DFLOPS:A Data Flow Machine for Production Systems

Fu-Chiung Cheng
Department of Computer Science
Columbia University
NY, NY 10027

Mei-Yi Wu
Department of Computer Science
New York University
NY, NY 10003

November 15, 1993

Abstract

Many production system machines have been proposed to speed up the execution of production system programs. Most of them are implemented based on conventional control flow model of execution which is limited by the "von Neumann bottleneck." In this paper we propose DFLOPS, a new multiprocessor data flow machine, for parallel processing of production systems. Rule programs are compiled into data flow graphs and then mapped into DFLOPS processing elements. Three levels of parallelism: Rule Level Parallelism, RHS Level Parallelism and LHS Parallelism are fully exploited to achieve high performance. The design and implementation of DFLOPS is presented in detail. The distinguishing characteristics of this proposed machine lies in its simplicity, fully-pipelined processing and fine grain parallelism. The initial results reveal that the performance of production systems is greatly improved.

Keywords: Parallel processing, Data flow machine, Production systems, Pipeline processor, and Compilation.

1 Introduction

Although production systems have been widely applied in the implementation of a number of knowledge-based expert systems[18, 32], the major problem faced by expert system technology is efficiency. The main technique to increase the processing speed of a production system is through parallelism and compilation. DFLOPS fully exploits both techniques.

Many efforts have been proposed to speedup the performance of production system programs. In software approaches Forgy presented the RETE match algorithm, developed for the OPS5 production system interpreter, to reduce redundant pattern matching[8]. Schor et al. modified the operations of RETE network to achieve improvement in efficiency and rule clarity[19]. Miranker proposed the TREAT matching algorithm[21] and lazy matching to improve the matching time[20]. In hardware approaches Stolfo gave four parallel matching algorithms and a parallel execution algorithm based on DADO machine, a tree-structured parallel machine for large rule-based expert systems [31, 30, 15, 28, 27]. Gupta implemented OPS5 production system on DADO to perform parallel matching[10]. Oflazer proposed partitioning algorithms for production systems to maximize the effect of parallel matching[22]. However, in those parallel approaches, most of the production system machines are implemented in conventional control flow model of execution which is limited by the "von Neumann bottleneck." Gaudiot illustrated that production systems can be mapped on data-driven architecture[9, 17] and pointed out that the data-driven model of execution has been proposed as a solution to solve the "von Neumann bottleneck." Instead of designing a new architecture for data-driven production systems, a RETE network was compiled into a data flow graph which was then mapped into the MIT Tagged Token data flow computer(TTDA) for parallel processing. There are some drawbacks in this approach. First, compiling a RETE network into ID language is complex; second, since a condition element with 3 attributes may generate more than ten times this number of actors in data flow graph, namely more than 30 nodes, the performance is degraded when the graph is mapped into a general purpose data flow architecture such as the MIT TTDA; third, conflict resolution strategies are hard to implement since it is control-flow-oriented; fourth, MIT TTDA does not work well for symbolic computations; and finally good performance requires not only parallel rule matching but also parallel rule firing.

The main contributions in this paper are as follows: first, DFLOPS, a new data flow machine with RISC-like processors, is developed to directly support parallel production systems; second, LHS and RHS of a rule program are directly compiled into DFLOPS instructions and a set of tokens and then map them into DFLOPS processing elements; third, three levels of parallelism: Rule Level Parallelism, RHS Level Parallelism and LHS level Parallelism are fully employed; fourth, DFLOPS exploits both compilation technique and parallelism to achieve high performance.

The distinguishing characteristics of the proposed machine lies in its simplicity, fully-pipelined processing and fine grain parallelism. The preliminary results reveal that higher performance can be achieved and the effectiveness may be superlinear.

Section 2 presents a brief introduction to OPS5 production systems and data flow computations. Some potential problems with data flow computation of production systems are also identified. The design of tokens and instructions and an algorithm to compile a rule program into DFLOPS tokens and instructions are presented in section 3. The design and implementation of DFLOPS is presented in detail in section 4. Section 5 gives an example showing how this machine works. The results evaluation is discussed in section 6. Finally, the conclusion and remarks are given.

2 Production Systems

A production system[3, 7] is defined by a set of rules, or productions, which form the production memory (PM), together with a database of assertions, called working memory (WM), and an inference engine which executes the rules with a cycle consisting of three action states:

1. Match: the interpreter identifies all the rules whose conditions are satisfied by the current contents of the WM.
2. Select: the interpreter applies some conflict resolution strategies to determine one dominant rule for execution.
3. Execute: the interpreter executes the rule selected and the working memory is modified by the action part of the selected rule.

OPS5[3, 7] supports forward chaining mechanism and LEX or MEA conflict resolution strategies to select the "best" rule to fire. Each unit of WM, consisting of a class name and several attribute-value pairs, is called working memory elements(WMEs). Each WME is associated with an integer called *time tag*. Time tag is used in conflict resolution strategies. For example, "Peter is the manager of Marketing department" can be expressed as

```
5: (manager ^name Peter ^dept Marketing)
```

The time tag of the WME is 5. Each rule in the PM consists of a rule name followed by a conjunction of condition elements, called its left-hand side(LHS) and a set of actions, called its right-hand side (RHS). There are two kinds of condition elements in production rules: *positive condition elements* that are satisfied when there exists a matching WME and *negative condition elements* that are satisfied when no matching WME is found. Negative condition elements are marked '-' in a production rule. For example,

```
(p Add-boss
   (manager  ^name  <mn> ^dept <d>)
   (employee ^name  <en> ^dept <d>)
-  (boss    ^bname <mn> ^ename<en>)
-->
   (make boss ^bname <mn> ^ename<en>))
```

The rule can be interpreted as: if there is a manager with name `<mn>` of department `<d>` and there is an employee `<en> of department <d>` but there is no fact like "`<mn>` is the boss of `<en>`" then create a new fact and insert it into the WM.

2.1 RETE Matching Algorithm

The RETE matching algorithm[8] is a very efficient approach for matching many objects against many rules. It compiles the LHS of the production rules into an augmented data flow network, called RETE network. The network contains a *root node* to distribute incoming tokens to other nodes, *a set of one-input nodes* to test intraelement features, *a set of two-input nodes* to test the interelement features, and *a set of terminal nodes* to report the rule instantiations that are satisfied and ready to fire. The RETE network is constructed based on *structural similarity.*

Since there is little change in the working memory between recognize-act cycles, *alpha* and *beta memories* are introduced to keep the partial match information. This strategy is called *temporal redundancy.* The RETE network of the add-boss rule is shown in Figure 1.

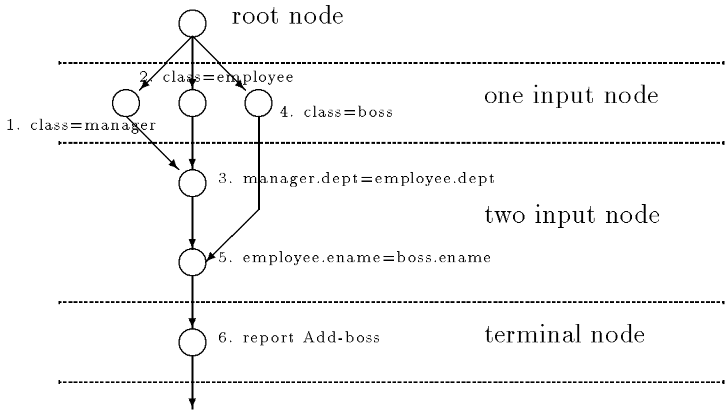

Figure 1: RETE Network

2.2 Parallelism of Production Systems

Many methods had been proposed to improve the performance of production systems. The software approaches[14, 21, 19, 12] can be classified as changing representation of important data structure and/or implementation of important operations, reducing processing by handling common cases efficiently, matching rules against working memory on demand, compiling and optimizing[21, 20]. In hardware approaches, some production system machines[25, 16, 11, 28, 30, 10] have been constructed. There are three applicable parallelism in production systems.

1. Rule level Parallelism: At each match state, the interpreter identifies all the rules whose LHS are satisfied by current contents of the working memory. Those matched rules that need not be synchronized can be fired in parallel. Thus several rules can be fired at the same recognize-act cycle if the final result is the same as firing one rule at each of several consecutive recognize-act cycle.

2. LHS level Parallelism: Production rules can be distributed to different PE, so that a SUBRETE network can be constructed in each PE to match the production rules. Matching the LHS of the production rules with the current working memory can be carried out in each PE simultaneously. Since the original RETE network in a single processor is divided into several SUBRETE networks, the time required for the matching phase can be considerably reduced. Performance evaluation of LHS parallelism can be found in [22].

3. RHS level Parallelism: The actions of a rule being fired can be assigned to different processors so that several differernt actions of the RHS can be processed simultaneously.

Among them, the most significant parallelism are the rule level parallelism and the LHS level parallelism. The latter reduces the matching time and the former reduces the total number of recognize-act cycles.

2.3 Data Flow Computations

Most data flow machines such as MIT Tagged-Token Dataflow Architecture[1], Monsoon[23, 24], EM-4[26], were designed for general-purpose parallel numerical computations. As for data flow symbolic computations, GAMMA[6], AGM[2], KARDAMOM[4] have demonstrated that the data flow principle can be applied in database applications. We will demonstrate that data flow principle can also be applied in expert systems.

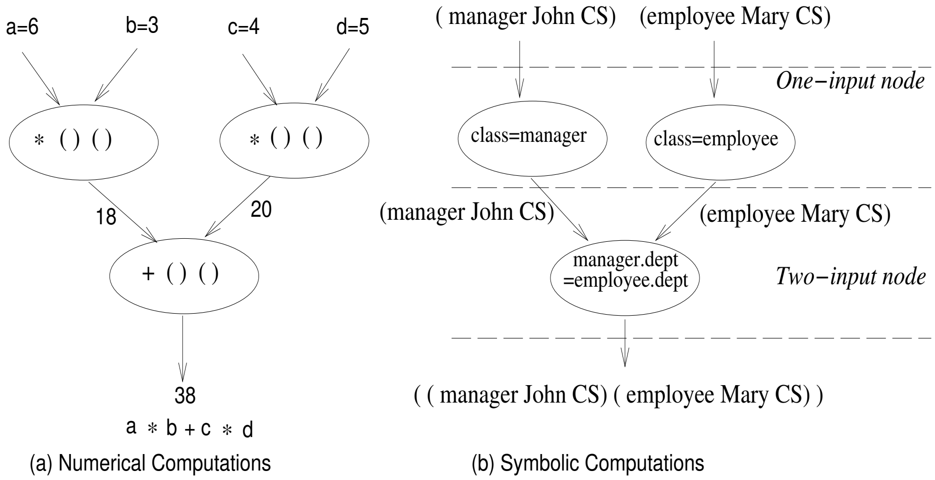

Figure 2: Data Flow Computations:

2.3.1 Data-driven Numerical Computations

Information items in a data flow computer appear as *operation packages* and *data tokens*[13]. Each *operation package or instruction* consists of an operator, operands, and the destinations to which the result(a new data token) will be sent. Instructions are activated by the availability of the data tokens and the enabled instructions can be executed simultaneously. A *data token* is formed with a result value and its destinations. Data flow programs are represented by directed graphs; nodes and arcs represent operation packages and the values in the arcs represent data tokens. When a token flows into a two-input node, it will check if its partner has been stored; If so, the operation package is executed and one or more new tokens are generated and sent to their destinations; Otherwise, it will discard the destination part and store *the value part* of the token. An example executing $a * b + c * d$ is shown in Figure 2(a).

2.3.2 Data-driven symbolic operations in production systems

The suitability of the data-driven execution model for production systems was discussed in [17]. Here we discuss some differences between data-driven numerical computations and symbolic computations of productions systems.

1. Data grain: Data grain is the unit of data to be processed in a token. A token passed between operation packages in numerical computations is **a value** and its destinations, while a token in production systems is **a working memory element** and its destinations. In Figure 2(a) the data grain of numerical computations is a value(such as $6, 3, 4, 5, 18, 20$ and 38). In Figure 2(b) the data grain of production systems is a WME(such as (manager ^name John ^dept CS), (employee ^name Mary ^dept CS) and ((manager ^name John ^dept CS) (employee ^name Mary ^dept CS))).

2. Data persistence: When a new token is generated, indicating whether old tokens are removed or not is called **data persistence**. In numerical computations, once the new data are calculated, the old ones are no longer needed. In the example of Figure 2(a), once $a * b$ is computed, a and b are no longer needed. In symbolic processing, the new and old data or WMEs always persist unless an explicit *delete* command is issued. In the example of Figure 2(b), the WME of left/right incoming token for the two-input

node(manager.dept = employee.dept) is first stored in a left/right *alpha* memory then matches against the WMEs stored in right/left *alpha* memory. If the test operation succeeds, a new token is generated. The WMEs stored in *left* and *right alpha* memories are kept whether the test operation succeeds or not.

3. One-to-many vs. one-to-one matching: Data-driven numerical computation is one to one matching, while data-driven symbolic computation may be one to many matching. For the two-input node in data-driven numerical computations, one right/left token can match only one left/right token, so the *present bit mechanism* used in MIT TTDA is sufficient to check if both incoming tokens are ready. In production systems, a left/right token may match many right/left tokens in the opposite arc. For example, suppose initially two tokens (employee ^name Mary ^dept CS) and (employee ^name Tom ^dept CS) have flowed into the two-input node shown in Figure 2(b). These two WMEs are kept in the right *alpha memory* of the two-input node. Now suppose a *manager token* flows into this node, this will cause two new tokens to be generated. So, we need more sophisticated mechanisms to deal with such join-like operations in data-driven symbolic computations.

4. Function units: In numerical computations, matrix computations such as matrix additions, multiplications are the critical points where the parallelism lies and these operations require hardware support, while in symbolic computations, data retrieval, number and string comparisons require hardware support to achieve high performance.

LR memory is introduced to solve the *data persistence* problem and *token matcher* is designed to deal with *one-to-many matching.* They are presented in detail in session 4. Based on these observations, we now design a new data flow machine for production systems.

3 Data Flow Programs for Production Systems

In order to execute production system programs in a data flow machine, a rule program must be compiled into a data flow graph and then the data flow graph must be mapped into the data flow processing elements. A rule program consists of a set of rules and a set of WMEs. The set of rules form a data flow graph and the set of WMEs form a set of tokens and a database which stores the real data. A data flow graph is composed of a set of nodes and a set of directed edges connecting these nodes. A node and the outgoing arcs of the node in a data flow graph is an operation package. In this section, we first design formats of *data tokens* and the *instruction sets or operation packages* for parallel processing of production system; then, we present an algorithm to compile rule programs into data flow graphs.

3.1 Tokens in DFLOPS

When a WME is created by a *make command*, the data is stored into the WM and a *positive token* is generated; when a WME is deleted by a *remove command*, the data is removed from the WM and a *negative token* is generated. A token carries the information of which instruction to be executed and which data to be operated. It has four fields:

$$(IP, Action, TimeTagList_1, TimeTagList_2)$$

where *IP* is a pointer to an instruction to be executed, *Action* is either *add*, denoted by '+', or *delete*, denoted by '-', operation. $TimeTagList_1$ and $TimeTagList_2$ are two time tags or

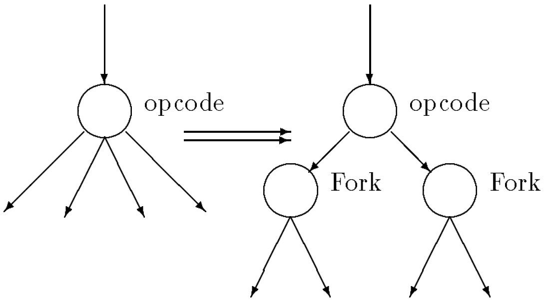

Figure 3: Multiple destinations problem and Fork instructions

a set of record pointers. A token with +/- in *Action* field is called a *positive/negative* token. Also a token sent to a two-input node from a left/right arc is called a left/right token.

3.2 DFLOPS Instructions

A DFLOPS instruction consists of 11 fields:

opcode	RI 1	FI 1	RI 2	FI 2	Dest. 1	Dest. 2	Flag 1	Ptr 1.	Flag 2	Ptr 2.

The *opcode* specifies which operation to be performed. A *record index* (RI), pointing to a WME or a record, and a *field index* (FI), indicating one field of the WME, together specify an operand. Thus, *RI 1* and *FI 1* specify the first operand and *RI 2* and *FI 2* the second operand. They will be used to fetch the data by *operand fetch units* and the instruction is executed in a *function unit. Dest. 1* and *Dest. 2* contain two pointers pointing to next instructions to be executed. In a RETE network, when a left/right token flows into a two-input node, it will match against all the tokens stored in the right/left *alpha memory.* A *LR memory flag* and a *LR memory pointer* together specify a *alpha or beta memory.* Thus, *flag 1* and *Ptr 1* specify the *alpha or beta memory* for the the instruction pointed by *Dest. 1* if it is a two-input-node instruction and *flag 2* and *Ptr 2* for the instruction pointed by *Dest. 2*.

In general, a new token may be forwarded to more than two destinations. *Fork* instructions are introduced to deal with this situation. For example, if a token will be forwarded to four different destinations, two *Fork* instructions are introduced and copy the incoming token into four outgoing tokens as shown in Figure 3.

The DFLOPS instructions have two kinds of instructions: one-input-node instructions and two-input-node instructions. One-input-node instructions are similar to the *selection* operations and two-input-node instructions *join operations* in database systems. Both one-input-node and two-input-node instructions have two operands, but the former contains one constant operand and one operand from a WME, while the latter has both operands from WMEs. The DFLOPS instruction set is shown in the Appendix.

3.3 Compiling Rule Programs

A rule program is compiled into five files: *token file, data file, instruction file, LHS constant file and RHS template file.* The toplevel commands: *make*, *remove* and *modify* commands create the token file and data file. These commands are compiled as following: A *make* command in production systems generates a positive token in *token file* and creates a WME in *data file*; a

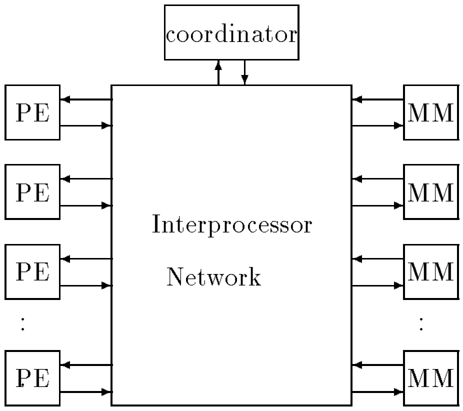

Figure 4: Architecture of DFLOPS:

remove generates a negative token but does not creates a WME, and a *modify* first generates a positive token and copies the bound WME to WM, and then generates a negative token to remove the old WME.

The set of rules of a program is compiled into *an instruction file, a LHS constant file and a RHS template file*. A *LHS constant file* holds all constants in LHS of rules and *A RHS template file* holds all templates in RHS. A template is a mixture of constants and variables. *An instruction file* is a data flow graph which contains set of nodes to be executed.

Compiling the LHS of a rule program into a data flow graph is similar to the process of generating a RETE network except there is no *root node*, instead each *class name test* becomes a parent node. The algorithm for compiling a production system program into DFLOPS instructions is shown in Algorithm I in the Appendix. Note that the compiler compiles both LHS match and RHS action into DFLOPS instructions.

A rule program to solve puzzle problem, the correspondent data flow graph, and the five output files are listed in appendix A.

The following section describes the architecture of proposed data flow machine for parallel processing of production systems in detail.

4 The Architecture of DFLOPS

DFLOPS, evolved from MIT TTDA, consists of a set of processing elements(PEs), a set of interleaved memory modules(MMs) and a switching network connecting PEs and MMs. The architecture is shown in Figure 4. The interleaved memory modules serve as buffers for instruction memory, local working memory and LR memory. Messages in the interprocessor network are simply tokens.

We first present the architecture of DFLOPS processing element in detail; then we show how DFLOPS processing element works in detail. Readers are encouraged to refer to Figure 5 while reading this section.

4.1 DFLOPS Processing Element

A DFLOPS processing element, shown in Figure 5, consists of eight pipeline stages: Instruction Fetch Unit(IFU), two Operand Fetch Units(OFUs), Function Unit(FU), Tag Computa-

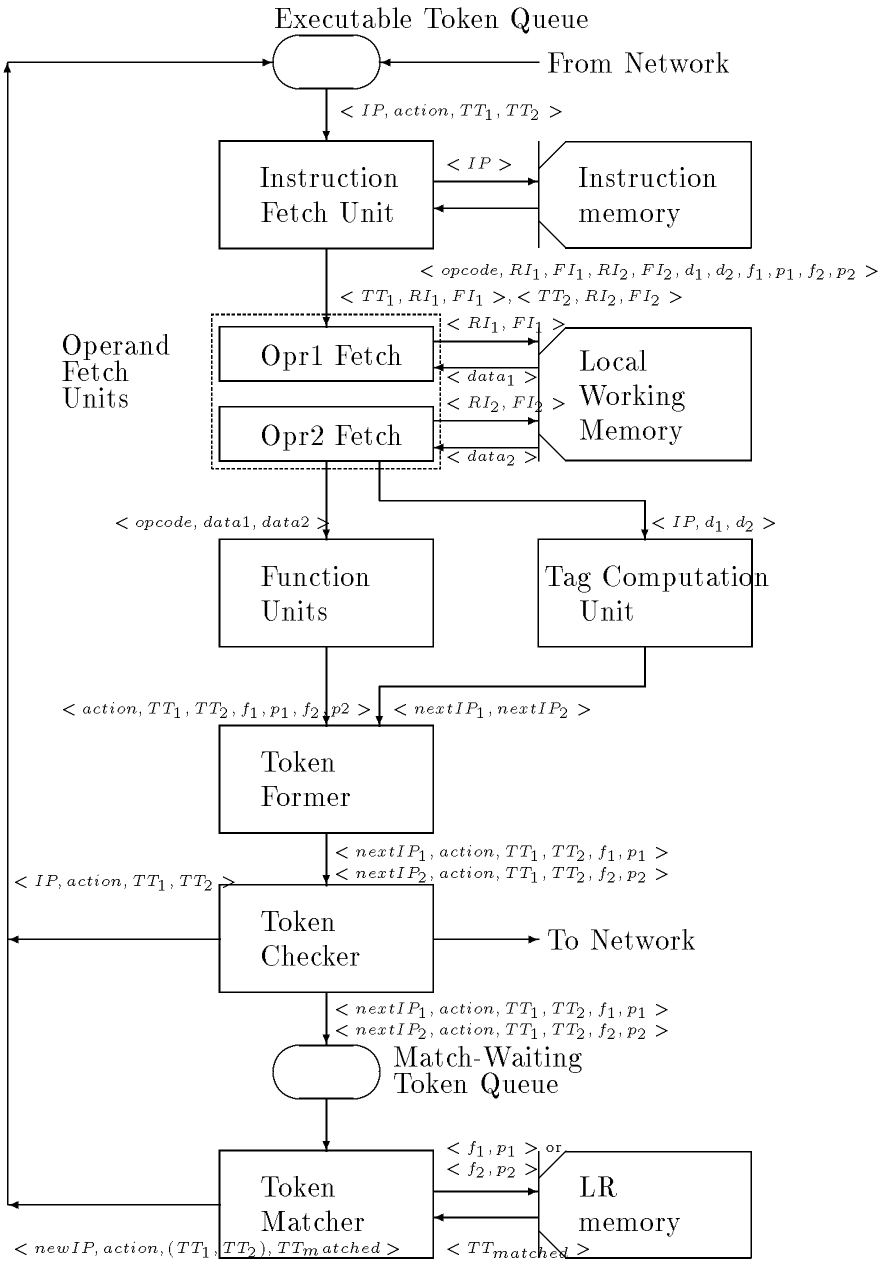

Figure 5: A Data flow Processing Element of DFLOPS:

tion Unit(TCU), Token Former(TF), Token Checker(TC), and Token Matcher(TM); three local memories: Instruction Memory(IM), Local Working Memory(LWM), and LR Memory(LRM); two Token Queues: Executable Token Queue(ETQ) and Match-Waiting Token Queue(MWTQ). The following subsections describes a PE in detail.

4.1.1 Instruction Memory

A DFLOPS instruction file is loaded and stored in an instruction memory before a PE starts execution. When a token is been processed by the instruction fetch unit of a PE, an instruction is fetched from *Instruction Memory* according to the IP field of the incoming token. The fetched instruction is sent to *Operand Fetch Units*.

4.1.2 Local Working Memory

Local working memory keeps the WMEs in *data file*, the constants in *LHS constant file* and the templates in *RHS template file*. During execution, WMEs are inserted, deleted and modified, while constants and templates remain unchanged. Operand Fetch Units retrieve the actual data to be operated from local working memory based on the *record index, field index* and *the WMEs bound in a incoming token*. The data will be operated in Function Unit in the next machine cycle.

4.1.3 LR Memory

LR memory is designed to handle the *one to many matching* (i.e. joining operations) and *data persistence* discussed in section 2. It consists of L and R Memories. L/R Memory serves as the Left/Right *alpha* and/or *beta* memories in the RETE network. Every entry in LR memory has two fields: *TimeTags* and *Link*. When a token flows into a two-input node, the TT1/TT2 of the token is stored in the entry of LR memory specified by *Flag 1/Flag 2* and *Ptr. 1/Ptr. 2*. Since there are more than one time tags that can be stored in an *alpha* or *beta* memory, tags are chained by the *Link* field. The function of the *Link* field of LR Memory is similar to the present bits mechanism in MIT TTDA. If the value of the *Link* of the start point of an *alpha* or *beta* memory is negative then this memory is empty, namely it contains no data; if the value is positive then there is more than two data stored in this memory; otherwise, the value is zero and that indicates there is only one data in this memory. *Link* = 0 in an entry also means that this entry is the last element in an *L or R memory.*

A token coming into a two-input node from the left/right is first inserted into the L/R Memory and then is matched against the corresponding *R/L Memory.* The combined tokens are generated and forwarded into the Executable Token Queue, waiting for processing. The above operations are repeated until the value of a *Link* is zero or negative.

4.1.4 Token Queues

There are two tokens queues– *Executable Token Queue*(ETQ) and *Match-Waiting Token Queue*(MWTQ)– in a DFLOPS PE. ETQ keeps all the tokens ready to be executed. Three sources, *token checker*, *token matcher* and other processing elements, can generate and forward tokens to ETQ.

MWTQ keeps all the tokens whose *IP value* points to a two-input node. *Token Matcher* will fetch tokens from MWTQ and matches against the LR memory. The matching operations is explained in subsection 4.2.2.

4.2 Pipeline Operations

There are eight RISC-like pipelined stages in a PE. They execute concurrently without suffering the *pipe-flushing phenomena* when a branch instruction executes in a RISC architecture. The DFLOPS can be subdivided into **execution pipe** and **match pipe** based on the functionalities. The **execution pipe** processes the tokens in Executable Token Queue, while the **match pipe** processes the tokens in Match-Waiting Token Queue. The execution pipe contains Executable Token Queue, Instruction Fetch Unit, Instruction Memory, Operands Fetch Units, Local Working Memory, Function Unit, Tag Computation Unit, Token Former, and Token Checker; the match pipe contains Match-Waiting Token Queue, Token Matcher and LR memory. The follow subsections show in detail how the DFLOPS PE works.

4.2.1 The pipeline operations of execution pipe

The execution pile is to execute the tokens ready to be executed in ETQ. The followings explain the pipeline operation of execution pipe in a DFLOPS PE.

- Instruction Fetch Unit fetches a token $< IP, action, TT_1, TT_2 >$ from ETQ and then fetches the instruction from the Instruction Memory specified by the IP field of the incoming token. The fetched instruction contains

  $< opcode, RI_1, FI_1, RI_2, FI_2, d_1, d_2, f_1, p_1, f_2, p_2 >$.

  where $< RI_1, FI_1, RI_2, FI_2 >$ is used by the Operand Fetch Units to fetch the operands, $< opcode >$ by the Function Unit to indicate which operation shall be performed, $< IP, d_1, d_2 >$ by the Tag Computation Unit to compute the instruction pointers of the next instructions to be executed, and two flags, $< f_1, f_2 >$, by the Token Checker to decide the new generated token to be sent to ETQ or MWTQ.

- The first Operand Fetch Unit fetches $data_1$ according to TT_1, RI_1 and FI_1 and the second Operand Fetch Unit fetches $data_2$ according to TT_2, RI_2 and FI_2.

- The Function Units executes $< opcode, data_1, data_2 >$. If the test succeeds, the *execution flag* will be set. In the same time, the addresses of next instructions to be executed are computed by the Tag Computation Unit and sent to the Token Former; $NextIP_1$ is calculated by adding IP and d_1 and $nextIP_2$ by adding IP and d_2. Note that d_1 and d_2 in the instructions of Table 5 are precalculated for convenient explanation; these values of the destinations shall be the offset of current instruction. For example, the first entry of the Table 5 shall be <TEQA,1,0,0,0,+1,+5,N,0,N,0>.

- The Token Former checks the *execution flag* set by FU. If the flag is "SUCCESS", then TF forms new tokens by replacing the IP of the incoming token with the new IPs(i.e. $nextIP_1$ and $nextIP_2$); otherwise, TF discards all the data sent from FU and TCU.

- Finally, the Token Checker checks if the IP of the new token is local. If not, the token is forwarded to the network and routed to the proper processor element. If so, TC further checks the f_1 or f_2 flags. If the flag is "N" (i.e. the instruction to be executed is an one-input-node instruction), the new token is sent to ETQ; otherwise, it must be "L" or "R" (i.e. the instruction to be executed is a two-input-node instruction) and the token is sent to MWTQ to find its partners.

The Execution pipe keeps processing tokens in ETQ until all the tokens are processed. Since these stages are executed in a pipelined fashion, the time to process a token is a machine cycle time.

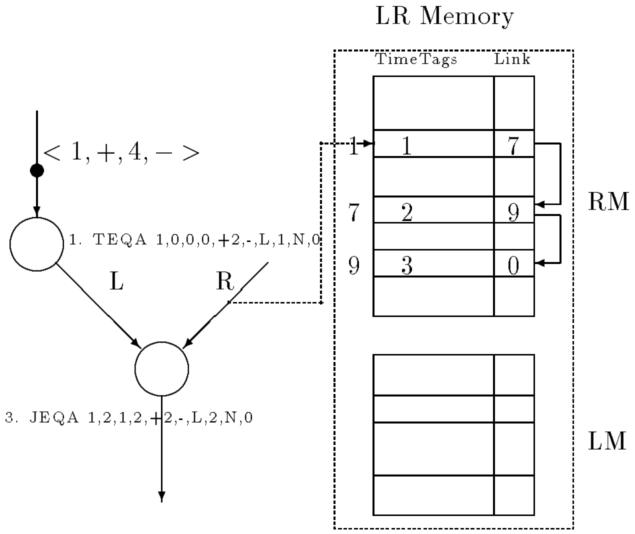

Figure 6: (a) The operation of Token Matcher:

4.2.2 The Operations of Match Pipe

Token Matcher first fetches a token form MWTQ and inserts the time tag of the token into L/R Memory specified by *LR memory flag and pointer*, (f_1, p_1) or (f_2, p_2.), then matches against the corresponding R/L Memory. If *Link* field in LR Memory is positive or zero, the matched time tag($TT_{matched}$) is fetched from LR memory. TT_1 and TT_2 are combined into a joint time tag and a new token, $< nextIP, action, (TT_1, TT_2), TT_{matched} >$, is generated and flowed to Executable Token Queue. Token Matcher repeatedly applies these operations until the last element is matched(i.e. $Link <= 0$).

The example shown in Figure 6 illustrates how Token Matcher works in detail. Part of the RETE Network for the add-boss rule, shown in Figure 1, is used as an example. The instruction node 1 is an one-input node which checks if the class name is equal to manager (i.e. class = manager) and the instruction node 3 is a two-input node which checks if the the manager and employee are in the same department (i.e. manager.dept = employee.dept). Suppose we have the following database:

```
1. (employee ^name Mary ^dept CS)
2. (employee ^name Ben  ^dept EE)
3. (employee ^name Tom  ^dept CS)
4. (manager  ^name John ^dept CS)
```

Figure 6(a) shows that three employee records with time tags 1,2,3 have been processed and stored in LR memory. They are chained by *link* fields. Now the manager token $< 2, +, 4, - >$ flows into the instruction node 1 waiting for execution.

After the token $< 1, +, 4, - >$ has been executed in node < 1. TEQA 1,0,0,0,+2,-,L,1,N,0>, a new token, $< 3, +, 4, - >$ is generated by the execution pipe as Figure 6(b) shown. This token is flowed to MWTQ because the the *LR memory flag*, f_1, of node 1 is "L". Token matcher works as follows: first it fetches this token from MWTQ and stores the time tag of this token(i.e. 4) into LM[1] specified by f_1 and p_1; second, it matches this token against the time tag list stored in RM[1]; three time tags are matched and three new tokens ,$< 3, +, 4, 1 >$, $< 3, +, 4, 2 >$ and $< 3, +, 4, 3 >$, are generated; finally, it flows the generated tokens to ETQ.

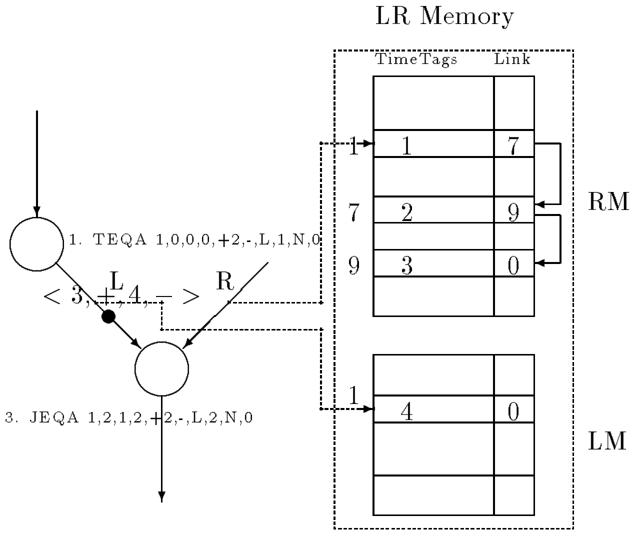

Figure 6: (b) The operation of Token Matcher

5 Parallel Executions

5.1 Execution Models

After a rule program is compiled into DFLOPS instructions, WMEs, tokens, LHS constants and RHS templates which are respectively stored in instruction file, data file, token file, constant file and template file, the coordinator loads the constants and the templates into local working memory of each PE. To achieve high performance, instructions, WMEs and tokens are partitioned. The more carefully the data and instructions are partitioned, the higher performance one can get. The partitioned instructions, WMEs and tokens are loaded into PEs. DFLOPS iteratively performs the following steps until the goal is satisfied or no more progress can be made.

1. match: All PEs perform LHS match. Whenever a rule is matched, DFLOPS's PE reports the instantiation to the coordinator and the coordinator performs the conflict resolution on the fly. When there is no more work to do, each PE send a *finish signal* to the coordinator.

2. Select: After receiving all the *finish signal* from each PE, the coordinator decides which rules should be fired in each PE and send the *go signal* to each PE.

3. execute: Each PE executes its RHS actions.

Several conflict resolution strategies can be applied to select a rule or a set of rules to fire. Thus we have the following execution models

- Model A: LEX or MEA conflict resolution strategy.

  All PEs perform LHS match and report the matched rules to the coordinator. When there is no more work to do, PEs send *finish signals* to the coordinator. Whenever receiving an instantiation, the coordinator performs LEX or MEA conflict resolution strategy. After receiving *finish signals* from all PEs, the coordinator chooses "the best rule" to fire based on the recency and specificity and sends a *go signal* to the PE who contains "the best rule." The PE executes the RHS actions and broadcasts the changed WMEs to other PEs.

- Model B: Fire-all strategy.

  All PEs perform LHS match but do not report the matched rules to the coordinator. When there is no more work to do, PEs send *finish signals* to the coordinator. After receiving *finish signals* from all PEs, the coordinator broadcasts a *go signal* to all PEs. All PEs execute the RHS actions and broadcast the changed WMEs to other PEs.

- Model C: LEX or MEA conflict resolution strategy with rule analysis.

  Rules dependency (i.e. which rules can be fired in parallel and which rule cannot) can be analyzed by the *dependency propagation algorithm* at compiled time[5], so a rule dependency matrix is constructed and kept in the coordinator to decide which rules can be fired in parallel without changing the semantics of production systems. This model works as follows: All PEs perform LHS match and report the matched rules to the coordinator. When there is no more work to do, PEs send *finish signals* to the coordinator. Whenever receiving an instantiation, the coordinator decides which rules can be executed in parallel based on the rule dependency matrix. After receiving *finish signals* from all PEs, the coordinator chooses "the best rule" and the rules that don't interfere with "the best rule", and sends *go signals* to PEs who contain one of the rules to be executed. The PEs executes the RHS actions and broadcasts the changed WMEs to other PEs.

- Model D: Meta rules.

  Meta rules are a set of rules to decide which instantiations in a conflict set shall be synchronized[29]. Meta rules can be compiled into token instructions as the production rules described in section 3. The compiled meta rule instructions are then stored in the coordinator. This model works as follows: All the PEs perform LHS match and report the matched rules to the coordinator. When there is no more work to do, PEs send *finish signals* to the coordinator. Whenever receiving an instantiation, the coordinator performs conflict resolution strategy by matching the instantiation to those meta rules. Some instantiations are deleted, others survives. After receiving *finish signals* from all PEs, the coordinator sends *go signals* to PEs who contain one of the survived rules to be executed. The PEs executes the RHS actions and broadcasts the changed WMEs to other PEs.

The Model D hasn't been implemented yet. In the example shown in next subsection, we uses a simple hash function, namely module N, to partition the WMEs and tokens but does not partition the instruction memory.

5.2 An Example

In this subsection we'll illustrate how DFLOPS works by using the puzzle program in Figure 9 and the initial database in Table 7. Although the following subsection describes the pipeline operations in a single PE, readers can imagine that a set of PEs execute concurrently. We assume each stage in a PE can finish its job in one machine cycle time.

Table 1 shows a time-stage execution snapshots of executing the puzzle program inside a PE. The values in column A indicates the *ith* machine cycle; column B shows the tokens to be executed at the *ith* machine cycle; Column C to column I are the pipeline stages of execution pipe; Column J is the pipeline stage of match pipe; column K describes the types of operations in FU for explanation purpose. If the IFU fetches an instruction at *ith* cycle time, then FOFU processes this token at *(i+1)th* cycle, SOFU at *(i+2)th* cycle, FU and TCU at *(i+3)th* cycle, TF at *(i+4)th* cycle, and TC at *(i+5)th* cycle.

Column C shows the instruction pointers fetched by IFU. The operands to be operated in FU are extracted from local working memory by FOFU and SOFU; the first operands are shown in column D and the second ones in column E. Column F holds the values of *execution flag* set by Function Units. 'S' means "test Succeeds", namely the instruction executed in FU is successful, and a new token will be generated by Token Former. 'F' means "test Fails" and this token is discarded. The values in column G are the next instruction pointers to be executed. The values in column H indicates whether new tokens are generated or not; 'Y' indicates one or two new tokens are generated and 'N' means no token is generated. The data in column I indicates the new generated token will be sent to ETQ, MWTQ or the network; 'M' in column I means Token Checker forwards the token to Match-Waiting Token Queue, 'E' to Executable Token Queue, 'G' to the interconnection network and '-' means no operation. 'N' in column J means the MWTQ is empty and Token Matcher is idle; 'Y' means a token is inserted to LR memory but no token on the other side can be matched, therefore no token is generated; a pair of numbers in column J means matched tokens are generated and forwarded to Executable Token Queue. Readers are encouraged to see the appendix for more detail explanations.

6 Results

6.1 Fine Grain Parallelism

DFLOPS exploits three kinds of parallelism: first, each PE executes in parallel; second, the execution pipe and the match pipe in a PE executes in parallel; finally, each pipeline stage in an execution pipe execute in pipelined fashion.

The machine normally executes MIMD mode i.e. each PE executes different node instructions or rules with different WMEs. It also can execute SIMD and MISD modes. SIMD mode in production systems machine is to execute the same rule with different WMEs in different PEs; MISD mode is to execute different rules with the same WMEs in different PEs. To execute SIMD mode, WMEs are partitioned into several buckets by a hash function and then load the same instructions and one of the buckets to a PE. To execute MISD mode, Rules are partitioned into the sets and then load one set and the same WMEs to a PE. MISD is similar to *Copy and Constraint* technique[29].

6.2 Simulation Environment

The DFLOPS simulator is implemented with execution models A, B and C. The software environment is shown in Figure 7. An OPS5 compiler is implemented to compile a rule program into instruction file, LHS constant file, RHS template file, data file and token file. we are writing SQL compiler for SQL languages and other compilers for some others high level languages such as Datalog.

Table 1: An example of pipeline operations in a PE

		Execution Pipe							Match Pipe	Comments
A	B	C	D	E	F	G	H	I	J	K
Time	Executable Token	IF IP	FOFU data1	SOFU data2	FU	TCU d1,d2	TF	TC	TM	
LHS Matching in the Recognize-Act Cycle: 1										
1	(1,+,1,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
2	(1,+,2,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
3	(1,+,3,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
4	(1,+,4,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
5	(1,+,5,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
6	(1,+,6,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
7	(1,+,7,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
8	(1,+,8,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
9	(1,+,9,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
10	(1,+,10,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
11	(1,+,11,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
12	(1,+,12,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
13	(1,+,13,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
14	(1,+,14,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
15	(1,+,15,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
16	(1,+,16,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
17	(2,+,1,0)	2	*	*	F	3, 3	N	-	N	shape ≠ *
18	(6,+,1,0)	6	*	*	S	7, 11	Y	E ,E	N	shape = *
19	(2,+,2,0)	2	*	*	F	3, 3	N	-	N	shape ≠ *
20	(6,+,2,0)	6	*	*	S	7, 11	Y	E ,E	N	shape = *
21	(2,+,3,0)	2	a	*	S	3, 3	Y	M	(3,3)	shape ≠ *
22	(6,+,3,0)	6	a	*	F	7, 11	N	-	N	shape = *
23	(2,+,4,0)	2	b	*	S	3, 3	Y	M	(4,4)	shape ≠ *
24	(6,+,4,0)	6	b	*	F	7, 11	N	-	N	shape = *
25	(2,+,5,0)	2	a	*	S	3, 3	Y	M	(5,5)	shape ≠ *
26	(6,+,5,0)	6	a	*	F	7, 11	N	-	N	shape = *
27	(2,+,6,0)	2	*	*	F	3, 3	N	-	N	shape ≠ *
28	(6,+,6,0)	6	*	*	S	7, 11	Y	E ,E	N	shape = *
29	(2,+,7,0)	2	*	*	F	3, 3	N	-	N	shape ≠ *
30	(6,+,7,0)	6	*	*	S	7, 11	Y	E ,E	N	shape = *
...	...				...				...	
46	(6,+,15,0)	6	*	*	S	7, 11	Y	E ,E	N	shape = *
47	(2,+,16,0)	2	*	*	F	3, 3	N	-	N	shape ≠ *
48	(6,+,16,0)	6	*	*	S	7, 11	Y	E ,E	N	shape = *
49	(7,+,1,0)	7	no	no	S	8, 8	Y	M	(1,1)	match=no
50	(11,+,1,0)	11	no	yes	F	12, 12	N	-	N	match=yes
51	(7,+,2,0)	7	no	no	S	8, 8	Y	M	(2,1)(1,2)(2,2)	match=no
52	(11,+,2,0)	11	no	yes	F	12, 12	N	-	N	match=yes
53	(3,+,3,3)	3	a	a	S	4, 0	Y	E	N	P1.shape=P2.shape
54	(3,+,3,3)	3	a	a	S	4, 0	Y	E	N	P1.shape=P2.shape
55	(3,+,4,3)	3	b	a	F	4, 0	N	-	N	P1.shape=P2.shape
56	(3,+,3,4)	3	a	b	F	4, 0	N	-	N	P1.shape=P2.shape
57	(3,+,4,4)	3	b	b	S	4, 0	Y	E	N	P1.shape=P2.shape
...	...				...				...	
133	(7,+,15,0)	7	no	no	S	8, 8	Y	M	(15,1)(15,2)...	match=no
134	(11,+,15,0)	11	no	yes	F	12, 12	N	-	N	match=yes
135	(7,+,16,0)	7	no	no	S	8, 8	Y	M	(16,1)(16,2)...	match=no
136	(11,+,16,0)	11	no	yes	F	12, 12	N	-	N	match=yes
137	(8,+,1,1)	8	1	1	S	9, 0	Y	E	N	P1.id=P2.id
138	(8,+,1,1)	8	1	1	S	9, 0	Y	E	N	P1.id=P2.id
139	(8,+,2,1)	8	1	1	S	9, 0	Y	E	N	P1.id=P2.id
140	(8,+,1,2)	8	1	1	S	9, 0	Y	E	N	P1.id=P2.id
141	(8,+,2,2)	8	1	1	S	9, 0	Y	E	N	P1.id=P2.id
142	(8,+,2,2)	8	1	1	S	9, 0	Y	E	N	P1.id=P2.id

Table 1: (continue):

		Execution Pipe							Match Pipe	Comments
A	B	C	D	E	F	G	H	I	J	K
Time	Executable Token Q.	IF IP	FOFU data1	SOFU data2	FU	TCU d1,d2	TF	TC	TM	
228	(8,+,12,16)	8	3	4	F	9, 0	N	-	N	P1.id=P2.id
229	(8,+,16,15)	8	4	4	S	9, 0	Y	E	N	P1.id=P2.id
230	(8,+,15,16)	8	4	4	S	9, 0	Y	E	N	P1.id=P2.id
231	(8,+,16,16)	8	4	4	S	9, 0	Y	E	N	P1.id=P2.id
232	(8,+,16,16)	8	4	4	S	9, 0	Y	E	N	P1.id=P2.id
233	(9,+,1,1)	9	1	1	F	10, 0	N	-	N	P1.side=P2.side
234	(9,+,1,1)	9	1	1	F	10, 0	N	-	N	P1.side=P2.side
235	(9,+,2,1)	9	2	1	S	10, 0	Y	E	N	P1.side=P2.side
236	(9,+,1,2)	9	1	2	F	10, 0	N	-	N	P1.side=P2.side
237	(9,+,2,2)	9	2	2	F	10, 0	N	-	N	P1.side=P2.side
238	(9,+,2,2)	9	2	2	F	10, 0	N	-	N	P1.side=P2.side
...	...				...				...	
257	(9,+,16,15)	9	16	15	S	10, 0	Y	E	N	P1.side=P2.side
258	(9,+,15,16)	9	15	16	F	10, 0	N	-	N	P1.side=P2.side
259	(9,+,16,16)	9	16	16	F	10, 0	N	-	N	P1.side=P2.side
260	(9,+,16,16)	9	16	16	F	10, 0	N	-	N	P1.side=P2.side
261	(10,+,2,1)	10	no	ppiece	F	28, 29	N	-	N	Report corner
262	(10,+,7,6)	10	no	ppiece	F	28, 29	N	-	N	Report corner
263	(10,+,12,9)	10	no	ppiece	F	28, 29	N	-	N	Report corner
264	(10,+,16,15)	10	no	ppiece	F	28, 29	N	-	N	Report corner
RHS Actions in the Recognize-Act Cycle: 1										
265	(28,+,0,16:15)	28	CORNER: 4	-	S	0, 0	Y	-	N	Print MSG
266	(29,+,0,16:15)	29	-	-	S	30, 35	Y	E ,E	N	Fork
267	(30,+,0,16:15)	30	16	-	S	31, 33	Y	E ,E	N	Copy wme 16
268	(35,+,0,16:15)	35	15	-	S	36, 38	Y	E ,E	N	Copy wme 15
269	(31,+,17,16:15)	31	no	yes	S	32, 0	Y	E	N	UPDT match
270	(33,+,17,16:15)	33	16	-	S	34, 0	Y	E	N	DELE wme 16
271	(36,+,18,16:15)	36	no	yes	S	37, 0	Y	E	N	UPDT match
272	(38,+,18,16:15)	38	15	-	S	39, 0	Y	E	N	DELE wme 16
273	(32,+,17,16:15)	32	-	-	F	1, 0	N	-	N	GEN Token
274	(34,-,16,0)	34	-	-	F	1, 0	N	-	N	GEN TOken
275	(37,+,18,0)	37	-	-	F	1, 0	N	-	N	GEN TOken
276	(39,-,15,0)	39	-	-	F	1, 0	N	-	N	GEN TOken
LHS Matching in the Recognize-Act Cycle: 2										
277	(1,+,17,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
278	(1,-,16,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
279	(1,+,18,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
280	(1,-,15,0)	1	ppiece	ppiece	S	2, 6	Y	E ,E	N	class=ppiece
281	(2,+,17,0)	2	*	*	F	3, 3	N	-	N	shape $\neq$ *
282	(6,+,17,0)	6	*	*	S	7, 11	Y	E ,E	N	shape = *
283	(2,-,16,0)	2	*	*	F	3, 3	N	-	N	shape $\neq$ *
284	(6,-,16,0)	6	*	*	S	7, 11	Y	E ,E	N	shape = *
285	(2,+,18,0)	2	*	*	F	3, 3	N	-	N	shape $\neq$ *
286	(6,+,18,0)	6	*	*	S	7, 11	Y	E ,E	N	shape = *
287	(2,-,15,0)	2	*	*	F	3, 3	N	-	N	shape $\neq$ *
288	(6,-,15,0)	6	*	*	S	7, 11	Y	E ,E	N	shape = *
...	...				...				...	
RHS Actions in the Recognize-Act Cycle: 10										
582	(28,+,0,2:1)	28	CORNER: 4	-	S	0, 0	Y	-	N	Print MSG
583	(29,+,0,2:1)	29	-	-	S	30, 35	Y	E ,E	N	Fork
584	(30,+,0,2:1)	30	16	-	S	31, 33	Y	E ,E	N	Copy wme 16
585	(35,+,0,2:1)	35	15	-	S	36, 38	Y	E ,E	N	Copy wme 15
586	(31,+,29,2:1)	31	no	yes	S	32, 0	Y	E	N	UPDT match
587	(33,+,29,2:1)	33	16	-	S	34, 0	Y	E	N	DELE wme 16
588	(36,+,30,2:1)	36	no	yes	S	37, 0	Y	E	N	UPDT match
589	(38,+,30,2:1)	38	15	-	S	39, 0	Y	E	N	DELE wme 16
590	(32,+,29,2:1)	32	-	-	F	1, 0	N	-	N	GEN Token
591	(34,-,2,0)	34	-	-	F	1, 0	N	-	N	GEN Token
592	(37,+,30,2:1)	37	-	-	F	1, 0	N	-	N	GEN Token
593	(39,-,1,0)	39	-	-	F	1, 0	N	-	N	GEN Token

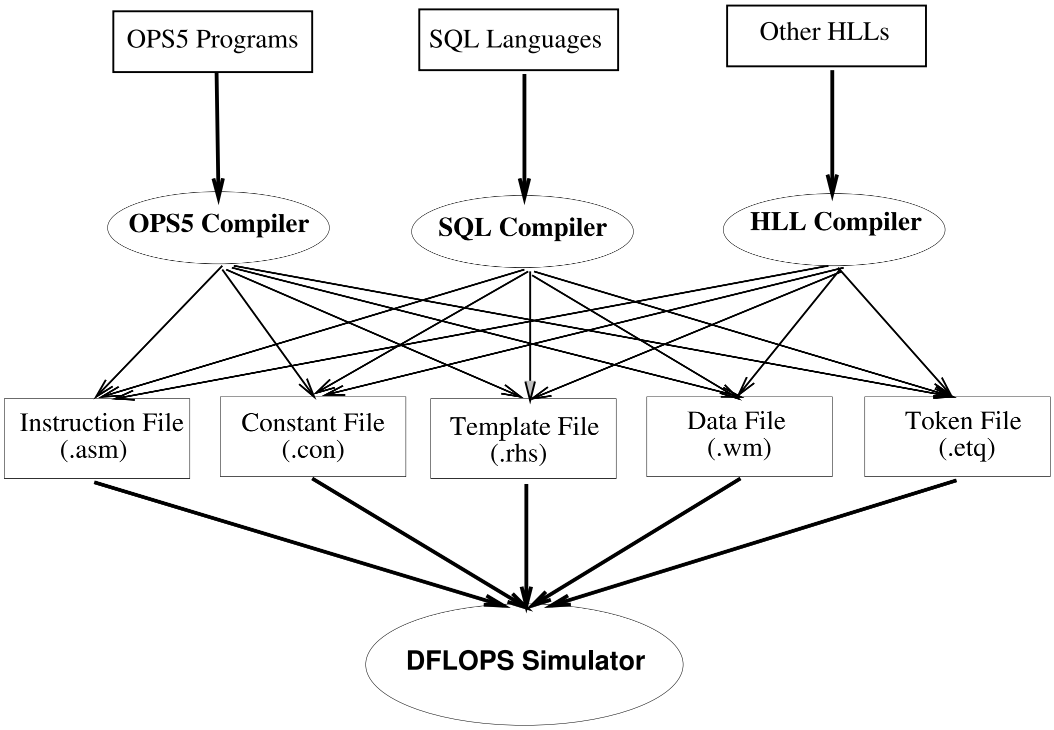

Figure 7: DFLOPS Software Environment:

6.3 Performance Evaluation

In this experiment, the sequential execution time of a run is calculated by the number of tokens executed in LHS match plus the number of tokens executed in RHS action. The parallel execution time of a run in DFLOPS is complicated, for the the matching phase overlaps selecting phase in each cycle and the action phase in ith cycle overlaps the matching phase in (i+1)th cycle. The parallel execution time is measured by recording the total machine cycles executed in DFLOPS.

The speedup of parallel execution is determined by the sequential execution time SET over the parallel execution time PET.

$$Speedup = average\ SET\ /\ average\ PET$$

The effectiveness is calculated by speedup over the number of PEs used,

$$Effectiveness = Speedup\ /\ the\ number\ of\ PEs\ used.$$

For the sequential execution time, we run five cases for the puzzle program: PP4, PP6, PP8, PP10 and PP12. The only difference among them is the number of puzzle pieces. PP4 has 4×4 puzzle pieces; Each puzzle piece has 4 edges; Thus the the number of initial WMEs is 64. The numbers of puzzle pieces, initial WMEs, tokens executed in LHS, tokens executed in RHS, and the total number of tokens executed are shown in Table 6.3.

For the parallel execution, there are 4 execution models proposed on previous section. The following evaluation of DFLOPS is based on Model C. The reason we use this model is that the results of this model are the same as those of sequential execution with LEX and MEA semantics.

Table 2: Sequential Execution Time:

	PP4	PP6	PP8	PP10	PP12
# of Puzzle Pieces	16	36	64	100	144
# of initial WMEs	64	144	256	400	576
# of tokens executed in LHS	4470	21350	68518	150838	245610
# of tokens executed in RHS	408	948	1728	2199	2235
Total # of tokens executed	4878	22298	70246	153091	247845
Total machine cycle time executed	4878	22298	70246	153091	247845

Table 3: Parallel Execution Time:

	PP4	PP6	PP8	PP10	PP12
# of PEs used	5	7	9	11	13
Total # of tokens executed	2676	8668	18704	34443	57448
Total machine cycle time executed	1028	2268	4292	7292	11460
Speed up	4.745	9.832	16.367	20.972	21.627
Effectiveness	0.949	1.405	1.819	1.907	1.664

To evaluate the of the performance of DFLOPS, we first partition the WMEs into two sets: border pieces WMEs(i.e. the pieces with $shape = *$) and non-border pieces(i.e. the pieces with $shape \neq *$); then, the non-border pieces are further partitioned into N buckets by module N function. The N+1 buckets are mapped into N+1 DFLOPS PEs. For example, in PP4, non-border pieces are hashed into 4 buckets, so four PEs is used to hold these four buckets and one more PE is used to hold the bucket with border pieces. By partitioning the data into several subsets, a lot of useless join operations are eliminated and higher performance can be achieved.

The parallel execution time, speedup and effectiveness are shown in Table 6.3. Figure 8 shows the speedup in different number of PEs used. Figure 8 also shows that DFLOPS scales well as the problem size increases.

Due to the preprocessing of the WM, i.e. partitioning the WMEs into several disjoint subsets, the effectiveness in case PP6, PP8, PP10, PP12 is superlinear. Considering joining two WME classes: A with M records and B with N records, we need $M \times N$ operations if we join them in one PE. If we can partition them into k disjoint subsets i.e. each with M/k records in A and N/k in B, then each PE takes only $M \times N/k^2$ operations. It is obvious that careful partitioning should be concerned to ensure correctness as well as to achieve high performance.

7 Conclusions

In this paper, we first analyze differences between data-driven symbolic and numerical computations, then propose a new data flow machine for parallel processing of production systems. A compiler is developed to compile production system programs into DFLOPS instructions. The architecture and an example to illustrate how this machine work are discussed in detail. Unlike traditional data flow machines which are suitable for numerical computations, our model is designed to perform data-driven symbolic computations efficiently. We also demonstrate that data flow principle can be applied in expert systems.

The distinguishing characteristics of this machine lie in its simplicity, fully-pipelined pro-

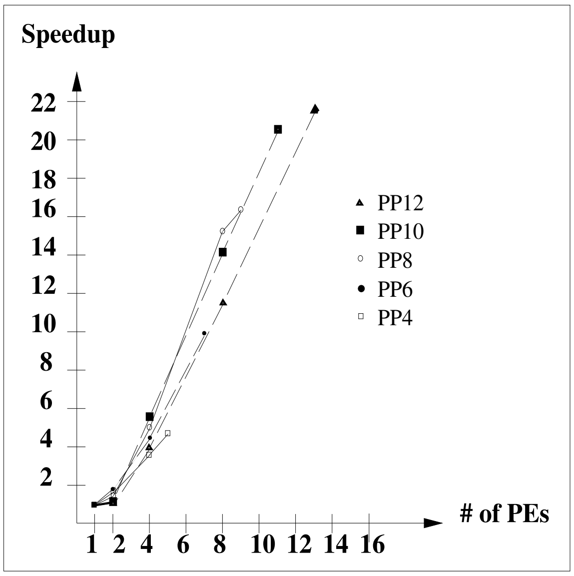

Figure 8: The Speedup of DFLOPS:

cessing and fine grain parallelism. Three levels of parallelism, rule level parallelism, LHS level parallelism and RHS parallelism, in production systems are fully exploited to achieve high performance. MISD, SIMD and/or MIMD modes of parallel processing can be exploited in this machine according to the properties of applications. This machine is suitable for distributed production systems as well as expert database systems. Our parallel execution model exploits not only parallel matching but also parallel firing and selection. Thus the performance of production systems is greatly improved. The initial results show high speedup is achieved and the effectiveness is superlinear.

References

[1] Arvind and Rishiyur S. Nikhil. Executing a program on the mit tagged-token data flow architecture. *IEEE Transactions on Computer*, 39:3, March 1990.

[2] L. Bic and R. Hartmann. Agm: The irvine dataflow database machine. In *Data Flow Computing*, pages 387–412, 1987.

[3] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison Wesley, Reading, Mass., 1985.

[4] G. Bultzingloewsn and Cirano Iochpe. Design and implementation of kardamom – a set-oriented data flow database machine. In *International Workshop on Database Machines*, pages 18–33, 1989.

[5] Fu-Chiung Cheng, Huei-Huang Chen, and Jing-Hwi Perng. Parallel execution on production systems. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 463–470, 1990.

[6] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, R. Jauhari, M. Muralikrishna, and Anoop Sharma. A single user evaluation of the gamma database machine. In *Database Machine and Knowledge Base Machines*, pages 370–386, 1987.

[7] C. L. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, July 1981.

[8] C.L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence*, pages 17–37, 1982.

[9] Jean-Luc Gaudiot and Andrew Sohn. Data-driven parallel production systems. *IEEE trans. on Software Engineering*, 16:3, March 1990.

[10] A. Gupta. Implementing ops5 production systems on dado. In *Proceedings of International Conference on Parallel Processing*, pages 83–91, 1984.

[11] A. Gupta. Parallel ops5 on the encore multimax. In *Proceedings of International Conference on Parallel Processing*, pages 271–280, 1988.

[12] B.K. Hillyer and D.E. Shaw. Execution of ops5 production systems on a massively parallel machine. *Journal of parallel and distributed computing*, 3:1:236–268, 1986.

[13] Kai Hwang and Faye A Briggs. *Computer Architecture and Parallel Processing.* MaGraw Hill, 1984.

[14] Toru Ishida and S.J. Stolfo. Optimizing rules in production system programs. In *Proceedings of the National Conference of the American Association for Artificial Intelligence*, pages 699–704, 1988.

[15] Stolfo S. J. Five parallel algorithms for production system execution on the DADO machine. In *Proceedings of the National Conference on Artificial Intelligence 1984*, pages 300–307. AAAI, 1984.

[16] M.A. Kelly and R.E. Seviora. A multiprocessor architecture for production system matching. In *Proceedings of the National Conference of the American Association for Artificial Intelligence*, pages 36–41, 1987.

[17] Jean-Luc Gaudiot S. Lee and Andrew Sohn. Data-driven multiprocessor implementation of the rete match algorithm. In *Proceedings of International Conference on Parallel Processing*, pages 256–260, 1988.

[18] J. McDermott. R1: A rule-based configurer of computer system. *Artificail Intelligence*, 1982.

[19] H.S.L M.I. Schor, T.P Daly and B.R. Tibbitts. Advanced rete matching algorithm. In *Proceedings of National conference of Artificial Intelligence*, pages 226–232, 1986.

[20] D. P. Miranker, D. A. Brant, B. Lofaso, and D. Gadbois. On the performance of lazy matching in production systems. In *Proceedings of AAAI-90*, pages 685–692, 1990.

[21] D.P. Miranker. Treat: A better match algorithm for ai production systems. In *Proceedings of National conference of Artificial Intelligence*, pages 36–41, 1987.

[22] K. Oflazer. Partitioning in parallel processing of production systems. In *Proceedings of the IEEE International Conference on Parallel Processing*, pages 92–100. IEEE, 1984.

[23] Gregory M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor.* MIT Press, 1991.

[24] Gregory M. Papadopoulos and David E. Culler. Monsoon: an explicit token-store architecture. In *Proceedings of IEEE Symp. on Computer Architectures*, pages 82–91, 1990.

[25] F. Schreiner and G. Zimmermann. Pesa i- a parallel architecture for production systems. In *Proceedings of the IEEE International Conference on Parallel Processing*, pages 166–169, 1987.

[26] Yoshinori Yamaguchi Shuichi Sakai and Kei Kiraki. An architecture of a dataflow single chip processor. In *Proceedings of IEEE Symp. on Computer Architectures*, pages 46–53, 1989.

[27] S. J. Stolfo. Initial performance of the DADO2 prototype. *IEEE Computer Special Issue on AI Machines*, 20:75–83, January 1987.

[28] S. J. Stolfo and Miranker D. The DADO poduction system machine. *Journal of Parallel and Distributed Systems*, August 1986.

[29] S. J. Stolfo, H.M. Dewan, and O. Wolfson. The PARULEL parallel rule language. In *Proceedings of the IEEE International Conference on Parallel Processing*, pages II:36–45. IEEE, 1991.

[30] S. J. Stolfo and D. P. Miranker. DADO: A parallel processor for expert systems. In *Proceedings of the 1984 International Parallel Processing Conference*, pages 74–82. IEEE, 1984.

[31] S. J. Stolfo and D.E. Shaw. DADO: A tree-structured machine architecture for production systems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 242–246, 1982.

[32] G.T. Vesonder and S. J. Stolfo. The ace system. In *Proceedings of the Workshop on Expert Systems*, Australia, August 1984.

A APPENDIX

A.1 DFLOPS Instruction set:

Table 4: DFLOPS Instruction Set

One-Input-Node Instructions:	
Format	Comment
TEQA string1 string2	test string1 == string2
TNEA string1 string2	test string1 != string2
TEQN number1 number2	test number1 == number2
TNQN number1 number2	test number1 != number2
TGEN number1 number2	test number1 >= number2
TGTN number1 number2	test number1 > number2
TLEN number1 number2	test number1 <= number2
TLTN number1 number2	test number1 < number2
FORK	duplicate tokens
TERM	report instantiation
DELE WME	delete the WME
COPY WME	copy a WME and attach a new time tag to it
UPDT WME.filed data	update the value of an attribute of a WME
SEND messagee	send a message to other processing elements
GENT	generate a new token
Two-Input-Node Instructions:	
Format	Comment
JEQA string1 string2	join if string1 == string2
JNEA string1 string2	join if string1 != string2
JEQN number1 number2	join if number1 == number2
JNQN number1 number2	join if number1 != number2
JGEN number1 number2	join if number1 >= number2
JGTN number1 number2	join if number1 > number2
JLEN number1 number2	join if number1 <= number2
JLTN number1 number2	join if number1 < number2

A.2 Algorithm I: The algorithm to generate DFLOPS instructions:

```
Algorithm: Compiling a rule program.
input: a rule program.
output: an instruction file, a LHS constant file and a RHS template file.
method:
compiler_rules()
{
   for each rule, r, in the program
   {
     /* compiling the LHS of a program */
     current_variables={}; /* set current_variables empty */
     for each condition element, ce, in the rule, r
     {
        if (there is a node testing class name of ce)
        then follow the link and call it parent node, p_node;
        else {
          generate one-input node to test the class name;
          insert the class name to LHS constant file;
          and call it parent node, p_node;
        }

        for each constant test in the condition element, ce
        {
           if (the constant test already exists)
           then follow the link and call it parent node, p_node;
           else {
             c_node = generate one-input node for this constant test;
```

```
            insert the constant to LHS constant file;
            check_and_link(p_node,c_node);
          }
          p_node = c_node; /* current node becomes parent node */
       }

       for each variable, var, in the condition element, ce
       {
          if (the variable, var, in current_variables) /* two-input node */
          then { c_node = generate a two-input node;
                 check_and_link(p_node,c_node); }
          else put the pair, (var, ce), into current_variable;
       }
    } /* end of LHS compiling */

    /* terminal node for LHS */
    generate a terminal node(t_node) for the rule, r;
    check_and_link(p_node,t_node);

    /* compiling the RHS of a program */
    for each RHS action, rhs_action in the rule, r
    {
       p_node = t_node;
       check and insert the template into RHS template file;
       switch(opcode of rhs_action) {
       case make:
         c_node = generate a MAKE node;
         check_and_link(p_node,c_node);
         p_node = c_node;
         gen_attribute_value_code(p_node,rhs_action);
       case remove:
         gen_remove_code(p_node);
       case modify:     /* create a new WME and then destroy the old one */
         c_node = generate a COPY node;
         check_and_link(p_node,c_node);
         p_node = c_node;
         p_temp_node = p_node;
         gen_attribute_value_code(p_node,rhs_action);
         gen_remove_code(p_temp_node);
       case /* others rhs actions */
      .......
       } /* end of switch */
    } /* end of rhs compiling */
  } /* end of for each rule */
}

check_and_link(p_node, c_node);
{
  if (its p_node has more than two destinations)
  then { generate a fork node(f_node);
         set the second child of p_node to first child of the f_node;
         set f_node to the second child of p_node;
         set c_node to the second child of f_node; }
  else link this node to p_node;
}

gen_remove_code(p_node)
{
  c_node = generate a DELETE node;
  check_and_link(p_node,c_node);
  p_node = c_node;
  generate a GENT node;
  check_and_link(p_node,c_node);
}

gen_attribute_value_code(p_node,rhs_action)
{
  for each attribute-value pair
  {
```

```
    c_node = generate one-input node for attributes;
    check_and_link(p_node,c_node);
    p_node = c_node;
  }
  generate a GENT node;
  check_and_link(p_node,c_node);
}
```

A.3 The Puzzle Program and The Five Output Files

The rule program to solve the puzzle problems is listed in Figure 9 and the corresponding data flow graph is shown in Figure 10. The oval nodes in the data flow graph are the LHS matching operations and the rectangle nodes the RHS actions.

The four rules are compiled into three files: the instruction file of the puzzle program is shown in Table 5; the LHS constant file and the RHS template file are shown in Table 6.

The sixteen *make commands* in the puzzle program create sixteen tokens stored in *token file* and sixteen WMEs stored in *data file* shown in Table 7.

A.4 Explanation of Pipeline Operations in a PE

The first 16 rows of Table 1 show the tokens generated at top level. The token in 17th row is generated by the first token in first row. The first row can be interpreted as the follows:

At the first cycle time, token $< 1, +, 1, 0 >$ is fetched by IFU. Since the IP of the token is 1, IFU fetches the first instruction $< TEQA, 1, 0, 0, 0, +1, +5, N, 0, N, 0 >$ from the Instruction Memory. The first operand *(ppiece)* is fetched from the first WME by the First Operand Fetch Unit at the 2nd cycle time. The second operand *(ppiece)* is fetched by the Second Operand Fetch Unit at the 3rd cycle time. FU performs the *(TEQA)* operation against $data_1$ and $data_2$ at the 4th cycle time. At the same time, the Tag Computation Unit computes the addresses of next instructions *(1+1 and 1+5)*. Since the test operation in FU succeeds, the Token Former replaces IP of the token with the addresses of next instructions to be executed and generate two new token $< 2, +, 1, 0 >$ and $< 6, +, 1, 0 >$ in the 5th cycle time. Finally, the Token Checker finds that both tokens, $< 2, +, 1, 0 >$ and $< 6, +, 1, 0 >$, shall be forwarded to ETQ and flows them at 6th cycle time. Since there are nothing in MWTQ the Token Matcher is idle.

The second token $< 1, +, 2, 0 >$ is fetched by IFU at the 2nd cycle time and flows through the pipelined stages and ends at the 7th cycle time. The execution of the second token in pipelined stages is similar to that of the first one except one machine cycle time behind.

After the first and second tokens shown in row 1 and 2 are executed, four tokens shown in rows 17-20 are generated; two of the four tokens (i.e. row 18 and 20) succeed and keep generating four tokens(rows 49-52), the others two tokens (i.e. row 17 and 19) fail and die. In fact, the first and second tokens(or WMEs) form a corner, so the tokens descended from them keep generating new tokens until they report that **puzzle piece 1 is a corner**(see row 261). This instantiation will keep in conflict set until the coordinator select it to fire; it is fired at the 582nd cycle time.

```
;; database schema
(literalize bucket p1 p2 shape status)

(literalize ppiece id shape match side)

(p hashtobucket   ;; find two puzzle pieces with the same shape
  (ppiece ^id <x1> ^shape { <e> <> *})
  (ppiece ^id {<x2> < <x1>} ^shape <e>)
-->
  (write HASH Put piece <x1> and <x2> in the BUCKET (crlf))
  (make bucket ^p1 <x1> ^p2 <x2> ^shape <e> ^status undone))

(p corner        ;; find the corner pieces
  (ppiece ^id <x> ^shape * ^match no ^side <y>)
  (ppiece ^id <x> ^shape * ^match no ^side <> <y>)
-->
  (write Corner:  Piece <x> is a corner.  Put it on puzzle (crlf))
  (modify 1 ^match yes)
  (modify 2 ^match yes))

(p cornerjoin     ;; Put two puzzle pieces together
  (ppiece ^id <x> ^shape * ^match yes ^side <a>)
  (ppiece ^id <x> ^shape * ^match yes ^side <> <a>)
  (bucket ^p1 <x> ^p2 <y> ^shape <e> ^status undone)
-->
  (write Connerjoin: join pieces <y> and <x> by shape <e> (crlf))
  (modify 3 ^status done))

(p bucketjoin     ;; Put two puzzle pieces together
  (bucket ^p1 <x> ^p2 <y>  ^status done)
  (bucket ^p1 <y> ^p2 <z> ^shape <e> ^status undone)
-->
  (write Bucketjoin: join pieces <y> and <z> by shape <e> (crlf))
  (modify 2 ^status done))

;; create the puzzle pieces database
(make ppiece ^id  1 ^shape * ^match no ^side 1)
(make ppiece ^id  1 ^shape * ^match no ^side 2)
(make ppiece ^id  1 ^shape a ^match no ^side 3)
(make ppiece ^id  1 ^shape b ^match no ^side 4)
(make ppiece ^id  2 ^shape a ^match no ^side 5)
(make ppiece ^id  2 ^shape * ^match no ^side 6)
(make ppiece ^id  2 ^shape * ^match no ^side 7)
(make ppiece ^id  2 ^shape c ^match no ^side 8)
(make ppiece ^id  3 ^shape * ^match no ^side 9)
(make ppiece ^id  3 ^shape b ^match no ^side 10)
(make ppiece ^id  3 ^shape d ^match no ^side 11)
(make ppiece ^id  3 ^shape * ^match no ^side 12)
(make ppiece ^id  4 ^shape d ^match no ^side 13)
(make ppiece ^id  4 ^shape c ^match no ^side 14)
(make ppiece ^id  4 ^shape * ^match no ^side 15)
(make ppiece ^id  4 ^shape * ^match no ^side 16)
```

Figure 9: A rule program to solve puzzle problem:

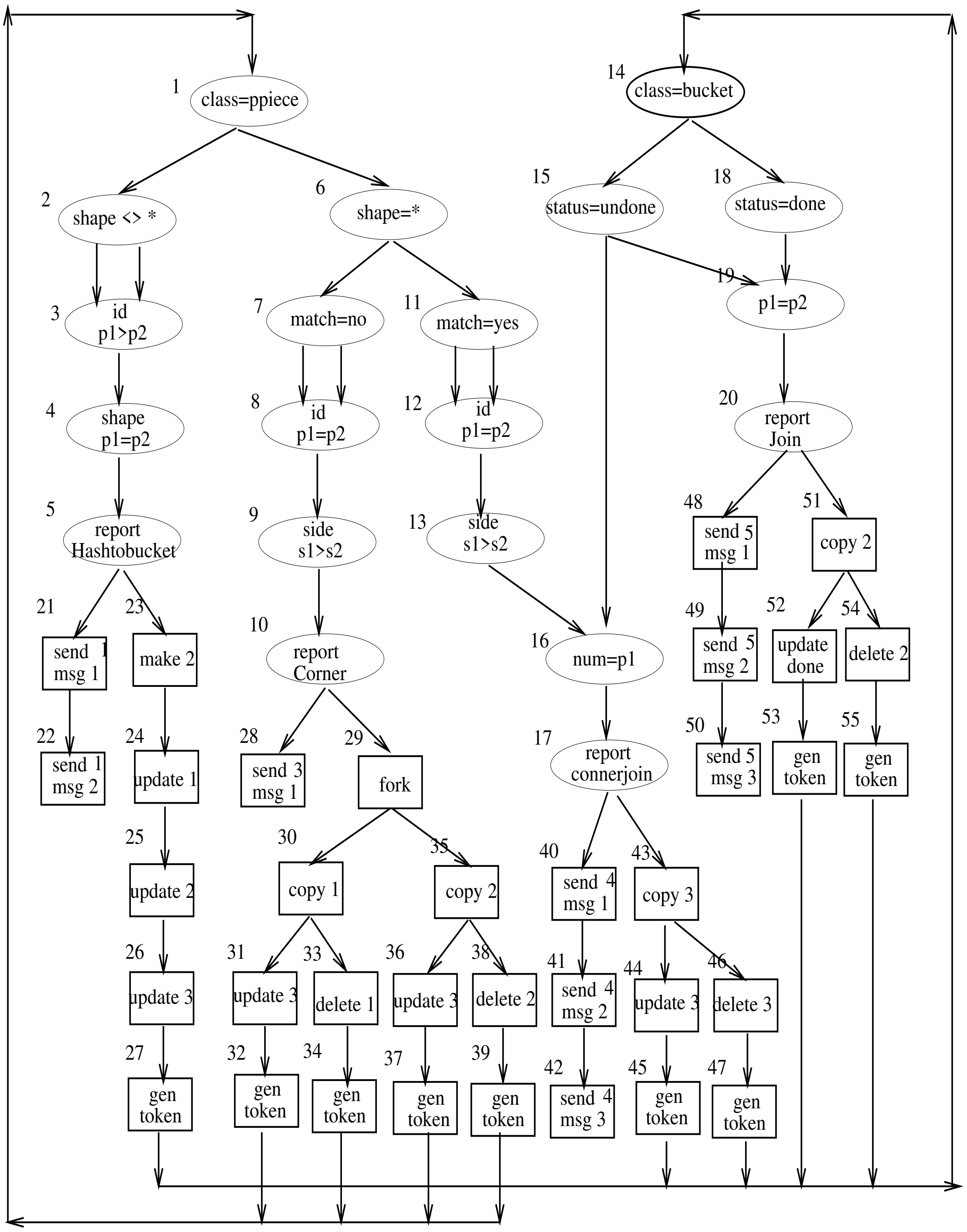

Figure 10: The corresponding data flow graph for the puzzle program:

Table 5: DFLOPS Instructions for the puzzle program

node id	opcode	RI1	FI1	RI2	FI2	d1	d2	fg 1	ptr 1	fg 2	ptr 2	comments
1	TEQA	1	0	0	0	2	6	N	0	N	0	C=PPIECE
2	TNEA	1	2	0	1	3	3	L	1	L	1	EDGE<>*
3	JEQA	1	2	1	2	4	0	N	0	N	0	P1EDGE=P2EDGE
4	JLTN	1	1	1	1	5	0	N	0	N	0	P1NUM¡P2NUM
5	TERM	0	1	0	0	21	23	N	0	N	0	R=HASHTOBUCKET
6	TEQA	1	2	0	1	7	11	N	0	N	0	P1EDGE=*
7	TEQA	1	3	0	2	8	8	L	3	L	3	MATCH=NO
8	JEQN	1	1	1	1	9	0	N	0	N	0	P1NUM=P2NUM
9	JGTN	1	4	1	4	10	0	N	0	N	0	P1SIDE>P2SIDE
10	TERM	0	2	0	0	28	29	N	0	N	0	R=CORNER
11	TEQA	1	3	0	3	12	12	L	5	L	5	MATCH=YES
12	JEQN	1	1	1	1	13	0	N	0	N	0	P1NUM=P2NUM
13	JGTN	1	4	1	4	16	0	L	6	N	0	P1SIDE>P2SIDE
14	TEQA	1	0	0	4	15	17	N	0	N	0	C=BUCKET
15	TEQA	1	4	0	5	16	18	R	1	L	7	STATUS=UNDOWN
16	JEQN	1	1	1	1	17	0	N	0	N	0	NUM=P1
17	TERM	0	3	0	0	40	43	N	0	N	0	R=CORNERJOIN
18	TEQA	1	4	0	6	19	0	R	2	N	0	STATUS=DOWN
19	JEQN	1	1	1	2	19	0	N	0	N	0	P1=P2
20	TERM	0	4	0	0	48	51	N	0	N	0	R=JOIN
21	SMSG	1	1	1	1	22	0	N	0	N	0	PRINT
22	SMSG	1	2	2	1	0	0	N	0	N	0	PRINT
23	MAKE	2	0	0	0	24	0	N	0	N	0	INSERTBUCKET
24	UPDT	1	1	1	1	25	0	N	0	N	0	INSERTBUCKET
25	UPDT	1	2	2	1	26	0	N	0	N	0	INSERTBUCKET
26	UPDT	1	3	1	2	27	0	N	0	N	0	INSERTBUCKET
27	GENT	1	0	0	0	13	0	N	0	N	0	GENTOKENBUCKET
28	SMSG	3	1	1	1	0	0	N	0	N	0	PRINT
29	FORK	0	0	0	0	30	35	N	0	N	0	FORK
30	COPY	0	0	1	0	31	33	N	0	N	0	INSERTBUCKET
31	UPDT	1	3	0	3	32	0	N	0	N	0	MATCH=YES
32	GENT	1	0	0	0	1	0	N	0	N	0	GENTOKENBUCKET
33	DELE	0	0	1	0	34	0	N	0	N	0	DELETE
34	GENT	1	0	0	0	1	0	N	0	N	0	GENTOKEN-
35	COPY	0	0	2	0	36	38	N	0	N	0	INSERTBUCKET
36	UPDT	1	3	0	3	37	0	N	0	N	0	MATCH=YES
37	GENT	1	0	0	0	1	0	N	0	N	0	GENTOKENBUCKET
38	DELE	0	0	2	0	39	0	N	0	N	0	DELETE
39	GENT	1	0	0	0	1	0	N	0	N	0	GENTOKEN-
40	SMSG	4	1	3	2	41	0	N	0	N	0	PRINT
41	SMSG	4	2	1	1	42	0	N	0	N	0	PRINT
42	SMSG	4	3	3	3	0	0	N	0	N	0	PRINT
43	COPY	0	0	3	0	44	46	N	0	N	0	INSERTBUCKET
44	UPDT	1	4	0	6	45	0	N	0	N	0	STATUS=DONE
45	GENT	1	0	0	0	13	0	N	0	N	0	GENTOKENBUCKET
46	DELE	0	0	3	0	47	0	N	0	N	0	DELETE
47	GENT	1	0	0	0	13	0	N	0	N	0	GENTOKEN-
48	SMSG	5	1	2	1	49	0	N	0	N	0	PRINT
49	SMSG	5	2	2	2	50	0	N	0	N	0	PRINT
50	SMSG	5	3	2	3	0	0	N	0	N	0	PRINT
51	COPY	0	0	1	0	52	54	N	0	N	0	INSERTBUCKET
52	UPDT	1	4	0	6	53	0	N	0	N	0	STATUS=DONE
53	GENT	1	0	0	0	13	0	N	0	N	0	GENTOKENBUCKET
54	DELE	0	0	1	0	55	0	N	0	N	0	DELETE
55	GENT	1	0	0	0	13	0	N	0	N	0	GENTOKEN-

Table 6: LHS constant file and RHS template file for the puzzle program:

LHS Constant File		**RHS Template File**		
index	constant	index	# of variables	RHS actions
0	ppiece			
1	*	1	2	HASH Put piece %s and %s in the BUCKET
2	no	2	3	bucket %s %s %s undone
3	yes	3	1	CORNER Piece %s is a corner. Put it on puzzle
4	bucket	4	3	Corner Assemble pieces %s and %s by edge %s
5	undone	5	3	Border Assemble pieces %s and %s by edge %s
6	done			

Table 7: The token file and the data file of the puzzle program:

	Token File				**Data File**				
index	IP	action	TT1	TT2	class	id	shape	match	side
1	1	+	1	0	ppiece	1	*	no	1
2	1	+	2	0	ppiece	1	*	no	2
3	1	+	3	0	ppiece	1	a	no	3
4	1	+	4	0	ppiece	1	b	no	4
5	1	+	5	0	ppiece	2	a	no	5
6	1	+	6	0	ppiece	2	*	no	6
7	1	+	7	0	ppiece	2	*	no	7
8	1	+	8	0	ppiece	2	c	no	8
9	1	+	9	0	ppiece	3	*	no	9
10	1	+	10	0	ppiece	3	b	no	10
11	1	+	11	0	ppiece	3	d	no	11
12	1	+	12	0	ppiece	3	*	no	12
13	1	+	13	0	ppiece	4	d	no	13
14	1	+	14	0	ppiece	4	c	no	14
15	1	+	15	0	ppiece	4	*	no	15
16	1	+	16	0	ppiece	4	*	no	16