TR-323

Efficient Stream Processing in GHC and Its Evaluation on a Parallel Inference Machine

by
N. Itoh, E. Kuno & T. Oohara

November, 1987

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

Efficient Stream Processing in GHC and Its Evaluation on a Parallel Inference Machine

Noriyoshi Ito, Eiji Kuno, and Teruhiko Oohara
OKI Electric Industry Co., Ltd.

ABSTRACT

A set of primitives to implement efficient stream processing in GHC, Guarded Horn Clauses, and its evaluation results on PIM-D, Parallel Inference Machine based on the Dataflow model, are presented. The language is efficient as the conventional procedural languages, because stream processing, the basic operation in GHC, is implemented as machine primitives rather than creating enormous merge processes. In order to support such primitives with preserving "logically correctness," only one extra bit shared flag added to every pointer is sufficient. The evaluation results show that performance is improved by introducing such primitives.

1. INTRODUCTION

The authors are investigating a parallel inference machine aiming at the Fifth Generation Computers. The target language is called KL1, Kernel Language version one, whose basis is on a stream AND-parallel logic language called GHC (Guarded Horn Clauses) [14]. GHC provides powerful description power and has clear semantics. In the GHC programs, goal activation is assumed as process invocation. Communication between these processes is performed via unification on logical variables shared among the processes. The processes invoked by the goal attempt to unify the shared variables with sequences of messages (streams) to be sent or received.

The typical applications of the parallel inference machine include generate-and-test problems. In these problems, a producer process generates a stream of candidate solutions, while a consumer process gets the candidates via the stream and tests if they satisfy the specified conditions. Again, new generate-and-test phases may be initiated using these solutions until the final solutions are obtained. Here, we can initiate multiple producer processes in parallel if possible; the solutions generated by these processes are merged into a single stream in nondeterminate manner and then sent to the consumer.

In GHC, such stream merging is logically performed by creating perpetual processes; a stream merge process is created when two or more streams are merged. The merge process waits for solutions from the multiple producers and produce a merged stream to send to the consumer. If such stream merging is implemented directly as the perpetual processes, heavy overhead to create and manage the merge processes may degrade system performance.

The proposed stream primitives can support efficient stream processing as in the conventional procedural languages, such as FORTRAN or language C. In order to support such primitives with preserving "logically correctness," only one extra bit shared flag added to every pointer is sufficient [9].

As the first step of the project, we developed an experimental machine called PIM-D (Parallel Inference Machine based on the Dataflow model) [10]. Programs in the dataflow model are represented by dataflow graphs, where nodes correspond to operators and directed arcs correspond to data paths along which operands are sent. Execution of the graphs is performed in a data driven manner. That is, each node becomes executable only when all the operands are arrived on its input arcs; it performs the operation and puts the results on its output arcs without side-effect. This functionality of the operators assures independent execution of the operators whose operands are ready; they are executed in parallel without affecting the other active operators [1] [2]

[6]. The machine, therefore, can easily exploit the parallelism in the programs.

The machine is constructed from multiple processing element module and structure memory modules interconnected by a network. Each processing element module interprets the data flow graphs in parallel and transfers packets to/from other processing element modules or the structure memory modules. Each structure memory module stores structured data and is responsible to the structure accessing command packets from the processing element modules. The evaluation results of the stream merging primitives on PIM-D show that performance is significantly improved.

The informal semantics of GHC and stream processing in GHC are outlined in Section 2, and the efficient stream primitives are proposed in Section 3. Section 4 shows a brief description of PIM-D machine architecture, and the evaluation results of the stream processing are described in Section 5.

2. SEMANTICS OF GHC

GHC is a basic language of KL1 (Kernel Language version one), which is a machine-independent language of the parallel inference machine. GHC is one of AND-parallel logic languages, such as PARLOG [5] or Concurrent Prolog [13]. Of these, GHC is simplest because it is a minimum extension of Horn Clauses; only the guards are added to Horn Clauses. In full GHC which allows any user-defined predicates in the guards, compile-time detection of suspensive unification is difficult; the compiler must generate codes so that unification operations to unify unbound guard variables with non-variable terms are suspended until the guard variables are instantiated. Therefore, flat GHC, which allows only built-in predicates in the guards, was selected as the basic language of KL1 [11].

But flat GHC still provides powerful description power and has clear semantics. In the GHC programs, goal activation is assumed as process invocation. Communication between these processes is performed via unification on logical variables shared among the processes. The processes invoked by the goal attempt to unify the shared variables with sequences of messages (streams) to be sent or received.

A simple producer/consumer problem is described as follows:

```
?- p(1, S), q(S).                                          ... Goal
p(N, S) :- N < 100 | S = [N|S1], N1 := N + 1, p(N1, S1).   ... C1
p(N, S) :- N = 100 | S = [].                               ... C2
q(S) :- S = [N|S1] | print(N), q(S1).                      ... C3
q(S) :- S = [] | true.                                     ... C4
```

The first statement is a goal to be solved. When the goal is invoked it activates subgoals p(1,S) and q(S). Each subgoal consists of a predicate ('p' or 'q') and zero or more arguments. Here note that an unbound variable S is shared among these subgoals. GHC has no sequential semantics; the subgoals p(1, S) and q(S) can be activated in parallel.

The second through fifth statements (C1 through C4) are called clauses. The symbol ':-' specifies implication, and the left side of this symbol is called a clause head. Each clause consists of a guard and body separated by a symbol '|', which is called a commit operator. The commit operator plays a role of the guarded command [7]; when a subgoal is given, the clauses whose head predicates are matched with the subgoal are invoked, and only one clause whose guard succeeds can proceed to its body. Execution of the other clauses may be terminated.

Another role of the commit operator is that it controls the direction of unification. A clause which attempts to unify the goal variable with an instance in its body can instantiate the variable, while a clause which attempts to unify the goal variable with a non-variable term or other goal variable in its guard will be suspended until the variable is instantiated.

In the above example, the subgoal p(1, S) will invoke the clauses with head predicate 'p' (i.e., C1 and C2). Of these, only the guard of C1 succeeds for the given subgoal because the first argument of the subgoal is a number less than 100. Clause C1, therefore, proceeds to its body and instantiates the shared variable to a list [1|S1]. The body of clause C1 also activates a new subgoal p(2, S1) recursively, which then instantiates the new shared variable S1 to a list [2|S2]. This recursion is repeated until the first argument of the subgoal becomes to 100. The clauses with head predicate 'q' (C3 and C4), on the other hand, will be suspended until the variable S is instantiated to a list or nil by clauses C1 or C2. If the variable S is instantiated to a list, the clause C3 prints the list elements.

Here, the clauses activated by the subgoal p(1, S) and q(S) play a role of a producer and consumer, respectively, of a sequence of messages which consists of a list [1,2,3,...]. The messages are sent from the producer to the consumer on each time the partial solutions are obtained. As the messages are asynchronously sent from the producer to the consumer, the sequence of the messages is said a stream. The streams play a role of I-structures (incremental structures) [3], where the elements of the stream are incrementally produced or consumed.

Stream merging, which is often used for parallel search, is written as follows:

```
?- merge(S, T, U), p1(S), p2(T), q(U).
```

where the definition of the merge predicate is given as:

```
merge(S, T, U) :- S = [E|S1] | U = [E|U1], merge(S1, T, U1).
merge(S, T, U) :- T = [E|T1] | U = [E|U1], merge(S, T1, U1).
merge(S, T, U) :- S = [] | U = T.
merge(S, T, U) :- T = [] | U = S.
```

The goal activates four subgoals; a merge process invoked by the subgoal merge(S, T, U) consumes two streams S and T produced by processes invoked by the subgoals p1(S) and p2(T), respectively, and generates a new stream U which is then consumed by a process invoked by the subgoal q(U). The streams, here, are represented by a list and have only one producer and one consumer.

3. EFFICIENT STREAM PRIMITIVES

As mentioned above, many GHC programs can be considered as producer-consumer problems, where producer processes produce streams of messages and consumer processes receive the messages from them. GHC programs use logical variables for message passing. Each logical variable can be bound to only one instance and may be immediately discarded after the consumers read the instance from it. That is, the life span of each logical variable is very short and frequent memory allocation and deallocation for the logical variables are necessary.

The main idea of this section is to hide the detailed structure of the stream from the programmer's point of view and to develop a more efficient stream structure and primitives instead of using logical variables directly. An efficient stream, which is called a packed stream, is represented by pointers and a buffer as shown in Fig. 1. Each pointer points to a stream

buffer entry to be accessed and the stream elements are stored in the stream buffer. The streams here are "packed" into the stream buffers (i.e., CDR parts of the lists are omitted) by using the CDR coding scheme [12].

3.1 Stream Creation

When a new stream is created, a create$stream primitive is executed which initializes a stream buffer and returns two or more pointers pointing to the buffer. Usually, one of the pointers is used in the producer to append stream elements to the buffer and the others are used in the consumers to get the elements.

The following goal shows an example:

```
?- create$stream(S, T), p(S), q(T).
p(S) :- true | ..., put$stream(S, E, S1), p(S1).
q(S) :- get$stream(S, E, S1) | ..., q(S1).
```

Logically, the create$stream predicate unifies its first argument with the second argument as defined:

```
create$stream(S, T) :- true | S = T.
```

Actual implementation of the create$stream primitive forces the variable S and T to be instantiated to two independent buffer pointers. They are initialized to point to the head of the buffer. The processes activated by p(S) and q(T) will execute put$stream or get$stream primitives. Logically, these primitives are defined as:

```
put$stream(S, E, S1) :- true | S = [E|S1].
get$stream(S, E, S1) :- true | S = [E|S1].
```

Actually, each put$stream primitive unifies the argument E with a stream element pointed to by the buffer pointer S; if the stream element is undefined, E is stored to the entry. Then the pointer is incremented to point to the next entry. The incremented pointer is unified with the variable S1. Each get$stream primitive gets a stream element from the stream buffer and also increments the pointer when the stream element is defined (i.e., if the element is already defined). If the element is undefined, the get$stream primitive will be suspended (hooked to the buffer entry) until the element is written by a put$stream primitive; the put$stream primitive activates the suspended get$stream primitive before it writes the element.

A special symbol 'EOS' (end-of-stream) is used to signal the end of stream; the producer processes put the 'EOS' symbol when no more stream elements are sent to the consumer processes. This will be done by executing a put$stream(S, 'EOS', S1), or a special primitive put$EOS(S), where S and S1 are streams.

3.2 Stream Sharing

When a stream is shared among processes, each process should have independent buffer pointers; the original buffer pointer is copied and distributed to these processes. The following clause shows an example when a variable is shared among processes.

```
p(S) :- true | p1(S), p2(S).
```

To ensure this pointer copying, share$stream primitive is used in the machine language:

```
p(S) :- true | share$stream(S, S1, S2), p1(S1), p2(S2).
```

The stream pointer S is copied to S1 and S2, which is then used as the arguments of the subgoals p1(S1) and p2(S2), respectively. To be noted here is that the share$stream primitive simply copies (passes) its first argument to its second and third arguments when the first argument is a stream pointer or an atomic value, but copying will be deferred until the object pointed to by the pointer is referred when its first argument is a reference pointer to an unbound variable or structure. In order to support such deferred pointer copying, the shared flag scheme is introduced.

3.3 Shared Flag Scheme

When a structure is shared among multiple processes, it is sufficient to copy its reference pointers to these processes. A problem, however, is caused when a structure including streams or unbound variables is passed among the subgoals. The following goal shows an example where a structure f(...S...) which includes a stream S is shared among subgoals.

```
?- p(S), q(S).
p(S) :- true | T = f(...S...), p1(T), p2(T).
```

Here, we can rewrite this program by using the stream primitives:

```
?- create$stream(S1, S2), p(S1), q(S2).
p(S1) :- true | T = f(...S1...), share$stream(T, T1, T2), p1(T1), p2(T2).
```

When an eager pointer copying scheme is used, the whole structure of f(...S1...) should be copied before activation of two subgoals p1(T1) and p2(T2), in order to create copies of the stream pointer S1. Thus, variables T1 and T2 will be instantiated to copied structures f(...S1'...) and f(...S1''...), respectively, where S1' and S1'' are copied buffer pointer of S1. If the following definitions are given, the invoked clauses p1 or p2 may put the stream elements to or get the stream elements from the stream buffer pointed to by the stream pointers S1' or S1'' included in the copied structures.

```
p1(T1) :- T1 = f(...S...) | put$stream(S, E, U), ...
p2(T2) :- T2 = f(...S...) | get$stream(S, E, U), ...
```

This eager copying may waste the processing time if structures to be copied is vary large or if they are frequently shared among processes. Furthermore, the copying will cause heavy communication traffic in the network, because the substructures may be distributed among many storage units in the parallel machine such as PIM-D.

The deferred pointer copying can be implemented by adding an extra bit to each pointer, which is called a shared flag; the shared flag is implemented by extending the tag field to specify the data type. The share$stream primitive in the above case will only set the shared flag of the reference pointer to f(...S1...), which is passed to the processes. This shared flag is inherited to its substructures; unification to decompose the shared structure into the substructures is extended to:

(1) create copies if the substructures are stream buffer pointers,
(2) set shared flags of the substructures on if they are reference pointers,
(3) perform normal unification if the substructures are atomic values.

Thus, if the clause p1 is defined as:

```
p1(T1) :- T1 = f(...S...) | put$stream(S, E, U), ...
```

and if its given goal argument is a structure f(...S1...), where S1 is a stream buffer pointer, copy of the buffer pointer S1 is created and instantiated to the variable S on its guard's unification between the original structure f(...S1...) and the structure f(...S...) in the guard.

3.4 Nondeterminate Stream Merging

Stream merging is performed by sharing the stream pointers. The following goal shows an example of stream merging, where two subgoals, p1(S) and p2(T), generate two independent streams S and T consisting of lists [a1,a2,...] and [b1,b2,...], respectively, that are merged into a single stream U consumed by two independent subgoals q1 and q2.

```
?- merge(S,T,U), p1(S), p2(T), q1(U), q2(U).
```

Figure 2 shows a direct implementation schema of this goal. The producer processes create lists containing the stream elements which may be immediately discarded after the merge process consumes them and creates a new list for the merged stream. In this implementation, the memory manager may suffer from heavy list cell allocation or reclamation overhead and stream merging is very expensive because it is performed by an independent process.

The efficient stream merging is implemented by the packed stream primitives depicted in Fig. 3. This figure shows that every producer process shares an indirect stream pointer cell and appends new stream elements to the stream buffer. In order to create the indirect stream pointer cell, which is called a merged stream descriptor, a merge$stream primitive is executed instead of the merge predicate as:

```
?- merge$stream(S,T,U), p1(S), p2(T), share$stream(U,U1,U2), q1(U1), q2(U2).
```

The put$stream and get$stream primitives are extended to handle both non-merged stream pointers and merged stream descriptors; they are identified by their tag field. The merge$stream primitive creates a stream buffer and a merged stream descriptor which will be unified with the first and second arguments (S and T), if its third argument U is undefined. If U is already instantiated to a descriptor, S and T are unified with the descriptor; processes referring S and T will share the descriptor. This merge$stream primitive is also used for consumer processes; the consumers sharing the stream descriptor will obtain independent stream elements from the stream in a nondeterminate manner.

When a stream is merged by multiple producer processes, the 'EOS' symbol should be put into the buffer when all the producer processes sharing the merged stream descriptor have executed the put$EOS primitives. In order detect this situation, a reference count scheme is used; a reference count field, which is called the descriptor reference count, to maintain the number of pointers pointing to the merged stream descriptor is stored with the indirect stream pointer as shown in Fig. 3. The descriptor reference count is

(1) incremented when a merge$stream primitive is executed (i.e., when a new process to merge the stream is created), and

(2) decremented when a put$EOS primitive is executed (i.e., when a process merging the stream is terminated, or when it no longer refers to the stream).

Note that update of the reference count is only necessary in the above cases and the ordinal put$stream or get$stream primitives don't affect the reference count. The 'EOS' symbol is stored into the stream buffer when the reference count is reached to zero. As the stream descriptor is no longer referred, it will be reclaimed.

3.5 Stream Buffer Management

Because almost all the process communication is performed via streams in GHC, frequent stream buffer allocation or reclamation will be necessary. A simple but inefficient solution is to leave the garbage collector to reclaim of such stream buffers. In this case, however, the garbage collector will suffer from exhaustive memory consumption.

Real-time garbage collection can be implemented using the reference count scheme; another reference count field, which is called a buffer reference count, is added to each stream buffer as shown in Fig. 4. The buffer reference count is

(1) incremented when a share$stream is executed, and
(2) decremented when the stream buffer pointer exceeds the tail buffer address by executing a put$stream or get$stream primitive, or when a process no longer refers to the stream.

When the buffer pointer exceeds the buffer, a new buffer is allocated if the buffer reference count of the old buffer is not zero, and the allocated buffer is chained to the old buffer as shown by a broken line in Fig. 4. Other processes referring the old buffer will follow the new buffer chain if their stream pointers exceed the old buffer. If the buffer reference count of the old buffer is zero, it is reused.

To ensure that the stream primitives can update the buffer reference count, the stream pointer may be extended to another stream descriptor as shown in Fig. 5, that holds the pointer to the buffer reference count field, the buffer size, as well as the buffer pointer; in this case, each process has a reference pointer to the descriptor. This scheme, however, needs an extra indirect memory access to put or get a stream element.

Therefore, we introduced an optimized version of packed streams called standard streams, that have fixed-sized buffers. The standard stream buffers are aligned to N-word address, where N is a buffer size, and the reference count field is located at the head of the buffer. Because the address of reference count field is easily obtained by the stream buffer pointer, no stream descriptor as shown in Fig. 5 is necessary for non-merged streams. Furthermore, memory management of the standard stream buffers is easily implemented because every buffer is fixed-sized and a simple free list management scheme can be used; buffers whose buffer reference counts are decremented to zero are chained to a free list, and new buffers are allocated from the free list.

4. THE MACHINE ARCHITECTURE

The machine is constructed of multiple processing element modules and multiple structure memory modules connected by a hierarchical network as shown in Fig. 6. Each processing element module consists of a packet queue unit, an instruction control unit, two atomic processing units, and a network interface unit; these units operate independently and construct a pipeline architecture. The programs, represented by dataflow graphs, are stored in the instruction control unit, which receives the packets from the packet queue unit, detects the readiness of the operands, and sends the executable instruction to one of the atomic processing units if all the operands of the instructions are ready.

The atomic processing units interpret the executable instructions, access the local memory unit if necessary, generate new results packets, which are again sent to the instruction control unit via the packet queue unit, and generate structure command packets sent to the structure memory modules.

The local memory unit is used to store the local information such as process control blocks or remote resource management tables; in order to perform remote resource allocation quickly, each local memory unit has a process management table containing reserved process identifiers of the other processing element modules, and a structure management table containing the reserved structure cell addresses of the whole structure memory modules. When a new process which should be distributed to other module is created, a process identifier of the other processing element module is obtained from the remote process management table; the packets for the new process are sent to the processing element module specified by the process identifier via the network. The structure memory allocation is performed in the same manner.

The structure memory modules are responsible to the structure command packets from the atomic processing units and used to store structure data. Current version of packed stream implementation uses local memories to allocate stream buffers, because locality can be exploited to access the stream buffers; if many producer and consumer processes are created and there is locality in communication among these processes, the processes can be allocated to the local processing element modules so that most of stream accesses can be performed on the local memories. To control such allocation, local process invocation primitives are introduced [10].

When a create$stream primitive is executed in an atomic processing unit, a new stream buffer is allocated and all the buffer entries are initialized to be undefined. The stream buffer pointer is sent to the next instructions to perform put or get stream elements. If the put$stream or get$stream primitives are executed and if the stream buffers are not resident in its own local memory, they are sent to the processing element modules designated by their operands (stream pointers).

5. EVALUATION RESULTS

Two versions of simple GHC benchmark programs are examined. The first one is a non-packed version, where all the streams are represented by lists and stream merging to collect every solution is performed by perpetual merge processes. The other is a packed stream version, where stream processing is performed by the stream primitives described in Section 3. The benchmark programs include 6-queens and prime number generator up to number 500.

Because only the top level stream is used to merge all final solutions in the 6-queens program, the packed stream version of this program uses the packed stream primitives to merge the top level solutions, in order to examine effectiveness of packed stream merging; all the other streams are implemented by non-packed streams. In the packed stream version of the prime number generator, on the other hand, no stream merging is performed and all the streams are implemented by packed streams; it is used to examine effectiveness of the put$stream or get$stream primitives.

Figure 7 shows performance comparison of these programs. In this figure, performance is given by T1/ti, where ti is the execution time needed to search all solutions when the number of both modules (the processing elements and structure memories) is i (i=1,2,4,8), and T1 is the execution time of the non-packed stream version when the number of modules is one. Because each cluster has up to four processing elements and also up to four structure memories, only one cluster is used when the number of modules is less than or equal to four, and two clusters are used when it is eight.

For all these programs, performance is significantly improved when the number of the modules is increased from one to four, but shows a tendency to be saturated when the number of modules is eight because parallelism inherent in these programs is not so large. Compared with the non-packed 6-queens program,

performance of the packed 6-queens program is sixteen to forty percent higher, even though only ten percent of the total executed instructions are reduced than that of the non-packed version; only the stream merge processes are replaced by the stream merging primitives.

Performance of the packed prime number program (packed primes) is significantly increased; it is about more than twice than that of the non-packed version(non-packed primes), independent of the number of modules. This improvement is achieved mainly by the reduction of the structure memory commands, that is about one third of that of the non-packed version.

6. CONCLUSION

A set of primitives to support efficient stream processing in GHC is presented. The detailed structure of the streams is hidden from the programmers and a more efficient stream structure and primitives are provided instead of using logical variables directly. An efficient stream, which is called a packed stream, is represented by pointers and a buffer. Each pointer points to a stream buffer entry to be accessed and the stream elements are stored in the stream buffer.

The language is efficient as the conventional procedural languages, because stream processing, the basic operation in the AND-parallel logic language, is implemented as machine primitives rather than creating enormous merge processes. In order to support such primitives with preserving "logically correctness," only one extra bit shared flag added to every pointer is sufficient. This shared flag scheme is currently extended to the multiple reference bit to reclaim structure cells [4]. The evaluation results on PIM-D show that performance is improved about ten to forty percent by introducing such primitives.

<Acknowledgments>

The authors extend their thanks to Director Kazuhiro Fuchi at ICOT, who afforded them the opportunity to pursue this research. Also much appreciated are Dr. Shun-ichi Uchida, Chief of the Fourth Research Laboratory, and Dr. Atsuhiro Goto (ICOT) for their valuable advice and comments, Masasuke Kishi and Masayuki Tomisawa (OKI) for their hardware implementation of PIM-D, and other ICOT research members for their fruitful discussion.

<References>

[1] Amamiya,M., R.Hasegawa, O.Nakamura, and H.Mikami, "A List-processing oriented Data Flow Architecture," National Computer Conference 1982, pp. 143-151, June, 1982.

[2] Arvind, K.P.Gostelow, and W.E.Plouffe, "An Asynchronous Programming Language and Computing Machine," TR-114a, Dept. of ICS, University of California, Irvine, Dec., 1978.

[3] Arvind and R.E.Thomas, "I-Structures: An Efficient Data Type for Functional Languages", TM-118, Laboratory of Computer Science, MIT, 1980.

[4] Chikayama,T. and Y.Kimura, "Multiple Reference Management in Flat GHC," Proc. of 4th Int'l Conf. on Logic Programming, May 1987.

[5] Clark,K. and S.Gregory, "PARLOG: Parallel Programming in Prolog," Research Report DOC 84/4, Imperial College of Science and Technology, April, 1984.

[6] Dennis,J.B. and D.P.Misnus, "A Preliminary Architecture for A Basic Data Flow Processor," Proc. of 2nd Symp. on Computer Architecture, Jan., 1975.

[7] Dijkstra,E.M., "A Discipline of Programming," Prentice-Hall, 1976.

[8] Ito,N., H.Shimizu, M.Kishi, E.Kuno, and K.Rokusawa, "Data-flow Based Execution Mechanisms of Parallel and Concurrent Prolog," New Generation Computing, Vol. 3, No. 1, 1985.

[9] Ito,N., M.Kishi, E.Kuno, and K.Rokusawa, "The Dataflow-Based Parallel Inference Machine To Support Two Basic Languages in KL1," Proc. of IFIP TC-10 Working Conference on Fifth Generation Computer Architecture, UMIST (Manchester), July 1985.

[10] Ito,N., S.Masatoshi, E.Kuno, and K.Rokusawa, "The Architecture and Preliminary Evaluation Results of the Experimental Parallel Inference Machine," Proc. of 13th Annual Int'l Symp. on Computer Architecture, Jun. 1986.

[11] Kimura,Y. and T.Chikayama, "An Abstract KL1 Machine and Its Instrcution Set," Proc. of 4th Symposium on Logic Programming, Aug. 1987.

[12] Knight,T., CONS, MIT AI Working Paper 80, 1984.

[13] Shapiro,E.Y., "A Subset of Concurrent Prolog and its Interpreter," TR-003, Institute for New Generation Computer Technology, Tokyo, Japan, Jan., 1983.

[14] Ueda,K., "Guarded Horn Clauses," TR-103, Institute for New Generation Computer Technology, Tokyo, Japan, 1985.

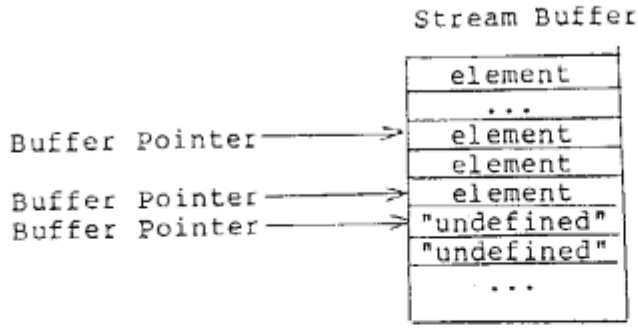


Fig.1 Representation of A Packed Stream

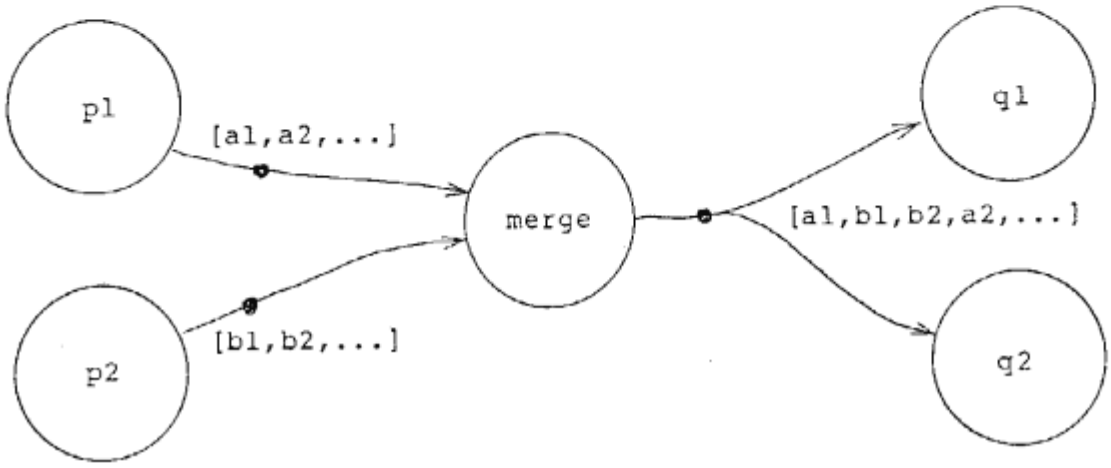
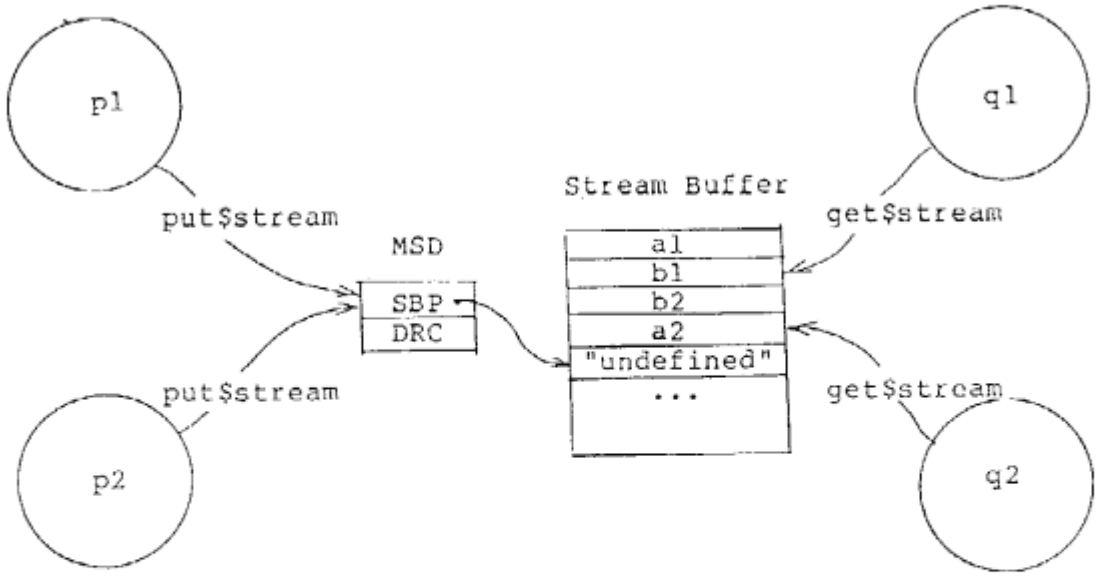


Fig.2 Stream Merging Schema



MSD: Merged Stream Descriptor
SBP: Stream Buffer Pointer
DRC: Descriptor Reference Count

Fig.3 Stream Merging by A Packed Stream

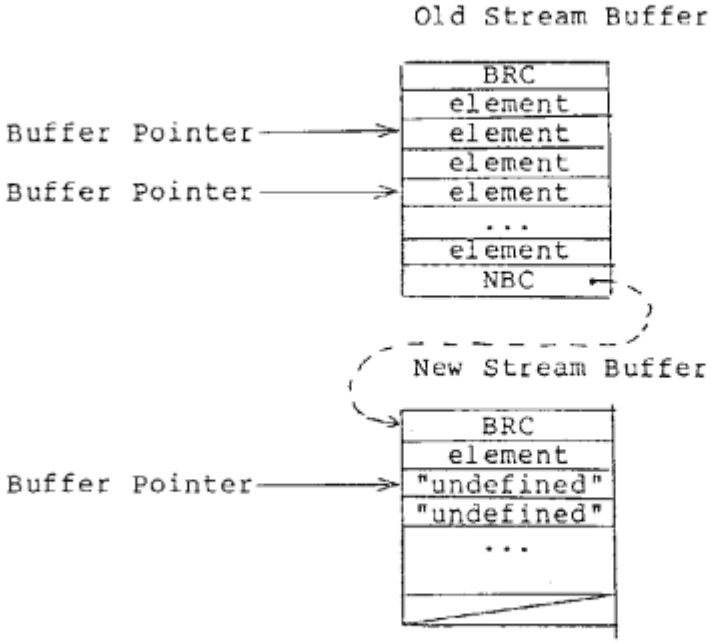


Fig.4 Representation of A Packed Stream

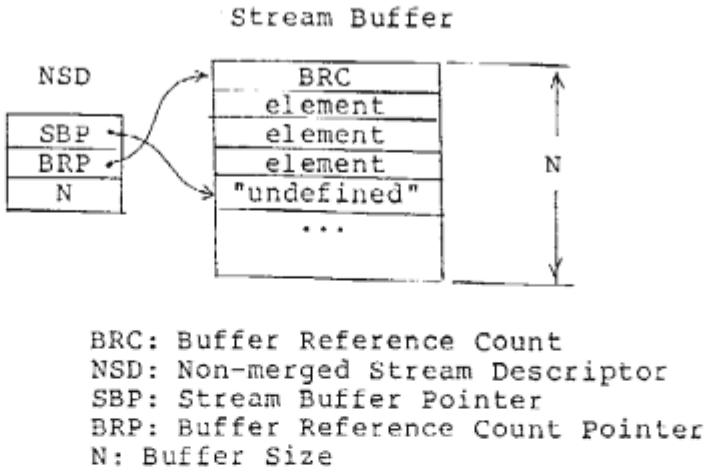


Fig.5 Stream Descriptor of the Non-merged Stream

PE : Processing Element
PQU : Packet Queue Unit
ICU : Instruction Control Unit
APU : Atomic Processing Unit
LMU : Local Memory Unit
SM : Structure Memory
SPU : Structure Processing Unit
SMU : Structure Memory Unit
NN : Network Node
BA : Bus Arbitor
FIFO : First-In First-Out Memory

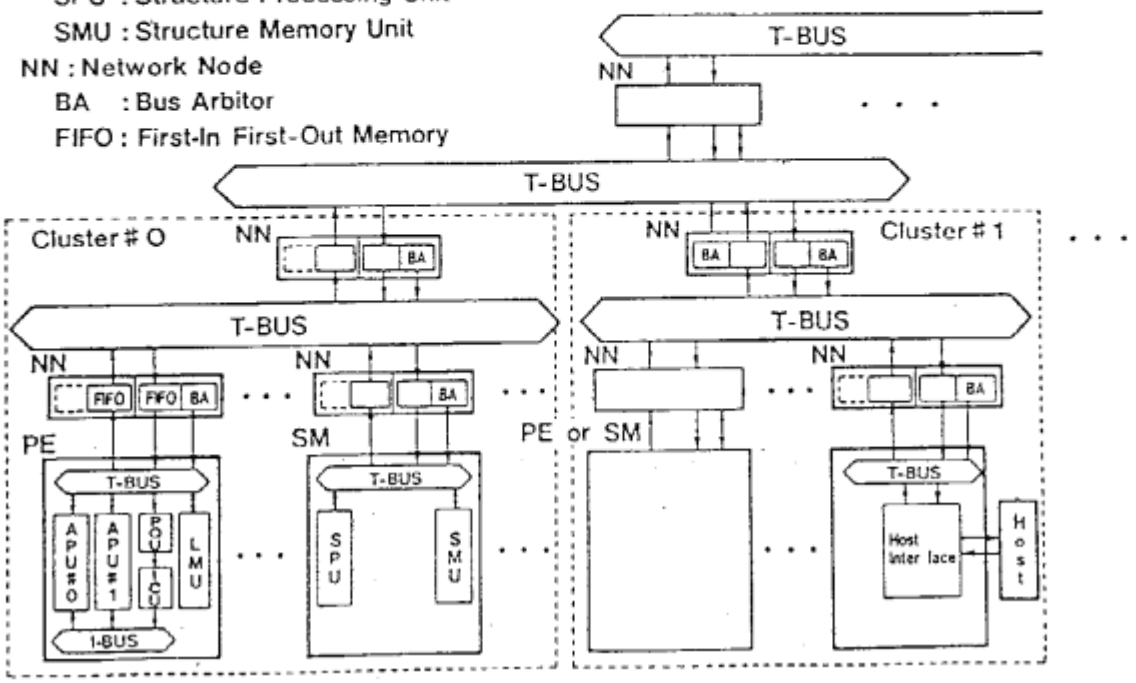


Fig.6 Machine Architecture of PIM-D

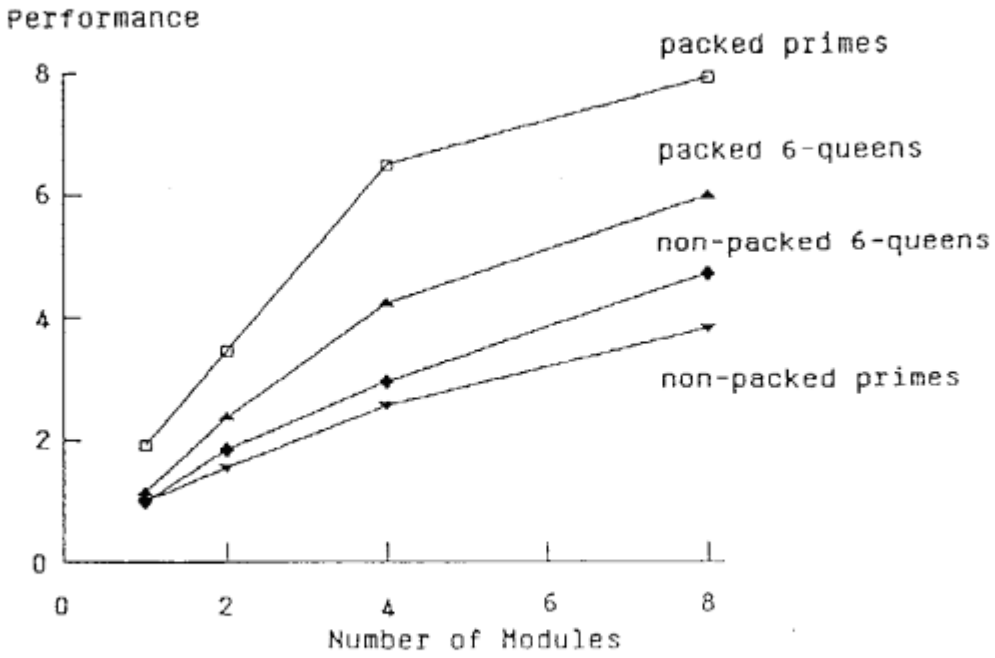


Fig.7 Performance Comparison between Non-packed and Packed Streams