Data Flow Computing and Parallel Reduction Machine

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

This paper discusses a parallel graph reduction model and its implementation in the relation to the data flow computing scheme. First, a parallel graph reduction mechanism and its relation to data flow computing are discussed. In the discussion, the data flow computing model, which introduces by-reference concept is shown to be a natural implementation of parallel graph reduction model. Then, a practical implementation of a parallel reduction machine is presented. In the implementation, a cell token flow model is proposed. By using the cell token flow concept, a given data flow program can be transformed to an efficient multi-thread control flow program. Last, an overview of the machine architecture is described.

1. Introduction

Recently, several machines were proposed for both serial and parallel reductions [1–9]. Almost all serial reduction machines are based on the stack control mechanism, and are much more practical than those for currently considered parallel reduction machines. The most serious obstacle for the current implementation of the parallel reduction machines is that they lack efficient parallel reduction control.

The data flow computing scheme offers a basis for parallel execution control in the most natural way [10,11,12]. Data flow based computation control can take advantage of parallelism quite naturally, since all operations are controlled according to data dependency relation. Programmers can describe the precise synchronization control using data flow concept.

Control in the reduction model lies in the choice of reduction strategy, i.e. the order of reduction. The best known reduction strategies are applicative order reduction and normal order reduction. In general, applicative order reduction is more efficient, while normal order reduction is safe for nonstrict evaluation. Therefore, it is suggested that the normal order reduction should be used only in a few parts where nonstrict evaluation is needed, i.e. lazy evaluation should be performed only where it is necessary.

This paper discusses the relationship between reduction and data flow computing, and proposes a model and implementation of a parallel graph reduction machine based on the data flow computing scheme. It is shown that the conventional by-value data flow computing scheme implements the applicative order reduction strategy. In the proposed model, the data flow computing scheme is extended so as to implement the normal order reduction, by introducing by-reference concept. The extended data flow computing model carries the data flow graph itself, which is called meta-data. It is also pointed out that the data flow computing model naturally implements the super-combinator [13].

Then, the paper discusses a parallel reduction machine implementation. The machine architecture is designed on a multi-thread control flow concept. The multithread control flow program, which is extracted from the given data flow program, reflects the data flow structure of the given program, and yet eliminates the need for redundant data flow controls, such as switch and gate controls, and overhead for operand matching. This makes the data flow machine much more efficient and practical. Last, an overview of the parallel reduction machine architecture is described. The machine is designed on the basis of packet

North-Holland
Future Generations Computer Systems 4 (1988) 53–67

communications concept, in which many processor and memory resources are connected with packet communications network. The processor is designed as a circular pipeline system which can execute the multi-thread control flow program in high speed.

2. Graph Reduction

Essential constructs of functional programming language are expressions and functions. Execution of a functional program consists of evaluation of expressions. The evaluation of expression is interpreted elegantly as a reduction process.

Graph reduction is performed on a graph which is a representation of a given functional program. Graph reduction is stated briefly as follows.

(1) Expression has a natural representation as a tree, or more generally a graph.

(2) Evaluation proceeds by means of sequence of simple steps, called reduction.

(3) Each reduction performs a local transformation of the graph. This process is called graph reduction.

(4) Reduction for subexpression may safely take place simultaneously since they do not interfere with each other.

(5) Evaluation is complete when no further reducible expressions.

Function application is also an expression. The reduction process for the function application substitutes it with its function body, in which bound variables are substituted with actual values. Then the reduction is performed to the function body.

We will inspect the graph reduction process using a simple example. Expression $(5 + 3) * (5 - 2)$ is represented as a tree, and the graph reduction on this expression is depicted as Fig. 1.

$(5 + 3) * (5 - 2) \Rightarrow 8 * 3 \Rightarrow 24.$

As an example of function application, when the function f

$f = \lambda(x, y)[(x + y) * (x - y)]$

is applied to (5, 2), then reduction proceeds as

$f(5, 2) \Rightarrow (5 + 2) * (5 - 2) \Rightarrow 7 * 3 \Rightarrow 21.$

2.1. Applicative Order Versus Normal Order

There are several reduction strategies. In those, normal order reduction and applicative order reduction are typical. The normal order reduction tries to reduces an expression from outer to inner, or from top to bottom in the graph reduction. The applicative order reduction, on the other hand, reduces an expression from inner to outer, or from bottom to top in the graph reduction. For example, normal order reduction of the expression $(5 + 3) * (5 - 2)$ tries first to reduce the expression $(\ldots) * (\ldots)$. Then, in order to reduce this expression, the reductions are tried for expressions $5 + 3$ and $5 - 2$. At this point, these expressions are reducible to 8 and 3, and the outer level expression is changed to $8 * 3$. As $8 * 3$ is now reducible, this expression is reduced to 24. In this reduction process, demands for reduction propagate from outer to inner (or from top to bottom) in the first phase. When the demand has reached innermost level (or bottom), the innermost expression is reduced to normal form, which is called a value. In the second phase, the reduction resumes from inner to outer (or values are transmitted from bottom to top), and final result is obtained when the outermost reduction completes. This process is shown in Fig. 2.

In the applicative order reduction, on the other hand, reductions for the expression $(5 + 3) * (5 - 2)$ are performed to the inner most expressions $5 + 3$ and $5 - 2$ first, then the reduction is

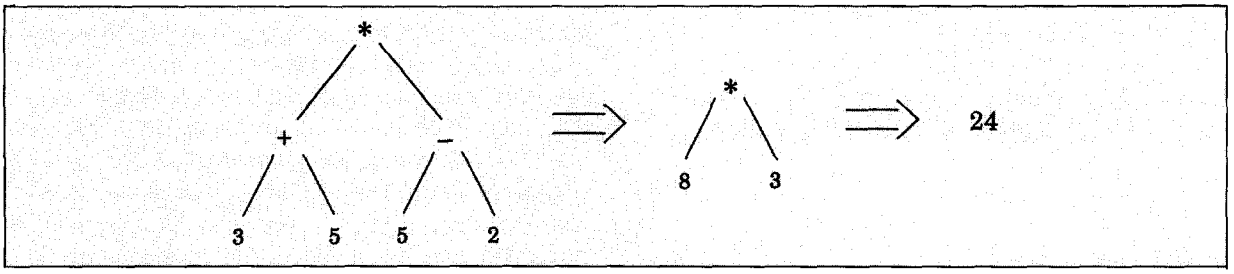

Fig. 1. Graph reduction Process.

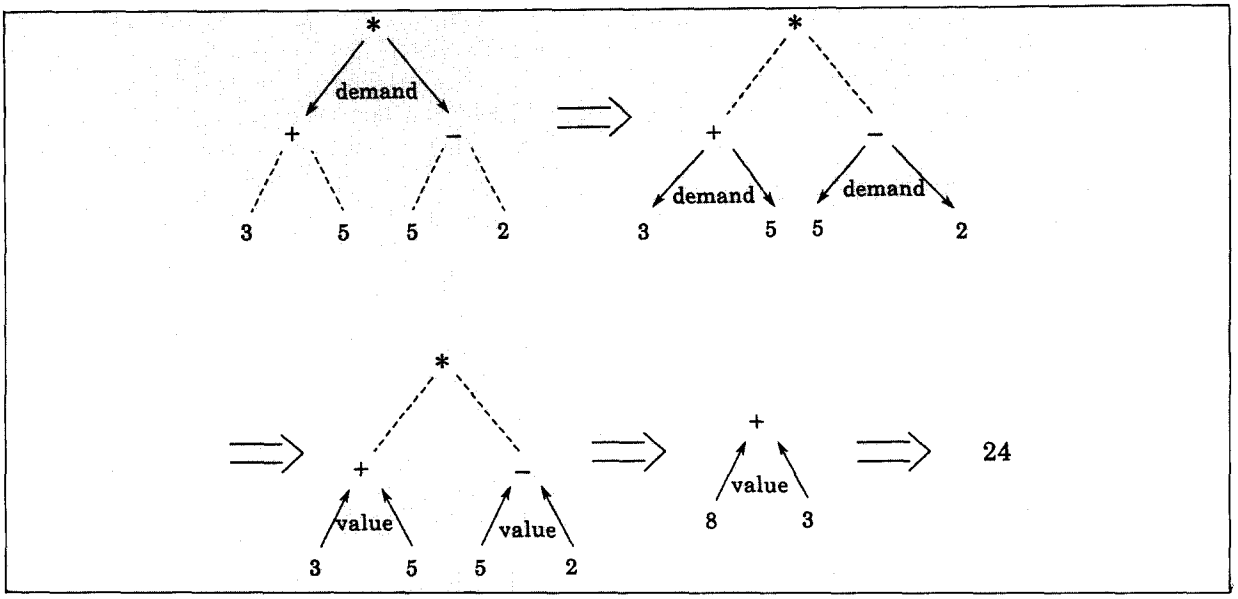

Fig. 2. Normal order reduction (demand driven).

performed to the outer level expression 8 * 3. This process is shown in Fig. 1.

We can easily recognize that applicative order reduction is more efficient than normal order reduction, since the demand propagation in normal order reduction is not needed in applicative order reduction. Nevertheless, normal order reduction is preferable because the normal order reduction is safe for nonstrict function. Typical examples of nonstrict function are conditional and structure data manipulation.

A conditional function *ite*(p, $e1$, $e2$), which is a short form of "*if* p *then* $e1$ *else* e", is defined as: If p is evaluated to *true* then its value is the value of $e1$, else its value is the value of $e2$. In the normal order reduction, the reduction of *ite* is performed to p at first. Then, according to its result, the reduction is performed to $e1$ or $e2$ selectively.

If the applicative order reduction is applied to the expression $f(0, 1)$ where $f = \lambda(x, y)[ite(x = 0, y, f(x-1, y))]$, the reduction is performed simultaneously to expressions $x = 0$, y and $f(x-1, y)$ when the reduction is performed to the body of $f(x, y)$. The last expression $f(x-1, y)$ is applied to itself recursively, and this reduction is performed to $f(x-2, y)$ again. This process continues forever.

$$
\begin{aligned}
f(0.1) &\Rightarrow ite(true, 1, f(-1, 1)) \\
&\Rightarrow ite(true, 1, (ite(false, 1, f(-2, 1))) \\
&\Rightarrow \ldots \\
&\Rightarrow ite(true, 1, (ite(false, 1, \\
&\qquad (\ldots(ite(false, 1, f(-n, 1)))\ldots))) \\
&\Rightarrow \ldots .
\end{aligned}
$$

On the other hand, if the reduction of $f(0, 1)$ is performed in normal order, $f(0, 1)$ is reduced to 1 safely.

Another example is nonstrict data structuring operations. The function *cons*(x, y) constructs a data with two elements x and y, and function *head* or *tail* take out the first or second element from the cons data. Using *cons*, *head* and *tail* functions, *intseq* and *nth* functions are defined as,

$$intseq = \lambda(n)[cons(n, intseq(n+1))],$$

$$nth = \lambda(x, n)[nth0(1, x, n)],$$

$$nth0 = \lambda(i, x, n)[ite(i = n, head(x), nth0(i+1, tail(x), n))].$$

When the expression *nth*(*intseq*(1), 2) is reduced in applicative order, the reduction is performed to *intseq*(1) first. In the course of reduction of *intseq*(1), the reduction is performed to *intseq*(2) before *cons*(1, *intseq*(2)), and this reduction is

again performed to *intseq*(3) recursively. Thus, reductions to *intseq*(n) is performed infinitely.

If the reduction is performed in normal order, *nth*(*intseq*(1), 2) is reduced to a normal form safely.

$$\begin{aligned} nth(intseq(1), 2) &\Rightarrow nth0(1, intseq(1), 2) \\ &\Rightarrow nth0(2, tail(intseq(1)), 2) \\ &\Rightarrow head(tail(intseq(1))) \\ &\Rightarrow head(tail(cons(1, intseq(2)))) \\ &\Rightarrow head(intseq(2)) \\ &\Rightarrow head(cons(2, intseq(3))) \\ &\Rightarrow 2. \end{aligned}$$

The normal order reduction is thus safe and more powerful especially for nonstrict operations. However, the problem is that the normal order reduction is less efficient than the applicative order reduction. Therefore, it is strongly suggested that applicative order reduction strategy should be used for the case in which safeness of computation is assured, while the normal order reduction should be performed only to the case in which computation will be unsafe.

It should be noted that redundant demand propagation can be eliminated at compile time by using data flow and nonstrictness analysis techniques [14,15]. Thus, almost all of the part of evaluation is performed in applicative order at run time. Normal order reduction is performed only for a few part of evaluation, in which lazy evaluation is needed.

2.2. Graph Sharing

Other important concept is sharing of the reductions. In the sharing of reductions, the reduction to an expression which is shared with several expressions is performed only once when the first demand for reduction is issued. Once the expression is reduced to a normal form, re-reduction is skipped for other demands and this normal form is used for subsequent reductions.

For example, consider the variable x which is defined as

$$x = (3 + 5) * (5 - 2).$$

If the reduction to an expression $x * x + x$ is performed in straightforward, the same reduction to $(3 + 5) * (5 - 2)$ is performed three times. But this redundancy of reduction can be avoided by sharing the result of reduction to $(3 + 5) * (5 - 2)$ as a value of variable x. This reduction sharing is elegantly controlled in the graph reduction scheme; Initially, the variable x has a pointer to the graph of expression $(3 + 5) * (5 - 2)$. Once a demand has triggered the reduction to x, the expression $(3 + 5) * (5 - 2)$ is reduced to 24, and x is rewritten with this result value. For the subsequent demands delivered to x, no reductions are performed.

2.3. Supercombinator

One of the problems in reduction is the occurrence of free variables. In order to treat these free variables, a set of variable-value binding informations, called environment, needs to be conveyed during the reduction. Managing such environments causes serious overhead. However, fortunately, it is possible to eliminate such a free variables from the original lambda expression, and transform to a lambda expression with no free variables, by using lambda-lifting technique [16]. Such a free-variable free lambda expression is called supercombinator [13].

For example, $f = \lambda(x)[x + y * y]$, which has a free variable y, is transformed into a lambda expression with no free variables $g = \lambda(w, x)[x + w * w]$ (or in supercombinator form, $g\,y\,x = +x(*\,y\,y)$). When h is defined as $h = \lambda(x)[g(5, x)]$, the reduction proceeds partially as

$$\begin{aligned} h = \lambda(x)[g(5, x)] &\Rightarrow \lambda(x)[x + 5 * 5] \\ &\Rightarrow \lambda(x)[x + 25]. \end{aligned}$$

This example shows that a partial evaluation is possible by performing the applicative order (eager) reduction.

3. Data Flow Computing

In data flow computing, programs are represented as a directed graph called a data flow graph. The data flow graph has several input and output ports. Nodes in the data flow graph represents operations. Operations on nodes are performed when their needed operands are available, and their results trigger the succeeding operations. Execution starts when initial tokens are put into

the input ports of the data flow graph, and is complete when result tokens are output from its output ports. This data flow computing process is interpreted as a reduction process performed on the data flow graph [17]. For example, graph reduction shown in Fig. 1 is also understood as a reduction on the data flow graph.

We can consider several data flow computing model according to what kind of data is assigned to tokens. Primitive data flow model assigns data itself to tokens. We call the computation mechanism built on this assignment as by-value mechanism. More advanced dataflow computing model assigns to the token a cell which is variable, pointer to structure data or program code. We call the computation mechanism built on this assignment as by-reference mechanism. Even more sophisticated data flow models, those that provide more elaborate computation control, are combination of by-value and by-reference mechanism.

In the by-value mechanism there is no notion of cells that hold data value or pointer. We can easily understand that the by-value data flow computing model is equivalent to the applicative order reduction.

3.1. By-Reference Mechanisms

In the by-reference mechanism, all operations have their own cell which hold their operand and result values. Each cell is shared between two operations, i.e. the cell is used as result value cell of one operation and also as the operand value cell of its successive operation.

Either of two types of implementation are considered depending on how to control the read and write access to this shared cells:

Cell-driven eager computation

One implementation is called cell-driven control, in which operations are initiated when operand cell is made available. The initiated operation tries to fetch its operand and waits for its value to be written by the preceding operation. A read-ready tag is used to control such concurrent read and write accesses. A simple example of cell-driven control is shown in Fig. 3.

This mechanism realizes maximal eager evaluation, since all operations are initiated eagerly when their cells become available. (It should be noticed

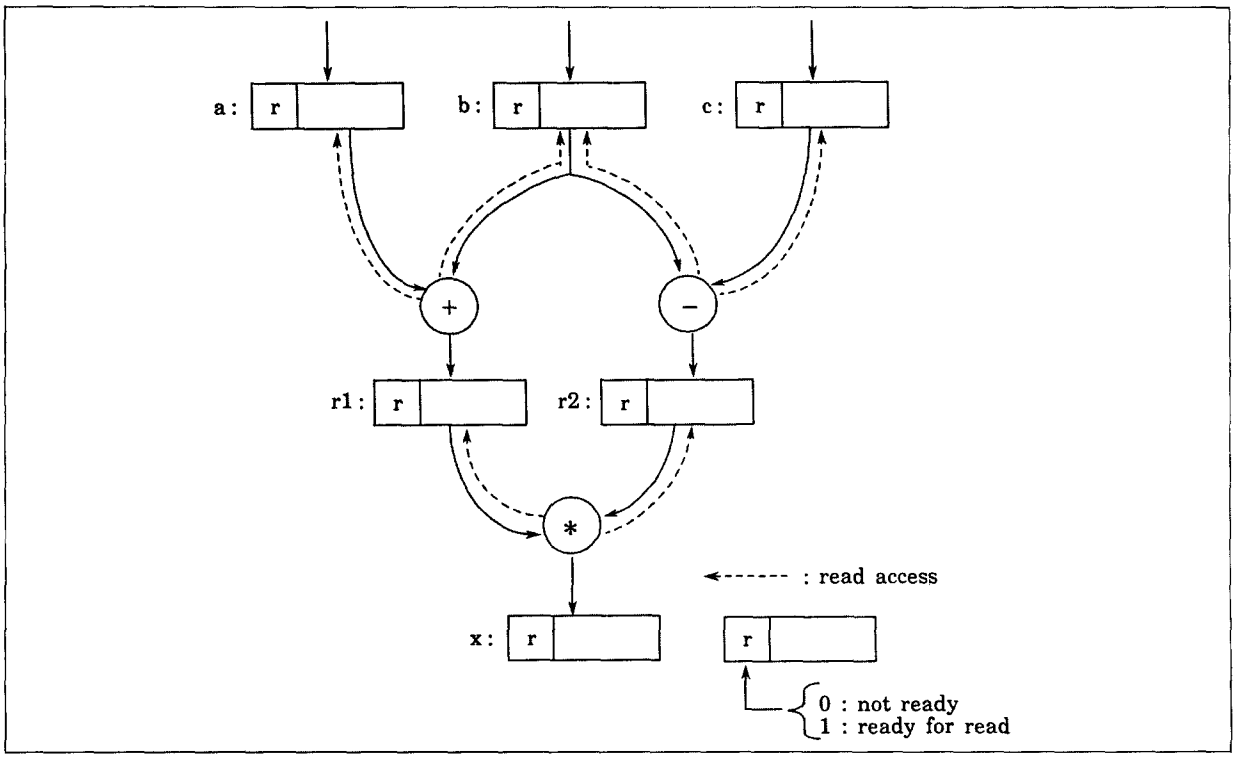

Fig. 3. By-reference mechanism (cell driven).

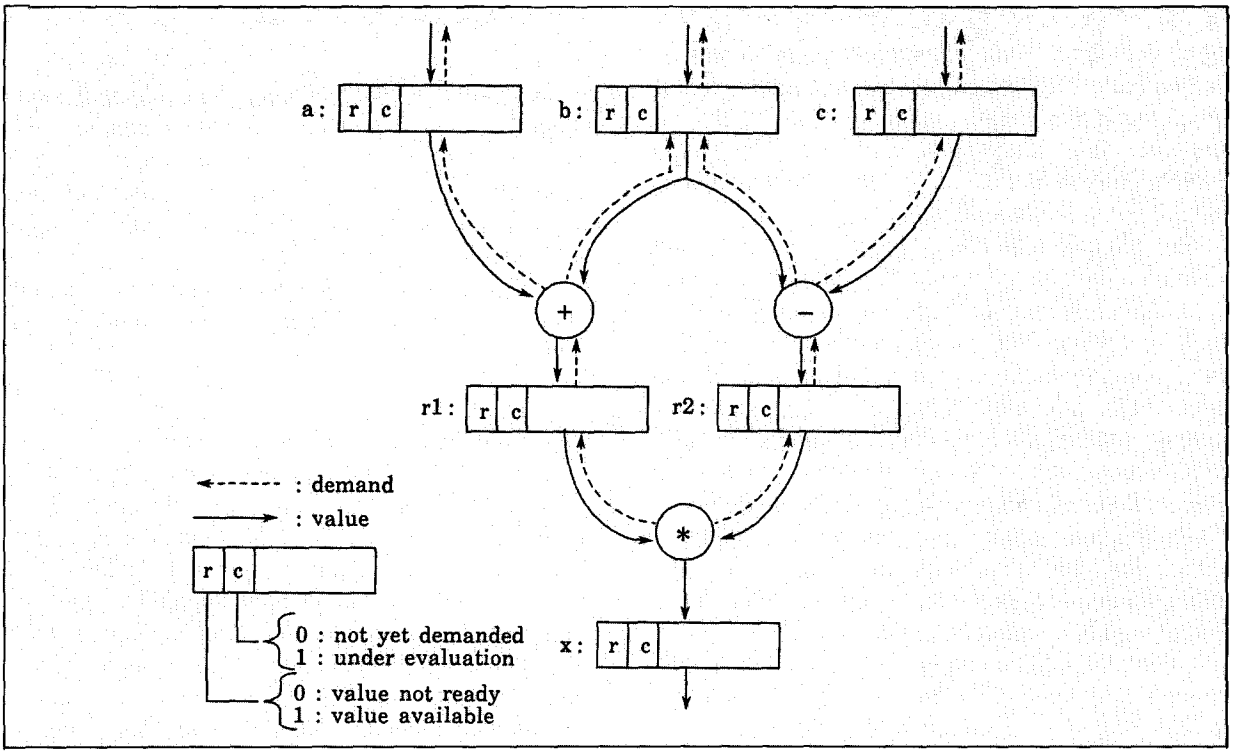

Fig. 4. By-reference mechanism (demand driven).

that all cells are possibly made available in advance before execution.) The cell-driven mechanism is also equivalent to the applicative order reduction.

Demand-driven lazy computation

Another implementation is called demand-driven control, in which every operation is initiated when its value is required from its destined operation. In the course of execution, demand tokens flow against the direction of arc, then result values are returned back. This mechanism is shown in Fig. 4.

Initially, each cell has pointer to the operation node that generates its value. When an operation is initiated, the operator accesses its operand cell. If the cell contains no value, the read access triggers a demand to the preceding operation and waits until the value arrives. In order to control this demand-triggering and value-awaiting mechanism, two tags, read-ready tag r, and demand-trigger tag c, are used in each cell.

The demand-driven computation mechanism is equivalent to the normal order reduction, and is safe for nonstrict function evaluation. However, this model has two problems in cost effectiveness. First, it is expensive to implement the demand-driven control mechanism. Second, computation takes much more time than data-driven control does because of the double flow of tokens.

3.2. Combined By-Value and By-Reference Mechanism

More effective mechanism of eager and lazy evaluation is to use the by-value and by-reference mechanisms selectively. This model is a data flow implementation of the reduction control which performs applicative order reduction and normal order reduction selectively. In this model, each token has as its value either a data flow graph which is called meta data, or a normal form which is called object data. Cells are allocated to variables and function argument, which are extracted by a compiler through data flow and nonstrictness analysis.

Each cell has two tags r and c, which are used for eager and lazy evaluation control. The content of each cell is either an evaluated value or a pointer to a program code entry, called recipe.

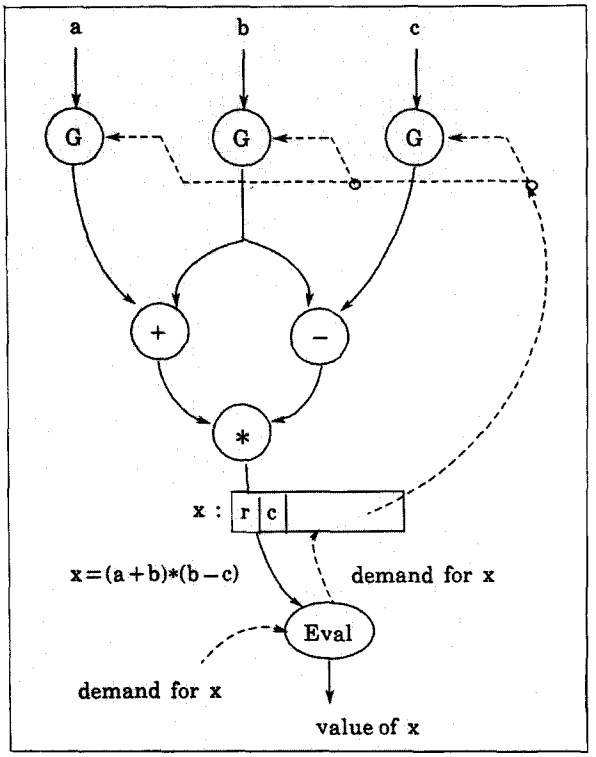

Fig. 5. Combined by-value and by-reference mechanism.

When tag r is on, the cell contains a value and thus read access is permitted. When tag c is on, the content of cell is recipe. A simple example is shown in Fig. 5. In this graph, by-reference mechanism is used for variable x. When the value of x is required, a demand signal is triggered from cell x to gate nodes to pass data a, b and c to the graph. Thus, the expression is evaluated in applicative order by the by-value mechanism.

By-reference mechanism also naturally implements the graph sharing in reduction. For example, suppose the expression $x + x * x$, where $x = (a + b) * (b - c)$, is evaluated by demand driven (Fig. 6). Even if operations + and * issue demands to the cell of variable x, the evaluation of x is triggered only once by the first arrived demand, and its result value is returned back directly to subsequent demands.

3.3. Nonstrict Evaluations

Nonstrict function application is implemented using the by-reference mechanism. The function argument which is required to be evaluated lazy are not evaluated but its data flow graph, which is

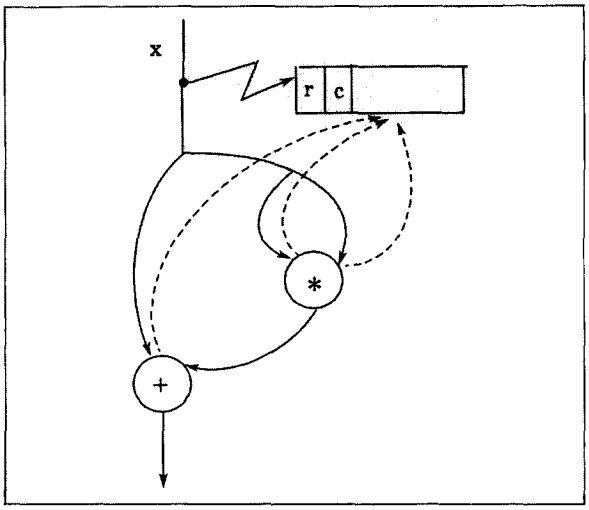

Fig. 6. Graph sharing.

a recipe, is passed to the function body. In this case, a cell that points to the data flow graph (i.e., program code) is treated as a token. Consider the *ite* function as an example. The data flow graph of *ite*(p, x, y) is represented as Fig. 7.

In this scheme, arguments x and y have data flow graphs as their values, while p is an evaluated

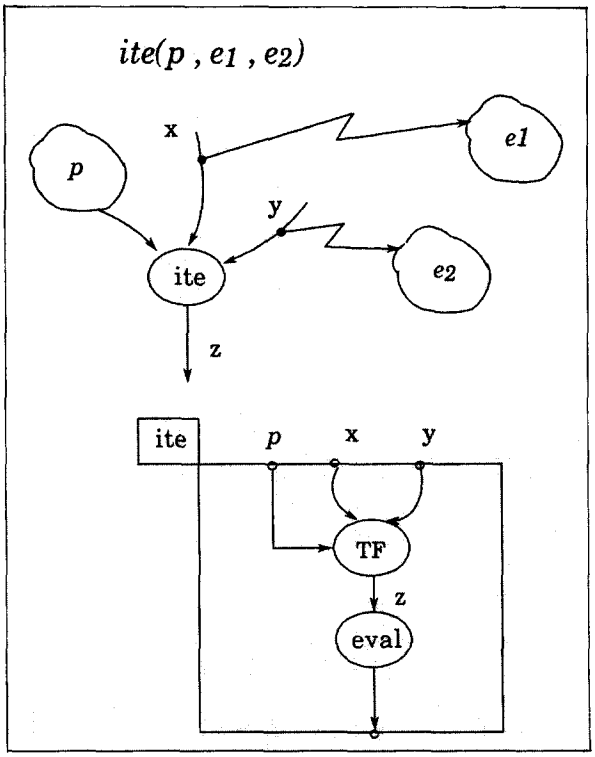

Fig. 7. Example of nonstrict function.

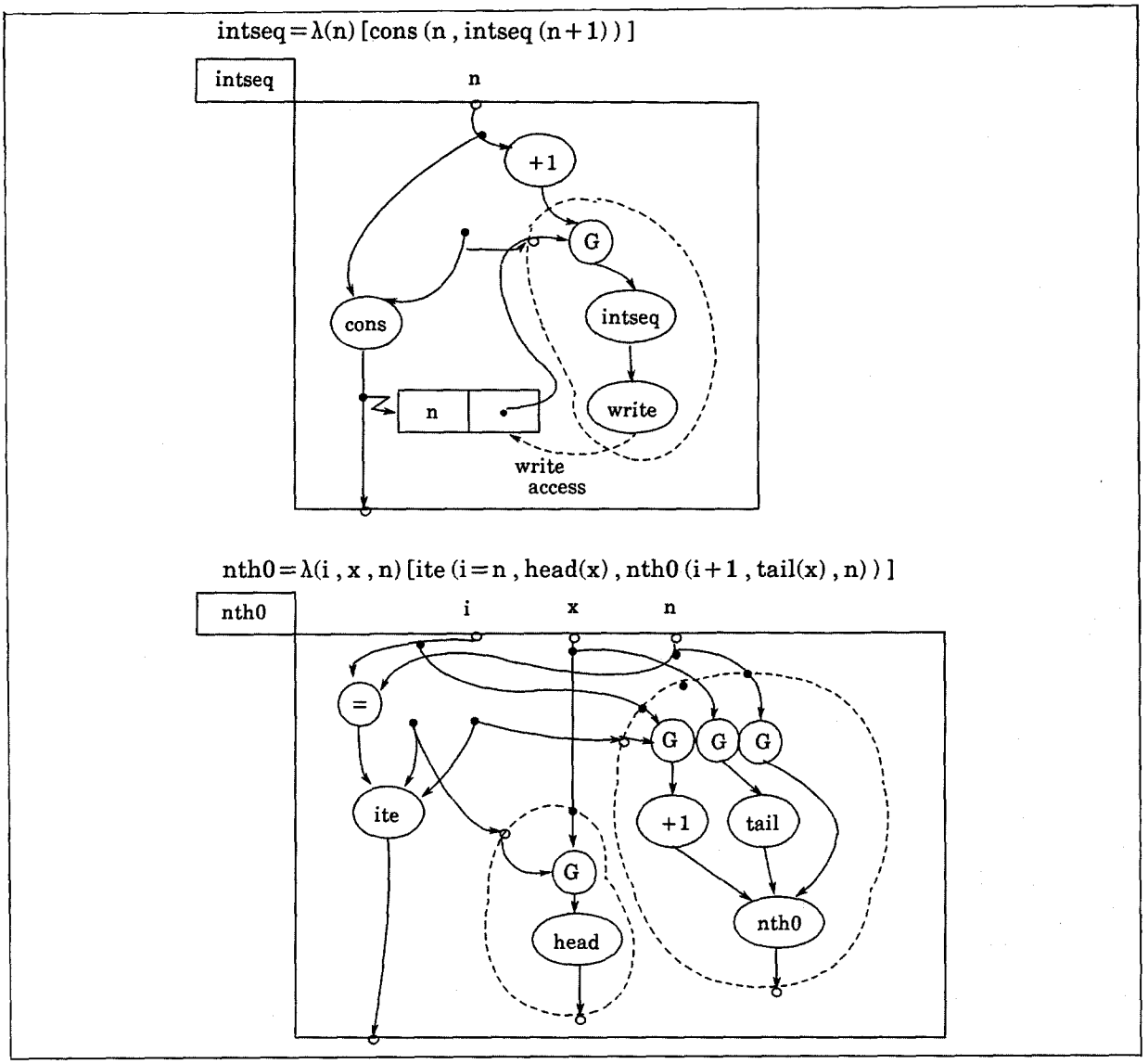

Fig. 8. Data flow computing on nonstrict evaluation.

Boolean value. The Boolean p controls the inbound switch to select x or y, and put it to z. Eval operator evaluates the data flow graph pointed to by z. This mechanism implements the normal order reduction for *ite* function described in section 2.1.

Nonstrict evaluation on structure data is also implemented on the data flow computation mechanism. In order to control the nonstrict operation of structure data, the read-ready tag and the demand-trigger tag are also used for data cells. In the nonstrict *cons* operation, its head and/or tail

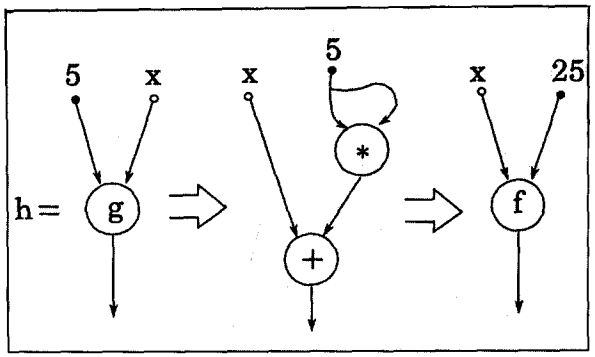

Fig. 9. Partial evaluation.

argument is not evaluated. Instead, the unevaluated data flow graph of the argument is written into head and/or tail field of the cons cell. The demand to evaluate the head and/or tail expression is issued when *head* and/or *tail* operation has accessed head and/or tail field of the cons cell [12,18,19]. As an example of the usage of nonstrict cons mechanism, a data flow implementation of the function *intseq* and *nth*0 is shown in Fig. 8.

3.4. Supercombinator and Data Flow Graph

As described in section 2.3, supercombinator is a lambda abstraction which has no free variables. The data flow model has quite similar properties to the supercombinator. All tokens in a data flow graph of function body are given explicitly as parameters, and no free variables exist in the data flow graph.

Data flow computing model, therefore, naturally implements supercombinator as a defined function. Eager and lazy evaluation mechanism in data flow computing model would enable the partial evaluation described in section 2.3, as shown in Fig. 9.

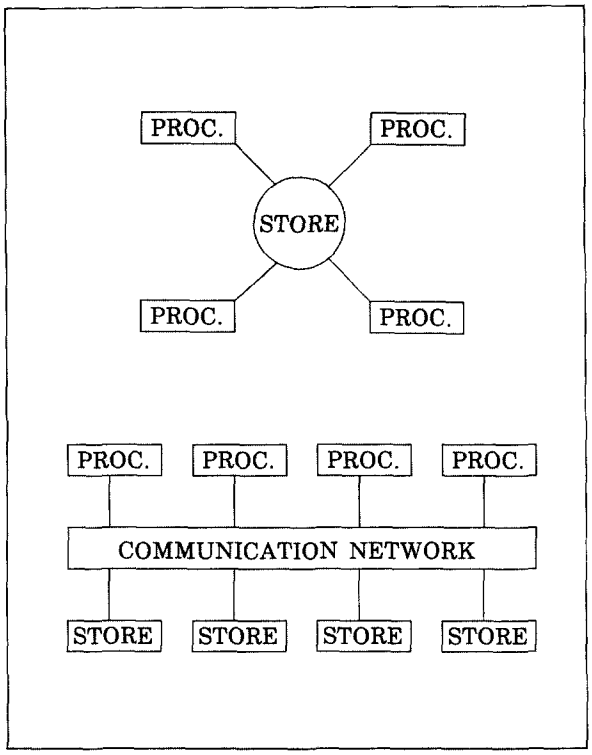

Fig. 10. ALICE machine architecture. (Top) Abstract machine structure. (Bottom) A simple distributed machine structure.

4. Parallel Reduction Machine

As opposed to the serial reduction control, stack based execution control mechanism does not work well for parallel reduction control. Instead, a packet based execution mechanism would be considered as a basis for parallel reduction control. The typical example of this type of implementation is ALICE machine [3]. ALICE machine is a physical implementation of graph reduction process described in Section 2. The ALICE machine structure is similar to that shown is in Fig. 10(Top).

The processors access the store, select a packet, perform a reduction and write new packet or modify old ones in the packet store. Each packet has three modes; active mode, suspended mode and dormant mode. The packet in active mode, indicated as !, is a candidate for further reduction. The packet in suspended mode, indicated as #n, is waiting for n of its arguments to reach where reduction is possible. The packet in dormant mode, indicated as –, is not currently a candidate for any action. An intelligent packet store could detect when packets had all the required arguments and make them processable. It could also detect when packets were no longer required and mark them for garbaging. In the physical implementation the packet store is partitioned and distributed within the machine as shown in Fig. 10(Bottom).

One of the problem in ALICE machine architecture is overhead of operations in packet store. When a reduction is performed on the packet store the processing agent must rewrite the argument field. When a reduced value is returned back, the agent must check whether the suspended packet has changed to active packet, and if so, the agent must put the the active packet to an active

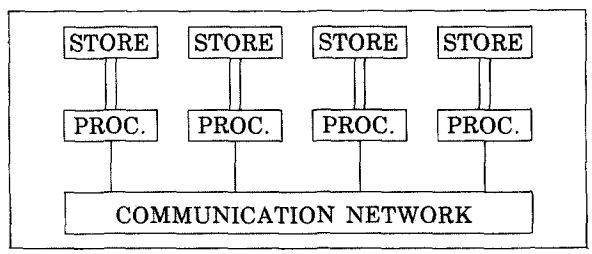

Fig. 11. Flagship machine architecture.

packet queue. The key problem in implementation is how to reduce the overhead in these operations on packet store.

The Flagship machine [20] is a successor to ALICE machine. The Flagship is designed to improve the above mentioned overhead in ALICE machine. In the Flagship architecture, redundant demand propagation is assumed to be eliminated at compile time, and run time execution could be mainly controlled by data driven. However, mechanisms of access and operation on packet store are essentially same as those in ALICE machine, except that Flagship has a local packet store, instead of the global shared packet store, as shown in Fig. 11. Therefore, Flagship still has the problem of overhead in data driven control and packet store operations.

5. Data Flow Based Reduction Machine

The important point in designing the parallel reduction machine is that the demand propagation in normal order reduction will be possibly eliminated in compile phase. By eliminating the redundant demand propagation, compiled code of a given functional program is executed essentially in data driven, except for a few part which needs to be executed in demand driven. Program analysis and compiler construction technique for optimizing the demand driven computation [14,15] are also important issues, but those are omitted in this paper because the theme is beyond the scope of this paper.

In this section, an implementation model is proposed, which will reduce the overhead in data flow control while preserving the parallelism maximally. The key idea is to extract a multi-thread control flow from the original data flow.

5.1. Features of Data Flow Architecture

Data flow architecture offers several beneficial features as a basis of parallel machine architecture [10,11,12].

(1) The data driven control mechanism is quite natural for parallel execution. It can maximally exploit the parallelism inherent in a given program.

(2) The tagged token mechanism, with which multi-process execution environments are managed on the hardware level, can efficiently control the parallel and concurrent execution of multiple function activations.

(3) The packet-communications-based circular pipeline architecture realizes pipeline and parallel operations between hardware modules, and this mechanism offers a basis for massively parallel MIMD machine architectures.

At the same time, the data flow architecture has several drawbacks:

(1) Communications overhead: There exists an overhead in communications that occurs within a processor and between processors. Low level communications are essential in data flow architecture, since all operations explicitly transfer data to their successors as tokens. (Here, it should be noted that the interprocessor communications overhead is not a problem in the data flow machine only, but is also essential in all multi-processor systems where communications are necessary among processors.)

(2) Overhead of fine grain data flow control: If the process granularity is set at a finer level, a larger number of switch and gate operations will be required. This results in an increase in data flow control overhead.

(3) Demerits of memory-less concept: The pure by-value data driven control mechanism produces needless data copies and redundant operations. A by-reference data sharing mechanism, on the other hand, never produces needless data, and therefore only the demanded operations are activated, since data are accessed only when they are needed. Furthermore, the data sharing and access-by-needs concept is essential for implementing the the lazy evaluation.

5.2. Implementation Model

In the practical implementation, the by-reference and by-value mechanisms are used for data tokens. All data tokens flowing in a data flow graph, unless having only one destination, denote memory cells that hold values, except for Boolean values and gate control signals. Boolean values and control signals are represented in one-bit data, and they are used directly for gate and switch controls [17].

This cell token flow implementation model offers the following good features compared with the conventional data flow model:

(1) Even if a result value produced by an operation is referred to by several operations, it is not necessary to create copies of the result value, since the by-reference mechanism permits those operations to share the operand data. The operand data is accessed only when it is needed.

(2) Any two nodes connected with a link have a data dependence relation. In the cell token flow

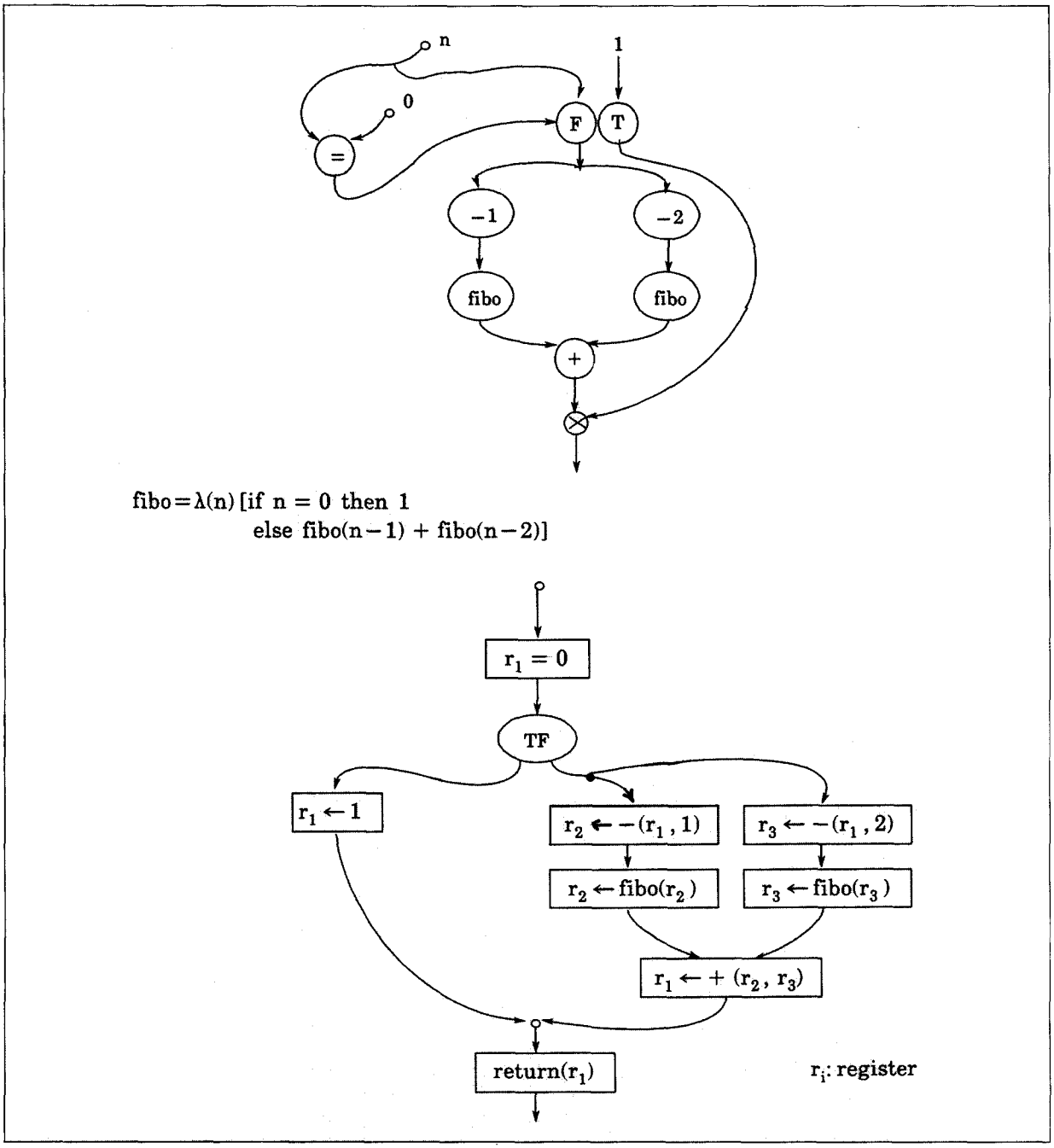

Fig. 12. Data flow program and its extracted multi-thread control flow program. (Top) Data flow. (Bottom) Multi-thread control flow program.

model, the connection link between two nodes represents a memory cell address. The address of the memory cell can be determined at compile time, and thus be free of the run time memory allocation.

(3) The by-reference mechanism assures that an operand data of some operation exists when it is produced (and written into its memory cell) by the preceding operation. Therefore, the availability of an operand, call it x, need not be checked, if its partner operand is a dependent of operand x. This reduces the overhead of checking the pair token availability in detecting enabled instructions.

Considering these characteristics of the cell token flow model, we can eliminate redundant token flow links, and can transform the data flow program to a multithread control flow program, which preserves the original data dependency.

5.3. Control Flow Extraction

The multi-thread control flow program is extracted from a functional style source program in the following process. First, the source program is translated into a data flow graph. Then, the data flow graph is transformed to a multi-thread control flow program, through memory cell assignment and data flow optimization. (It should be noted that memory cell assignment and multithread control flow extraction can be performed on the source program directly. This is the issue of compiler construction technique, therefore detailed discussion is omitted here.) An example of the data flow program and its extracted multithread control flow program is shown Fig. 12. An architecture of the processor which executes the extracted multi-thread control flow program will be described in later.

5.4. Function Activations and Memory Blocks

In the course of computation, a new function instance is created for each function activation. These function instances share the same function body (program code). A memory block, which is used as a working register file, is assigned to an activated function instance. All operand data used in the instance are stored in the allocated memory block.

Each memory cell in a block is accessed by a local displacement address in the block. This local displacement is determined at compile time. At run time, memory cells are accessed with a two-level addressing mechanism; the dynamically specified block address and the statically decided local displacement.

5.5. Partial Evaluation and Function Closure

The evaluation environment for a newly created function instance is maintained in the memory block assigned to the function instance. In the course of execution, graphs always shrink except for function application, and when a function application occurs, a new memory block is allocated to the newly created function instance. Therefore, during execution, a larger memory block is never needed than was initially allocated.

5.6. Demand Driven Lazy Evaluation

Memory cells have two tags, r and c, for controlling the lazy evaluation discussed in Section 3. Tag r shows whether the memory cell is a data value or code pointer. Tag c shows whether the memory cell has been demanded already or not. When an eval operator $eval(v)$ is activated, it accesses a memory cell allocated to variable v. If the cell has a value, i.e., r is on, the data flow graph has already been reduced to its value. If the cell has no value and has not been accessed yet, i.e., r and c are off, then a demand signal is delivered toward the entry point of data flow graph to start the evaluation. If the cell has been demanded already, i.e., r is off and c is on, the reduction has already been triggered by some other eval operator, and the eval operator waits for the value to be written to the cell.

6. Machine Architecture

6.1. System Architecture: Parallel and Pipeline Multi-processor System

An architecture of parallel reduction machine is shown in Fig. 13. The machine is constructed with a number of processing elements and structure memory elements. Structure memory is set as a local memory for a specific processor. At the same

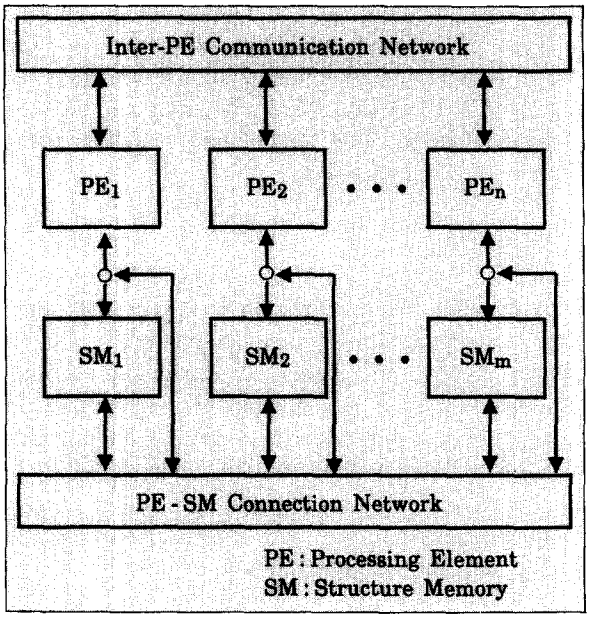

Fig. 13. System architecture.

time, the structure memory can be accessed from other processors. The local/global access switching is simply controlled using the structure memory address.

Processors and memories are connected with multi-stage packet switching network. Processing elements, called PEs, execute a program. The program code and operand data are stored in Instruction Memory and Data Memory in the PEs. Structure data such as lists and arrays, on the other hand, are stored in structure memory, called SM, and the pointer to structure data stored in SM is treated as operand data within each PE. This mechanism for handling structure data is the same as that in the data flow machine DFM [18,19].

The inter-processor connection network is designed so as to realize a flexible tree structure in logical, while its physical connection is set regular and local. The network also controls the processor work load to balance among processor resources. The network watches the work load of each processor, and allocates an activated function instance to the least loaded processor so as to distribute the work load over processor resources.

6.2. Processor Architecture: Multi-thread Control Flow Executor

The design philosophy of the processor is as follows:

(a) Separation between data token and control signal: Only control tokens are used for execution control. Data values are stored in memory cells or pipeline latch register. The address of operand data is specified in the operand field of an instruction. Boolean values are treated as control token, so that switch operation is operated in the execution control module.

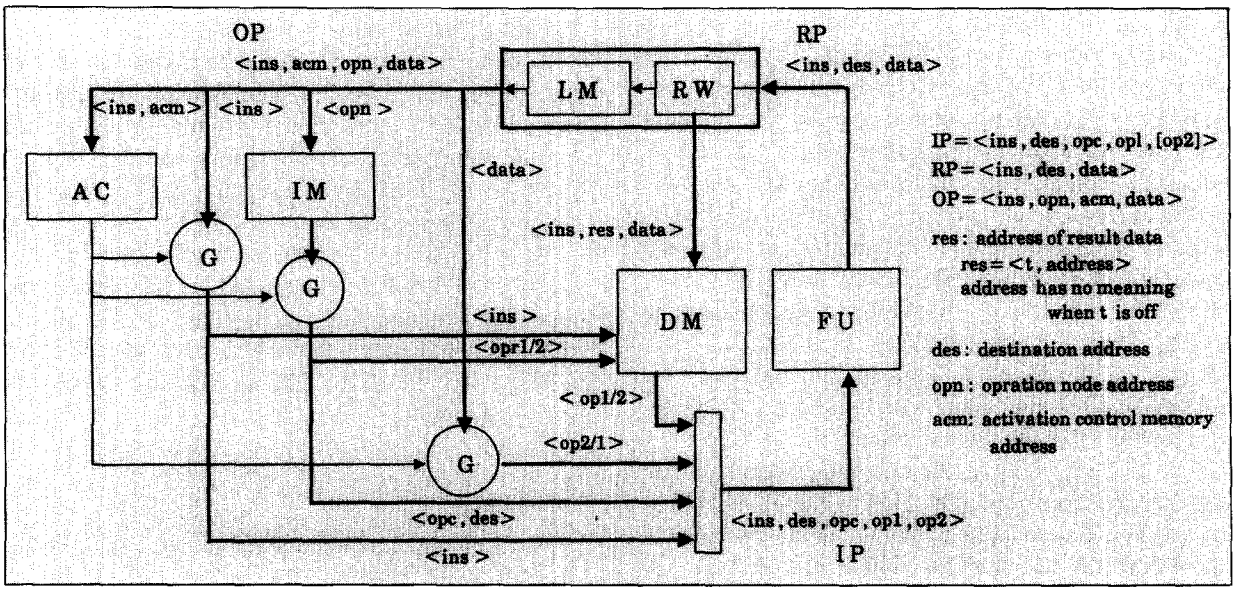

Fig. 14. Processor architecture.

(b) Large capacity register file: The concept of a large capacity register file is used for implementing the operand memory system. The register file is divided into multiple blocks. Each block is assigned to each function instance at run time, while the local cell address in each block is obtained at compile time.

The processor is designed on the basis of a circular pipeline architecture constructed with Function unit (FU), Link memory unit (LM), Result write unit (RW), Instruction memory unit (IM), Activation control unit (AC), and Operand data memory (DM). These units are connected in a circular pipeline, as shown in Fig. 14.

The execution of an instruction in the circular pipeline proceeds as following:

(1) The function unit executes operations specified by the operation code *opn* in Instruction packet (IP). The execution result is constructed as a Result Packet (RP), and sent to the RM.

(2) The RM writes the result data (*data*) to Operand data memory (DM). At the same time, the LM decides the IM-address of its destination operation node (*opn*), and constructs an Operand packet (OP).

(3) Instruction memory unit (IM) fetches the instruction code (*opc*), operand address (*opr*) and the destination node name (*des*) from the Instruction memory cell specified by the address *opn*.

(4) Activation control unit (AC) checks whether the partner operand has been written in Operand data memory. AC memory is used for token matching. AC memory is also divided into multiple blocks, each of which is assigned to a function instance. The AC memory cell is constructed with only two bits. One bit is used as a Boolean value, and the other bit is used as a tag to show the presence of a partner operand. The AC memory cell is accessed by two level address; the block address specified by the instance name *ins*, and the cell address *acm* which is local to the block. It should be noted that almost all two-operand operations need not examine the matching of pair tokens, as discussed in section 5.3. Only a few two-operand instructions are needed to examine the readiness of the pair operand.

(5) For enabled instructions, their operand data are fetched from DM. The DM cell is accessed by two levels of address: the DM block address which is specified by the instance name *ins*, and te local displacement in the DM block which is specified by the operand address *opr*. One problem is access contention to DM since DM access occurs at both result data write and operand data fetch. However this problem can be solved designing a multi-port memory device with high access speed using SRAM device technology.

7. Conclusions

A parallel graph reduction model and its machine implementation were discussed. First, a parallel graph reduction mechanism and its relation to data flow computing were discussed. In this paper, an extended data flow model, which introduces by-reference concept, was proposed. In the discussion, it was pointed out that the data flow computing model with by-reference concept was a natural implementation of parallel graph reduction. Then, an implementation of a parallel reduction machine was presented as a practical model. In the implementation, a cell token flow model was used. By using the cell token flow model, a given data flow program can be transformed to a multi-thread control flow program.

Last, an overview of the machine architecture was presented. The machine is designed on the basis of the packet communications concept, in which many processor and memory resources are connected with packet communications network. The processor is designed as a circular pipeline processor, which performs high speed execution of the multi-thread control flow program.

Another important issue, which was omitted in this paper, is program analysis and compiling technique for functional program; the algorithms to analyze data dependency and nonstrictness, to optimize the functional program and to extract a multi-thread control flow.

By integrating the researches on the parallel machine architecture discussed in this paper and compiler construction technique for functional language, a high performance parallel reduction machine will be developed as a future supercomputing system.

References

[1] K. Berkling, Reduction language for reduction machines, Proc. IEEE Internat. Symp. Comp. Arch., 1975, pp. 133–140.

[2] G. Mago, A cellular computer architecture for functional programming, Proc. IEEE Compcon, 1980, pp. 179–187.

[3] J. Darlington and M.J. Reeve, "ALICE": a multiprocessor reduction machine, in: Proc. Conf. Functional Programming Languages and Computer Architecture, 1982, pp. 65–75.

[4] S. Vegdahl, A survey of proposed architecture for the execution of functional languages, *IEEE Trans. Computer* **33** (1984) 1050–1071.

[5] I. Watson, Parallel data-driven graph reduction, in: J.V. Woods, Ed., Fifth Generation Computer Architecture (North-Holland, Amsterdam, 1985) 203–220.

[6] E.A. Ashcroft and R. Jagannathan, Operator nets, in: J.V. Woods, Ed., *Fifth Generation Computer Architecture* (North-Holland, Amsterdam, 1985) 177–201.

[7] P.J. Landin, The mechanical evaluation of expressions, *Comput. J.* **6** (4) (1966) 308–320.

[8] R.B. Kieburts, The G-machine: A fast, graph-reduction evaluator, *Proc. Functional Programming Languages and Computer Architecture*, Lecture Notes Comput. Sci. **201** (Springer, Berlin,1985) 400–413.

[9] D. Turner, A new implementation technique for applicative languages, *Software Practice and Experience* **9** (1979) 31–49.

[10] J.B. Dennis, First version of data flow procedure language, in: Lecture Notes Comput. Sci. **19** (Springer, Berlin, 1974) 362–376.

[11] Arvind, K.P. Gostelow and W. Plouffe, An asynchronous programming language and computing machine, TR-114a, Dept. Information and Computer Science, Univ. California Irvine, 1978.

[12] M. Amamiya, Dataflow computing and eager and lazy evaluations, *New Generation Computing* **2** (1984) 105–129.

[13] J. Hughes, Super combinator—A new implementation method for applicative languages, Proc. 1982 ACM Symposium on Lisp and Functional Programming, 1982, pp. 1–10.

[14] S. Ono, N. Takahashi and M. Amamiya, Optimized demand driven evaluation of functional programs on a dataflow machine, 1986 Internat. Confer. Parallel Processing, 1986.

[15] C. Clack and S.L. Peyton Jones, Strictness analysis—a practical approach, in: *Proc. Functional Languages and Computer Architecture*, Lecture Notes Comput. Sci. **201** (Springer, Berlin, 1985) 35–49.

[16] T. Johnson, Lambda lifting: transforms programs to recursive equations, *Proc. Functional Languages and Computer Architecture*, Lecture Notes Comput. Sci. **201** (Springer, Berlin, 1985) 190–203.

[17] M. Amamiya, A new parallel graph reduction model and its machine architecture, in: *Programming of Future Generation Computers*, Proc. First Franco-Japanese Symp, 6–8 October, 1986 (North-Holland, Amsterdam) to appear.

[18] M. Amamiya, R. Hasegawa, O. Nakamura and H. Mikami, A list-processing-oriented data flow machine architecture, *Proc. NCC* (AFIPS, Reston, VA, 1982) 148–151.

[19] M. Amamiya, M. Takesue, R. Hasegawa and H. Mikami, Implementation and evaluation of a list-processing-oriented data flow machine, Proc. 13th Ann. Int. Confer. Computer Architecture, 1986, pp. 10–19.

[20] I. Watson, J. Sargeant, P. Watson and V. Woods, Flagship computational models and machine architecture, *ICL Techn. J.* (May 1987) 555–574.