	23	60	43	45195	39	5	12	113	278
Waltz	48	198	100	174891	90	5	40	245	297
8-Queen	19	68	71	151985	36	6	11	1044	3866

Table 1: Characteristics of benchmark production systems, where PS=production system program, a=No of rules, b=No of CEs, c=No of acts, d=OINs executed, e= No of TINs, f=No of groups, g=Avg CEs/group, h=Rule firings, i=WMEs generated.

The information collected from the above five production system programs characterize various aspects of the benchmark programs. Our purpose is to measure the relative performance of the MRN approach in terms of execution time along production cycles. What is important in our performance evaluation is the information on groups of a production system program. Indeed, we find that even a small size production system program such as Brick Sorting problem would suffice.

4.2 Measurements on grouping

Grouping the condition elements (CEs) based on the number of Attribute Value Pairs (AVPs) is central to the MRN approach. This would allow us to partition the production systems into many pieces each of which can be processed independent of the incoming newly generated wmes. Measuring the distribution of condition elements over groups at compile time, we can predict the potential parallelism in the given production systems. Figure 5 depicts the distribution curve, where the *x*-axis shows group numbers and the *y*-axis the percentage of each group in a particular production system program.

In the Brick Sorting problem, there are four different groups, where a group-*n* contains condition elements, each of which has *n* Attribute-value Pairs. For example, Group2 occupies slightly above 30% whereas Group 5 does 6% of the total number of CEs. Condition elements of Monkey and Banana are relatively equally distributed over four groups, compared to that of Waltz Labeling where one group is dominant over other groups. This dominance of a group over another is not desirable and does not yield a good performance. We shall come back to this analysis shortly.

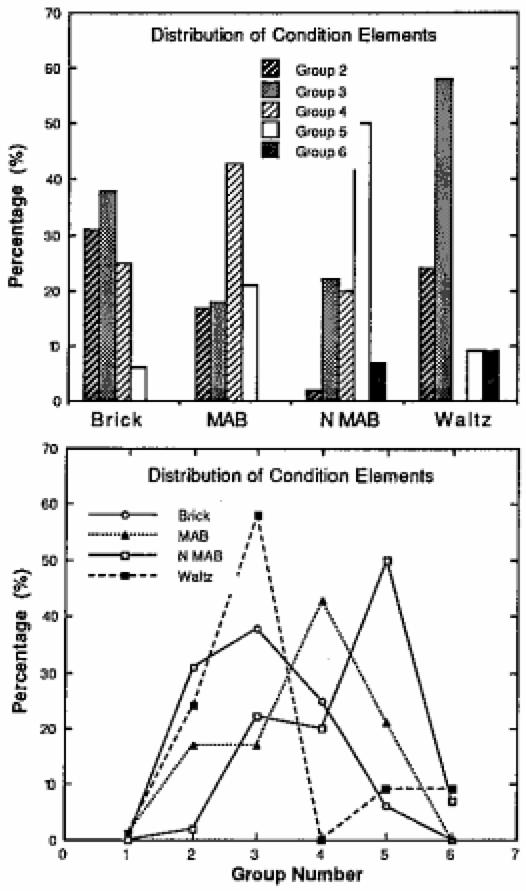

Figure 5: Distribution of CEs over groups measured at compile time.

4.3 Execution time on one-input nodes

Figure 6 shows the execution time of matching one-input nodes measured at each production cycle. There are several points at which execution time run off the boundary. Several points running off the boundary are unimportant since our purpose is to show the relative performance.

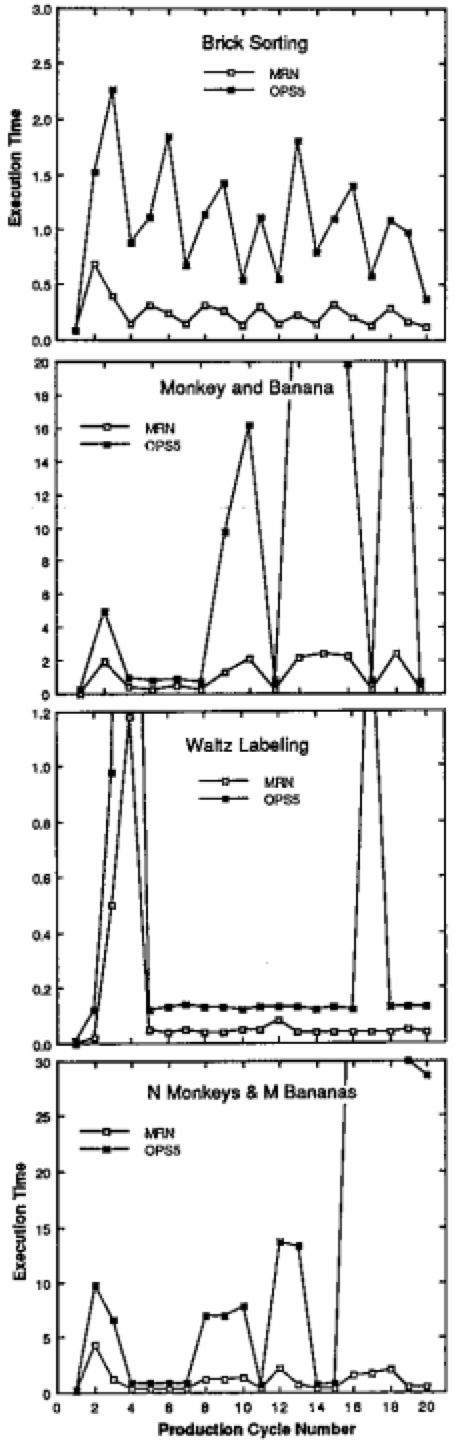

Figure 6: Execution time profile of matching one-input nodes.

For Brick Sorting and Waltz, it appears that both the MRN and OPS5 show a rather regular behavior while they maintain a reasonably constant distance between the two curves along the *x*-axis. For example, in the Waltz, the differences between two execution time curves for the cycle numbers 5 to 16 are relatively constant, except at the cycle numbers 3, 4, and 17. A similar behavior is also observed in Brick. This kind of proportional distance between two curves is important in predicting the possible outcome of the MRN approach.

The MAB and NMAB, however, exhibit a slightly different behavior compared to Brick and Waltz. For example, the MRN curve in MAB gives an amplification factor higher than the one for Brick or Waltz. This irregular behavior is due partly to the memory management policy, *garbage collection*, in Lisp which contributes to inaccurate performance measurements. We shall give a more accurate measurement shortly. Over all, it is obvious that the MRN outperforms the OPS5 in any of the four problems.

4.4 Number of comparison operations

Another criterion to measure statistics at runtime is counting the number of comparison operations. Consider the following simple Lisp function member:

```
(defun member (a l)
    (cond  ((null l) nil)
           ((equal a (car l)))
           (t (member a (cdr l)))))
```

Suppose that the function is called with (member 1 '(2 6 4 7 1)). It is clear that the function member will be called five times and therefore, the number of comparison operations will be five. Figure 7 shows the number of comparison operations for four programs. When we considered the execution time, we discussed that the behaviors of the four programs are rather irregular. The MRN curve of MAB in Figure 6 gave an amplification factor higher than the one for Brick or Waltz. However, that irregular behavior no longer persists in Figure 7. This consistent behavior is due mostly to the new criterion. Figure 7 again demonstrates that MRN outperforms OPS5 for all programs.

4.5 Distribution of groups

Figure 8 shows the runtime distribution of wmes and condition elements for four programs. Take the Brick Sorting, for example. At runtime, there is no wme generated for group2, group4, and group6. Those wmes generated for Brick at runtime fall into either group3 or group5. As we can observe from Figure 8, there is a considerable amount of discrepancy between the runtime wme distribution and the compile CE distribution.

For MAB, however, the situation becomes different. As we can observe from Figure 8, the discrepancy for MAB becomes much smaller compared to that for Brick. MAB and NMAB show a relatively low discrepancy whereas Brick and Waltz show a rather high discrepancy in terms of the wme distribution and the CE distribution.

Contrary to the compile time distribution of condition elements, most of the wmes generated at runtime fall into a few distinctive groups. All the four problems have basically two groups actively working at runtime. However, these distribution curves are problem dependent and there is no single rule which can predict the behavior of the runtime distribution of wmes. A simple conclusion we could draw from these discrepancy plots would be that more the discrepancy there is, more the improvement there will be.

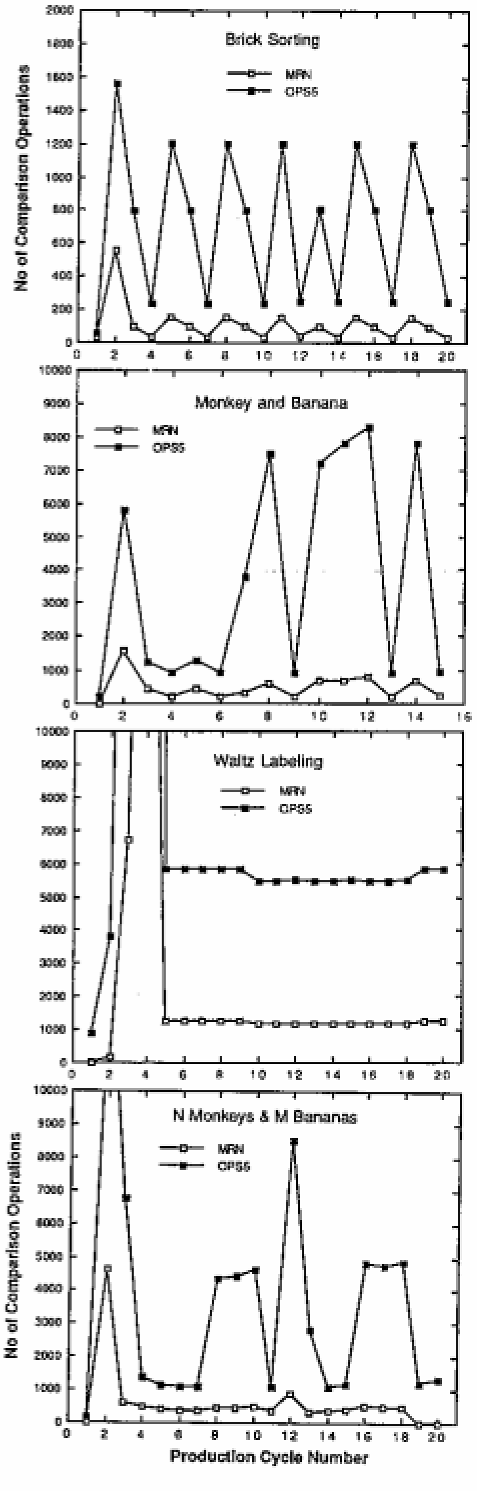

Figure 7: Number of comparison operations on one-input nodes.

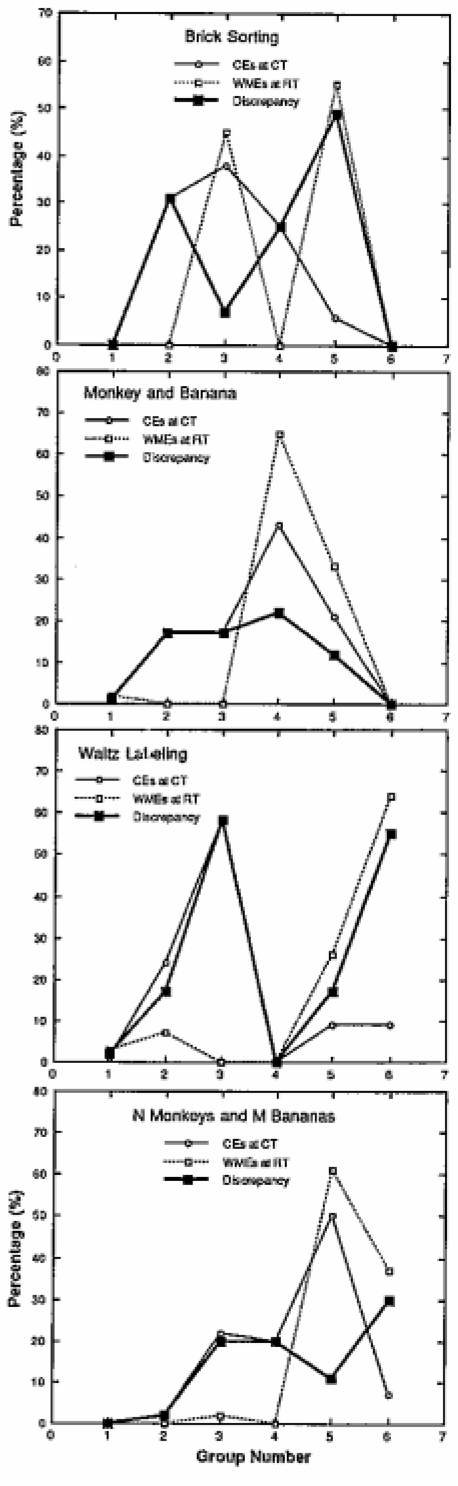

Figure 8: Runtime distribution of wmes.

5 Performance Evaluation

Based on the foregoing three different types of observations, i.e., one-input match time, number of comparison operations, and distribution of wmes, we analyze the performance of both interpreters.

5.1 Comparison of MRN and OPS5

Figure 9 shows the ratio of MRN to OPS5 on one-input match time for four different programs. Here, the ratio means simply the comparison of two match time units for one-input nodes. Again, x-axis is plotted against the production cycle numbers whereas y-axis indicates the ratio of two different approaches.

For Brick Sorting, for example, the one-input match time of OPS5 at production cycle number 13 is about 8 times more than that of MRN. For NMAB, the one-input match time of Ops5 at the cycle 13 is about 17 times more than that of MRN. It is clear that there is a substantial improvement, ranging from 2 to over 20, depending on the programs and the production cycle number.

Figure 10 gives a more accurate performance measure. It uses the number of comparison operations at each production cycle. The x-axis is plotted against the production cycle number while the y-axis is again the ratio of the MRN-based match to Rete-based match. To closely examine the improvement curve, consider again the production cycle number 13 as we did for Figure 9.

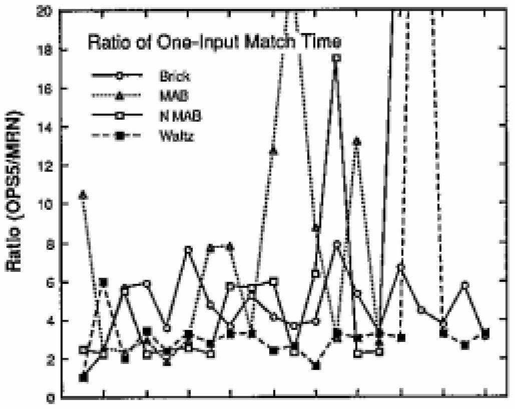

Figure 9: Ratio of one-input match time.

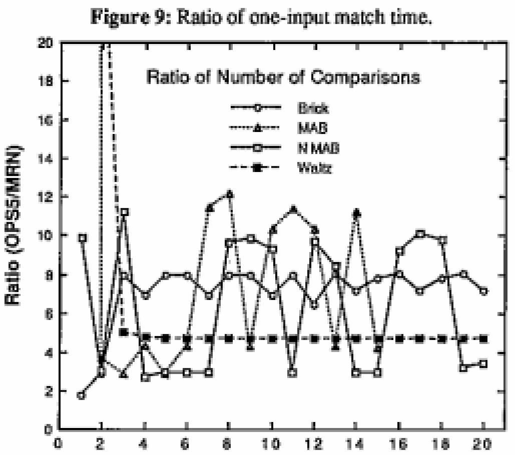

Figure 10: Ratio of number of comparison operations.

For Brick Sorting of Figure 10, the number of comparison operations performed by the Rete-based match at cycle 13 is about 8 times more than that by the MRN-based match. This improvement of 8 is exactly the same as the one we obtained from Figure 9. For NMAB, the improvement, however, becomes different from what we would expect. Closely examining the NMAB curve of Figure 10 at cycle 13, we find that the improvement is 8! We remember that the improvement we obtained from Figure 9 for NMAB at cycle 13 is 17. This is not surprising because measuring the real time can be affected by many factors which we iterated several times. Nevertheless, it is clear from the two improvement curves plotted in Figure 9 and Figure 10 that the MRN-based match algorithm outperforms the Rete-based match algorithm. Since the objective here is to compare the performance of the two match algorithms, the two figures would suffice the stated objective.

There are some other experimental results which are of particular interest but due to space constraints we shall have to be content with what we have presented thus far. When the two figures are summed and averaged along the production cycle number of x-axis, it gives an eventual improvement of *six*. The average improvement of the MRN approach over OPS5 on one-input match time would reach to *six* fold for the four production programs considered in this study.

5.2 Discrepancy in the distribution of wmes and CEs

It is interesting to observe the discrepancy between the compile time distribution of condition elements and the runtime distribution of wmes. By finding the discrepancy between them, we can more accurately locate the behavior of each production system, thereby identifying the potential improvement for a given production system program.

Figure 11 displays all the discrepancies for the four programs. Note the discrepancy curves in Figure 11 and the improvement curves of Figures 9 and 10. Among the four discrepancy curves, the Brick curve has a high and regular behavior, which can in turn translate to a high im-

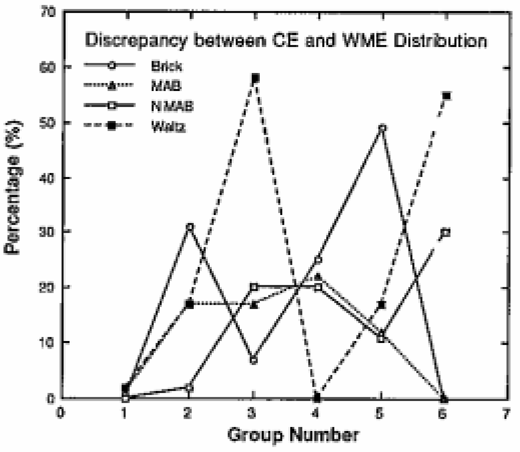

Figure 11: Discrepancy between CE and wme distribution.

provement. The curve for Brick in Figure 11 verifies this relation in which the improvement is high when the fluctuation is low.

However, the above statement on the relation between the discrepancy and the improvement would have to be further substantiated by more experimental results. Most problems are runtime dependent and a simple prediction rule would be problematic. Based on our observations, it could be concluded that if there is more discrepancy between the compile time distribution of CEs and the runtime distribution of wmes, the production system program would have more potential parallelism in match step.

A complete execution time of production cycles is illustrated in Figure 12 to help give an overall view of the two interpreters. Again, the x-axis is plotted in production cycle numbers but the y-axis at this time is plotted in the total production cycle time. The total matching time is dominantly high compared to the selection time or action time as Forgy has indicated [2]. In any case, the MRN-based production system interpreter outperformed the Rete-based OPS5.

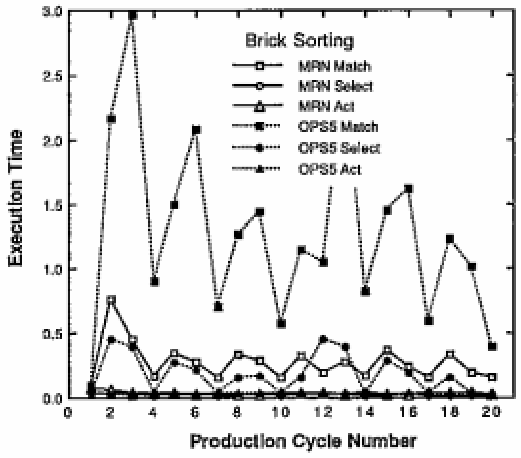

Figure 12: A complete execution time for two approaches.

6 Conclusions

The main purpose of this paper was to evaluate the performance of the multiple root node-based pattern matcher for production systems. A bottleneck was identified on the most efficient pattern Rete matcher. A solution to the bottleneck was proposed by introducing multiple root nodes to the Rete matcher. The MRN-based production system has been completely implemented in Lisp. To measure the relative algorithmic performance of the MRN-based matcher, benchmark production system programs were selected and executed on Sun 4/90 using both the MRN-based interpreter and Rete-based OPS5. Experimental results indicated that the MRN approach would give a multiplicative effect on the Rete-based production systems. The two criteria used in this study, one-input match time and number of comparison operations in a window of 20 production cycles, have shown that the MRN-based matcher can indeed give on the average a 6 fold improvement over the Lisp implementation of the Rete-based OPS5. Our experimental results suggest that production systems have more potential parallelism than what has been known. To further identify the complete source of parallelism in production systems, we have been implementing the MRN-based production system interpreter in SISAL, a pure functional language targeted to data-flow multiprocessors. The implementation is near completion and when complete we will be able to identify very fine-grain parallelism in production systems.

References

[1] Brownston, L., Farrell, R., Kant, E., and Martin, N., Programming Expert Systems in OPS5, Addison-Wesley Publishing Company, Reading, MA, 1985.

[2] Forgy, C.L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence* 19, September 1982, pp.17-37.

[3] Gaudiot, J-L., and Sohn, A., "Data-Driven Multiprocessor Implementation of the Rete Match Algorithm," in *Proc. of the Int'l Conf. on Parallel Processing*, St. Charles, IL, Aug. 1988, pp.256-260.

[4] Gaudiot, J-L., and Sohn, A., "Data-Driven Parallel Production Systems," *IEEE Transactions on Software Engineering*, March 1990, pp.281-293.

[5] Gupta, A., Parallelisms in Production Systems, Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

[6] Gupta, A., Forgy, C.L., Kalp, D., Newell, A., and Tambe, M., "Parallel OPS5 on the Encore Multimax," in *Proc. of the Int'l Conf. on Parallel Processing*, St. Charles, IL, August 1988, pp.271-280.

[7] Ishida, T., "Methods and Effectiveness of Parallel Rule Firing," in *Proc. of the IEEE Conference on AI Applications*, Santa Barbara, CA, March 1990.

[8] Kuo, S., and Moldovan, D. I., "Performance Comparison of Models for Multiple Rule Firing," in *Proc. of the 12th IJCAI*, Sydney, Australia, August 1991, pp.42-47.

[9] McDermott, J., and Forgy, C.L., "Production System Conflict Resolution Strategies," in Pattern Directed Inference Systems, D. Waterman and F. Hayes-Roth (Eds.), Academic Press, NY, NY, 1978.

[10] Miranker, D.P., Treat: A New and Efficient Match Algorithm for AI Production Systems, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[11] Sohn, A., and Gaudiot, J-L., "A Macro Actor/Token Implementation of Production Systems on a Data-flow Multiprocessor" in *Proc. of the 12th IJCAI*, Sydney, Australia, August 1991, pp.36-41.

[12] Sohn, A., "Parallel Processing of Production Systems on a Macro Data-flow Multiprocessor" *Technical Report*, EE-Systems Dept., University of Southern California, August 1991.