Invited Paper

A Data Flow Machine Architecture for Highly Parallel Symbol Manipulations

MAKOTO AMAMIYA, RYUZO HASEGAWA, MASARU TAKESUE and HIROHIDE MIKAMI

The data flow architecture has good features as a basis for highly parallel symbol manipulation machine. We have proposed an architecture of list-processing-oriented data flow machine, called DFM, and built a prototype machine.

This paper reports the implementation and evaluation of the DFM architecture. First, the paper clarifies the parallel and pipeline executions structure inherent in list processing written in a functional language. Then, the paper offers the DFM architecture which exploits the parallelism inherent in the list processing. And last, evaluation of DFM architecture is shown through the prototype machine implementation and register transfer level simulation of the DFM.

It is pointed out that the data flow machine is very effective for implementing the parallel and pipeline executions in list processing. The data flow machine can fully exploit the parallelism inherent in list processing, due to the ultra-multi-processing mechanism, packet communications-based parallel and pipeline executions mechanism, and lenient cons mechanism for non-strict structure data manipulation.

1. Introduction

A new machine architecture which achieves high performance in symbol manipulations is required in order to realize advanced AI systems [1, 2]. This new architecture must satisfy the following requirements: First, parallelism inherent in problems, especially the parallelism in symbol manipulations, must be exploited to achieve high performance; second, functional language, which is essential to improving software productivity for complex systems such as AI systems, should be executed efficiently; third, it should be relatively easy to implement a distributed control mechanism and thus make use of a large number of VLSI devices.

Data flow architecture [3] is very attractive for meeting these requirements, and researches are pursued in the United States [4, 5], in Europe [6] and in Japan [7-11]. Parallel list processing using the lenient cons concept has been proposed along with the data flow machine architecture needed to support such processing, and several issues concerning the architecture have already been discussed [11-15].

The list-processing-oriented data flow machine can exploit the parallelism inherent in list processing in the following ways:

(1) The data driven control mechanism realizes ultra-multi-processing facility, which allows thousands of processes to be executed concurrently.

(2) The packet communications mechanism enables parallel and pipeline executions among multiple processors.

(3) The lenient cons mechanism, which implements the non-strict operation of structure data such as list and array, enables the parallelism to be fully exploited. The data flow machine also effectively exploits parallelism in functional programs, which have a high degree of recursion [14].

First, this paper discusses the parallel and pipeline computation structures in functional programs and list processing, by classifying the evaluation structures into tree-structured evaluation and linear evaluation. Then, the data flow machine architecture, which exploits the parallelism inherent in these computation structure is proposed. The data flow machine is called DFM. And last, the evaluation of the DFM architecture is shown through the prototype DFM machine implementation and register transfer level simulation of the DFM architecture. The DFM has attractive features from the view point of high performance list processing: (1) It offers a mechanism for exploiting both function-level and instruction-level parallelism. This fine and medium grain parallelism is essential to obtain highly parallel and pipeline executions in symbol manipulation. (2) The structure memory incorporates list processing operations which employ lenient and lazy cons mechanisms. These high level operations incorporated within the structure memory enable the highly parallel and pipelined operations between structure memories and processing elements.

NTT Software Laboratories 3-9-11 Midoricho, Musashino-shi, Tokyo 180, Japan

2. Parallelism in List Processing

Parallel computation structures in list processing written in functional languages are discussed in this section.

2.1 Parallelism in Functional Programs

Parallelism can be achieved on the function activation level in two ways; parallel evaluation of function arguments and eager evaluation of the function body.

(1) Parallel evaluation of function arguments

In the function application $f(e_1, e_2, \ldots, e_n)$, all arguments $e_1, \ldots e_n$ are evaluated in parallel. The parallel evaluation of arguments induces a high degree of parallelism, because the functional program has a deeply nested function application structure.

(2) Eager evaluation of function body

The execution of a function is initiated when one of the arguments of the function is evaluated, and the evaluation of the caller function resumes its execution when one of the return values is obtained during the execution of the applied function. This eager evaluation scheme, in combination with the parallel evaluation of arguments, enables concurrent computation between caller and callee functions, thus inducing a greater degree of parallelism.

2.2 Recursive Computation and Parallelism

Parallelism is closely related to the recursive computation structure of functional programs. These recursive structures can be classified into either tree structure or linear structure, and parallel computation can be exploited for both computation structures.

(1) Tree-structured evaluation and parallel computation

A typical example of tree-structured evaluation is the divide-and-conquer algorithm. In divide-and-conquer computation, multiple function activations are carried out for evenly divided sub-problems. The following is an example of a divide-and-conquer algorithm.

function *solve* (*problem*)
$=$*construct* ($r_1, r_2, \ldots r_n$)
where {($prob_1, prob_2, \ldots, prob_n$)
$=$*divide* (*problem*),
r_1=*solve* ($prob_1$),
r_2=*solve* ($prob_2$),
$\ldots$
r_n=*solve* ($prob_n$)}

In this scheme, the function *divide* generates n equal-sized sub-problems $prob_1, \ldots, prob_n$, and the function *construct* creates a final result from the partial results $r_1, r_2, \ldots, r_n$. A high degree of parallelism exists in this scheme since multiple instances of the *solve* function are activated at each recursion level. This computation structure is shown in Fig. 1.

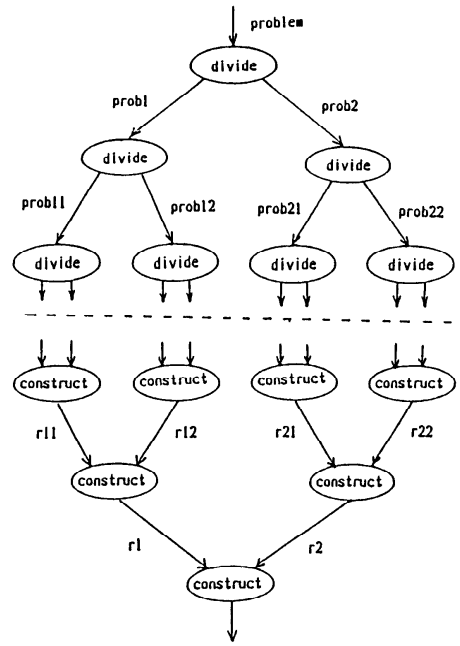


Fig. 1 Tree-structured Computation Scheme.

Merge-sort, quick-sort, and various recursive doubling computations using the divide-and-conquer method have this computation structure. OR-parallel evaluation of logic programming languages also falls into this category [16].

(2) Linear evaluation and pipeline computation

Linear evaluation occurs in linear recursion. The following is an example of linear evaluation.

function *solve* (*problem*)
$=$*construct* (res_1, res_2)
where {(*firstprob, restprob*)=*split* (*problem*),
res_1=*solve* 1 (*firstprob*),
res_2=*solve* (*restprob*)}

In this scheme, the function *split* divides the problem into the first sub-problem (*firstprob*) and the rest of the sub-problems (*restprob*). *Firstprob* is solved directly by the function *solve* 1, while the evaluation of *restprob* is carried on recursively with the function *solve*.

Multiple instances of *solve* 1 are activated successively and executed in parallel for the split sub-problems. If the construct operation is non-strict, a greater degree of parallelism will be possible due to the highly concurrent computation, i.e., a stream-oriented computation scheme, in which partial results are created and returned before result res_2 is obtained from *solve* (*restprob*). This computation structure is shown in Fig.

2.

It should be noted that the following tail-recursive scheme is a practical version of linear evaluation.

function *solve* (*problem, pares*)
=*solve* (*restprob, res*)
where {(*firstprob, restprob*)=*divide* 1 (*problem*),
res_0=*solve* 1 (*firstprob*),
re_s=*attach* (*pares, res*$_0$)}

The bubble-sort program and various set manipulation programs have this computation structure. AND-stream evaluation of logic programming languages also falls into this category [16].

2.3 Data Structures and Parallel Computation

In list processing, the evaluation structure is closely related to the list's data structure. Since the list's data structure is defined recursively, and the function is applied to this data structure, the function can be evaluated recursively thus reflecting the data structure. This scheme is described as follows,

$$f([l_1.l_2])=[f(l_1).f(l_2)]$$

where $[l_1.l_2]$ represents a cons list, i.e., *head* $([l_1.l_2])=l_1$ and *tail* $([l_1.l_2])=l_2$.

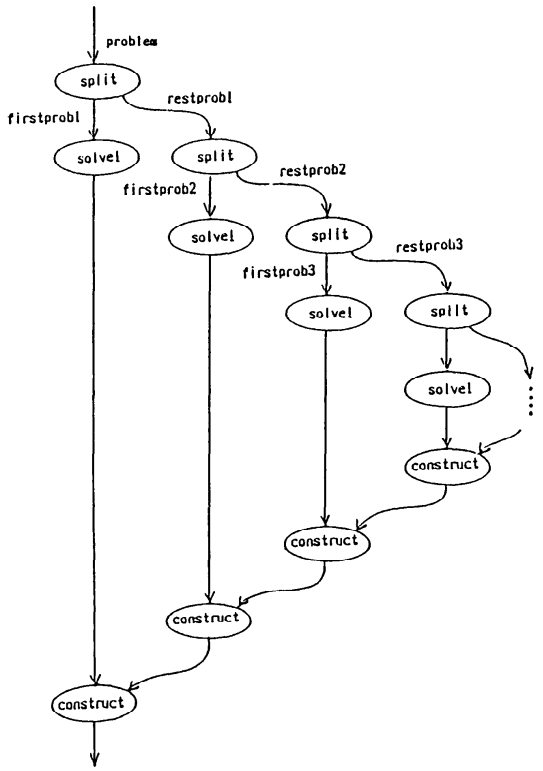


Fig. 2 Linear Computation Scheme.

In this example, the results of function applications $f(l_1)$ and $f(l_2)$ are constructed to a final value $[f(l_1).f(l_2)]$.

If l_1 and l_2 are equal sized lists and the recursive function is applied to each list element, the resulting evaluation will be a tree-structured evaluation. If l_1 is an atom and l_2 is a list, i.e., $[l_1.l_2]$ is a linear list, then the function applied to each list element (e.g., apply-all or mapcar) can be described as a linear recursive structure,

$$\begin{aligned} f([a_1, a_2, \ldots a_n]) &= f([a_1.[a_2.[\ldots.[a_n.[\,]]]]]) \\ &\Rightarrow [f_1(a_1).f([a_2.[\ldots.[a_n.[\,]]]])] \\ &\Rightarrow [f_1(a_1).[f_1(a_2).f([a_3.[\ldots.[a_n.[\,]]]])]] \\ &\Rightarrow \ldots \\ &\Rightarrow [f_1(a_1), f_1(a_2), \ldots f_1(a_n)], \end{aligned}$$

where f is defined as $f([a.l])=[f_1(a).f(l)]$ and $f([\,])=[\,]$.

If a non-strict cons operation is introduced, $f_1(a_1), \ldots f_1(a_n)$ can be carried out as a stream, and highly concurrent execution is obtained between the producer and consumer functions. Such producer and consumer concurrency is depicted in the following,

$$\begin{aligned} g(f([a_1, a_2, \ldots a_n])) &= g(f([a_1.[a_2.[\ldots.[a_n.[\,]]]]])) \\ &\Rightarrow [g_1(f_1(a_1)).g(f([a_2.[\ldots.[a_n.[\,]]]]))] \\ &\Rightarrow [g_1(f_1(a_1)).[g_1(f_1(a_2)).g(f([a_3.[\ldots.[a_n.[\,]]]]))]] \\ &\Rightarrow [g_1(f_1(a_1)), g_1(f_1(a_2)), \ldots g_1(f_1(a_n))] \\ &= [c_1, c_2, \ldots c_n] \end{aligned}$$

where f and g are defined as

$$f([a.l])=[f_1(a).f(l)] \text{ and } f([\,])=[\,],$$
$$g([b.l])=[g_1(b).g(l)] \text{ and } g([\,])=[\,].$$

In this example, list $A(=[a_1, \ldots a_n])$ is transformed to list $C(=[c_1, \ldots c_n])$ by functions f and g. The function f generates an intermediate result $f(A)(=[f_1(a_1), f_1(a_2), \ldots f_1(a_n)])$, and the function g consumes the result to generate the final result C. In this computation, partial results from the producer f are passed to the consumer g, and f and g are executed concurrently.

2.4 Data Flow Computation and Lenient Cons Concept

All parallel computation structures discussed above are maximally exploited by data flow computation and lenient cons mechanisms. In addition to parallelism on the primitive operation level, data flow computation can extract parallelism on the function application level, since it enables parallel evaluation of function arguments and eager evaluation of the function body.

The lenient cons mechanism, by which a non-strict cons operation is implemented in the data flow computation framework, extracts the maximal parallelism in list processing. Configurations of the parallel and pipeline computations on list data structures are depicted in Fig. 3. For more detailed discussions, see [11, 12, 13, 17].

2.5 Examples

Typical examples for parallel and pipeline executions are depicted in the following. The programs are written in **Valid** [15, 18].

(1) Tree-structured evaluation

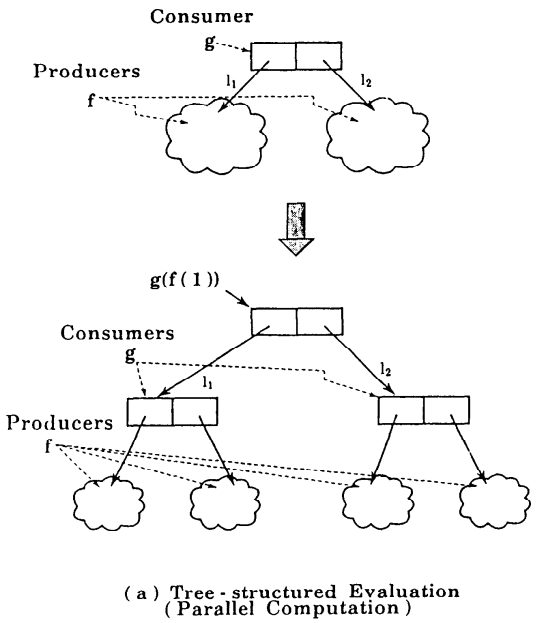
A typical example is the well known Quick-sort program.

function *sort(x)* **yield** *(list)*
=**if** *null(x)* **then** [ ]
else *append(sort(y1), append(y2, sort(y3)))*
where {*y*=*head(x)*,
(*y*1, *y*2, *y*3)=*partition(tail(x),y)*}.
function *partition(x, y)* **yield(***list, list, list***)**
=**if** *null(x)* **then** ([ ], [*y*], [ ])
else case
{*x*1=*y* → (*w*1, [*x*1.*w*2], *w*3),
*x*1 < *y* → ([*x*1.*w*1], *w*2, *w*3),
*x*1 > *y* → (*w*1, *w*2, [*x*1.*w*3])}
where {(w_1, *w*2, *w*3)=*partition(y*1, *y*),
(*x*1, *y*1)=(*head(x)*, *tai(x)*)}
function *append(x, y)* **yield(list)**
=**if** *null(x)* **then** *y*
else [*head(x)*.append(*tail(x)*, *y*)].

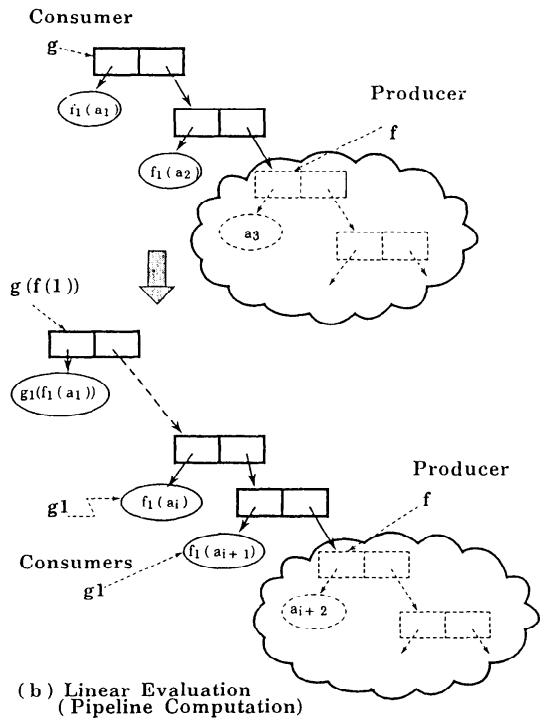
In this program, the function *partition* in sort body divides a list into three lists, *y*1, *y*2, *y*3, each of which contains elements less than, equal to, and greater than the first element. As the *sort* and *append* are activated immediately after each of *y*1, *y*2, *y*3 is generated, it is expected that the maximal parallelism among functions is obtained. However, parallelism by partial execution of function body does not work well for reducing the execution time in the order, since the time spent to sort the list of length n is proportional to the square of n in the worst case. The reason is that since each of the values *y*1, *y*2, and *y*3 is not returned until the append operation is completed in the partition body, the execution of the *sort* function, which uses those values, must wait until they are returned, the waiting time is proportional to the length of the list data made by the *append* function.

If the former parts of the list, which are partially generated, are returned in advance during the period when the latter parts are appended, the execution which uses the former parts of the list can proceed. Thus the producer and consumer operations overlap each other. As the append is the repeated application of cons, as the prorgam shows, this problem can be solved by the lenient cons mechanism.

As the result, the quick-sort program can sort a list in linear time, if sufficient processing elements are pro-



(a) Tree-structured Evaluation (Parallel Computation)



(b) Linear Evaluation (Pipeline Computation)

Fig. 3 Parallel and Pipeline Executions on List Data.

vided.

(2) Linear evaluation

Typical examples are set manipulations such as union, intersection, deletion and complement, and list compactions in which the duplicated list elements are filtered out. It should be noted that the set is represented as linear list.

function *union*(*u, v*) **yield**(*set*)
  =**if** *null*(*u*) **then** *v*
    else if *member*(*y,v*) **then** *x* **else** [*y*·*x*]
  where {*x*=*union*(*tail*(*u*), *v*),
      y=*head*(*u*)}.
function *member*(*e, v*) **yield**(*Boolean*)
  =**if** *null*(*v*) **then false**
    elsif *e*=*head*(*v*) **then true**
    else *member*(*e, tail*(*v*)).

The computation structure of the union program is depicted in Fig. 4. The *union* function checks whether the head of u, which is denoted by y, is a member of the set v. At the same time, the *union* function is applied recursively to the rest of u. Since the invocation of the *union* function is carried on in concurrent with the execution of the *member* function, executions of *union* and *member* overlap each other. By using the lenient cons mechanism, partially calculated union set is returned immediately after the execution of *member* function is terminated in the deepest recursion phase of *member* function. Thus, due to the lenient cons mechanism and concurrent execution between *union* and *member* functions, union of two sets, which would consume square time unless the lenient cons is used, can be obtained in linear time.

Another example is the list compaction, in which the effect of the lenient cons and stream processing scheme is more intuitive. The list compaction program is described in the following.

function *compaction*(*u*) **yield**(*set*)
  =**if** *null*(*u*) **then** [ ]
    else [*head*(*u*).*compaction*(*y*)]
    where {*y*=*remove*(*head*(*u*), *tail*(*u*))}.
function *remove*(*e, v*) **yield**(*list*)
  =**case** {*null*(*v*) → [ ],
      e=*head*(*v*) → *remove*(*e, tail*(*v*)),
      others → [*head*(*v*).*remove*(*e, tail*(*v*))]}.

The computation structure of the compaction program is depicted in Fig. 5. The well known prime number sequence calculation by Eratosthenes's sieve method has essentially the same computation structure [12].

3. Machine Architecture and its Implementation

3.1 Issues in Parallel List Processing

Several problems remain to be solved in the development of parallel list processing machines. These include:

(1) Ultra-multi-processing: Processors must have an ultra-multi-processing facility, in which thousands of processes can be executed simultaneously. In parallel list processing, the medium and fine grain process concurrencies should be implemented effciently within a processor, since a large number of function instances are created dynamically. These instances are treated as concurrent processes.

(2) Load balancing: In multi-processor systems which support ultra-multi-processing, the dynamically created processes should be allocated evenly to each processor, to obtain load balance among processors. Dynamic process allocation control is essential in order to achieve load balance during execution.

(3) Inter-processor communications: Another problem is how to reduce the interprocessor communications overhead, because communications among processes (function instances) assigned to different processors is essential in function linkage. One way to eliminate this overhead is to make process-processor assignments in such a way as to reduce inter-processor communications. Another is to make the inter-processor data transfer overlap the intra-processor execution.

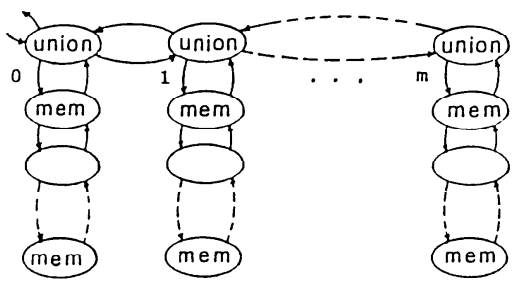


Fig. 4 Parallel Computation Structure of Union Program.

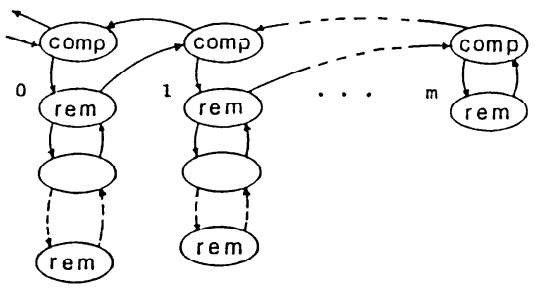


Fig. 5 Parallel Computation Structure of Compaction Program.

(4) Memory access latency: Multi-processor systems must have a common memory if they handle dynamic data structures. Such multi-processor systems with shared memory have a memory access latency caused by memory access contention. This latency is much more serious in systems with high level memory operations such as in list processing, since the latency adversely effects the performance of linear list traversing.

3.2 Design Philosophy and Basic Architecture

Packet-communications-based pipeline systems can be used to solve the problems of ultra-multi-processing (e.g., process switching overhead), memory access latency, and interprocessor communications overhead. The data driven control mechanism makes it easier to design such packet-communication-based pipeline systems, i.e., a single pipeline within a processor and multiple pipelines between processors and memories.

There are, however, problems which remain unsolved in designing data flow machines. These are: (1)how to implement history-sensitive operations such as I/O and memory read-write operations, (2)how to handle structure data effectively, and (3)how to reduce the hardware implementation costs, especially in operand matching memory implementation.

The list-processing-oriented data flow machine, called DFM was designed as a solution to the problem of parallel list processing and data flow machine implementation. The DFM design philosophy is as follows:

(1) The DFM exploits parallelism on both the function application level and instruction level. Function level parallelism is extracted through parallel execution among multiple processors, while instruction level parallelism is extracted through pipeline execution among hardware modules in a processor and memory.

(2) The cost of implementing an operand matching memory is reduced by extracting a computational locality. In the DFM architecture, each instance of a function is treated as a local context for evaluation, and the operand matching memory is designed on the basis of the semi-content-addressable memory concept which makes use of the local context. The operand matching memory is constructed with a number of content addressable memory (CAM) blocks. The name of the activated function (instance name) is used as the address of its CAM block, and the instruction identifier (address) in a function body is used as an access key for operand matching in the CAM block. The CAM block consits of only 32 words, and thus the CAM block is built in low cost using off-the-shelf devices.

(3) List operations (e.g., Lisp's primitive functions) are treated as structure memory (SM) access operations, and they are executed within SM. The SM is one of the hardware modules in a function unit, and is designed to execute instructions issued from processor (PE). Pipelining between the PE and SM would solve the memory access latency problem. The SM is also divided into multiple banks which can be accessed from any PE, thus reducing memory access contention by distributing the access demand.

(4) Work load information of a processor is given as a combination of the number of activated functions and the number of packets circulating in the processor pipeline. These work load factors can easily be measured by counting the number of activated instances and the number of token packets circulating in the pipeline.

The architecture of the DFM is shown in Fig. 6. PEs (Processing Elements) are data flow processors each of which is implemented as a circular pipeline consisting of a single IM (Instruction Memory Unit), OM (Operand Matching Memory Unit) and FU (Function Unit). SMs (Structure Memories) store list data and execute list operations. PEs and SMs are connected through a multi-stage packet switching network.

The highly parallel and pipeline execution described in Section 2 is thus implemented effectively by the DFM architecture, in which the parallel and pipeline executions are carried on among numerous circular pipeline processors and structure memories.

The prototype DFM machine was designed and built for the purpose of detailing and evaluating the DFM architecture. The machine configuration is shown in Fig. 7. In the practical implementation, common buses are used instead of a multistage packet switching network. A CCU (Cluster Control Unit) is also introduced which provides dynamic process allocation and functions as a host machine interface. For more precise information of prototype DFM hardware implementation, see [17].

4. Evaluation of DFM Architecture

The prototype DFM hardware and its RTL (Register Transfer Level) simulator are used for complementary evaluation of the DFM architecture. The former offers absolute performance data, while the latter simulates the multiple PE-SM system on the register transfer level.

4.1 Benchmark Programs

Performance of the DFM architecture was evaluated using several list processing benchmark programs. The times needed to execute the programs were measured and compared with that of conventional machines. Benchmark programs were written in **Valid**. Those **Valid** programs are compiled into the DFM machine code by **Valid** compiler [18]. Same programs were also written in Lisp, and their compiled codes were executed on conventional machines (e.g., VAX-11/750 and DEC-2060).

The benchmark programs used for evaluation are:

(a) tree-evaluation type: *sigma, merge-sort, quick-sort.*

(b) linear-evaluation type: *bubble-sort, sieve, union, list-compaction.*

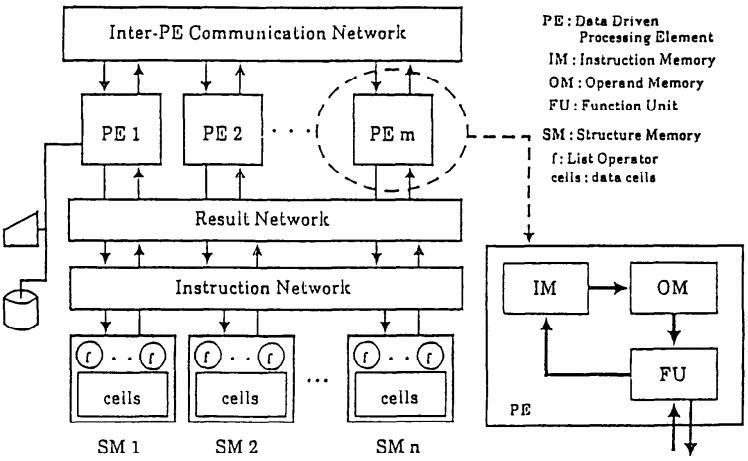


Fig. 6 DFM Architecture.

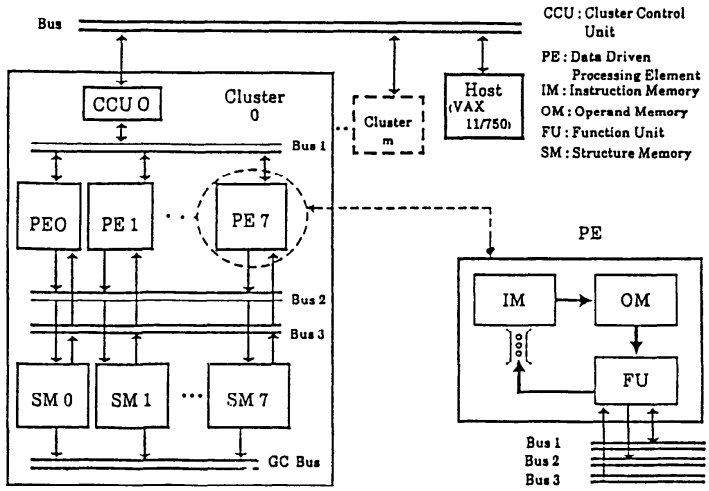


Fig. 7 DFM Prototype Machine.

(c) mixture of (a) and (b): *parse*.

The sigma program, which calculates summation of integers from 1 to *n* by the divide-and-conquer method, is adopted as a typical example of the well known recursive doubling computation structure. It should also be noted that set manipulation programs such as intersection, deletion, and complement have the same linear-evaluation structure as the *union* program. The *compaction* program eliminates duplicated list elements. The *sieve* program calculates prime numbers using *Eratosthen's sieve* method. The *parse* program is a syntax analyzer for simple arithmetic expressions.

All programs are composed of relatively small recursive function bodies, and 40–50% of the instructions in their compiled codes are related to function linkage. List operation instructions amount to about 10%. The program code size of the lenient cons version is 10-20% larger than that of the conventional cons version. Almost all the remaining instructions are switch and gate operations.

4.2 Performance of Single PE-SM System

The performance of a single PE-SM system was measured on the prototype DFM machine and the result was compared with that of conventional machines. Some of the result data are shown in Table 1.

Compared with the performance of list processing by conventional machines, the single PE-SM system is 5 to 7 times faster than the VAX11/750 and half as fast as the DEC-2060. These results demonstrate that the circular pipeline mechanism of the data flow processor is effective in achieving ultra-multi-processing and low memory access latency.

4.3 Performance of Multi-PE and Multi-SM System

The most important factor in the performance of

Table 1 Performance of Single PE-SM System

program	V (μsec)	M (μsec)	DFM		V/DFM		M/DFM	
			C(μsec)	L(μsec)	V/C	V/L	M/C	M/L
bsort (60)	776,860	88,200	121,188	79,147	6.41	8.00	0.73	0.91
msort (60)	196,833	11,233	27,677	29,932	7.11	6.58	0.41	0.38
qsort (60) Best	214,800	12,500	26,435	29,678	8.13	7.24	0.47	0.42
union (40, 40)	509,223	21,923	45,667	45,948	11.2	11.1	0.48	0.48
compact (60) A	25,823	1,530	8,376	6,064	3.08	4.26	0.18	0.25
compact (60) B	415,133	54,710	118,717	87,820	3.50	4.73	0.46	0.62
arith	29,193	2,118	4,469	4,820	6.53	6.06	0.47	0.44

V: VAXLISP(on VAX 11/750)
M: MACLISP(on DEC 2060)
C: Conventional cons
L: Lenient cons

multiple PEs multiple SMs systems is the linear speed-up ratio which is related to the number of PEs and SMs. This is given by the coefficient α in $p = \alpha^* m$, where m is the number of PEs or SMs, and p is the performance normalized with $m = 1$. The same programs examined on the single PE-SM system are also examined on the RTL simulator to evaluate the linear speed-up in multi-PE-SM systems.

(1) Linear speed-up in tree-structured evaluation

The effect of divide-and-conquer computation on linear speed-up is shown in Fig. 8. The divide-and-conquer program *sigma(n)* yields a linear speed-up of $\alpha = 0.87$ for $m < 32$ in the case of $n = 1024$. This data shows good linearity, since the theoretical upper limit on α is 0.92 for $n = 1024$ and $m = 32$.

The linear speed-up in list processing is not explicit for tree-structured evaluation (Fig. 9(a)). The reason is that there was not enough data volume to determine the system performance, because of the capacity of the simulator. When $n = 60$, which is the upper limit of the simulation's capacity, there is too little data in the simulation for the multi-PE-SM system to take advantage of the pipeline effect for *quick-sort* and *merge-sort* programs. If a larger size of data is fed into the system, the linear speed-up will be more explicit.

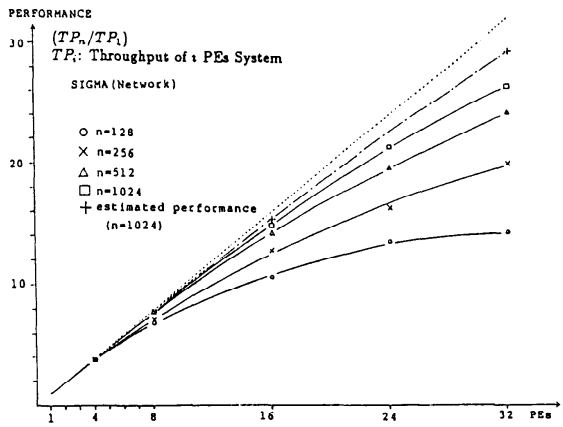


Fig. 8 Performance of Multi-PE-SM system (Sigma Program).

(2) Linear speed-up in linear evaluation

Linear speed-up is typically achieved in linear evaluations as shown in Fig. 9(b). This demonstrates that ultra-multi-processing and memory access latency problems in multiprocessor systems can be solved by the DFM architecture. In the *union* program, which has less data dependency, linear speed-up can be achieved both for conventional and lenient cons due to the concurrent computation among processes. On the other hand, for *bubble-sort* and *compaction* programs which have data dependency, the effect of lenient cons on linear speed-up is remarkable. In *list-compaction* programs, parallelism is determined by the nature of the input data. Case(A), in which all 60 list elements are different, has a parallelism of degree 60, while case(B), in which only four different elements are duplicated, has a parallelism of degree four.

A linear speed-up of $\alpha \approx 0.7 \sim 0.9$ was achieved for all of the benchmark programs.

(3) Linear speed-up for mixed evaluation

The lenient cons effect is also remarkable for mixed evaluation. The *parse* program achieved a linear speed-up of $\alpha \sim 0.6$ in the lenient cons case. If a larger size of data is put into the system, α will also increase, as in the case of tree-structured evaluation.

5. Conclusions

Highly parallel and pipeline computation structures

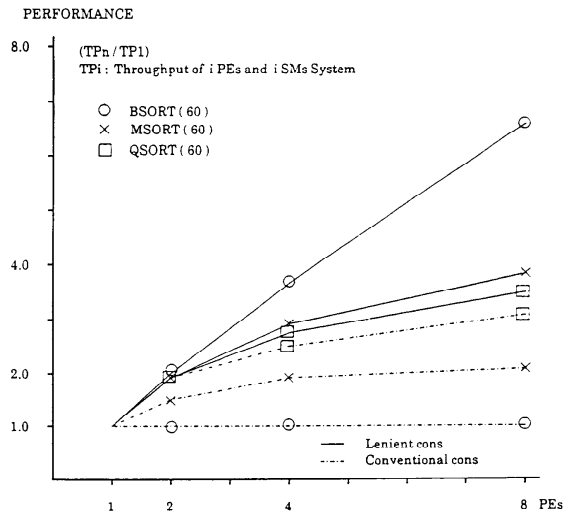


Fig. 9(a) Performance of Multi-PE-SM System.

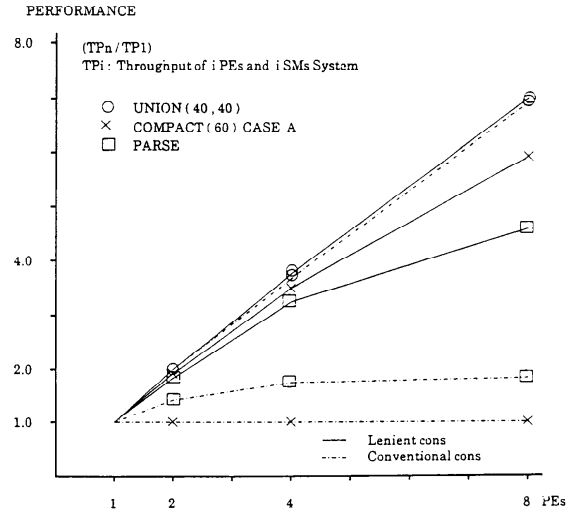


Fig. 9(b) Performance of Multi-PE-SM System

were discussed for the structured data such as the list. The computation structures were classified into tree-structured evaluation and linear evaluation. Then, it was shown that parallel and pipeline executions could be fully exploited by using data flow computing scheme with lenient cons concept i.e., divide-conquer execution for tree-structured evaluation and stream-oriented computation for linear evaluation.

The data flow architecture, which exploits the parallel and pipeline computation structures inherent in functional programs applied for list processing, was also proposed and issues for its practical implementation was described.

The data flow prototype machine, called DFM, has been implemented, and DFM performance was evaluated in a simulation of the register transfer level using several benchmark programs [17]. The DFM single processor system was shown to be several times faster than conventional sequential machines which use the same device technology, and a multi-processor DFM system was shown to achieve a linear speed-up ratio of $0.6 \sim 0.9$.

The **Valid** compiler has been developed, and **Valid** source programs are compiled into DFM machine code and executed on the DFM prototype machine automatically. Currently, various benchmark programs are written and tested on the DFM machine.

Next step of the research is:

(1) to build a DFM prototype II system, which will be constructed with hundreds of PEs and SMs, using VLSI chips, and

(2) to develop a coordinated computation system, a parallel-based object-oriented programming paradigm, on the basis of the **Valid** language system and the DFM machine architecture. The parallel-based object-oriented programming is a solution to the history sensitivity problem in functional programming.

Acknowledgement

The authors wish to thank Mr. T. Naruse and Mr. M. Yoshida, for their efforts in implementing the DFM prototype machine.

References

1. T. MOTO-OKA ed., Fifth Generation Computer Systems: Proc. International Conference on Fifth Generation Computer Systems, North-Holland (1981).
2. B. W. Wah, ed, New Computer Architectures for Artificial Intelligence Processing, *IEEE Computer*, **20**, 1 (1987).
3. J. B. DENNIS. A Preliminary Architecture for a Basic Data Flow Processor, *The Second Annual Symposium on Computer Architecture*, *IEEE*, pp. 126-132 (1975).
4. ARVIND, K. P. GOSTELOW and W. PLOUFFE. An Asynchronous Programming Language and Computing Machine *Report TR* 114a, *Department of Information and Computer Science*, University of California, Irvine, California, December, (1978).
5. R. M. KELLER, G. Lindstrom and S. PATIL, An Architecture for a Loosely-Coupled Parallel Processor, *UUCS*-78-105, *University of Utah*, Salt Lake City, Utah, 1978.
6. I. WATSON and J. GURD, A Prototype Data Flow Computer with Token Labeling, *Proc. of NCC, AFIPS* (1979) 623-628.
7. N. TAKAHASHI and M. AMAMIYA. A Data Flow Processor Array System: Design and Analysis, *Proc. 10th Ann. Int. Symp. Computer Architecture, IEEE* 1983, 243-250.
8. T. SHIMADA, K. HIRAKI and K. NISHIDA. An Architecture of a Data Flow Machine and Its Evaluation, *Proc. COMPCON* 84 (Spring) *IEEE* (1984), 486-490.
9. Y. YAMAGUCHI, K. TODA and T. YUBA. A Performance Evaluation of a Lisp-based Data-driven Machine (EM-3), *Proc. 10th Ann. Int. Symp. Computer Architecture, IEEE* (1983), 363-369.
10. N. ITOH, Y. MASUDA and H. SHIMIZU. Parallel Prolog machine based on Data Flow Model, *ICOT Tech. Rep. RT*-035, *ICOT* (1983).
11. M. AMAMIYA, R. HASEGAWA, O. NAKAMURA and H. MIKAMI, A List-processing-oriented Data Flow Machine Architecture, *Proc. NCC*, 51, *AFIPS* (1982) 143-152.
12. M. AMAMIYA, R. HASEGAWA and H. MIKAMI. List Processing with a Data Flow Machine, *Lecture Notes in Computer Science*, Springer-Verlag (1983), 165-190.
13. M. AMAMIYA and R. HASEGAWA, Data Flow Computing and Eager and Lazy Evaluation, *New Generation Computing*, **2**, 8, (1984), 105-129.
14. R. HASEGAWA, H. MIKAMI and M. AMAMIYA, A List-processing-oriented Data Flow Machine Architecture and Its Evaluation, *Trans. IECE Japan*, **67**-D, 9, (1984), 957-964.

15. M. AMAMIYA, R. HASEGAWA and S. ONO. VALID: A High-Level Functional Language for Data Flow Machine, *Rev. Electrical Communications Laboratories*, **32**, 5, (1984), *NTT* 793-802.
16. R. HASEGAWA and M. AMAMIYA. Parallel Execution of Logic Programs Based on Data Flow Concept, *Proc. Int. Conf. Fifth Gen. Computer Systems*, ICOT (1984), 507-516.
17. M. AMAMIYA, M. TAKESUE, R. HASEGAWA and H. MIKAMI, Implementation and Evaluation of A List-Processing-Oriented Data Flow Machine, *Proc. 13th Ann. Int. Symp. Computer Architecture, IEEE* (1986), 10-19.
18. R. HASEGAWA and M. AMAMIYA. Design and Implementation of A High Level Functional Language **Valid** for Data Flow machine, *Proc. 1986 Riken Symposium on Functional Programming*, FP86-01, Japan Society for Software Science and Technology (1986).

(Received October 27, 1987)