PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

PARALLELISM IN THE PESA I MULTIPROCESSOR

Franz Schreiner
Gerhard Zimmermann
Universität Kaiserslautern, SFB 124
6750 Kaiserslautern, Postfach 3049
Federal Republic of Germany

ABSTRACT

Many multiprocessors work inefficiently, because the individual processing elements are idle most of the time and the exploitation of parallelism on high level is restricted by synchronization mechanisms, while low level parallelism is not worth a distribution. The PESA I multiprocessor which is designed for the execution of production systems, avoids these disadvantages by the projection of a data flow based graph on a folded pipeline of parallel processors. The pipeline stages are interconnected by a bus system. Several experiments about details of implementation have been performed. This paper describes the evaluation of the simulation results which are used to enhance the performance and optimize the configuration of the PESA-I architecture. The performance could be increased by about 20 percent. Synchronization has to be performed only in a few cases that occur rarely during execution. Furthermore, the amount of utilizable parallelism is increased. Thus the number of processors performing nonredundant computation can be extended.

1 INTRODUCTION

The PESA-I architecture (Schreiner and Zimmermann 87) is based on the production system language OPS5 (Forgy 81, Brownston et al. 85) which is supported by the RETE pattern matching algorithm (Forgy 82). A detailed simulation of the pattern matching unit of the PESA-I showed the ability to fasten up the OPS5 execution by a large factor. This gain was reached by parallel computing on processors with a special instruction set. But we also recognized that the RETE algorithm is not optimal for our architecture, because it is tailored for a single processor architecture. We describe the main changes to the algorithm in order to enable more parallel computation. Another impediment of our implementation are the Not Nodes. They were executed sequentially and therefore caused congestions inside the architecture. By changing the synchronization concept we could solve this problem too. These improvements are located in the Match Phase of the OPS5 execution which takes most of the computation. The two other phases could also be accelerated. They can be performed nearly parallel to match. Their inplemention transforms the PESA-I from a coprocessor to an independent machine.

2 OPS5 AND THE RETE ALGORITHM

The only OPS5 programming primitives are disjunct rules, called productions. They consist of the left hand side (LHS), a condition part and the right hand side (RHS) which is an unconditioned action part. The facts called Working Memory Elements (WME) are stored in the Working Memory. Each fact is a database assertion comprising a time tag representing the order of creation and a number of attributes dependent on the class of the WME. The LHS is a conjunction of at least one positive and zero or more negated condition elements (CE). A CE is also assigned to a class and corresponds to all WMEs of the same class. A CE can contain pairs of attributes and predicates with constants and variables. A WME satisfies a CE if its attributes fulfill all predicates related to constants. These kind of tests are called constant tests. If a variable occurs in a CE, it is bound to the attribute of the WME. Identical variables must have the same values inside the LHS of the same production. Therefore these variables occuring in different CEs of the same LHS result in further tests, called intertests. A production is satisfied if for every positive CE at least one WME exists that matches and no WME matches a negated CE.

During one execution cycle of the production system, all productions with a satisfied LHS are instantiated by the time tags of the matching WME and form the conflict set (CS). This part is called match phase and needs most of the execution cycle. The next step is the selection of one entry of the conflict set by a strategy that uses actuality of time tags and number of condition elements as selection criterion. In some derived OPS execution models (Gupta et al. 86, Oshisanwo and Dasiewicz 87) more than one entry of the CS can be selected. The last step of the cycle ´fires´ the selected production by executing its RHS. The actions of the RHS (make, remove, modify) will create, delete or change elements of the working memory. The executability of rules may have changed. Therefore a new match phase starts the next execution cycle, called recognize act cycle. It stops when the CS is empty.

The main disadvantage of production system execution is the expensive match phase caused by the intertests. A single intertest between two condition elements of different classes with 100 WME each would result in 10000 comparisons to be performed in each execution cycle. Because production systems contain many intertests, a brute force match phase would prohibit an acceptable production system interpretation.

The RETE pattern matching algorithm exploits the fact that working memory in production systems changes very slowly. Multiple tests between the same data in different cycles are avoided by the introduction of a state which saves the result of intermediate tests (figure 1). The sequence of tests inside a production is fixed. If all constant tests of a CE are satisfied,

the WME is stored in an α-memory node. The first intertest has two a-memory nodes at the input and stores the results in a b-memory node as output. The following intertests have one b-memory node and one a-memory node at the input. The two input memory nodes associated with the same intertest are called partner nodes according to that test. The test and the memory nodes form a discrimination network, called RETE graph which is compiled from the LHS of all productions and interpreted by the match phase. The state of the match phase corresponding to the current working memory is stored, so that only the few changes to the working memory have to be considered for updating the state. The output of the RETE graph are the changes to the CS which has to be stored too. The results to be stored in a memory node inside the RETE graph consist of a concatenation of the matching WMEs. Attributs not needed for further computation can be stripped off to reduce memory space. If different CEs have tests in common these test nodes are shared. If the tests are totally equal, the following memory node can be shared too. This saves computation time and memory space. The shared nodes in figure 1 are marked.

If in figure 1 a token passes the constant test T1 it becomes activating token of the intertests T5, T7 and T9. It also becomes resident token of memory node S1. The intertest T5 is performed between the activating token and all resident tokens of the partner node S2. S2 may contain several residents, therefore zero or more positive results may activate T6 and become residents of S5.

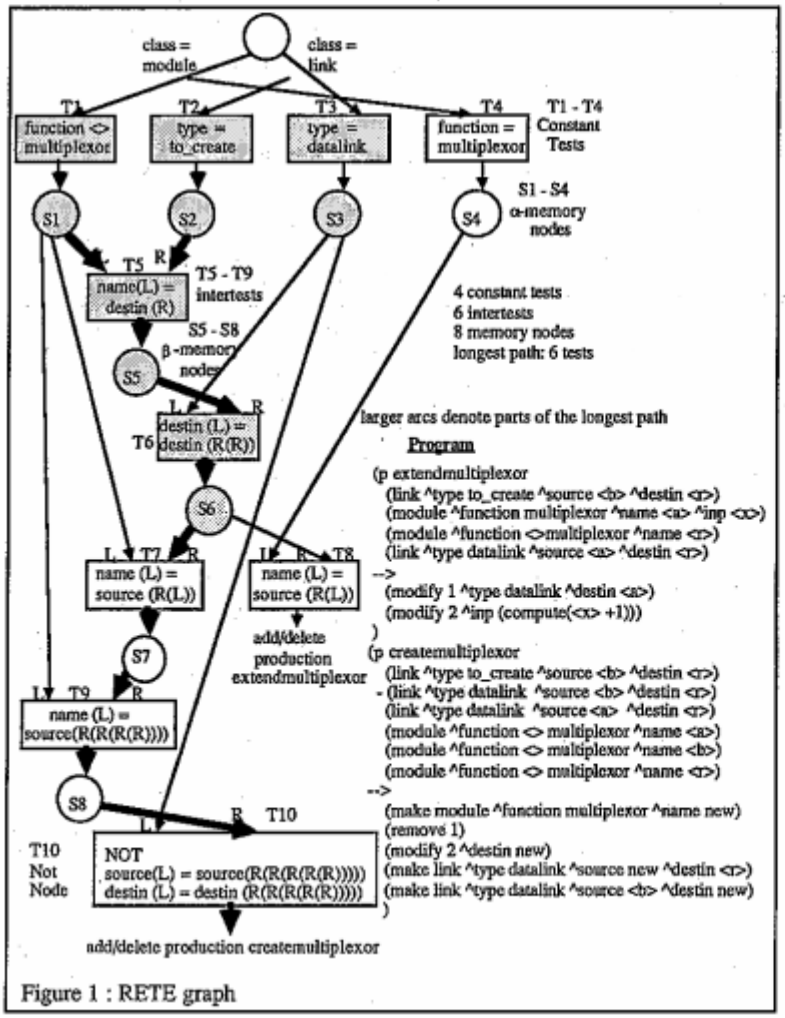

Figure 1 : RETE graph

3 PARALLELISM IN RETE

Although RETE reduces the number of tests during an execution cycle significantly, the match phase takes about 90% of the run time on a sequential computer. Therefore, we started introducing parallelism by taking advantage of detailed performance measurements of large production systems described by Forgy and Gupta 86. The constant tests are harmless, the main problem are the intertests. Generally there are three sources of parallelism:

(P1) Activations of different testnodes
(P2) Activations of the same testnode by different tokens
(P3) Comparison of the activating token with the resident partners.

Several other projects (Gupta et al. 86, Oshisanwo and Dasiewicz 87, Stolfo 87, Miyazaki et al. 87) concentrate on (P1) and (P2), while we try to utilize all sources. The reason is the local behavior of production systems. A firing production effects few changes to the WM, therefore only a small percentage of different tests is activated. The parallelism of (P1) is low. On the other hand the memory nodes typically contain about 30 residents, so that a single activation (P3) results in the comparison of many tokens. This is the essential part of the inherent parallelism present in OPS5. In order to utilize it, we distribute the contents of the memory nodes to parallel processors. Each processor contains all test nodes of the same depth in the RETE graph and a copy of the input memory nodes associated to that test, but only a part of the entries. If we provide an equal distribution of entries to those processors, the filling of their local memory nodes is divided by the number m of processors ($m < n$). Only (n/m) intertests are performed sequentially in every processor.

The processors form the stage of a pipeline which is connected via a bus to the next stage that contains the following testnodes of the RETE graph. Pipelining effects can be observed when positive results activate a chain in the graph. The result of a test may be several tokens. They are broadcast to the next stage. Each token is stored by one processor and the according tests are performed in all processors. To select the processor that has to store the data, an address has to be adjoined. The message contains also a sign and a starting point (tag) of the following testnode. This data format is called token, after storing it in a memory node address, tag and sign are stripped of. A bus access can be interpreted as a m-fold procedure call, where m is the number of processors on the following stage with a nonempty partner node.

The expression "pipeline" may be misleading. The processors store data that represent a state. A single input token can produce an avalanche of tokens inside the pipeline or may not affect anything in this execution cycle. Therefore an external scheduling like in standard pipelines is impossible.

4 THE MATCH PIPELINE

Figure 2 shows the PESA-I architecture. Most of the processing elements (PEs) are assigned to the match pipeline. The number of stages does not restrict the number of condition elements because the pipeline is folded back via bus 0. The depicted configuration is typical but not definitive. The final configuration will be choosen as late as possible. The PE is microprogrammed with a special instruction set. Functionally, the PE can be divided in three synchronous automata or agencies that communicate via common buffers and flags (figure 3).

The bus read agency (RBA) reads data from the bus and stores it in the readbuffer. The dataready flag signals that the bus carries valid data. If the readbuffer of a SBA is filled, the bus is blocked by the receiverbusy flag. The match agency (MA) performs the storing and testing. It has separate program and local tokenmemories of 64K words each. The tokenmemory contains the memory nodes of the RETE graph. After a positive test, the result is written into the sendbuffer. The bus send agency (SBA) sends these data to the next stage if it has permission to access the bus and if no receiverbusy flag is set. The MAs and SBAs of one stage are connected to manage synchronization (MA) and bus write access (SBA). There are two ways a MA can be blocked in presence of valid data in the readbuffer: In both cases the own writebuffer is filled and the bus can't be accessed because first there is no permission or second the bus is blocked by a filled readbuffer in the next stage. The special instruction sets for match, act and conflict resolution are different but can be microprogrammed on the same PE-type. In Match and Conflict Resolution the only arithmetic operation is a comparison, therefore the PE does not need complex arithmetical operators. In Act the arithmetical operations are performed by a commercial coprocessor.

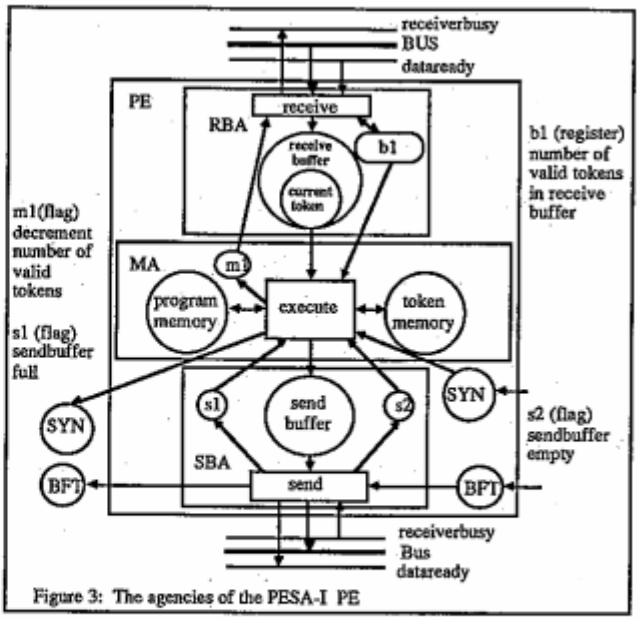

Figure 3: The agencies of the PESA-I PE

The results quoted in the following derive from a two stage simulator. The first stage simulates the PE, the input parameters are the delays for basic operations and the time for the different memory accesses. The output consists of the runtime for the microinstructions which are used as input of the stage two simulator. The other input parameters for the second tool are buffersizes, the configuration of the match pipeline (number of PEs per pipeline stage) and the buscycle. The main outputs are statistics about bus accesses and conflicts, idle times of individual PEs, fillings of memory nodes and buffers and number and time of instruction usages.

Although the contents of memory nodes is distributed over the whole stage, the assignment of tokens to processor-memories is unique and can be reconstructed. PEs storing results are selected by a sum of tags and time tags that can be reconstructed. This leads to an equal distribution of tokens to the memories of all stages because the time tags are equally distributed. The fact that the same token uses the same path is very important for fault detection and for avoiding unnecessary synchronization, because the same tokens with different signs cannot pass each other. Using the unchanged RETE algorithm only one kind of synchronization would be needed for the reason of detecting the end of the computation in the match phase.

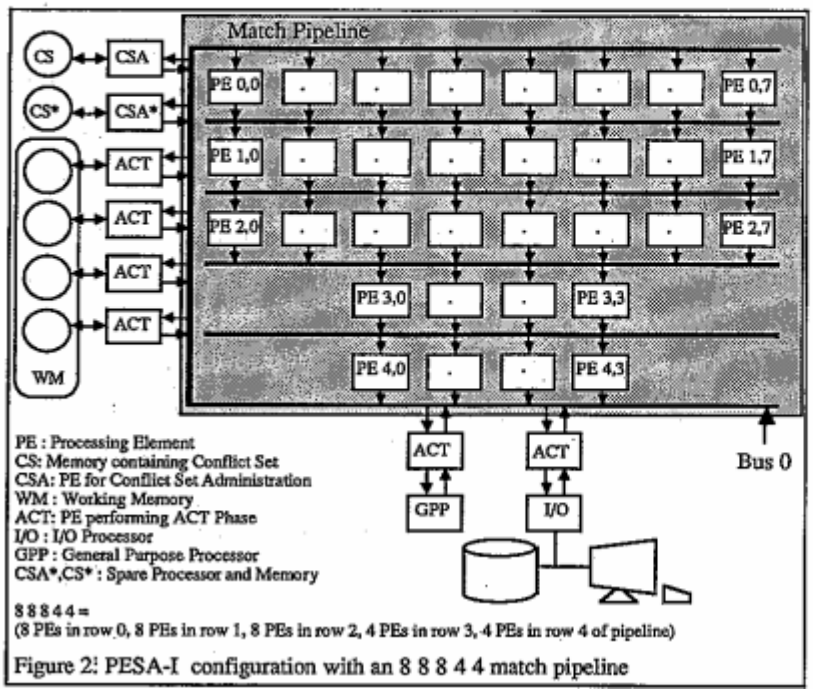

Figure 2: PESA-I configuration with an 8 8 8 4 4 match pipeline

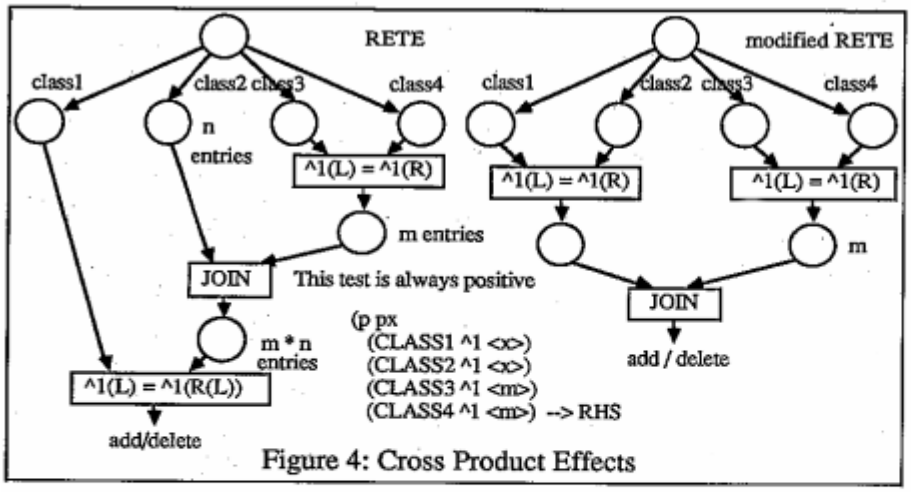

Figure 4: Cross Product Effects

5 CHANGES TO RETE

The fixed sequence of tests causes two effects reported by Gupta 86. First, early join operations could evoke data overhead (cross product effects). Replacing the fixed sequence of tests by a relative order (figure 4) softens the impacts of join operations. They are executed as late as possible. The number of partners will be reduced by the previous tests.

The second effect is the activation of long chains of tests.

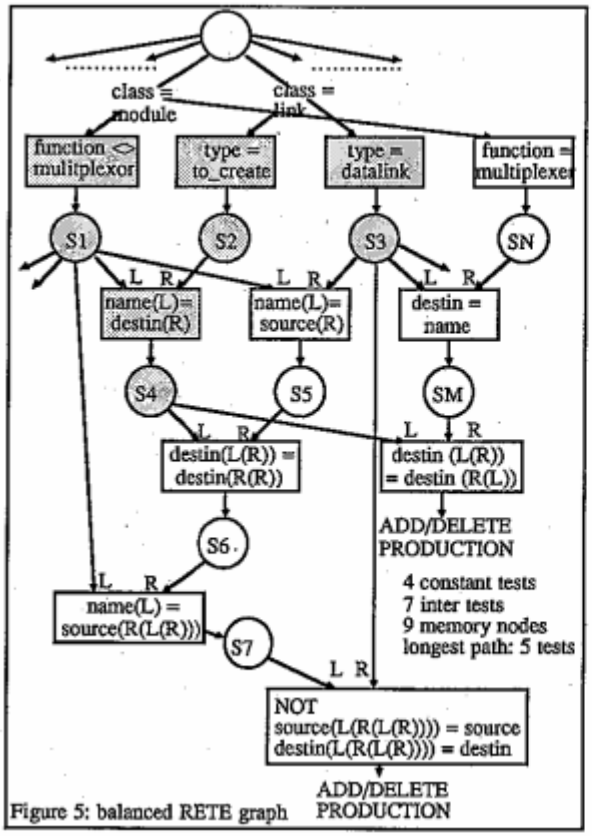

Figure 5: balanced RETE graph

Thereby the longest path (figure 1) is important. Our compiler arranges the tests to a tree which is as balanced as possible. Forced balancing by the introduction of new join operations is avoided. Figure 5 shows the modified RETE graph of figure 1. The balancing changes the sharing possibilities, we are currently testing the impacts. In this example the amount of sharing is reduced by our implementation, but it could also increase.

A third problem is that beside the store/delete operation there is no difference in the treatment of tokens with positive and negative sign. For a negative token all intertests and concatenations have to be performed resulting in an unacceptable run time. This time can be reduced by a factor between 2 and 10. In our approach we perform only the constant tests on negative tokens. As described by Forgy and Gupta 83, about 85% of the tokens fail in the constant tests, because most of them are mutually exclusiv. If a constant test has a positive result, all memory nodes on the chain following this test are affected. Tokens within these memory nodes containing the time tag are deleted, no further test is performed. This operation is supported by our distribution concept and can be performed parallel in one stage and nearly parallel in the whole pipeline, because the token containing the time tag can be transported to the next stage before the delete operation starts. The affected production nodes of the CS can be looked up at the same time with the exception of productions including negated CEs. These will be discussed later. In contrast to the old delete operation, tailored for a sequential machine, the new solution is nearly independent from the result of intertests, therefore the gain factor differs.

The negative changes to the conflict set do not have to be computed by the pipeline. This is a very important performance gain. The delete action in the conflict set takes place in a time when the Conflict Set Resolution would be inactive in the conventional RETE interpretation. Thus the computations of the phases overlap. Measurements have shown that the CS is not very large. Almost every time the delete operation is finished before the first positive update arrives.

By the introduction of this concept we loose the nonpassing feature mentioned in the last chapter. The delete token which contains only the time tag, is transported once from stage to stage. In the original RETE algorithm a token is sent from different PEs of stage n to stage n+1 once per positive result. We need a synchronization that guarantees that these delete tokens are not passed by positive tokens. A (positive) concatenation containing the element to be deleted could be passed by its delete message, resulting in a faulty match state. Our first attempt was to send a special synchronization token after the last delete token that blocked all PEs until all their sendbuffers were empty (the communication inside the stage was done by the syn flag; see fig3). The PE was busy waiting which should be avoided. So we changed the blocking of the PE into the blocking of the sendbuffer. The number of arrived

synchronization tokens is stored in a counterregister, the sendbuffer can only be accessed if its value is zero. The counterregister is decremented on the condition that the right neighbor signals an empty sendbuffer and the own sendbuffer is empty. The decrementing is done during the operation decoding and incremented by an operation. So the accesses to this register are exclusive and cannot result in conflicts. The PE can now work until the first result of a positive test has to be written. Measurements show that most of the tests are negative. If inside a stage this synchronization is performed twice in sequence, all sendbuffers are empty at the time at which PE 0 gives the syn flag to PE n. This double synchronization is used to protect tokens from passing each other. A single synchronization is used to detect the end of a match phase. Programs exist in each stage to activate both kinds of synchronization in the following stage by sending a special token.

The delaying impacts of the synchronization between the token with - and + sign is restricted by the fact that -WME changes have just to be found in the working memory while +WME changes have to be constructed first, so that there is a natural time gap between these inputs.

6 THE NOT NODES

The Not nodes are those intertests that combine a positive and a negated CE. A production with matching WMEs for all positive CEs is satisfied if there exists no WME that matches the negated CE. To avoid iterations over the whole working memory, RETE computes the WME that satisfy the negated CE. The Not nodes are performed after all intertests between positive CEs have been finished. If there are more than zero matches between both inputs, the production failes. A counter, adjoined to the positive part, represents the number of matches between the positive and the negated CE. The negated part is no longer needed because there are no data derived exclusively from the negated part, also no time tag can be associated with a negated match.

The actions of the Not node differ depending on the sign and kind of input. A token from the positive input effects the calculation of the counter if its sign is +, else the delete operation does not differ very much from the normal case described in the last chapter. An activation by the negated input updates the counters assigned to the positive partners instead of sending results. If a match takes place and the sign of the negated input was + the counter is incremented else the sign was - and the counter is decremented. By modifying the counter from -or to- zero, the executability of the production is altered. The CS has to be updated accordingly.

Our problem is that the distribution of the input representing the negated CE leads to various counters of same positive part which have to be added to obtain the correct result. These tokens are assigned to the same stage and besides the two flags mentioned earlier there is no instage communication. Therefore we have two alternatives to realize the Not nodes:

- no distribution of the negated input
- full distribution of the negated input, counter computation in the following stage

The first solution abandons the performance gain by the low level parallelism and leads to effects within the pipeline that are similar to "hot spots" in interconnection networks. The second solution is hard to realize because of following deviations from standard intertests:

- only the first token with the same contents has to be stored
- the next arriving token with same contents have to update the counter of only this token
- test activations by the positive and negated input can take place at the same time
  (because they are performed in different stages)
- tokens for these different activations can pass each other

After initial tests without distribution of the negated input, we decided to use the second alternative. The parallelism that we utilize is not as high as in normal intertests but the hot spot effect can be evaded. The influencing features of the match pipeline have to be exposed:

- The number of positive intertests is very low, therefore the bus is accessed in only 3 to 5% of the time and the filling of the sendbuffer is very low (0 to 2 tokens)
- All PEs are nearly at the same point of execution, caused by the equal distribution of data
- The activities are the highest in the upper stages and decrease in lower stages

The Not nodes are assigned to the last two stages. In the first of them the WME matching the Constant tests of the negated CEs are stored. Even if there is more than one negated CE they cannot be combined, except by joining. The matching WME fulfill the CE, as if it had a positive sign. While the negated CEs are connected by logical AND these matching WME are connected by logical OR, as it can be shown by the application of de Morgans rule:

$$CE1 \;\&\; \overline{CE2} \;\&\; \overline{CE3} \quad \overset{dM}{=} \quad CE1 \;\&\; \overline{(CE2 \vee CE3)}$$

If there is more than one negated CE an arriving positive partner performs the tests with all of them sequentially during the same activation. The treatment of the Not node is selected by one of the four cases mentioned before:

A) Activation by a negated input with sign -
The activating token is sent to the next stage before it is deleted in the memory node. In the next stage all partners are tested, if the tests are positive, the counter is decremented. A counter changing to zero evokes the sending of the associated token to the conflict set with sign +. The production is now satisfied.

B) Activation by a negated input with sign +
The handling is similar to A), the token is stored and the counter of the partner tokens in the next row can be incremented by one. A change from zero to one induces the sending of the instantiation with sign - to the CS. The production is no more satisfied.

C) Activation by a positive input with sign -
This differs from a normal delete action only by the fact that the tokens are sent with sign - to the CS before deleting.

D) Activation by a positive input with sign +
In the upper row the counters are computed after a double

synchronization. The PE matching the address of the incoming token sends it with the count to the next row afterwards it invokes another double synchonization. The other PEs invoke two double synchronizations before sending only their count if it is not zero. All these tokens get the same address, computed only from the unique old address. By this synchronization order, the whole token arrives on the next row before the other counts. It gets stored by the PE matching the address. If the countvalue is zero the token is send to the CS. A countertoken results in adding the value to the counter of the first token in the PE matching the address. If the old countvalue was zero the whole token is sent with sign - to the CS.

As it can be seen only the case D) is expensive. It is only activated, if the whole positive part of the production becomes true. The other cases depend on just one WME change with the positive constant test for the affected production. They could also activate more than one test because the possibility of shared negated CEs is high. In diffence to productions without negated CEs both the + and - changes to the CS have to be send. The concurrent CS update is not sufficient because by case A) a positive change to the CS could be produced by a negative negated token. This positive change could be deleted afterwards by the deletion of a WME in the positive part. By a concurrent update in the CS the second delete action could be finished before the positive token to be deleted arrives, therefore C) differs from other delete actions. This is the reason why the concurrent update is only performed for productions without negated CEs.

Because the not nodes produce tokens of different signs their passing has to be avoided. In case A) and B) the same path is used, C) and D) is handeled by the normal synchronization, A) and D) by the special synchronization which also provides that results of B) and D) pass each other. C) may pass or be passed by every token execpt D).

Only by this implementation we can distribute the lower stage too. This effects that in case A), B) and C) we reach the same gain than in intertests. The gain of D) depends on four factors:

the number of activations of case D)
the number of negated entries
the number of other tokens passing the upper stage
the number of PEs in the upper stage

If we have many PEs on the upper stage (e.g. 16), one negated entry and there is only one activation of D) or no other tokens, the runtime of the distributed solution is even minimally higher. The result token may be send twice to the CS and there is transportation and synchronisation overhead. In the old solution the hot spot would not appear or did not effect anything. This is one of the reasons why we adjoin fewer PEs to the upper Not-Stage. In the case of many negated entries, many activations of D) or many other tokens the gain rises to that of intertests on stages with fewer PEs.

7 DISTRIBUTED COMPUTATION OF THE THREE PHASES

The main problem in reducing the execution time of the match phase could be solved. The next aim is the reduction of the execution time for the two other phases by utilizing parallelism. The entire recognize act cycle may be accelerated

P1, P2, P3, P4, P5: different productions
(the unsorted instantiations and the pointers from sons to fathers are not shown in this figure)
Situation before (a) and after (b) the creation of the instantiation of production P5 with the time tags (12 11 4)

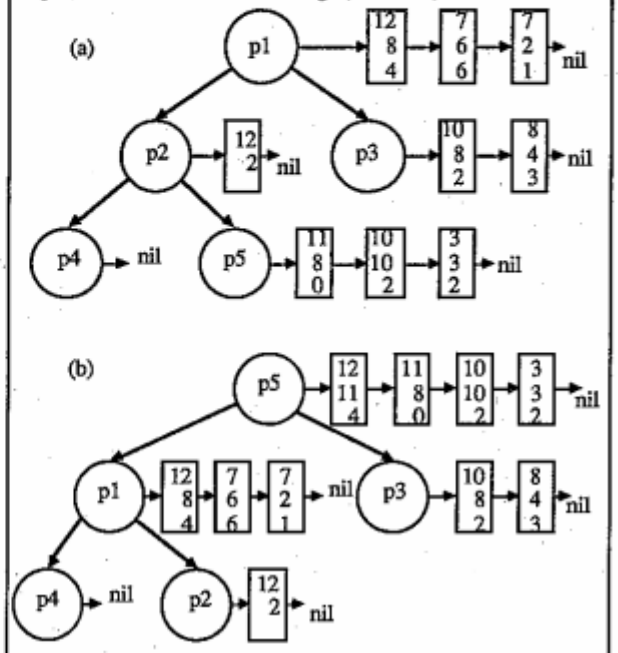

Figure 6: Conflict Set Administration

by an overlapped computation of the phases. Chances are good because the only necessarily synchronous action is the delivery of the dominant instantiation from the CS to ACT. At the end of the match phase the conflict resolution has to compute only this instantiation. A complete order of all instantiations has not to be provided. In our implementation we distinguish between conflict resolution and administration of the conflict set.

7.1 Administration of the conflict set

The data structure of the conflict set as shown in figure 6 is a heap of ordered lists. The lists contain the instantiations of one production. The instantiations have to be stored sorted for comparisons with other incarnations and unsorted in order to assign the time tag to the correct action in the RHS. A positive conflict set change has to be included in this list. In most cases this inclusion is performed at the top part of the list, because the CS change is the result of WME change that contains a time tag with a high value. Therefore, the effort for updating the list is low. The only exceptions are the CS changes generated after application of case A) of the previous chapter, they may contain only lower time tags. If the top of the list alters, the top element has to be compared with that of the "father" in the heap. If the new element dominates, the heap is updated without changing its balanced structure. The comparison continues upward until the "father" dominates or the top of heap is reached. In both occurences the update is finished, in the second the last change is the new dominant instantiation. The changes with sign - actualizes the list and if the top of list is deleted, the heap is updated in the downward direction. The filling of the CS is typically between 2 and 500 elements. By subdividing them to productions and by the logarithmic

runtime features of the actions adjusting the heap (Bentley 85), the time spent in the administration of the CS is considered to be low. The process is assigned to CSA-PE of figure 2. The second CSA-PE is only used if the CS is to large for one memory, additionally it is used as reserve if one of the ACT and CSA PEs should fail. The sorting of different instantiations is distributed to the ACT and CS PEs. It is supported by a special microinstruction, only positive changes have to be sorted.

OPS prevents firing of a production with the same time tags more than once. The top element of the heap has to be removed, leading to further administration steps. These steps are also performed in parallel to ACT.

7.2 Conflict resolution

The fact that only the dominant instantiation is needed for ACT can be utilized further. After the update of the CS by the tokens with negative sign the currently dominant instantiation is sent to the ACT PEs. Each of them stores and compares it to the incoming changes of the conflict set. If the change dominates it replaces the old instantiation. After the termination of the match phase, the ACT PEs contain the dominant instantiation. No time consuming delivery takes place, because the ACT PEs can start the execution of the RHS, while the administration of the conflict set may be still busy. The phases overlap. This mechanism does not work, if the actual dominant instantiation is deleted. Then the conflict resolution is performed by the PE administrating the CS.

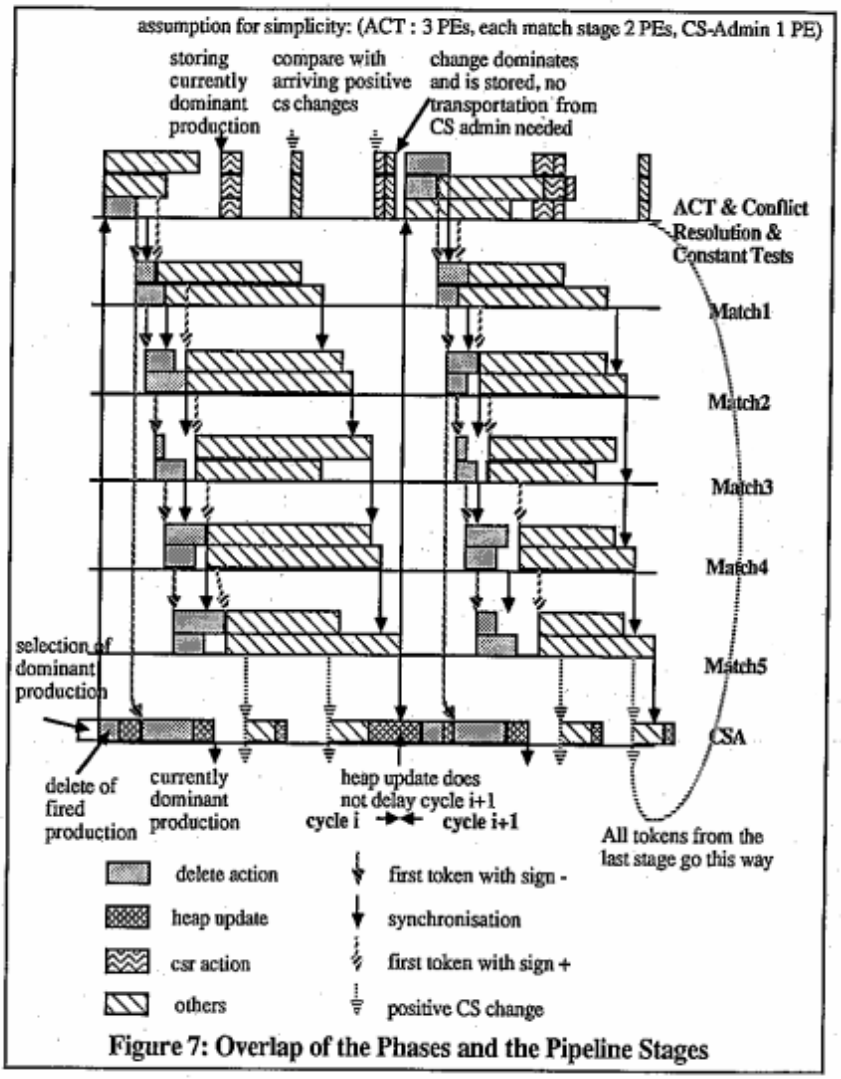

Figure 7: Overlap of the Phases and the Pipeline Stages

7.3 ACT

The act phase comprises the generation of new WME and the administration of the working memory including deletion and storing. Because of the 16bit address space of our PE we have to allocate at least four of them to ensure a sufficient working memory size. In consequence we have to divide the working memory and have to show how the PEs can be used in parallel. The solution of both problems is the same. The working memory is divided by assignment of different classes to different PEs. The WMEs of one class are stored, dependent on their time tag, in one of several memory nodes that belong to their class. This kind of hashing is unique and alleviats the searching. The number of nodes is decided by the number of classes. To supply parallel make and remove actions, classes that occur on the same RHS have to be assigned to different PEs. This is not always possible. We use an algorithm based on repulsiv forces, known from the design automation area, for the assignment.

The parallelism in the RHS is not very high, the ACT PEs would be idle most times. Therefore, we transfered the constant tests and the sorting of positive CS changes to the ACT PEs. After the first result that ACT performs, the match pipeline can start. In almost every production, this is a delete action. Thus the ACT PEs have just to search for the element by its time tag and then perform the constant test. After the first satisfied constant test, match and the conflict resolution can start updating their memory nodes. Figure 7 shows the overlapped computation.

8 RESULTS

The changes of the previous chapters impact both, code and instructions. The program of an intertest is very simple, it mainly consists of one loop with several exit points. Only in the case of shared nodes, several loops may exist, but they are not nested. Parameters like loop entry points or token lengths can be held in registers that are addressed implicitly resulting in few operands per instuction. Additionally several sequential instructions could be combined. Together with the complex instructions for storing, deleting and transport of tokens, this leads to a very dense code and few accesses to the program memory. A typical intertest contains 15 to 20 words and exists for each of its two entries. Our 64K word memory provides

enough space for 1500 -2000 intertests per stage.

The results that we published 87 could be increased from 35000 to 42000 working memory changes per second. Figure 8 shows the difference of the old intruction set and Not-Node implementation to the new one. For a given configuration the gain can be higher. By assigning the NOT nodes to the lower stages, the optimal configuration for a fixed number of PEs also changes. The new concept of parallelism and distribution is adapted more to the common situations that appear during production system interpretation. Currently our simulator can only handle unfolded pipelines. Therefore we assume that the folding results in additional performance gain, because the communication overhead cannot be neglected.

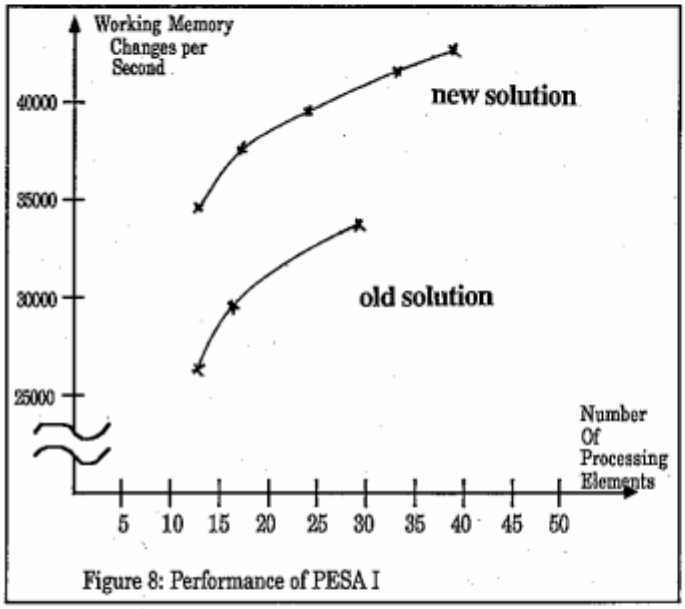

Figure 8: Performance of PESA I

Our results are the highest for all currently published OPS machines, although our number of PEs is lower. This is influenced by the special instruction set and the local memories instead of shared memory solutions. A little bit higher results could be reached by adding more PEs, but the relation between effort and result is not worth a discussion. Another possible improvement is the extension of OPS by parallel rule firing. Oshisanwo and Dasiewicz use it (19000 working memory element changes per second with 256 PEs) as well as Forgy et al.(9400 WMEC/s, 32 PEs). As long as we reach improvements with single rule firing we will concentrate on it. Parallel firing of rules concerns only the conflict resolution and can be introduced by a few changes.

CONCLUSION

It has been shown that by utilizing very detailed performance results, the major part of parallelism inherent in production systems can be observed. The simulation, although expensive, has to preceed a prototype. In our special case, we improved the treatment of the delete actions and the implementation of the NOT nodes. This results in a higher performance for a given configuration.

We added the concept of distributed computing to our parallel pipeline and avoided assigning the processes of the recognize act cycle 1:1 to processors. The ACT processors of the PESA I perform processes of every cycle. The original conflict resolution process is divided and distributed. This results in a better balancing of work to processors and avoids unnecessary bottlenecks.

Our next step is the introduction of fault tolerance mechanisms. Therefore we have to include some switches to the PEs. An interresting idea is to compute the optimal configuration for a given production system (maybe rule based) and use the switches to arrange this configuration before starting the execution. Another point of interest is the design of OPS derivations and environments that can be mapped onto the same hardware. The pattern matching has not to be changed by introducing new selection mechanisms (e. g. selection by possibilties) and undoing of performed actions which is needed in the area of non monotonic reasoning.

REFERENCES

[ScZ87] Schreiner, F.; Zimmermann, G.: "PESA I A Parallel Architecture For Expert Systems", 16th International Conference On Parallel Processing, 1987, p 166-169.

[For81] Forgy, C.: "OPS5 Users Manual", Technical Report CMU-CS-81-135, Carnegie-Mellon University, 1981.

[BFK85] Brownston, L.; Farrel, R.; Kant, E.; Martin, N.: "Programming Expert Systems in OPS5: An Introduction to Rule Based Programming", Addison Wesley 1985.

[For82] Forgy, C.: "RETE: A Fast Algorithm for the Many Pattern/ Many Object PatternMatch Problem", Artificial Intelligence 19 (September 82), p 17-37.

[GFN86] Gupta, A.; Forgy, C.; Newell, A.; Wedig, R.: "Parallel Algorithms and Architectures for Rule-Based Systems" 13th Intern. Symp. on Computer Architecture, Tokyo 1986, p 28-37.

[OsD87] Oshisanwo, A.; Dasiewicz, P.: "A Parallel Model and Architecture for Production Systems", 16th International Conference On Parallel Processing, 1987,p 147-153.

[Sto87] Stolfo, S. "Initial Performance of the DADO2 Prototype", Computer, January 1987.

[FoG83] Forgy, C.; Gupta, A.: "Measurements on Production Systems", Technical Report CMU-CS-83-167 Carnegie-Mellon University, 1983.

[Gup86] Gupta, A.: "Parallelism in Production Systems", Technical Report CMU-CS-86-122, Carnegie-Mellon University, 1986.

[Ben85] Bentley, J.: "Thanks, Heaps", Communications of the ACM Vol 28 Number 3, March 1985, p 245-250.

[MAA87] Miyazaki, J.; Amano, H.; Aiso, H.: "Manji: An Architecture for Production Systems" 20th International Hawaii Conference on Systems Sciences 1987, p 236-245.