A PARALLEL CONFIGURABLE LISP MACHINE

ADOLFO GUZMAN AND RAYMUNDO SEGOVIA
National University of Mexico
Mexico

A configurable computer architecture based on the concept of dynamic modification of the machine structure to tailor it to the algorithm being executed is presented. The machine consists basically of: a set of operational units performing basic functions, a dispatcher to handle the calls on these units, a tree stack that provides the running environment to the program and a memory where data and code resides. The application of this architecture to an expression language -LISP- is considered in some detail.

INTRODUCTION. Centralized sequential control in a computer acts in a limiting fashion for some algorithms that are naturally parallel.

Linear memory. The addressing schemes of most machines are not natural to handle list and trees.

Faster computers appear each year[1]. But there is a limit in the speed attained by electronic circuits, when you take into account switching times, propagation delays, and other effects[21,33].

But even for a machine that performs much of its computation in a parallel fashion, the programmer (the user) has to command it properly. The languages available to this user fall at least in three classes: a) Parallelism is handled explicity by the user[7]. This gives him the power to specify in detail the flow of a program; b) The user writes his code in "normal" language such as Fortran. The Fortran compiler tries to parallelize as much as possible the source code[30]. If the language was originally designed for sequential coding, the compiler has to deduce which statements are sequential, and which others can be processed simultaneously. Sincronization, dead-locks and other problems are handled by the compiler c) High level language fits naturally into a computer architecture, because it is well tailored to its characteristics, and it is also convenient for the user[22,17].

A good example of the later is the lambda-calculus[17], and a language that embodies it; LISP[23]. We refer here to pure LISP, without the program feature PROG, GO TO's, operators such as RPLACA and RPLACD, and other impurities.

Finally, most of the hardware produced today is fixed. But a given program could profit, at certain stage of its execution, from more adders than those bought with the machine.

THE PROBLEM. We would like to address the design of a machine that: 1) Executes in parallel several portions of the same program; 2) Frees the programmer from the need to specify what things are computed before, simultaneously or after other things. The machine should allow a more "natural" way of multiprocessing.; 3) Dynamically reconfigures itself to the algorithm being executed. The machine should sense the need of different execution boxes at different times, and should convert the non-used boxes to more useful ones.; 4) Exhibits tolerance to failures.; 5) Has self-checking capabilities.

THE SOLUTION. The model proposed consists basically of 4 elements: 1.-The memory (passive memory). It contains programs, data and results that are not in the grill at this moment., 2.-The grill (active memory). It holds the program(s) being executed., 3) The boxes (operational units). Each box performs a primitive operation of the language such as MULTIPLY, SINE, LIST. A MULTIPLY box searches the grill, through the blackboard, for a multiplication ready to be done, executes the operation if finds one and leaves the result in the grill., 4.-The blackboard (dispatcher). Points to places in the grill containning functions to be evaluated. The boxes look in the blackboard for work to do.

An expression is placed in memory by an I/O mechanism. At the end of the I/O a mark on the blackboard is placed. Such mark is a pointer to the head of the expression in memory. A mark on the blackboard is a flag for attention to the boxes. The boxes can be micro-computers that implement complex instructions and bookkeeping functions. The boxes are either busy doing their function or else they are continuously searching the blackboard for things to do. In the particular case of the initiation of an execution a START box is called in order to build the head node in the grill, from the code (the expression) placed in memory, and from then on a tree is built (a tree of stacks) on the grill[5]. Each node can be regarded as a STACK, and represents a function name with its arguments. The tree expands and contracts as the evaluation of the program goes on. Demands for evaluation of a function are placed on the blackboard whenever the arguments of such function are

Artículo 30

all evaluated. This means that the evaluation of the function is ready to proceed.

Each call for a function (a tree-node) has a counter of unevaluated arguments that is decremented in one for each evaluated argument. (Notice that the arguments are evaluated in parallel). This decrement is performed by each box that evaluates an argument. In addition, the box that reduces such counter to zero will place a mark on the blackboard, meaning that this function, is now ready for evaluation.

The values replace (take the place of) the evaluated expression, in the grill. Eventually the whole tree of the original program will collapse to a data-object (a number, a list), this being the end of the evaluation of the expression.

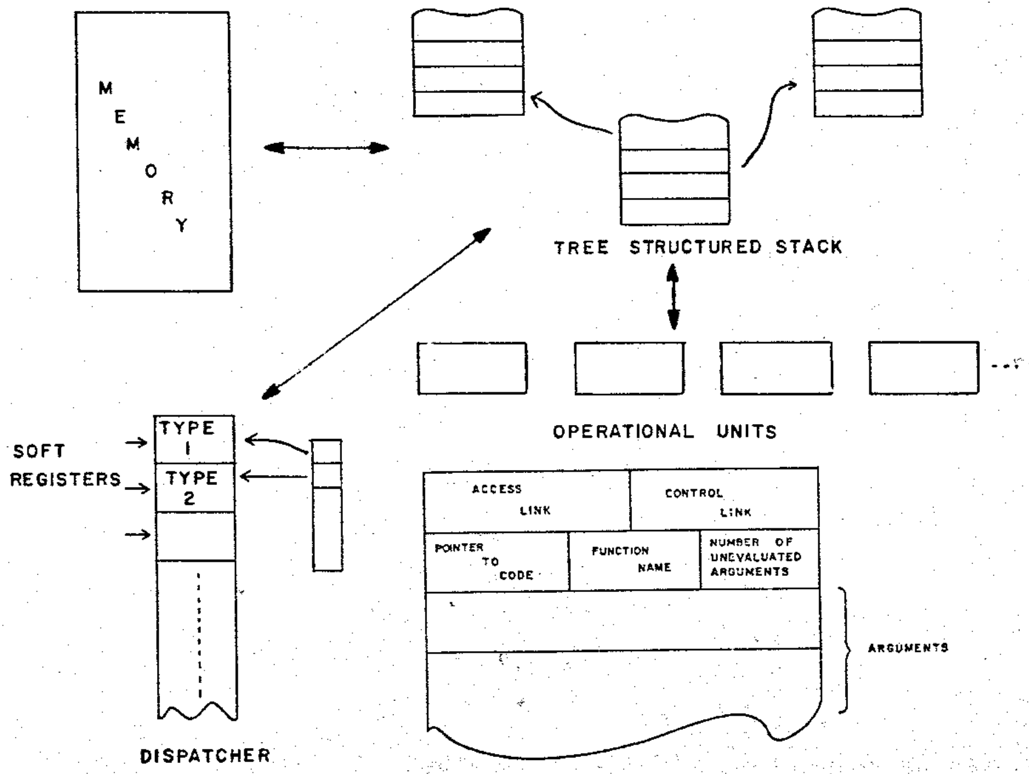
One has to notice that this process is begin held in parallel; the grill is growing and the boxes are placing constant demands on the blackboard, where other boxes are continuously searching for operations to perform. The limiting factor in parallellization is the amount of boxes. A process of reconfiguration (explained later) takes place whenever demands for certain type of boxes exceed its availability.

In order to easily incorporate concepts of high level languages, the grill's architecture is based on a tree structured memory[5] where the nodes [fig. 1] are stacks, providing an environment to handle free variables, bindings and return's to the calling environment (static and dynamic links) and a set of pointers to the arguments.

Boxes.- All boxes realize primitive operations. They have a tag that identifies them; eg., SIN or TAN box. They are organized in affinity groups for reconfiguration purposes. The sequence of operation of a box is as follows: it searches the blackboard in a specific area according to its tag; finds a demand from some node in the grill; erases the mark on the blackboard; goes on the evaluate the node, placing in it, its value; decrements the counter of unevaluated arguments of its father by one; if zero places a mark on the blackboard telling that the father is ready for evaluation; the sons are returned to a pool of free.

Grill.- The grill is a place where a program is slowly converted into results by the action of the boxes. It is a tree of nodes each of which has a structure as seen in figure 1, mainly an access link that points to higher level free variables and parameters bindings, accesible within the environment; a control pointer that points to the calling environment[6]; a pointer to code in the pasive memory; a function name or identification; a counter of unevaluated arguments; a set of arguments.

The grill is organized in active nodes that are being processed, and a pool of free nodes. We may



F I G U R E 1. THE CONFIGURABLE COMPUTER MODEL AND A NODE.

think that each node is a memory bank with asyncronous access to it. The blackboard is pointing to the nodes ready for evaluation. Nodes may be exhausted causing a machine error.

Blackboard.- The blackboard is a linear memory divided into areas where boxes place marks of nodes demanding evaluation. Areas are divided by box type; the division is with soft registers that can be moved as demands grow. The blackboard may be exhausted (eg. in an infinite recursion), causing the machine to abort.

Memory.-Passive memory contains programs, data and results. The grill usually points to some of these. Certain boxes copy programs from memory to the grill for evaluation, and copy results from the grill. I/O transactions take place between memory and external devices.

To poinpoint some of these ideas we propose a LISP machine, that is, a machine that process LISP programs residing in memory as s-expressions, (We consider "pure LISP").

Boxes are the LISP primitives CAR, CDR, CONS, ATOM, EQ, plus some programming functions LAMBDA, IF, AND, etc.[23]. The following classes of boxes are needed:

-Primitive and programming operations CAR, LAMBDA...
-Boxes that handle calls from user defined functions.
-Converter
-I/O (special box).

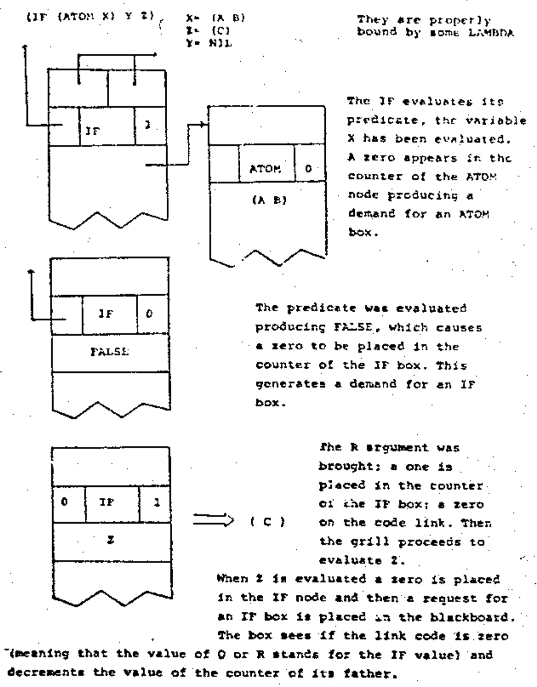
An s-expression is read into memory by the I/O box, and a node pointing to it is created in the grill; then execution begins.

-VARIABLES. A variable is evaluated by finding its value in the current a-list. In our case a stack is formed in each LAMBDA node, and a pointer is held to the calling environment.

-IF. The format of this expression is (IF P Q R). If P is true, Q is evaluated; otherwise R is evaluated[23]. Initially an IF node is created, having only the argument P and a pointer to the rest of the code in passive memory. If the pointer is nil it means that the argument just evaluated was Q or R, so the IF has to deliver it as a value, thus completing its evaluation. Else,P is evaluated leaving a TRUE or FALSE value. The IF box goes into action bringing Q or R and also setting the pointer to the code to nil, the counter of unevaluated arguments to 1 and then allows the evaluation of the argument to proceed. Since the IF delivers values at various stages (first P and then Q or R), it frees nodes(Fig. 2).

-LAMBDA binds variables in LISP and evaluates an expression (form). It can be used as a primitive: (λ (X) (CDR X)) (QUOTE (A B C))) or in a function definition. The binding takes place in the stack built in the node. Notice that the access and control links both point to the calling frame. Bindings for free variables are traced up in the call structure chain. A special case is the FUNARG problem where access and control links do not point to the same places [6].

LAMBDA has a set of variables; a form (its body) and its arguments (to be bound with its variables). Whenever a LAMBDA is placed on the grill, the arguments are evaluated, producing a zero in its counter of unevaluated arguments. Then a LAMBDA box is demanded, which in turn binds the arguments


(IF (ATOM X) Y Z)
X= (A B)
Z= (C)
Y= NIL
They are properly bound by some LAMBDA
IF
1
ATOM
0
(A B)
The IF evaluates its predicate, the variable X has been evaluated. A zero appears in the counter of the ATOM node producing a demand for an ATOM box.
IF
0
FALSE
The predicate was evaluated producing FALSE, which causes a zero to be placed in the counter of the IF box. This generates a demand for an IF box.
0
IF
1
Z
(C)
The R argument was brought; a one is placed in the counter of the IF box; a zero on the code link. Then the grill proceeds to evaluate Z.
When Z is evaluated a zero is placed in the IF node and then a request for an IF box is placed in the blackboard. The box sees if the link code is zero (meaning that the value of Q or R stands for the IF value) and decrements the value of the counter of its father.

F I G U R E 2.

with its variables, placing a 1 into its counter and having as argument a pointer to the form, which is now placed on the grill by the LAMBDA box. At the same time a zero is placed in the code link indicating that when evaluation of the form has ended, the LAMBDA box also ends and leaves the node transformed into an s-expression (a result). As soon as a node is not used it gets into the pool of free nodes

-RECURSION is handle by expanding the tree in the grill, replacing the name of the function (each time is called), by its definition. We examine the case where user-defined functions are evaluated. Assume that the name of a function and its arguments are already on the grill, in some node. Its arguments are evaluated and a zero is placed in its counter. The box that decrements the counter and makes it zero also places a demand in the blackboard in the section of the DEFINED-FUNCTION boxes. Once this demand is attended such box erases the name node replacing it by a LAMBDA node; in its counter places a zero and the grill continues its evaluation as a regular LAMBDA. When it finds the recursive call, the process is repeated.

Other LISP functions are computed in a similar manner. For instance, (AND F1 F2 F3...)places F1 on the grill and holds its other arguments in passive memory. If F1 is FALSE, the AND becomes FALSE itself; otherwise F2 is placed on the grill, etc.

RECONFIGURABILITY. A given machine of the type described above consists of a mixture of boxes; for instance, 4 CAR's, 6 CONSes, 2 COND's, 3 AND's,etc. This mixture may not be optimal for the whole execution of a program, hence the need to change the composition of the machine. It is through this change of composition that the machine reconfigures itself. We are not considering more complex ways to reconfigure: the creation or proposition of new primitives; new datapaths among the boxes, etc.

Tp make these changes, the machine needs:

1) To detect which type of boxes are iddle most of the time t, where t is small interval (a few milliseconds). Reconfiguration occurs at the end of each period t.

2) To detect which type of boxes are very busy during the same period t. These two actions (1) and (2) are performed with the help of a "laziness" counter attached to the blackboard of each type of boxes. It contains the number of times boxes of the type in question didn't find anything to do, in the period t.

3) To decide what box(es) to change into what other(s).

4) To make the change, that is, to reconfigure. A CONVERTER box will seize the (iddle) box chosen in step 3, and plants in it a new microprogram, converting it into one of the desired type. In addition, it will change its "type" identifier: such box will no longer be regarded as a "SINE" box, but as a "TANGENT" box. This change will affect some datapaths of the box; for instance, if the box (with a new function) searches the whole blackboard and finds nothing to act upon, it will add a one to the TANGENT counter, and not to the SINE counter, as it was previously doing.

5) To reset to zero the "LAZINESS" counters of (2). To initiate a new reconfiguration period t.

6) To have some statistics ready to answer the owner's question "what new boxes should I buy next?". This feature is optional.

INPUT OUTPUT. We can regard our LIP machine can be thought as sharing its passive memory with a channel or some I/O mechanism with performs I/O.

OTHER PROPERTIES. We list several properties this machine has: (a) The machine can process any language expressed as a lambda-calculus formalism.; (b) Unless the user explicity wishes to do it, there is no way to specify sequential evaluation when it is not needed or desired.; (c) There are no GOTO's, LABELS or CALL EXIT (STOP).; (d) The machine does not need a compiler or interpreter to execute the high level language.; (e) There is no central control, cpu, program counter. In fact, there is no machine cycle. It is an asynchronous architecture.; (f) There are no interrupts, deadlocks or priorities.; (g) The primitive boxes need not be so primitive. Boxes of different speeds, configurations and complexity can coexist.; (h) The machine is modularly expandible.; (i) The machine performs maximal parallelization. Of course, faster computation could be achieved if we: change the algorithm; make (some of) the boxes faster; add more boxes.; (j) The machine has self-checking capability. A TESTER box periodically checks the boxes. We realize that there are problems such as "who tests the TESTER?".

RELATED WORK. There is much activity in the field of parallel computation. We mention some recent books on the subject [21,3,9]. Most computer companies offer machines with some kind of parallelism [11,19,29,32,33]. A network of computers is another way to make parallel computations[14,34]. There have been recent meetings[27,28,30]. Several universities engage in this research[26,31,34].

Lambda-calculus machines. In 1971, a paper[5] formalizes the way to evaluate lambda-expressions in a machine. McCarthy's paper[16] is classical in the subject.

High level language machines. The idea to use high-level language as a machine language is not new. A Fortran machine[2] has been proposed. Hewlett-Packard has a BASIC machine. APL machines also exist. B5500 and B6700 are inspired in Algol[13]. LISP machines have been proposed[8].

Pipelines. The idea of many units jointly transforming a set of data is used in pipeline architectures[19,29]. We can regard the pipeline as a special type of grill, where flow of partial results follows rigid paths.

Tag machines. B6700 already has tag bits[13]. Feustel[10] generalizes this concept.

Asynchronous computation. Patil[26] studied the asynchronous evaluation of lambda-expressions. This line of work continues. Rumbaugh[39] designed a highly parallel machine for programs expressed in a data flow language. He has dormant as well as active programs; his data structures are vectors of values. A structure is shared instead of copied, for use by several concurrent activations.

Register transfer modules. Bell[4] postulates boxes that can be easily inter connected to perform arbitrary computations.

Architectures resembling ours. Miller[24,25] describes a configurable machine based on a searcher that feeds the operational units with new tasks to perform. It is not configurable in our sense. He uses an n by n interconnection network for reconfiguration. Glushkov[12] presents a recursive computer architecture that is similar to ours in the sense that all program elements for which operands are available are to be executed by boxes. As in our machine, his architecture allows the removal of interruption processing programs. Kautz[18] also places logic into the memory of the machine. A theory which is also relevant is described by Landin[20].

CURRENT TROUBLES. We do not like the complexity surrounding each box. They are simple in themselves, but they have to search the blackboard, substract one to the father of the node just evaluated, have a "laziness counter", etc. To support such overhead, the boxes will have to be big (perform large operations), thus increasing the probability that large parts of the hardware are not doing any thing useful, because inside each box computation takes place most likely in a serial fashion. Hence, a cheaper overhead will allow more "elementary" boxes to access the blackboard directly, provoking a better utilization of hardware.

CONCLUSSIONS AND RECOMMENDATIONS FOR FUTURE WORK. We have proposed a computer structure where several boxes "cook" in parallel the program laid in the active memory. There is no need to explicity synchronize such boxes; the program itself provides the time constraints. In this machine, there is no central administration: each box knows that to do.

We have described the behavior of such machine for programs written in LISP, although the approach is valid for computations expressed in the lambda notation, or expression type languages.

This machine departs from the traditional architectures in a number of ways described in the paper. Also, the proposed architecture presents new problems, some of which also received attention.

More simulations will be required in order to fully understand the effects of certain parameters of the machine.

ACKNOWLEDGMENTS. Many of the ideas presented here were generated in discussions with Mario Magidin, who is entitled to equal credit of the Metropolitan University (México).

REFERENCES.

1. Auerbach Computer Technology Reports. Auerbach Publishers Inc. Philadelphia, Penn.
2. Bashkow, T.R.; Sasson, A, and Kronfeld, A. System design of a FORTRAN machine. Chapter 31, 363-381 in[3].
3. Bell, C.G., and Newell, A (Eds). Computer Structures: Readings and Examples. McGraw Hill. 1971
4. Bell, C.G.; Eggert, J.L.; Grason, J., and Williams, P. The description and use of register transfer modules (RTM's), IEEE Trans. on Computers, C-21, 5, 494-500, May 1972.
5. Berkling, Klaus. A Computing machine based on tree structures. IEEE Trans. Computers, Vol C-20, 4, April 1971.
6. Bobrow, D.G., and Wegbreit, B. A model for control structures for artificial intelligence programming languages. Proc. Third I ntl. Jt. Conf. on Art. Intelligence, Stanford Research [illegible] 1973, 246-253
7. Cornell, J.A. Parallel processing of ballistic missile defensive radar data with PEPE. COMPCON 72, Sept. 1972. 69-72. We refer to the PFOR language of this machine.
8. Deutsch, L. Peter. A LISP machine with very compact programs. Proc. Third Intle Jt Conf on Art Int, S.R.I., Stanford, Cal. 1973. 697-703.
9. Enslow, P.H. (ed). Multiprocessors and parallel processing. John Wiley. New York 1974.
10. Feustel, E.A. On the advantages of a tagged architecture. IEEE Trans on Comp C-22, 7, Jul 73, 644-656.
11. Gleary, J.G. Process Handling on Burroughs B6700. Proc Fourth Australian Comp Conf 1969.
12. Glushkov V.M., et al. Recursive machines and computing technology. Proc IFIP 1974, North Holland, 65-70.
13. Hauck E.A., and Dent, B.A. Burroughs B6500/B7500 stack mechanism, Proc. SJCC 1968.
14. Heart, F.E., et al. A new minicomputer/multiprocessor for the ARPA network. Proc AFIPS 1973 Natl Comp Conf. AFIPS Press, Montvale, N.J. 1973.
15. Higbie, L.C. The OMEN computers: associative array processors. COMPCON 72, 1972, 287-290. We refer to Fortran and Basic of this machine.
16. John McCarthy et al. LISP 1.5 Programmer's Manual. MIT Press. 1962.
17. John McCarthy. Recursive functions of symbolic expressions and their computation by machine. Comm ACM 3,4 April 1960.
18. Kautz, W.H., and Pease, M.C. Cellular logic-in-memory arrays. National Tech Information Serv, Nov 71, AD763710.
19. Keller, Robert E. Look ahead processors. Computer Surveys 7,4, Dec. 1975.
20. Landin, P.J. A program machine symmetric automata theory, in Machine Intelligence 5, Edinburgh Uni. Press. 99-120.
21. Lorin, Harold. Parallelism in hardware and soft ware: real and apparent concurrency. Prentice-Hall, N.J. 1972.
22. Lawrie, D.H. et al. GLYPNIR- A programming language for ILLIAC IV. Comm. ACM, March 1975, 157-164.
23. Magidin, M., and Segovia, R. LISP B6700 Implementation. Technical Report 5, 70, Computer Science Dept., IIMAS, National University of Mexico. 1974.
24. Miller, Raymond E. A comparison of some theoretical models of parallel computation. IEEE Trans Computers, Aug 1973, 710-717.
25. Miller, R., and Cocke, J. Configurable computers: a new class of general purpose machines. Intl Symp on Theoretical Programming. Ershov and Nepomniaschy (Eds) Springer Verlag, New York 1974, 285-298.
26. Patil, S.S. An abstract parallel-processing system. S.M. Thesis, E.E. Dept. M.I.T. June 1967.
27. Proc. Symp on Computer Architecture Dec 73 IEEE.
28. Proc. 1975 Sagamore Computer Conference on Parallel Processing, August 19-22. IEEE.
29. Raytheon Data Systems Array Transform Processor. Raytheon Data Systems. Norwood, Mass.
30. Record of Project MAC Conference on concurrent systems and parallel computation. June 1970. Available from ACM.
31. Rumbaugh, J.E. A parallel asynchronous computer architecture for data flow programs. Ph. D. Th, EE Dept., M.I.T.
32. Thornton, J.E. Design of a computer: the Control Data 6600. Scott, Foreman & Co.
33. Thurber, J.K. Associative and Parallel Processors. Computer Surveys (ACM) Vol 7, No. 4, Dec. 1975.
34. Wulf, W., and Bell, C.G. "C.mmp - a multi-mini processor". Proc. AFIPS 1972 FJCC. AFIPS Press Montvale, N.J. 1972.