TR-035

Parallel Prolog Machine Based on the Data Flow Model

by
Noriyoshi Ito, Kanae Masuda
and Hajime Shimizu

August, 1983

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191 - 5
Telex ICOT J32964

Institute for New Generation Computer Technology

Parallel Prolog Machine Based on the Data Flow Model

Noriyoshi Ito, Kanae Masuda, Hajime Shimizu

Institute for New Generation Computer Technology

1. INTRODUCTION

Equipped with basic functions such as pattern matching and non-deterministic control, the logic programming language like Prolog (Programming in Logic) [10] seems suitable for knowledge information processing system. For such inference-based processing systems, generally, a sequence of processing is not determined in advance; such systems, therefore, require a function to carry on processing, while heuristically seeking appropriate procedures. This function is inherent in Prolog.

Prolog provides the potential for parallel processing [6]. Conventional sequential processing systems are based on the depth-first search approach, or sequentially controlled non-deterministic search approach with backtracking. By contrast, the breadth-first search approach, simultaneously trying multiple possible searches, can solve problems faster by initiating a number of searching processes on multiple processors.

The data flow model can naturally implement parallel processing [1] [2] [8]. The execution mechanism of the data flow model is closely related to that of Prolog. This paper describes a machine architecture based on the data flow model to implement parallel execution of Prolog. First, a Prolog's parallel execution model is discussed in section 2. Section 3 describes the abstract architecture of the machine. Section 4 and 5 explain the control mechanisms of unification processes and structured data, respectively; both are basically required to implement the parallel execution of Prolog. Finally, an architecture of a parallel Prolog machine is outlined in section 6.

2. PARALLEL UNIFICATION MODEL

Unification, or pattern matching, is a basic function of Prolog. A Prolog program is executed by repeatedly performing unifications between goal statements which correspond to queries and a set of clauses which correspond to given knowledge.

Each clause consists of a head and a body as following:

P <- Q1 & Q2 & ... & Qn. (n>=0)

where P denotes a head literal, Qi denotes a body literal, and the symbol '<-' means implication.

When clauses are restricted to Horn clauses as in Prolog, the head consists of at most one literal. A clause without head is called a goal statement. The body is also optional; a clause without its body is called an unit clause. Multiple literals in the body, if so, are connected with each other through ANDs.

When a goal statements has multiple literals, these literals are ANDed. The unification for the whole goal statement, therefore, does not succeed unless it succeed for all literals in the goal statement. That is, the goal statement cannot be successfully solved unless all the literals can be solved.

As for each literal in the goal statement, clauses which are potentially unifiable with the goal literal must have a head literal which has the same predicate as the goal literal. A subset of such clauses is referred as the definition of that predicate. A set of clauses constituting a definition are ORed. In other words, a goal literal can be solved if successfully unified with at least one clause.

When a goal literal in a goal statement is given, the primitive unification operation is executed by invoking one of clauses in the definition of the goal literal's predicate, and by unifying (i.e. pattern-matching) between the goal literal and the head literal of the selected clause. This operation will

produce a common instance of two literals, if the pattern-matching has succeeded; or a 'fail' signal, otherwise.

When a given goal literal is successfully unified with the head literal of a non-unit clause, then another unification is initiated taking its body as a new goal statement. On the other hand, the unification operation terminates (i.e. the solution is obtained) when the goal literal is successfully unified with the head literal of an unit clause; the result is returned to the parent process which has called the unification process.

In Prolog, parallelism concerning the unification can be divided into three types:

- OR parallelism
- AND parallelism
- parallelism among arguments

They are described below.

2.1 OR Parallelism

Given a goal literal and the definition of its predicate, unification between the goal and the definition can be performed in parallel on all clauses in the definition.

A sequential Prolog interpreter uses backtracking to control this unification. In this system, the order for calling clauses in a definition has previously been determined. When a goal literal is given, the first clause is invoked according to the order and unified with the goal literal. When the unification fails (i.e. no solution exists), the backtracking mechanism invokes the clause in the next order and unification is retried on this clause. On the other hand, our parallel Prolog machine aims at high-performance inferrence processing by performing unification in parallel on ORed clauses in the

definition.

This OR parallel execution can be performed simply by simultaneously initiating unification for individual clauses in the definition of goal predicate. Then, the successful unification results (i.e. solutions), are merged to form an output. When only a single solution is required, one of the successful solution, perhaps first obtained, can be output. This don't care non-determinism can be implemented by guarded clauses mechanism described later. The merge operation generally outputs solutions in a non-deterministic order; it may output them in the order obtained. Our machine will employ structured data called stream which plays a role of a "pipe" through which merged solutions are delivered [3] [11].

The stream is a non-strict structured data and provides a mean of asynchronous communications between producer processes, which generate the elements of a stream, and consumer processes, which refer these elements. For our machine, producer processes correspond to unification processes operating on an OR-parallel basis and generating merged solutions, while consumer processes correspond to other unification processes which input the merged solusions. The stream can be empty, when all the OR-parallel unifications of the goal literal have failed.

2.2 AND Parallelism

ANDed literals in a goal statement can be solved in parallel. This AND parallelism, however, involves the following problem. When goal literals being executed on an AND-parallel-basis have shared variables, the solutions for these variables must be consistent. In this consistency checking, another unification operation, such as a join operation in a data base system, must be performed on sets of the solutions for the literals with shared variables. When these sets of solutions are big, then the consistency checking operation tends to require longer overhead and larger amount of resourses. For this reason, we will introduce AND parallelism making use of the pipeline effect in our machine as

described below. Two approaches are possible for AND pipeline execution:

- Pipeline execution of all goal literals,

- Parallel execution of goal literals without shared variables.

These approaches are described below.

(1) Pipeline Execution of All Goal Literals

When a given goal statement has multiple literals, this approach determines the execution order of these literals according to certain rules. The order must be determined uniquely by, for example, specifying input/output relation of variables like in the Relational Language by K.L. Clark and S. Gregory [4], or by selecting literals from left to right.

With an uniquely determined execution order, a goal statement can be solved by transferring unifiers, which are the lists of the variables and its instances (i.e. the substitution information of the variables) in the goal literal, among the literals according to the execution order, and by initiating unifications of each literal when the unifiers have arrived. Pipeline processing can be implemented by connecting these literals with streams in the execution order.

Assume that a goal statement given has multiple literals, and unifications between a goal literal and clauses of its definition are executed in parallel, and they create a stream of unifiers for variables in the goal literal. In this case, the unification of each goal literal will be processed in the following order:

(a) When an unification on the previous goal literal succeeds, the unifiers are sent along the stream, and the next goal literal fetches the unifiers from the stream.

(b) According to the unifiers, the substitution is performed on the variables in the literal concerned. Using the substituted literal as a new goal literal, the next unifications are carried out between the new literal and the clauses of its definition. The results construct another stream, and are sent to the succeeding literal. If there is no succeeding literal, the new unifiers obtained satisfy the goal statement.

(2) Parallel Execution of Goal Literals without Shared Variables

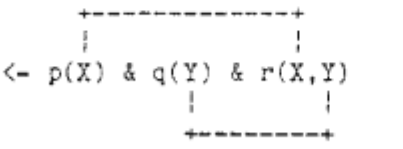
Literals without shared variables do not have to solved on a pipeline basis; the unifications of them can be executed in parallel. This approach is based on this idea to improve parallelism, and also to avoid the duplicated execution of same unifications in vain.

Suppose that a goal statement

<- p(X) & q(Y) & r(X,Y).

is given. Where p, q, and r are predicates, and X and Y are variables. As the literals p(X) and q(Y) have no shared variable each other, they can be solved independently. If a sequential execution method of AND literals, as described in (1) above, is used, and if the p(X) produces multiple instances of variable X, then the same unification of the literal q(Y) must be applied for all the results of the unification of literal p(X). In this approach, however, the unification of q(Y) may be applied only once, and is initiated not waiting for the results of the unification of p(X).

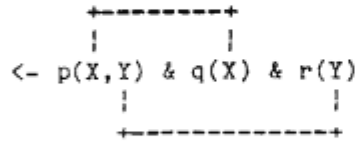
In the above goal, p(X) and q(Y) become the producers of the instances for variables X and Y, respectively, and r(X,Y) becomes a consumer of these streams. This will be represented as following graph:



The unification process of r(X,Y) inputs two streams for variables X and Y, and

performs unification on all the combinations of instances of them. This can be easily be achieved by recursive calling of the litaral p(X,Y) by unfolding both streams to their elements.

In this approach, however, involves some problems. Suppose that another goal statement below is given:



In this case, when one of solutions to the goal literal p(X,Y) is obtained, the instances for variables X and Y are sent to q(X) and r(Y), respectively. Then, q(X) and r(Y) will be solved independently.

However, if the instances of X and Y are non-ground terms (i.e. if they include some unbound variables before calling p(X,Y)), and if an unit clause below is given as the definition of p:

p(Z,Z) <-

where the head literal has multiple occurrence of the variable Z. Then the returned instances from p(Z,Z) may also be non-ground term, and be bound to the same unbound variable Z.

This means that, when execution of q(X) causes X to be bound some instances, execution of r(Y) is affected to it (i.e. involves side effect operation bteween q(X) and r(Y)).

It is difficult to detect such a side effect unification and to determine the execution order of literals at compile time. Therefore, operators, those that detect whether the instances from the literal p(X,Y) include the shared unbound variables or not, and those that control dynamically the stream flows in the goal statement, will be necessary.

By default, which selects the execution order of literals with shared variables from the left to the right, the literal q(X) will produce the instances of shared unbound variable Z, and r(Y) will consume them as follows:

```
          +---------+
          |         |
<- p(X,Y) & q(X) & r(Y)
             |      |
             +------+
```

The unification procedure of a clause where its head literal has multiple occurrences of same variable, like p(Z,Z), is represented by the following data flow graph:

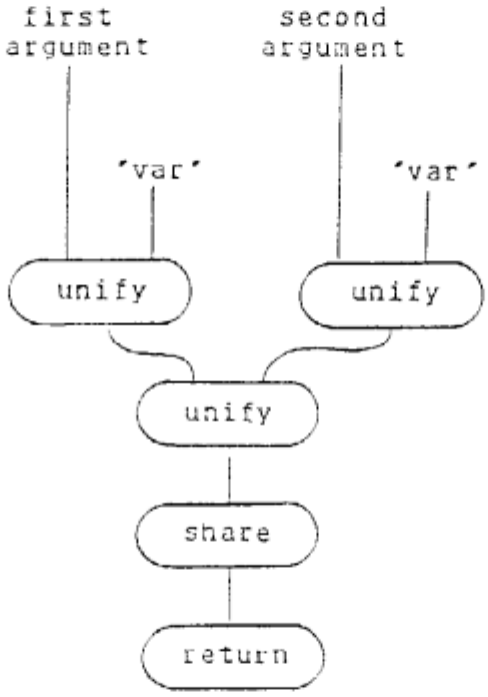


Figure 2.1 unification procedure of clause p(Z,Z) <-

As an unbound variable can be unified with any form of term, the unify operators between two input arguments and the unbound variable Z can be omitted. In order to guarantee the identity of the first and second argument of predicate p in the above clause, the independent instances of their unifications must be unified again. This unification will produce an instance of variable Z, which is sent the body of the clause, if the body exists.

After the body unification has completed, the share operator checks the final instance of Z whether it is a ground term or not. In order to perform this check operation faster, the data type field of the instance has an unbound flag, which shows whether the instance is an unbound variable or the instance has any unbound variable as its substructure. So the structure construction operator, which is used when constructing a new structured data from existing data, will check all the substructure's unbound flags, and will produce a new data structure with the unbound flag depending on them; if any of them is on, the new flag is set on, otherwise, the new flag is set off. The share operator executes the following procedure:

(a)If the argument's unbound flag of the share operator is off, the operator returns the argument itself.

(b)If the flag is on, and if the argument is an unbound variable, the operator returns a new shared unbound variable, which has an unique shared unbound variable name in its data part.

(c)If the flag is on, and if the argument is a shared unbound variable, the operator returns the argument itself.

(d) If the flag is on, and if the argument is a structure, the operator recursively calls the share operations to all its substructures, and returns a new structure constructed by their results.

2.3 Parallelism among Arguments

When a goal literal and one of head literal of its unifiable clause consist of multiple arguments, the unifications between the arguments in the goal literal and the corresponding position's arguments in the clause head literal can be performed independently by representing the unification procedure by data flow graph. When an argument of the goal literal and the corresponding

position's argument of the head litaral are structure, then the parallel unification of their substructures can also be implemented.

In this case, all the independent unification results must be checked for consistency, that is, the literal unification succeed only if all the argument unifications have successfully completed. The reqired consistency checking enables the parallelism among arguments to be considered a variant of AND parallelism.

If the goal literal has shared unbound variables as its arguments, the consistency checking must guarantee that the same shared variables are bound to the same instances. Here, we will show some example. Suppose that a goal statement

```
<- p(Z,Z) & ...
```

is given, and also a clause of definition p

```
p(f(U),g(V)) <- ...
```

is given. Then unification process of this clause must know that its input arguments have some shared unbound variables each other. We will use a share operator, like just described in (2), to change the unbound variable Z to the shared unbound variable, before calling the definition of p.

The unify operation of the clause can be represented by the data flow graph:

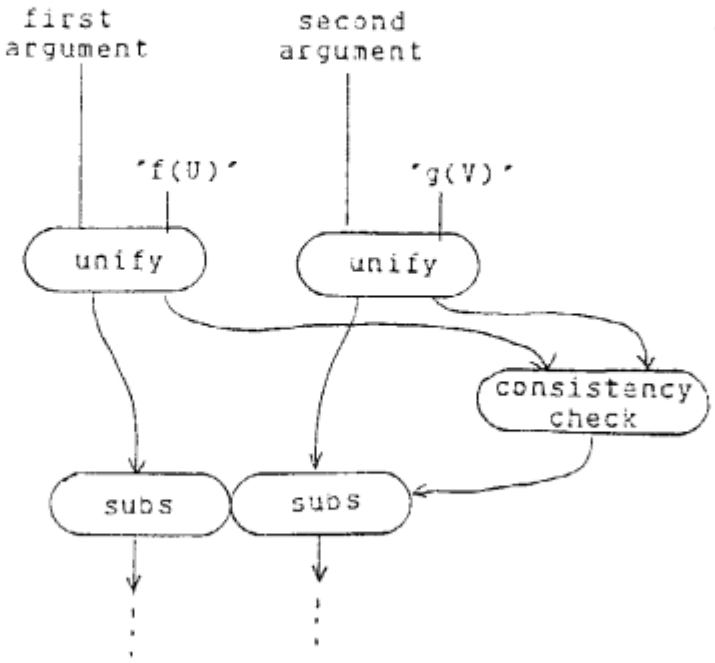


Figure 2.2 Data Flow Graph of the Clause p(f(U),g(V)) <- ...

The graph shows that an unify operator of each argument has two outputs; one is an instance between an argument of goal literal and its corresponding position's argument of the head literal, and other is a binding environment of the unbound variables included in the goal argument. The binding environments, which consist lists of the unifiers of the shared unbound variables (i.e. the shared unbound variable names and their instances), are sent to a set of consistency check operators. A consistency check operator receives a pair of the binding environments, and constructs a new binding environment as following:

(a) If one of the input environments is 'fail' at least (i.e. if its unification has failed), then outputs 'fail' as a new environment.

(b) If one of the input environments is 'nil' (i.e. if its goal argument has no shrared variable), then outputs another input environment as a new environment.

(c) If both environments are lists, then checks consistency among two environments; it get a first unifier from the first emvironment, and search associatively whether the second environment has the same shared unbound variable as the one in the first unifier; if the associative seaching succeeds, the two instances in the first unifier and the unifier in the second environment are checked for consistency, i.e. these two instances are unified; this unification will produce 'fail' if the the unification has failed, or a new instance for the shared unbound variable if the unification has succeeded.

The above consistency check operation is executed for all the unifiers in the first environment, and will produce a new environment for all the shared variables included in both of the input environments. This enviroment is to be a most general unifiers of the shared variables.

On the other hand, an unify operator of the arguments will execute the following procedure; it tests the goal argument's data type whether the argument has any shared unboud variable or not. The data type field has a shared flag, which indicates that the data has any shared unbound variable as its substructure, just like the unbound flag described above. If the flag is off, the operator produces the two outputs: one is an instance between the goal literal's argument and the head literal's argument, and other is a 'nil' value which shows a binding environment is empty. If the shared falg is on, the operator produces the two results as following:

(a) If the goal literal's argument is a shared unbound variable, it produces the shared unbound variable as the instance, and a new structure cell address as its unifier. This cell is used to store the shared variable and the contents of the head argument, which is an instance of the variable.

(b) If the goal literal's argument is a structured data, it decomposes the structure to the substructures, executes unifications upon these substructures by recursively calling the unify operators, and constructs two structures from these unification results: one is a construction of their substructure's instances, and other is a new environments.

If a goal literal has the shared unbound variables, the instance of the unification between the goal literal and head literal includes the shared unbound variables themselves. The instance of these variables may be gotten from the binding environment, by searching again associatively from the environment with the shared variable names as the keys. This operation may be done by the subs (abbreviation of substitute) operator in the above graph.

3. ABSTRACT MACHINE ARCHITECTURE

The machine is constructed by multiple processing elements with multiple shared structure memories. They can operate independently each other, and connected by asynchronous communication networks. A unification procedure represented by a data flow graph is loaded in one of the processing elements, and executed. The processing element detects the executable instructions in the graph, and interpretes them in parallel. If multiple procedure invocations are issued, these procedure invocations may be distibuted among processing elements. Each processing element can execute multiple instances of the procedures when so many procedure are activated. These instances are distinguished by the processing element number and their process identifiers.

The structured data is stored in the structure memories, which is accessible from the processing elements. The structured data, therefore, is represented by a pointer to the structure memory, and can be shared by the multiple unification processes. In this section, we will show the data types and the execution modes of the machine.

3.1 Data Types and Basic Unification Primitives

The data types implemented on our machine include symbols, integers, variables, lists, vectors, streams, and strings. Of these, structured data - lists, vectors, streams, and strings - is represented as a pointer to structure memory. To improve unification performance, our machine employs a tagged architecture, and represents data using a tag field showing its data type and a value field.

The basic unification primitive, 'term-unify', is used in the unification between one argument in a goal literal and the corresponding position's argument in the head literal of its definition. This primitive can be represented by a data flow graph in Figure 3.1. As the figure shows, this primitive inputs a pair of arguments and, when one argument is a variable, returns the other. When both arguments are atoms (symbols or integers), it returns the unification result of two arguments. (When they are same atom, it returns the atom itself; otherwise, it returns the special atom 'fail', which means the unification has failed.) When both are lists or vectors, it calls the list- or vector-unify primitive, respectively, which, in turn recursively calls the term-unify. In other cases, the term-unify returns 'fail'.

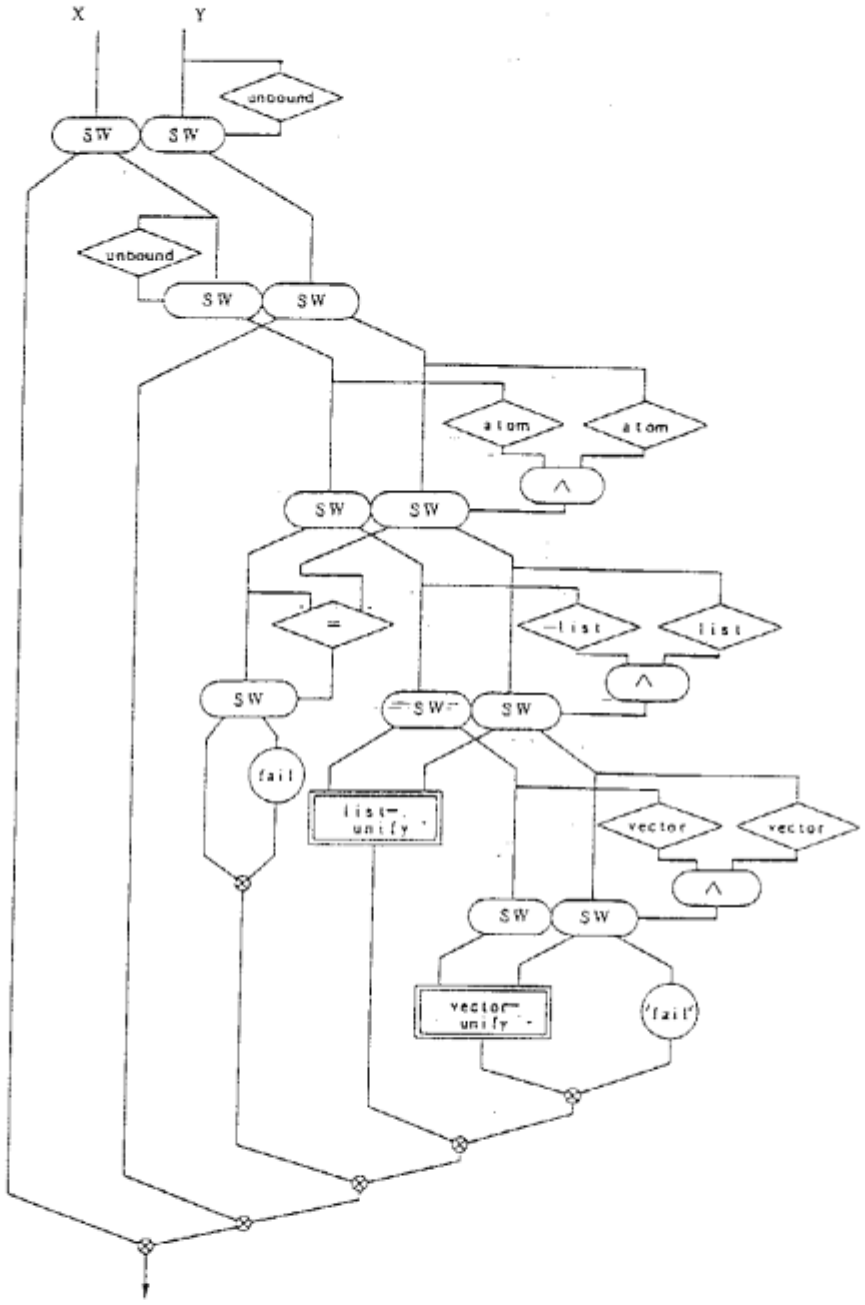


Figure 3.1 Data Flow Graph of 'term-unify' primitive

Another basic primitive is a merge operation of a non-deterministic stream. In order to implement a stream as an arbitrary incremental structure, a stream body is represented by a list type data structure; it has a first part where a stream element is stored, and rest part where a pointer of rests of the stream is stored. This merge operation can be implemented as follows. Before invoking

clauses of the goal literal's definition, a create-stream operator is executed; this oparator returns two outputs: one is a pointer of a stream head cell, and other is a pointer of a stream tail pointer cell. The stream head cell is a beginning of the stream, and its pointer is sent to the consumer processes, which can get stream elements from the pointer. The stream tail pointer cell is initialized to point to the stream head cell, and is shared by the invoked OR-parallel processes. Each OR process, if successfully terminated, appends a new solusion to the end of stream by updating the stream tail pointer to point to the new stream end. This append-stream operation executes the following cycles; it allocates a new stream body cell, read the contents of the stream tail pointer, updates it to point to the new stream body cell, writes a solusion to the first part of the body cell, and finally writes the pointer of the body cell to the rest part of an old stream tail cell. As the stream tail pointer cell is shared by the OR processes, the append-stream operator must lock the cell against other append-stream operators while updating the contents of the cell.

An append-stream operator of a failed OR process does no operation, and only decrements the reference count of the stream tail pointer cell. When all the OR processes of the stream have terminated, i.e. when the reference count of the stream tail pointer cell has reached to zero, the final operator writes 'fail', which is used for a symbol of end-of-stream, to the rest part of the stream tail.

3.2 Prolog Execution modes

Prolog program is executed in either compiler or interpreter modes. These modes differ only in the level used as the internal representation of program. In other words, in compiler mode, a Prolog program is converted into a data flow graph, the equivalent of machine language of our machine, before being executed, while in interpreter mode, it is converted into an intermediate code like vector, which is then interpreted by an interpreter written in a data flow

graph.

In general, the processing in the compiler mode requires rather complex compiler, but has an advantage of faster execution speed. By contrast, the interpreter mode is suitable for interactive operating environments where processing is carried on while dynamically updating the program. In this mode, as the program may be represented by structured data like vector, generation or update of intermediate code is relatively easy. Due to frequent accesses to structure memory and code interpretation overhead at runtime, however, the execution speed is slower.

Figure 3.2 shows an example of compiled code of a Prolog program represented by data flow graph. This example program is a definition of append as follows:

```
append([],X,X) <- .
append([H|X],Y,[H|Z]) <- append(X,Y,Z).
```

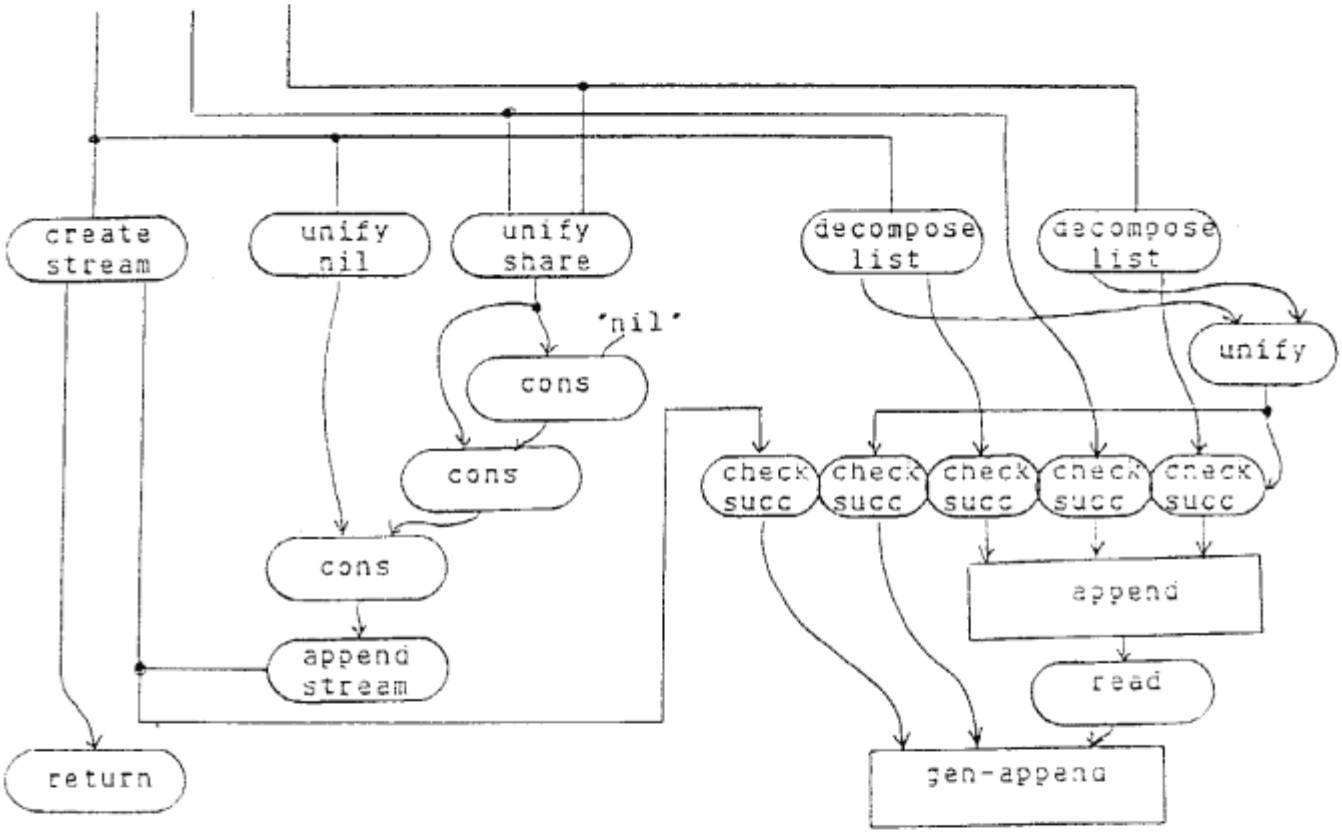
In this figure, a rectangular block means a procedure invocation, and the above-described 'term-unify' can be replaced with more efficient primitives, such as unify-with-nil, which unifies the input with nil, or decompose-list operator, which unifies the input argument with list and, if succeeded, decomposes it to left part and right part.

Figure 3.2 (a) is a graph of append where the first and second clauses are executed in parallel, and Figure 3.2 (b) is a graph of gen-append, which is invoked in the second clause of the definition. The second clause calls append recursively, which will generate a stream. The gen-append graph unfolds this stream into elements, and generates a new stream.

After the uifications of the head literal's arguments have completed, their results are tested whether all the unifications have succeeded or not. In the first clause of append definition, which is an unit clause, this operation is executed by cons operators, which test its inputs and, if all of them are not 'fail', generate a construction of them; otherwise, generates a 'fail'. The

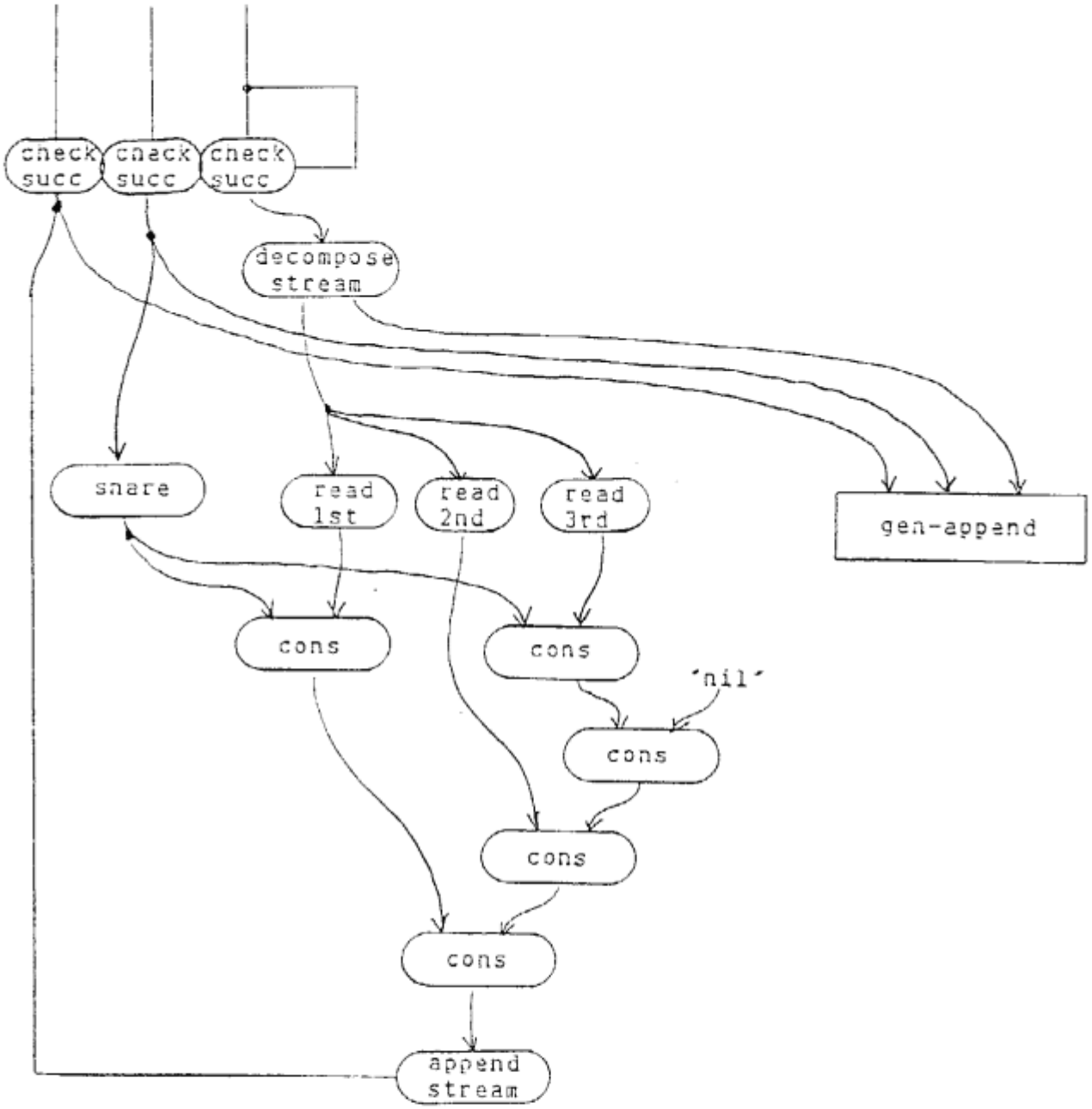
output of cons operators initiates an append-stream operator, described above, which in turn appends the result to the end of stream.

In the second clause, which is a non-unit clause, the head literal's unification results are passed to its body if all the argument unifications have succeeded. A check-succ operator passes its top input to the output port, when its right input is not 'fail'.



(a) Data Flow Graph of append

Figure 3.2 Data Flow Graph Example



(b) Data Flow Graph of gen-append

Figure 3.2 Data Flow Graph Example (cont.)

4. CONTROL OF PARALLEL PROCESSES

In the parallel execution environment of Prolog, multiple unification processes are simultaneously executed. This requires the process control function described below.

4.1 Sharing of Procedure Codes

To implement a recursive call in Prolog, procedure codes have to be shared either by copying the original codes or by assigning a different process identifier to the procedure invocation. Dynamic management of code memory allocation and code copying in the former method are generally thought to have longer overhead than management of process identifiers in the latter method. We, therefore, have chosen the latter method (colored token method) for our machine.

4.2 Control of the Number of Active Processes

As described earlier, there are two possible non-deterministic search approach: depth-first and breadth-first. Breadth-first search approach simultaneously tries multiple solutions, which may drastically increase the number of active processes. For an actual limited resouce system, this may cause a deadlock status because of explossive resouce exhaustion.

The system, therefore, has to be equipped with a mechanism to automatically control the number of active processes. Two mechanism are possible:

- Mechanism to restrict stream length,
- Mechanism to control process priority.

(1) Mechanism to Restrict Stream Length

This method prevents resources from being explosively consumed by restricting the length of a stream. For example, assume a stream with a maximum length of N. The stream length can be restricted by initiating simultaneously up to N processes of OR processes which output their results as elements of the stream and by suspending execution of the remaining processes.

(2) Mechanism to Control Process Priority

This method prevents the number of active processes from drastically increasing by assigning an appropriate priority to each process and controlling its initiation according to the priority.

In general, process creation can be controlled with process queues arranged by their priorities. This priority-based control is performed in the following way: first, a process management table like Figure 4.1 is established and, when a process invocation request (procedure call instruction) is issued, the request is chained in the queue with the corresponding priority on the table; meanwhile, the dispatcher, which is a process manager in operating system, monitors the current state of every processing element and, having judged some processing element in idle state, fetchs the request from the queue with the highest priority and allocates that element to it.

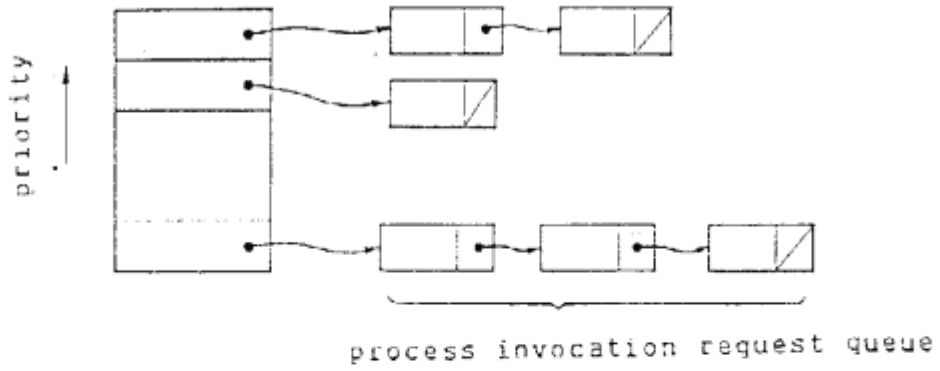


Figure 4.1 Process Management Table

The execution of a Prolog program can be represented by an AND-OR search tree shown in Figure 4.2. For an AND branch, a rectangular box in the figure, the solution cannot be obtained until all its successive AND processes are solved. Therefore, the same priority may be assigned to these AND processes.

On the other hand, an OR branch can be solved if any of the successive OR processes has successfully terminated; thus, some of its processes may be lasy evaluated. For example, by assigning higher priorities to left OR processes than its right, the right processes may be invoked only if the processing elements are relatively idle, otherwise their invocation may be delayed until higher processes terminate. This approach can prevent an explosive increase of OR processes.

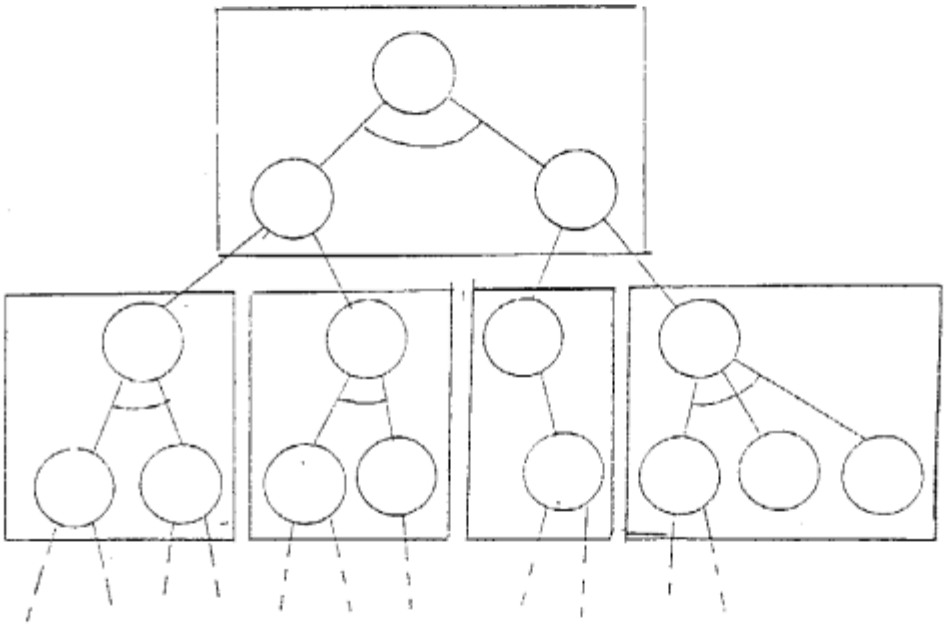


Figure 4.2 AND-OR Searching Tree

4.3 Termination Control of Processes

At the point of view of OR parallelism, our machine can be regarded as "eager evaluator", which execute OR candidate processes in parallel. It tries to obtain solutions, not by activating OR processes in a sequential manner until an OR process terminates successfully, but by activating multiple unification processes in parallel, although some of process executions may be waste if only one solution is necessary.

In order to support guarded clauses like in Concurrent Prolog [11], which is used to dynamically reduce the search space, it is necessary to terminate OR processes running in parallel, for efficient use of resources.

A definition of guarded clauses may be represented as follows:

H1 <- G1 | B1

H2 <- G2 | B2

. . .

Hn <- Gn | Bn

where H1,H2,..., and Hn are head literals, G1,G2,..., and Gn are guard parts, B1,B2,..., and Bn are body part, and "|" is a guard bar. If one of unification of guard parts succeeds and control has passed to its body part across the guard bar, then the solution of this definition is a result of this body part. Other results are discarded.

THis guard mechanism can be implemented using semaphore shown in Figure 4.3.

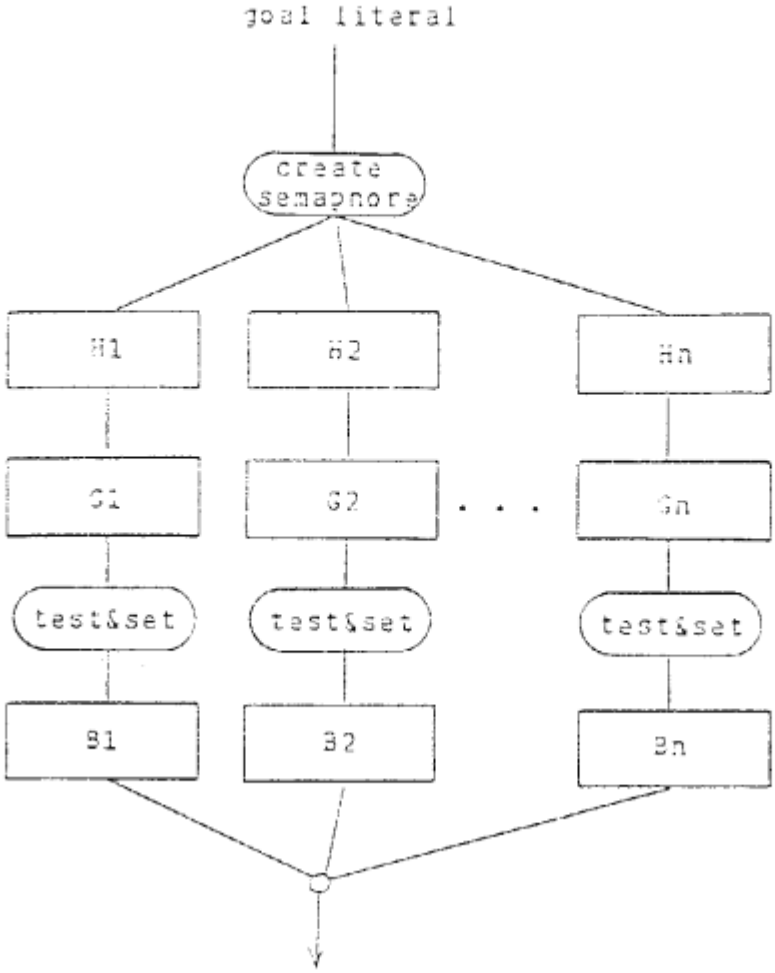


Figure 4.3 Unification of guarded clauses

The create semaphore operator allocates and initializes a memory cell shared by every guarded clauses. The test & set operator in each clause is executed when the unification of its guard part succeeds. It tests the flag in the allocated memory cell, and sets it on if the flag tested was off. While this opeartion is proceeding, other operator cannot interrupt to this operation. If the result of tested flag was off, the unification of its body part proceeds, otherwise, the token will be absorbed in this operator.

As soon as the first test & set operator is executed, the unification processes of other clauses may be forced to be terminated, instead of waiting for their terminations. We will provide token communication paths with filters which will absorb the tokens belonging to particular processes. The filter can be implemented using, for example, associative memory. Whether to absorb the

token can be determined storing process identifiers for tokens to be terminated in the associative memory and by searching associatively using token's process identifier as a key when a token arrived to the filter.

When the result of test & set operator above was off, it broardcasts all the process identifiers of its brothers and their decendants to all the filters.

4.4 Process Allocation Control

Another problem concerning process control is how to allocate resources to activated processes. This resource management problem is closely related to the network structure of the system. Ideally, the network structure should be capable of not only minimizing overhead for the resouce management and interprocess communications but also of permitting system load to be distributed evenly throughout the system.

Generally, it is very difficult to statically determine resource allocation at compile time in the applications such as inference system, where dynamic program behavior is not clear. The machine, therefore, may have to dynamically control resource allocation, while examining the following points; how to manage communication costs and load state of processing elements and how to control program loading.

(1) Management of Communication Costs and Load State

Resource management can be simplified by adopting a network topology in which communication distances among any pair of processing elements are same, because, with such a network, resource manager can allocate a relatively idle processing element to a new process without considering communication costs. One possible network structure of this topology is a multi-stage network. With this network, however, the more the number of processing elements in the system increases, the more the inter-element communication costs becomes expensive.

On the other hand, when there is some locality in inter-element communications, the communication costs can be lowered by allocating closely-related processes to neighbor processing elements. Compared with the network described above, this network requires rather complex resouce manager to balance the load. Load balancing could be achieved, for example, by distributing resource manager to each processing element and by communicating load status among neighbor local managers so that processing load can be evenly distributed among these processing elements, which may approximately disribute the processing load to all the processing elements.

Our machine, therefore, will use a network topology with local communications among processing elements.

(2) Program Loading Control

Copy method which broadcasts the entire program to all processing elements at initialization could eliminate dynamic program loading; process allocation could be controlled by simply considering inter-process communication costs and load balance among the processing elements. For large-scale application, however, this requires large-capacity program memory in each processing element; thus, hardware costs would become prohivitive in large-scale multi-processor system.

Programs, therefore, will have to be distributed among processing elements. This requires dynamic program loading control, since, as described above, static program allocation is generally difficult.

In the dynamic program loading, program loading overhead can be reduced by, for example, introducing following approaches:

(a) When a new process is initiated, allocation of the process to a processing element which has already the process's program codes.

(b) distinguishing the program loading unit from the process's program code unit.

In approach (a), process allocation will be controlled by balancing the inter-process communication costs and load concentration to the particular processing elements against program loading overhead.

The approach (b) involves loading a set of relatively-closely-related procedures in a processing element (or in a set of processing elements) in one loading time.

The allocation of new process may cause a swapping out of another processes. This swapping mechanism requires a dynamic relocation function of programs, which translates a logical address to the pysical address of program memory in the processing element.

5. STRUCTURED DATA CONTROL

Many data manipulated in the Prolog application has a complex data structure. The key point of a Prolog machine design will be efficient processing of structured data.

Generally, the copy method, in which the structured data is copied to all the processes referencing that data, enables the subsequent processes to be executed independently of each other. For handling complex and large structured data, however, this method suffers from large overhead caused by data copying.

Therefore, a function which can share structured data among processes is required. In a system with this function, processes will be able to share the same structured data by communicating pointers to the structure memory where the structured data is stored.

A system with shared structured data will require a control function capable of distibuting the structured data among multiple structure memories to avoid access concentration. For example, assume that multiple processes will simultaneously traverse the shared structured data. In this case, simultaneous accesses to the different portions in the same structured data and distribution of structure memory load can be achieved by mapping the logical structured data over the multiple structure memories.

As described above, efficient structured data manipulation can be achieved by two control functions to share and distribute the structured data.

5.1 Sharing Control of Structured Data

When sharing structured data in a parallel processing system, such as a data flow machine where multiple processes are running independently of each other and activities (instructions and tokens) are distributed over various portions in the machine, the problem involved is how to perform memory garbage collection.

A method proposed to solve this problem sets up a reference count in each memory cell [6]. This reference count method stores the number of activities referencing a memory cell in its reference count memory, and, when the number reaches zero by termination of an activity, makes the cell reusable as a garbage cell.

This method, however, requires reference counts to be updated not only when structured data is manupulated but also when a process is invoked or a conditinal branch occurs. Therefore, we must evaluate the overhead of reference count manipulation.

5.2 Distribution Control of Structured Data

Sharing structured data among several processing elements will be accompanied by the problem of structure memory access concentration described above.

Structure memory distribution gives rise to the problem of how to map a logical structured data on physical memory banks. For example, suppose that multiple processing elements simultaneously try to access a single data structure. If the structure is stored over several banks, access contension will be reduced and structure manipulation load will be evenly distributed among these memory banks.

Meanwhile, structure traversing operations, such as updating control of reference count or some sort structure traversing like a share operator described above, require transfer control of operations between memory banks, if the structure mapped over several banks. In this type of operations, execution can be done efficiently by introducing a network that interconnects memory banks with a local topology, and by carrying out the structured data mapping to exploit the locality.

When a two-dimensional mesh structure is used as this network, structured data like a frequently-used tree structure could be mapped as shown in Figure 5.1, where circles and squares denote memory banks and memory cells, respectively. Notice that unbalanced load could be reduced by introducing more flexible memory cell allocation strategy than the one shown in the figure. For example, a strategy which allocates the subtrees to memory banks located at one or less distance from its root node can be used.

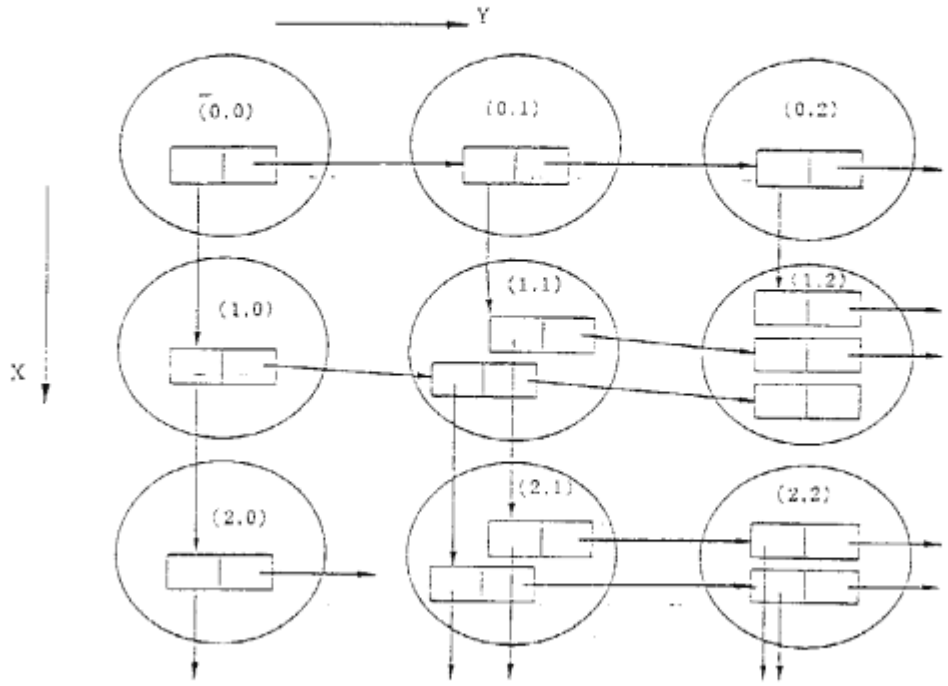


Figure 5.1 Distribution of Structured Data

6. MACHINE ARCHITECTURE

Our machine, constructed on a basis of multiprocessors with shared structure memory banks, consists of Processing Element Modules (PEMs), Structure Memory Modules (SMMs), and three types of networks connecting these modules (Figure 6.1).

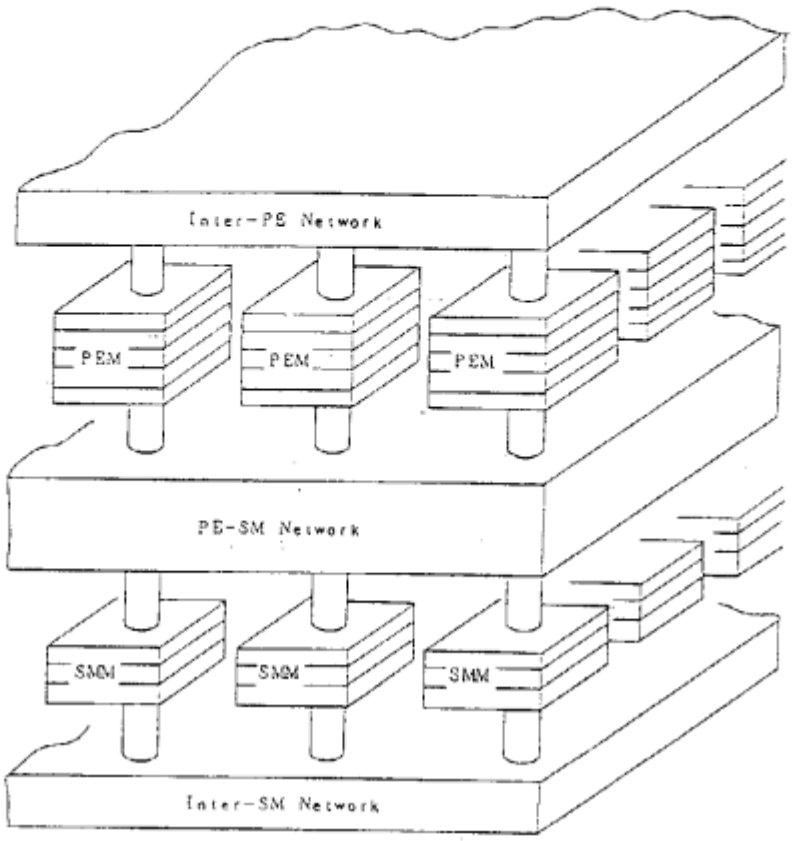


Figure 6.1 Machine Configuration

Each module can operate independently. Spatial parallelism (parallelism due to simultaneous execution) can be achieved by distributing a set of active processes (activities) among modules. Since asynchronous communications among these modules and among functional blocks in a module are adopted, this machine can make use of pipeline parallelism.

The basic structure of the modules are discussed below.

6.1 Processing Element Module (PEM)

PEM interprets the unification procedure and controls instruction execution, procedure calling, and basic pattern matching operation, as well as executes some built-in predicates. It is further divided into two submodules forming a circular pipeline structure shown in Figure 6.2.

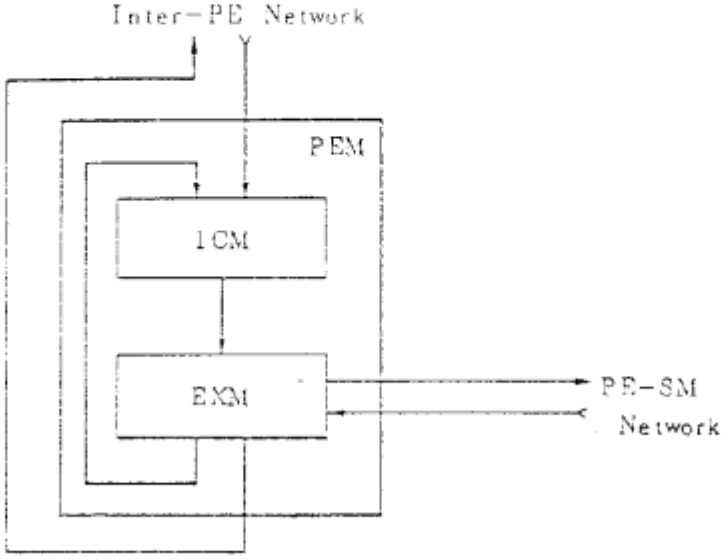
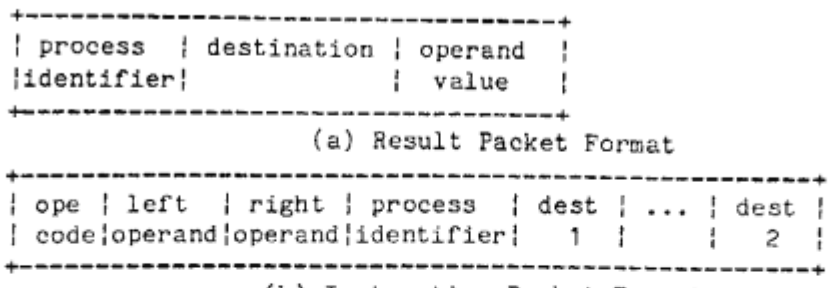


Figure 6.2 Configuration of a Processing Element Module

(1) Instruction Control Module (ICM)

The ICM is activated when a result packet (see Figure 6.3(a)) representing a token arrived on its input port. A result packet consists of a process identifier which specifies a process, a destination address of an instruction to which the result is transferred, and a result value which is an operand of the instruction. Using a process identifier and a destination address of the result packet, ICM determines whether all operands for the instruction are now present (i.e. whether the instruction becomes executable) or not. If the instruction turns out to be executable, the ICM generates an instruction packet (see Figure 6.3(b)) and sends it to the Execution Module (EXM). As Figure 6.4 shows, an ICM consists of a Result Packet Filter (RPF) that filters out result packets belonging to the termination processes, a Result Packet Queue (RPQ) with a queuing function for result packets, and an Instruction Control Unit (ICU) with a control function which determines whether an instruction is executable and, if it is, generates and outputs an instruction packets.



(a) Result Packet Format

(b) Instruction Packet Format

Figure 6.3 Result and Instruction Packet Formats

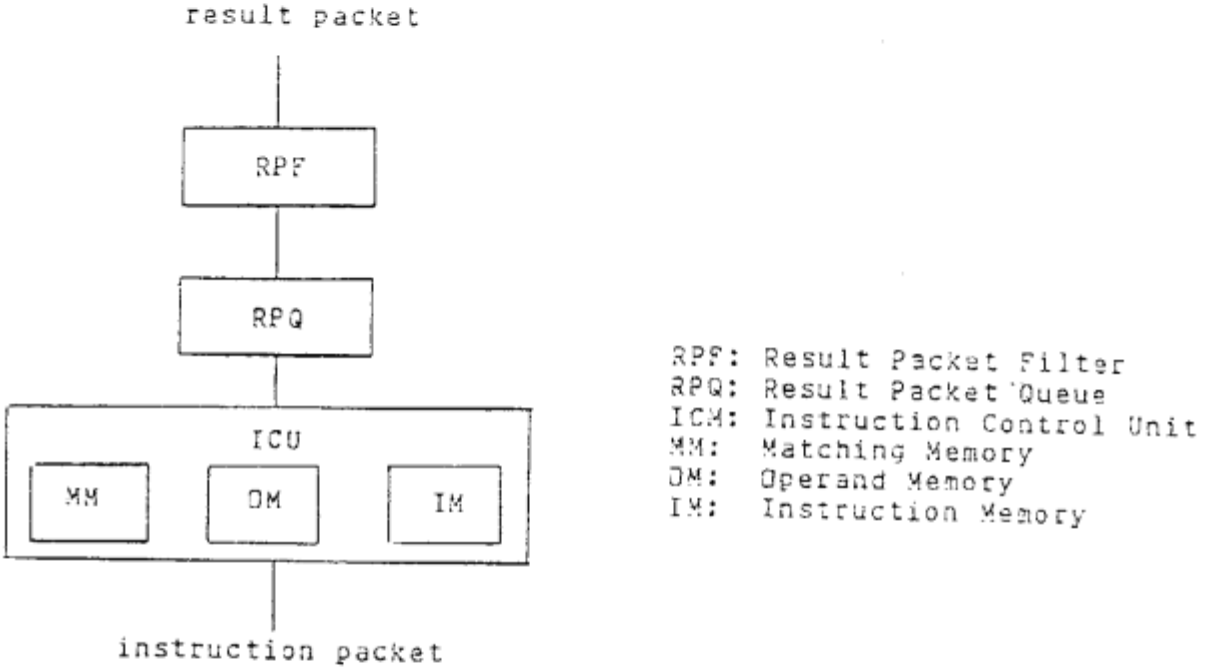


Figure 6.4 Configuration of Instruction Control Module

(a) Result Packet Filter (RPF)

The RPF has an associative memory to store process identifiers belonging to the active processes, and when a result packet arrives, performs associative search on the memory using the process identifier in the packet as a key field. If an associative search succeeds, the token belongs to an active process, otherwise, it belongs to a terminated process or a swapped out process. According to the search result, it determines whether it should pass the result packet onto the next RPQ. The RPF also has an address conversion function to implement the dynamic relocation of program. The associative memory, therefore, contains, in addition to the identifiers of the processes being executed, the

physical base addresses of the code for the processes. The RPF can perform address conversion by adding this base address to the destination address, which specifies a relative address from top of code.

(b) Result Packet Queue (RPQ)

The RPQ consists of first-in first-out memory and stores result packets temporarily until the next stage unit, ICU, becomes ready to input the result packet.

(c) Instruction Control Unit (ICU)

The ICU consists of a Matching Memory (MM) which is used to determine whether an instruction is executable or not, an Operand Memory (OM) which temporarily stores operands until their instructions become executable, and an Instruction Memory (IM) to store the instruction code.

To simplify the detection of instruction's executability, each instruction receives only one or two operands. An instruction with single operand becomes executable when its result packet arrives, and an instruction with two operands becomes executable when two result packets, which carry the left and right operands, arrives. To which type an instruction belongs is specified by the firing control tag subfield in the destination address field in the result packet.

When a result packet arrives from the RPQ, the ICU tests the firing control tag in the result packet. If the tag specifies an instruction with single operand, the ICU constructs the instruction packet immediately, from the result packet and the instruction code fetched from the IM, and send the instruction packet to the next stage unit, EXM.

In the case of an instruction with two operands, the MM, which consists of an associative memory, is searched associatively using the process identifier and the destination address in the result packet as a key field. The

associative search fails only for a packet which carries the first-arrived operand at the instruction, and the process identifier and the destination address in the packet are stored in an empty MM's entry, and the operand value in its corresponding OM's entry. For other packets, associative searches succeed and the instruction is executable; the first-arrived operand stored in the OM and the operand just arrived along with the instruction code constitute an instruction packet.

(2) Execution Module (EXM)

The EXM receives an instruction packet from the ICM, decodes the operation code, and controls its execution. Figure 6.5 shows the structure of an EXM. The EXM consists of Atomic Processing Units (APUs), the basic functional units responsible for instruction execution, an Instruction packet Distribution Unit (IDU) which distributes instruction packets among APUs, and a Result packet Arbitration Unit (RAU) which collects result packets generated in APUs and transfers them to the next destination.

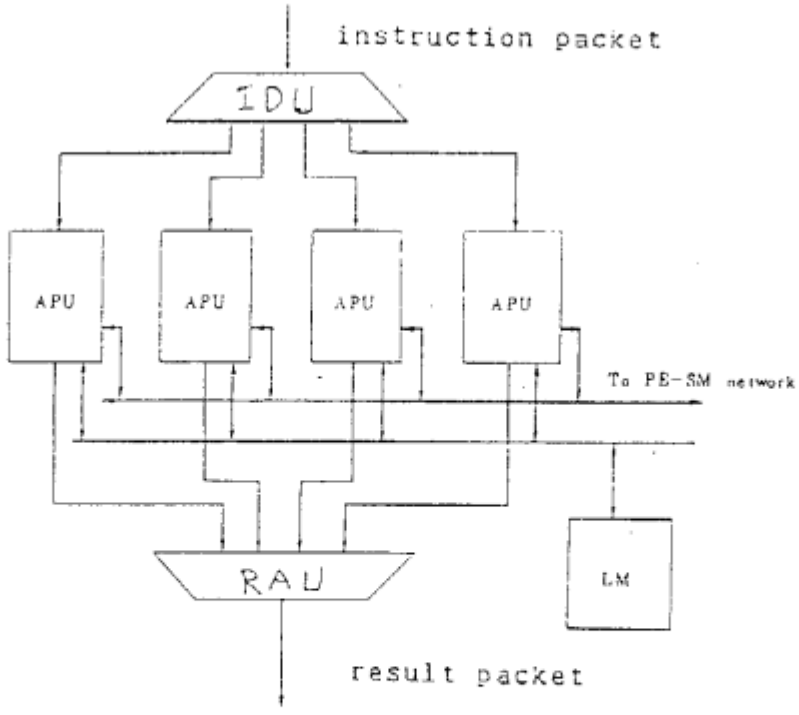


Figure 6.5 Configuration of Execution Module

(a) Atomic Processing Unit (APU)

Instructions executed in APUs include those for process identifier allocation control related to procedure calling and those for instructions or built-in predicates without accesses to structure memory. The execution results of the instructions, along with the destination addresses in the instruction packets, form result packets, which are sent to the RAU. Instructions, which access the structure memory, are transferred to the designated SMM via PE-SM Network.

Procedure call instructions reference a process control table, which maintain parent-children relationship of processes, and allocate unique process identifiers for created processes. The process control table is stored in Local Memory (LM), which are shared among APUs in the EXM. The LM is also used for free cell list described later.

Structure memory instructions, which include read, write, allocate, or reference count control instructions, are transferred from the APU to SMMs as SMM instruction packets.

In order to control pipelinig between the APU oparation and various packet transmissions, each APU has a result packet register and a SMM instruction packet register.

The number of APUs in an EXM will eventually be determined by balancing the throughput of the ICM and the throughput of the APU.

(b) Instruction Packet Distribution Unit (IDU)

The IDU receives instruction packets from the ICM and distributes them to the appropriate APUs. Using a load-distribution strategy, in which each APU has same functions, the IDU can send the instruction packets to the idle APUs. The packet transmission is suspended, if all the APU are busy.

(c) Result Packet Arbitration Unit (RAU)

The results of an instruction execution are sent as result packets, to the specified destination addresses in the instruction code. The RAU waits for result packets from every APUs, arbitrate the result packet sending requests, and select one of them. According to packet's destination address, which includes the PEM number and the IM address of the destination instruction, the RAU determines whether circulates the packet in its own PEM or sends it to another PEM via Inter-PE Network.

6.2 Structure Memory Module (SMM)

Structured data manipulated in SMM includes lists, vectors, streams, and strings, whose logical structures are represented in Figure 6.6. A user defined structure is represented by a vector or a list. A vector is composed from its size and one or more elements, and is represented by a pointer with a data type 'vector' of the structure memory, where its size and these elements are stored. The first element of a vector is called as a functor, the second element as a first argument, the third element as a second argument, and so on. A list is a special vector which is assumed to have a construct operator as its functor and have two arguments, left and right parts. Only two arguments are stored in the structure memory, and a list is represented by a pointer of this memory address with a data type 'list'. Lists are introduced to implove the performance of list operation. While a list and vector are able to have any type of data for their individual elements, a string must has a same data type for their all elements.

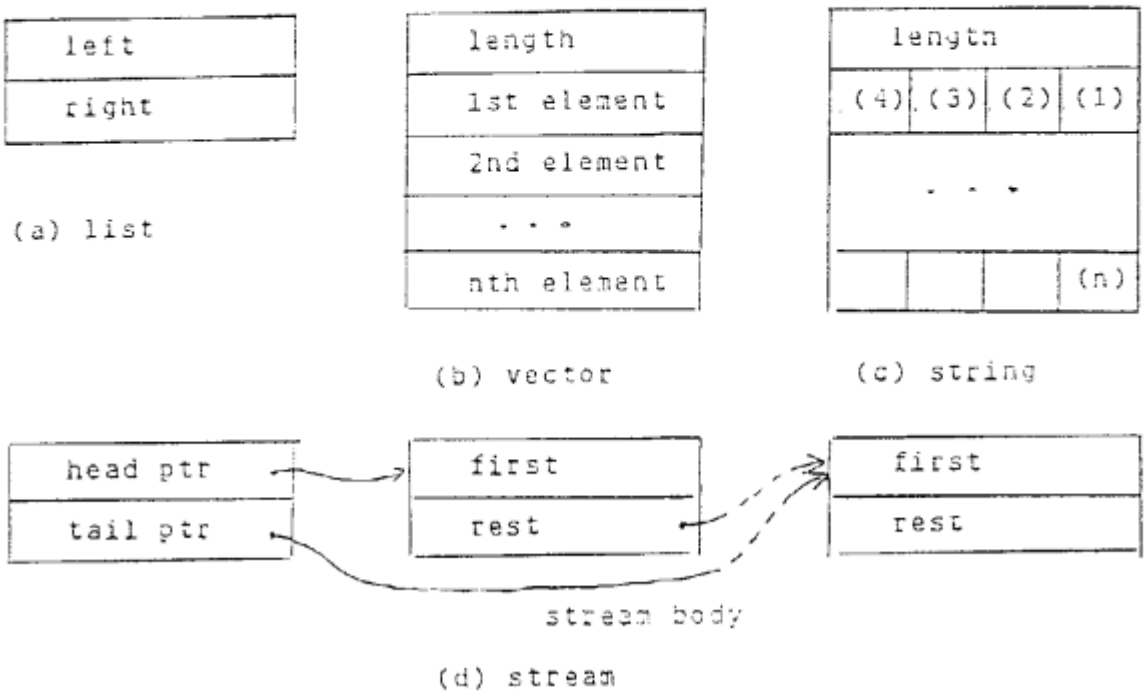


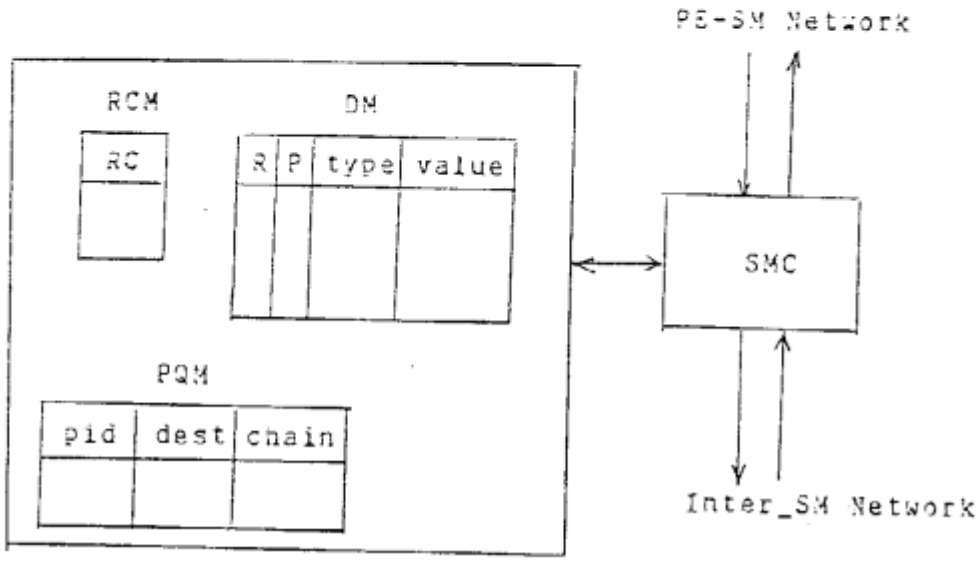
Figure 6.6 Representation of Structures Data

Streams basically employ the representation of the difference list [5]. As in Figure 6.6 (d), a stream is represented by a head pointer indicating the beginning of the stream and a tail pointer indicating its end. Like a list, a stream's body is represented by a chain of cells each having two arguments, first and rest parts. A stream element is put on the first part, and a subsequent part of stream is pointed by the rest part.

A producer process of the stream elements adds a new element after the current tail pointer and updates the contents of the tail pointer to the address of the new element. A consumer process of the stream can get the elements of the stream by traversing the stream body starting with the head pointer. When the consumer reached at the current stream end, it waits for the next element

which will be appended by the producer. That is, the stream provides an asynchronous communication means from the producer to the consumer.

Figure 6.7 shows the configuration of a Structure Memory Module (SMM).



SMC: Structure Memory Controller
DM: Data Memory
RCM: Refernce Count Memory
PQM: Packet Queue Memory

Figure 6.7 Configuration of a Structure Memory Module

As the figure shows, a SMM constists of a Structure Memory Controller (SMC), Data Memory (DM), Reference Count Memory (RCM), and Pending Queue Memory (PQM).

The SMC is initiated when a SMM instruction packet sent from an APU via the PE-SM Network, and controls execution of the instruction. Instructions can be roughly classified into three types:

- Reference count control instructions

- Data read/write instructions

- Free cell control instructions

(1) Reference Count Control Instructions

To perform efficient gargabe collection on structure memory, our machine uses a reference count method. This method requires a reference count, which is stored in the Reference Count Memory (RCM), corresponding to each structure memory cell stored in the Data Memory (DM). A structure memory cell has multiple entries where its substructures are stored. Since a reference count control instruction is often performed simultaneously with other structure operations, the SMC will be designed so that updating the reference count and accessing the DM can be carried out in parallel. The updating of the reference count involves a series of operations: reading from the RCM, calculating a new reference count, and writing the result into the RCM. Using a high-speed memory device as the RCM, these operations can be completed in the same or less machine cycles as a DM operation.

When a reference count for a structure memory cell reaches to zero, that is, when the structure becomes a garbage, all of its substructures are tested. If an substructure is found to be a pointer to another structured data, the reference count to that substructure must be decremented. If the pointer of substructure points to other SMM, the reference count decrement instruction packet will send to the specified SMM via Inter-SM Network.

(2) Data Read/Write Instructions

In addition tc a data type tag and value, each entry in the DM contains a Ready (R) bit, which shows whether the contents of the corresponding data value is valid, and a Pending (P) bit, which shows whether any waiting instruction for data is exist. The R bit is provided to allow the producer and consumer processes to asynchronously communicate messages.

Having received a read instruction, the SMC first checks the R bit at the specified DM address. If it is on, the SMC reads the data type and value in the DM, and constructs a result packet to send them to the destination address specified in the instruction. Otherwise, the read operation is suspended until data is written into that entry. The suspended read operation is temporarily stored in the Pending Queue Memory (PQM); First, the SMC gets an free queue cell in the PQM and stores the process identifier and destination address in the cell. Then it also stores the address of the queue cell in the value field of the DM entry and turns its P bit on. If the P bit has been already on, the new queue cell is chained to the previous queue cell chain.

Having received a data write instruction, the SMC checks the P bit at the specified DM address. If it is on, which indicates some suspended read requests has already been issued, then the SMC fetches all the queue cells chained by the value field and produces result packets to be sent to the specified destination addresses in the queue cells. Then, the SMC writes the data value into the DM entry, while reseting the P bit and settig the R bit simultaneously.

More complicated instructions, such as test & set operation or structure traversing operation, will be also provided.

(3) Free Cell Control Instructions

To execute a structure construction instruction faster, a free cell list to maintain the free cell addresses of strucure memory will be set up in the LM of each PEM. When this instruction is executed in one of the APU, the APU gets a free cell address from top of the local free cell list and constructs result packets immediately. After the sending of result packets has completed, the APU sends a free cell instruction packet to the SMM in order to add a new free cell address into the free cell list, and also sends the data write instruction packets of the substructures. Notice that addresses held in the free cell lists are different each other.

In order to map the structured data over the SMMs evenly, the APU must be able to select an arbitrary SMM when allocating a free cell list. All the SMMs in the system provides a single address space. A SMM cell address, therefore, consists of a SMM number and a local address in the SMM. Figure 6.8 shows the free cell list stored in each LM, where free cell addresses are maintained by their source SMMs. Allocating a new free cell, the APU selects one of the SMMs according to the allocation strategy described in section 5, and get the top of free cell list of that SMM. The free cell list in the LM should be long enough to prevent the list from being emptied by free cell requests.

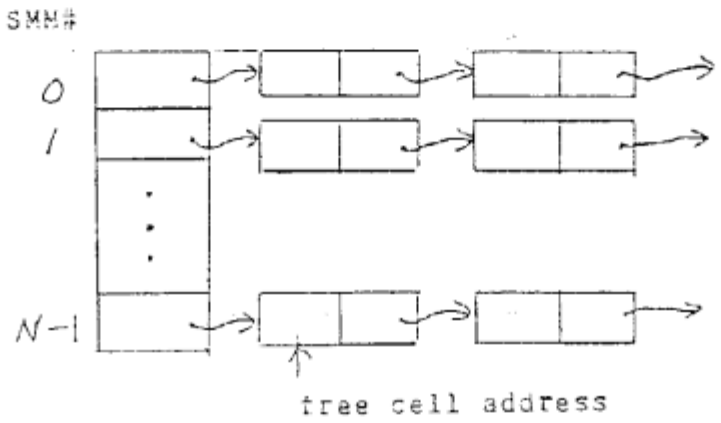


Figure 6.8 Free Cell List in a Local Memory

Having received a free cell instruction, the SMC gets one of free cells and constructs a free cell packet, which is sent to the requested PEM to add a new free cell to the free cell list in its LM. All the cells, whose reference count is zero, are chained in the DM beginning at a free cell header. The SMC, therefore, can get a free cell from this header.

6.3 Network Structure

We will use an asynchronous, disributed, packet exchange network structure, so that all the modules in the system can operate independently. Another factor which determines the network structure is a network topology. The network

topology will depend upon the characteristics of data transfer requests. Our machine will use tree types of networks: Inter-PE Network, PE-SM Network, and Inter-SM Network.

As the Inter-PE Network and Inter-SM Network, we select a network topology with a two-dimensional mesh structure, because, as mentioned in Section 4 and 5, this type of network has a potential to allocate resources to optimize localized communications.

On the other hand, we have chosen a multiple-stage network topology as the PE-SM Network, where every distance between a PEM and a SMM is equal. A PE-SM Network based on a localized structure might cause resource allocation to be very complex, because this structure requires testing over process allocation status among PEMs as well as structure memory allocation status among SMMs, when allocating any resources.

6. CONCLUSION

This paper has described a processing model of a data-flow-based Prolog machine and its architecture. In the description, we have shown that the introduction of a stream concept can implement OR-parallel and AND-parallel processing of Prolog.

At present, we are developing a software simulater for this machine and designing a more detailed structure.

Acknowledgements

We would like to thank Dr. Kunio Murakami, Cheif of the First Research Laboratory in ICOT, and other ICOT research members for their valuable comments.

References

[1] Amamiya,M. and Hasegawa,R.,"Data Flow Machine and Functional Language", AL81-84,PRL81-63, IECE of Japan, Dec. 1981 (in Japanese).

[2] Arvind, Gostelow,K.P. and Plouffe,W.E., " An Asynchronous Programming Language and Computing Machine", TR114a, Dept. of Information and Computer Science, University of California, Irvine, Dec. 1978.

[3] Arvind and Thomas,R.E.,"I-Structures: An Efficient Data Type for Functional Languages", TM-178 Laboratory for Computer Science, MIT, Sept. 1981.

[4] Clark,K.L. and Gregory,S., " A Relational Language for Parallel Programming", Research Report of Imperial College of Science and Technology, DOC 81/16, Jul. 1981.

[5] Clark,K.L. and Ternland,S.A., "A First Order Theory of Data and Programs", IFIP 77, North-Holland Publishing, 1977.

[6] Cohen,J., "Garbage Collection of Linked Data Structures", Computing Survays, Vol.13, No.3, Sep. 1981.

[7] Conery,J.S. and Kibler,D., "Parallel Interpretation of Logic Programming", Proc. of Conf. on Functional Programming and Computer Architecture, ACM, Oct. 1981.

[8] Gurd,J.R. and Watson,I., "Data Driven System for High Speed Parallel Computing", Computer Design, Jul. 1980.

[9] Tanaka,H. et al.,"The preliminary Research on the Data Flow Machine and Data Base Machine as the Basic Architecture of Fifth Generation Computer Systems", Proc. of International Conference on Fifth Generation System, JIPDEC, Japan, Oct. 1981.

[10] Kowalski,R., "Predicate Logic as Programming Language", IFIP 74, North-Holland Publishing, 1974.

[11] Shapiro,E.Y., "A Subset of Concurrent Prolog and its Interpreter", TR-003, ICOT, Japan, Jan. 1983.