MCC Technical Report Number: PP-355-86

AHR: A Parallel Computer for Pure Lisp

MCC Non-Confidential

Adolfo Guzman

October 1986

This paper was submitted for publication in a book "Parallel Computation and Computers for Artificial Intelligence" written by Dr. Janusz S. Kowalik of the Boeing Corporation.

The AHR computer, proposed in 1973 in Mexico City and operational by 1981, yielded data that helped to establish the feasibility of parallel computers. Its name, an acronym for Arquitecturas Heterarquicas Reconfigurables, emphasizes its reconfigurable, heterarchical (single-level) architecture.

This paper describes the design and construction of AHR, a multi-microprocessor that executes pure Lisp in parallel. Each microprocessor can execute any Lisp primitive, and the computer can support as many as 64 microprocessors. Each has its own private memory, and all share access to three common memories as well: 1) the *grill*, where the Lisp program(s) being executed resides; 2) the *passive memory*, which contains data; and 3) the *variables memory*, where bindings of variables to values reside. By using pure Lisp, the programmer does not need to be aware that the program is being executed in parallel, for it is not necessary to give a parallel command explicitly. All communication with the user is handled by a host machine, for which AHR is a back-end processor.

The architecture of AHE is predicated upon several premises. The operating system is small due to the synchronization among tasks being done, through hardware, by simple counters. No processor needs to communicate explicitly with another. A processor is never blocked; while there is still work to be done, a processor can proceed without wasting any time. In addition, a *process* itself is never blocked: either it has not yet started, or, if it is running, it is guaranteed to run until completion. Finally, this architecture has been realized: the AHR was built, and it worked as predicted.

MCC TECHNICAL REPORT

AHR: A Parallel Computer for Pure Lisp

October 29, 1986

Adolfo Guzman

Abstract

The AHR computer, proposed in 1973 in Mexico City and operational by 1981, yielded data that helped to establish the feasibility of parallel computers. Its name, an acronym for Arquitecturas Heterarquicas Reconfigurables, emphasizes its reconfigurable, heterarchical (single-level) architecture.†

This paper describes the design and construction of AHR, a multi-microprocessor that executes pure Lisp in parallel. Each microprocessor can execute any Lisp primitive, and the computer can support as many as 64 microprocessors. Each has its own private memory, and all share access to three common memories as well: 1) the *grill*, where the Lisp program(s) being executed resides; 2) the *passive memory*, which contains data; and 3) the *variables memory*, where bindings of variables to values reside. By using pure Lisp, the programmer does not need to be aware that the program is being executed in parallel, for it is not necessary to give a parallel command explicitly. All communication with the user is handled by a host machine, for which AHR is a back-end processor.

The architecture of AHR is predicated upon several premises. The operating system is small due to the synchronization among tasks being done, through hardware, by simple counters. No processor needs to communicate explicitly with another. A processor is never blocked; while there is still work to be done, a processor can proceed without wasting any time. In addition, a *process* itself is never blocked: either it has not yet started, or, if it is running, it is guaranteed to run until completion. Finally, this architecture has been realized: the AHR was built, and it worked as predicted.

1. Objectives of the AHR Project

Parallel processing now constitutes a major direction for computer development, and the future of this field is very bright. At the same time, the field is very difficult due to the complexities inherent in parallel systems. In 1973, these complexities made parallelism seem to be impractical, yet results obtained from a project undertaken that year at the Institute for Research in Applied Mathematics and Systems (IIMAS) at the National

† This work was carried out at the Institute for Research in Applied Mathematics and Systems (IIMAS), National University of Mexico, Mexico City. Other members of the AHR team were Luis Lyons, Luis Peñarrieta, Kemer Norkin, David Rosenblueth, Raul Gómez, Manuel Correa, Dora Gómez, and Norma A de Rosenblueth.

University of Mexico demonstrated that parallel systems were more feasible than had previously been assumed.

In 1973, researchers at IIMAS outlined several organizational premises for parallel computers, specifically one for a reconfigurable Lisp machine [1,2]. At that time, the proposal showed little promise to be fruitful. However, in spite of all the difficulties intrinsic to parallel systems, a computer designed along the lines proposed in 1973 was built and tested. Specific conditions existing in Mexico influenced the final form of AHR, primarily in terms of staffing and the curtailment of second-generation work. Yet results obtained from the prototypes at IIMAS in 1980 and 1981 yielded information useful for future developments in parallel processing.

The laboratory prototype of the AHR computer consisted of approximately 600 components, many of which were Large Scale Integration chips. Consequently, the AHR project represented, under classifications prevailing at that time, a complicated electronic design. This demand for such a huge amount of hardware devoted to a unique goal was new for IIMAS, and the project was a pioneering effort in computer research in Mexico.

Although the main goal of AHR was educational, the project had a multi-purpose aim. AHR was designed as a vehicle for future development, one that would, for instance, allow a programmer to write programs for a parallel machine without having to worry explicitly about parallelism. The first implementation of AHR supported pure parallel Lisp, but the ultimate goal for the machine was as a development tool for new languages and hardware for parallel systems. As part of its evolution as a development tool, the machine was available for students to use to learn and practice parallel concepts in hardware and software.

Four principle uses were envisioned for the machine:

- to develop hardware and parallel processing languages,
- to explore new ways to perform parallel processing,
- to provide parallelism in a way that was transparent to the user, and
- to provide a machine for students to use.

1.1. Characteristics of the AHR Machine

The design of AHR was predicated upon several architectural considerations, the chief of which are listed in Table 1.

AHR Design Premises
A general purpose parallel processor.
The absence of hierarchical distinctions.
An asynchronous operation.
The use of pure Lisp as the main programming language.
The absence of processor-to-processor communications.
A gradually expandable hardware design.
An allocation of input/output to a host.
A small operating system.

Table 1: AHR Design Premises

These design premises made the complexity of a parallel system more manageable. Because all the processors were at the same hierarchical level, no one of them was the "master"; hence, the AHR machine used a heterarchical organization, not a hierarchical one. Having pure Lisp as the main programming language eliminated side effects, setq's, and goto's. Up to a point, the computing power of the machine could be increased by simply adding more microprocessors. Communication between processors was minimized: since the design of the machine did not require processors to communicate directly with each other, they simply "left work" for some other processor to do, without knowing or talking to such a processor. Finally, all input/output was conducted by a host computer to which the AHR machine was attached as a slave or back-end processor.

Another factor in the management of the complexity dealt with the allocation of tasks normally associated with an operating system. The AHR machine, as such, had no software-written operating system. A normal operating system existed in the host processor, but it was not considered a part of the AHR machine. If the term *operating system* is taken to mean the "system's resources administrator," then the *grill*, or active memory—together with the *fifo* and the *distributor*—constituted an operating system.

Even this small operating system, however, was embodied in the hardware. The majority of the Lisp operations, as well as the garbage collector, which was not parallel, were written in Z-80 machine language. Special hardware also helped with handling list structures, free-cell lists, and queues. The efficiency of the system was further increased by the use of "intelligent memories" that could be accessed in several modes, among them *read, write, free this cell,* and *give me a new cell.* All modes were implemented in the hardware.

Since the purpose of the AHR Project was mainly educative and scientific, not much attention was paid to the normal things that give good service to users, such as I/O facilities, utility programs, and service routines. Because such resources are very well known, the AHR team concentrated instead on the *new* aspects of the computer, its unique and novel parts. By doing so, the team insured that the new ideas performed correctly, given the constraints of time and funds for the AHR project. The final report [3] of the AHR project contains further details of the IIMAS effort; the summary thus far in this article is based on that report.

1.2. Project Status

By the end of 1981, the AHR was operational to the extent of performing short test programs, which processed correctly. As a prototype, the AHR computer was not destined for normal use, but for verification of design ideas, a purpose for which it was completely sufficient. Consequently, the prototype became the "living proof" of the feasibility of the AHR premises. Since the machine had only a modest amount of memory, no large programs were run on it. As a prototype, it proved the correctness of its design assumptions and the validity of its way of operation. As a learning tool, it was invaluable because, in addition to confirming many design choices, it taught the AHR researchers many things to *avoid.*

The things to avoid primarily involved the issue of reliability for a sustained operation of several hours. The prototype failed frequently, due to the designers' insufficient experience in some technical aspects of its construction; some of the problems were due to the connections, weak contacts, a card size that was too large, short circuits, and reflections of pulses.

Because the prototype proved the success of the design, together with the fact that the machine was not able to support practical applications, Phase 1 of the AHR Project

terminated; version 1 of AHR ceased to exist at the end of 1982. As detailed in the final report [3] of Phase 1 of the AHR project, a second version of the machine—improved, robust, and reliable—was proposed, but due to a lack of resources, work on this second version never began.

2. The Parallel Evaluation of Pure Lisp

Since the order of evaluation of the arguments of a function does not matter in a pure applicative language, such as pure Lisp, it is indeed possible to evaluate them in parallel. The only rule that a designer has to follow is *applicative order evaluation*, meaning that a program has to evaluate all of the arguments of a function before evaluating the function itself.

Thus, a pure Lisp program is analogous to a tree in which the leaves are ready for evaluation. This evaluation can proceed in parallel by several Lisp processors: as each leaf is converted into a value, the "parent" of that leaf must receive not only the value but also the notification that one more of its arguments has been evaluated. When a node has *all* of its arguments evaluated, that node becomes a leaf node, and is itself ready for evaluation. Figure 1 shows the tree corresponding to a particular Lisp expression. Notice that next to each node there is a number, called a *nane*, indicating the number of arguments not yet evaluated.

2.1. Idealized Evaluation Model

If the "tree" represented by Figure 1 is placed in a common memory, which is accessible to all Lisp processors, then the processors can be controlled with an instruction of the following generalized form:

> Look for a node with *nane* = 0, and evaluate it.
> After finding its value, give the value to its parent.
> Then subtract one from the *nane* of the parent.

This rule permits the parallel evaluation of the tree from the leaves towards the root. As the internal nodes have their *nanes* diminished, the *nanes* eventually become 0 and, consequently, ready for evaluation. In other words, the nodes can then be captured by whatever Lisp processors are not busy evaluating a node; these processors search through the memory looking for "work to do," namely nodes with a *nane* of 0. Eventually, the whole tree is transformed into a Lisp result, namely a list or an atom.

Figure 2 illustrates the foregoing principle: the tree has been drawn with nodes corresponding to each Lisp primitive, and each node points to its parent. Initially, only the nodes named *VAR*, or the variables, are ready to be evaluated, but soon the evaluation proceeds towards the root. As a node is evaluated, its value is inserted in the corresponding slot of its parent, and the *nane* of the parent subsequently decremented. VAR Z, for example, can be evaluated as having the value of 5; this value is then inserted in the slot for its parent process, and the value of the *nane* of the parent decreases by one.

The above design, as is, contains the possibility of having sixty-four Lisp processors fighting for access to the common memory where the program to be evaluated resides. In such a case, the memory port becomes a bottleneck. However, the problem can be avoided if another memory, called

fifo or *blackboard*,

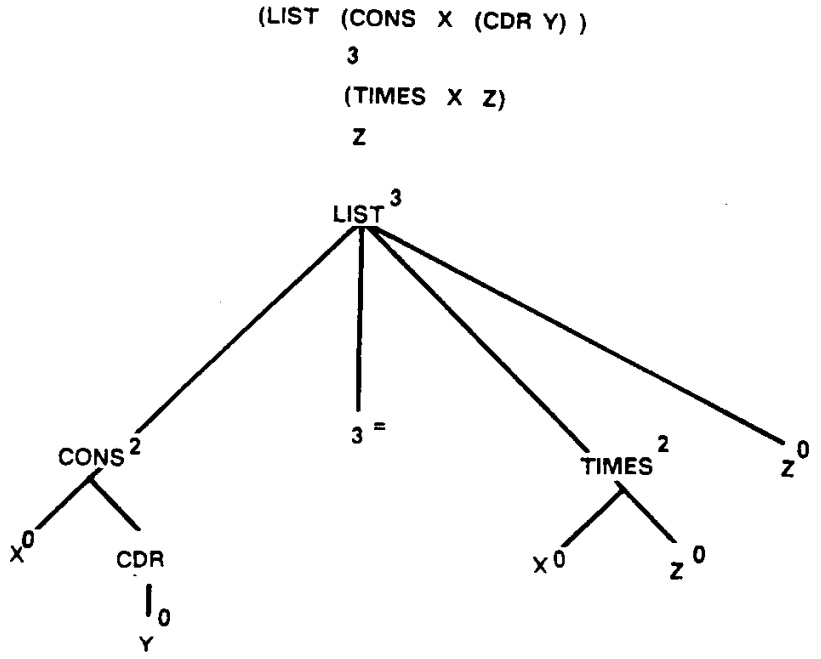

Figure 1: A Lisp Program
A Lisp program can be represented as a tree. The numbers near the nodes are the nanes, and nodes with a nane of "0" are ready for evaluation. Nodes with nane "=" are already evaluated (such as the constant 9).

contains only pointers to nodes with *nane* = 0. Additional details about the *fifo* can be found in the third section of this paper, entitled "The Parts of the AHR Machine".

In the above model, the common memory where the program resides—called the *grill*—has the property to destroy the programs, since it converts them into Lisp results; therefore, a master copy of the definition of each user-defined function has to be kept outside the *grill*. As discussed later, these definitions are kept as list structures, together with the other Lisp data, in another common memory called the

passive memory.

The bindings of variables to values must also be kept outside the *grill*; consequently, there is another common memory, called the

variables memory.

In addition to having a separate logical use, these memories are actually separate memories in the AHR machine in order to allow simultaneous access by different Lisp processors.

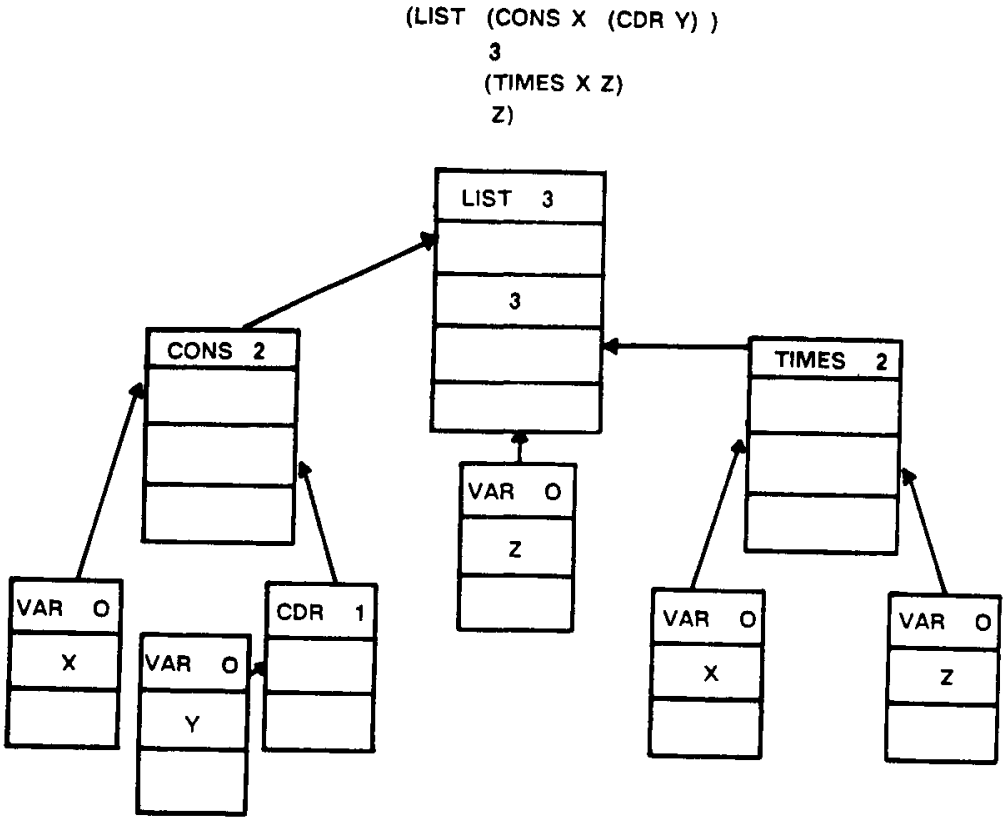

Figure 2: Representation inside the *Grill*
The nodes inside the grill represent the program to be evaluated in parallel.

2.2. The Handling of Conditionals

Evaluations of Lisp conditional expressions must be treated in a special manner for processing in a parallel system. For example, pure Lisp evaluates (*if p q r*) by first evaluating the predicate *p*. Then, exactly one of *q* or *r* is evaluated, depending on whether the value of *p* is "True" or "False." Consequently, the tree of (*if p q r*) can not be placed into the *grill*, because this action calls for the parallel evaluation of *p*, *q*, and *r*. However, if the expression (*if1 p*) is placed into the *grill*, then whenever *p* is evaluated, *if1* decides to copy *q* or *r* on top of itself. Thus, *if1* becomes either *q* or *r*.

COND, AND and OR are handled in similar ways, because their "arguments" can not be evaluated in parallel. Consequently, the program does not copy an entire list, as in *AND p q r s* ..., into the *grill*; instead, *p* is copied while *q r s*... stay in the *passive memory*, waiting, perhaps, for a later evaluation.

In conclusion, a function can be *evaluated* merely by copying it into the *grill*; there the evaluation takes place "automatically" with the help of the Lisp processors that search for nodes with a *nane* = 0.

2.3. Handling Recursion in an Idealized Lisp Machine

When the call to a user-defined function, as in, for instance, *factorial z*, is placed on the *grill*, the node then has the name *factorial*, a procedure that allows the evaluation of *z* to occur. For example, suppose the value of *z* is 4; if so, the node (factorial 4) is replaced by the node

((lambda (n) (if (eq n 0) ...)) 4)

In effect, this operation replaces "factorial" with its definition. The evaluation of a lambda expression, in which all of its arguments are already evaluated, produces no special problem, except that new bindings have to be registered in the *variables memory* prior to the evaluation of the lambda body.

3. The Parts of the AHR Machine

Various parts of the AHR machine are discussed below: a) the *mailbox* and injection mechanism for avoiding access bottlenecks, b) the various forms of memory, c) the Lisp processors, d) the communication media, and e) the host. Following the discussion of the parts is a section that describes how the machine works.

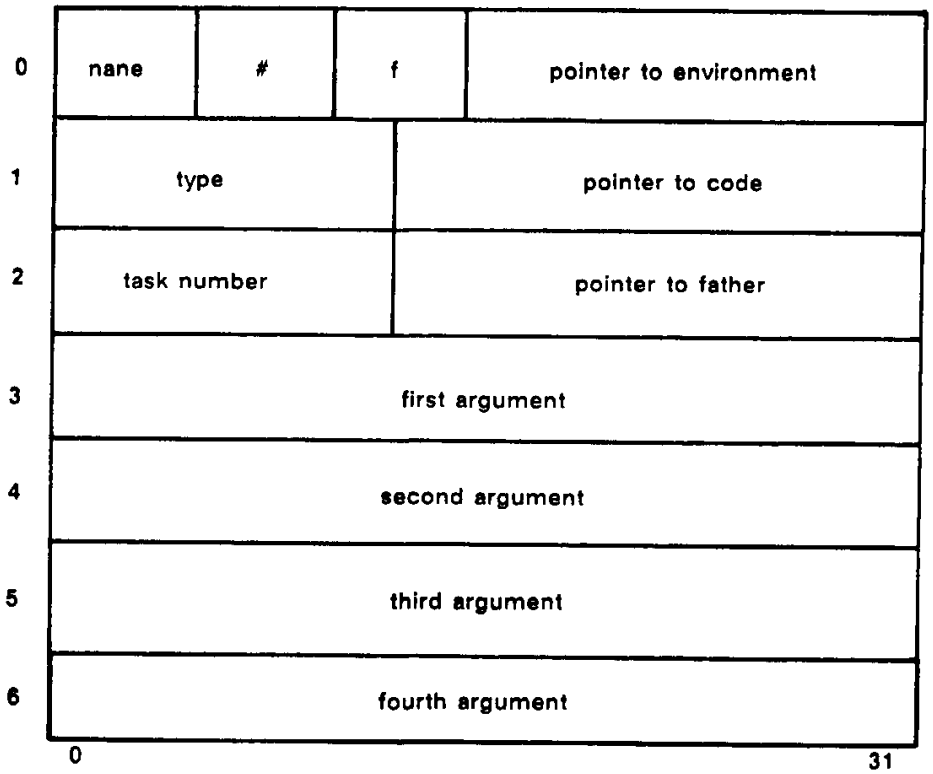

Figure 3: A Node
This node contains seven words of 32 bits; the last four are for arguments to the Lisp primitive, as indicated in the field type. The fields "#" and "f" indicate number of arguments and flags, respectively. Nane contains the number of arguments not yet evaluated.

3.1. The Mailbox and Injection Mechanism

One of the most critical parts of the AHR design involves the management of the memory known as the *grill*. To make efficient use of the *grill*, a Lisp processor has to access it for as little time as possible. On the other hand, what is transmitted from the *grill* to the Lisp processor is a node—that is, a primitive function—with all its arguments already evaluated. Figure 3 depicts such a node.

A node contains seven words of 32 bits each; however, in the instance of the AHR, a Lisp processor (a Z-80A) can only access 8 bits at a time. Thus, fetching a node from the *grill*, in this configuration, is quite slow. In addition, several Lisp processors may be trying to access the *grill*. To avoid this bottleneck, the situation is reversed: when a Lisp processor requests more work, the *grill* actually *injects* the node into the microprocessor.

This "injection" is done through a *mailbox* in the private memory of each Lisp processor. The *mailbox* is a 7-word (32-bits each) register that *shadows* some addresses in the Lisp processor memory. When a Lisp processor requests a new node, the processor goes into a wait state. The *grill* notices this request, extracts the previous result from the *mailbox*, and stores it in the corresponding slot of the parent node. The *grill* then subtracts one from the *nane* of the parent; this subtraction is an indivisible operation. The *grill* then checks to see if this new *nane* is zero; if so, the *grill* registers the parent in the *fifo*. At this point, the grill obtains a new node, with the help of the *fifo*, from the *grill* and injects this node into the *mailbox* of the (waiting) Lisp processor. Finally, the grill takes the Lisp processor out of its wait state.

Thus, the *grill* operates as a *smart memory*, and its capability as such is made possible by the construction of an additional piece of hardware, the *distributor*, which does all these things on behalf of the *grill*.

3.2. The Memories of AHR

There are five different types of memory:

1) *grill*,
2) *fifo*,
3) *passive memory*,
4) *variables memory*, and
5) *private memories* within each Lisp processor.

Each of these is discussed below, and Figure 4 shows a diagram of the overall machine.

3.2.1. The Grill

The *grill*, also known as the "active memory," holds the programs that are being evaluated. With the help of the *fifo*, the *distributor* of the *grill* passes data to/from the Lisp processors requesting access to the *grill*. The *grill* has to be very fast, in order to distribute nodes quickly to the Lisp processors. The *grill* can consist of up to 512K words of 32 bits and is divided logically into nodes, each with seven words. Version 1 of AHR, which contained only 8K words, had an access time of 55 nanoseconds per word [4, 5].

3.2.2. fifo

Also called the "blackboard," the *fifo* holds pointers to the Lisp nodes that are ready for evaluation but have not yet been evaluated. When the *distributor* decides to send a new "ready for evaluation" node to some Lisp processor requesting it, it pops the top of the *fifo* to obtain its address.

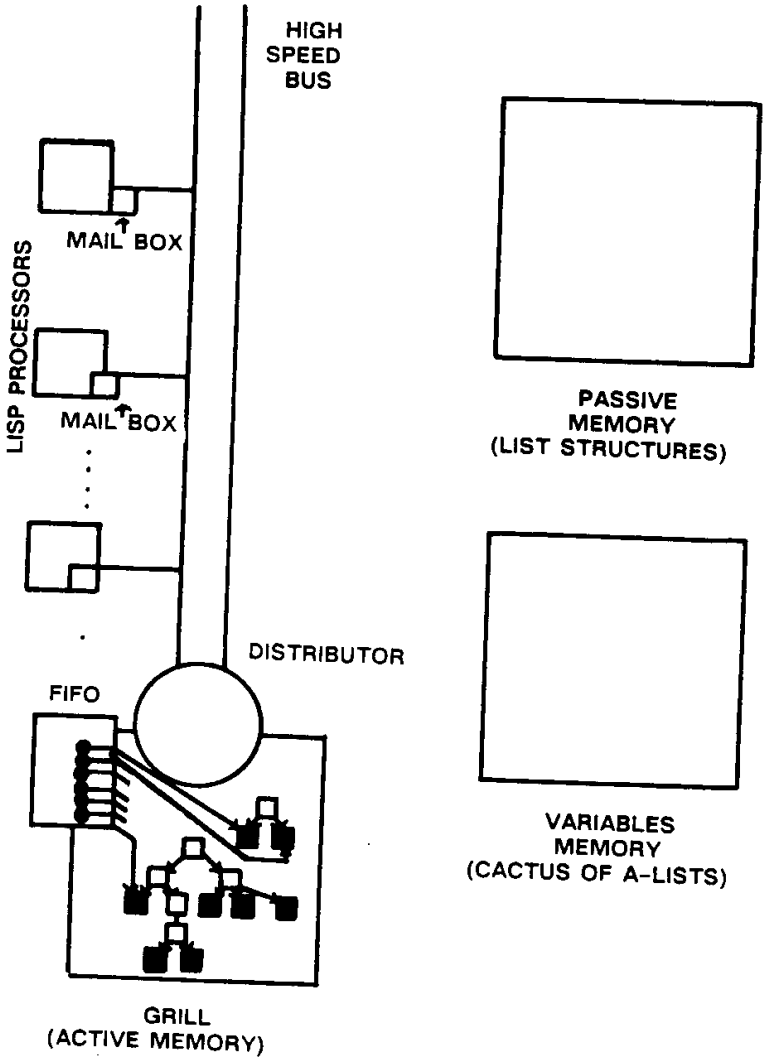

Figure 4: Diagram of the AHR Lisp Machine
The fifo contains pointers to nodes with nane = 0. There are connections (not shown) from each Lisp processor to passive memory and also to variables memory. The distributor sends nodes with nane = 0 to each Lisp processor requesting for more work to do. These nodes are placed directly in the mailbox of the requester.

3.2.3. Passive Memory

The *passive memory* contains "data," namely lists and atoms; this memory also contains the Lisp programs in list form. In the beginning of a process, the programs to be executed reside in *passive memory*. From there, they are copied by the Lisp processors onto the *grill* for their execution. As new data structures, otherwise known as "partial results," are built, these structures also come to reside in the passive memory, which can

have up to one million words of 22 bits, plus a parity bit. The tag bits were not implemented in hardware, but in software, as indicated in Figure 3.

In addition, the *passive memory* communicates with the memory of the *host computer* through a *window*. Version 1 of AHR had only a modest 64K words of passive memory with an access time of 150 nanoseconds per word. This memory is a single-port memory; an arbiter handles simultaneous requests from the Lisp processors, according to a fixed priority.

3.2.4. Variables Memory

The *variables memory* contains a tree of a-lists (association lists). Each element of an a-list is a variable name paired with its value. If AHR were a sequential machine, the *variables memory* would be a stack; instead, the variables memory looks more like a cactus, where branches and subbranches grow and shrink in parallel. A branch of this "cactus" grows after each Lambda binding.

When the value of a variable is needed in a node, the Lisp processor in charge of the evaluation of that variable uses the appropriate part of the "cactus" to begin searching for its value. Accordingly, the node also has a "pointer to the environment."

The *variables memory* consists of up to 512K words of 32 bits. Its lower half contains floating point numbers, and its upper half has has "environments" (the cactus of a-lists), which are lists of cells of 5 words each. Version 1 of AHR had a variables memory of 16K words with an access time of 150 nanoseconds. The variables memory is a single-port memory with a fixed-priority arbiter.

3.2.5. The Private Memory of Each Lisp Processor

Each Lisp processor has 16K bytes, with a maximum of 64K, of private memory (RAM + ROM). This "private" memory is where the processor's own machine stack resides, as well as the code in Z-80 machine language that executes each Lisp primitive.

3.3. The Lisp Processors of AHR

Each Lisp processor is a Z-80A with 16K bytes of private memory, each one connecting to the AHR machine through a *coupler*. This coupler contains:

a) a *mailbox* for fast access to the *grill*, and

b) latches to indicate petition access either to the *grill*, the *passive memory*, or the *variables memory*.

The Z-80 not only accesses these memories in the *write* or *read* modes, but the chip also addresses the *grill* in several other modes, namely *give me new work* and *take my previous result* and others [6]. This plurality of modes to access memories proved to be valuable for simplifying the design of AHR.

All the Lisp processors are connected to the *distributor* of the *grill* through the *high-speed bus*, which transfers a word (32 bits) in 55 nanoseconds. Note that this transfer occurs directly between the *grill* and the *mailbox* of the Lisp processor, while the latter is in a **wait** state.

Each Lisp processor knows how to execute every Lisp primitive; each one works asynchronously, without communicating directly with other processors. The processors "communicate" by leaving their results in the corresponding slot of the parent process, as

shown in Figure 3. Synchronization takes place whenever the *nanes* of nodes become 0. Nodes with a *nane* of zero signal a request, after their inscription in the *fifo*, for their evaluation.

Each Lisp processor is always either occupied in evaluating a node or ready to accept more work (another node). Only nodes with a *nane* = 0 come to the Lisp processor for evaluation; hence, the processor never has to wait, since all its arguments have already been computed. In the process of evaluation, the Lisp processor may have to access the *passive memory*, as in, for instance, taking CADR of a list. Likewise, the Lisp processor may also have to access the *variables memory* to obtain the value of a variable. If a processor wants to access the node that it is evaluating, that node is already in its *mailbox*; consequently, the node is available through the processor's own private memory.

Up to 64 Lisp processors are possible, but version 1 of AHR had only five. Figure 5 gives an overall view of version 1 of AHR.

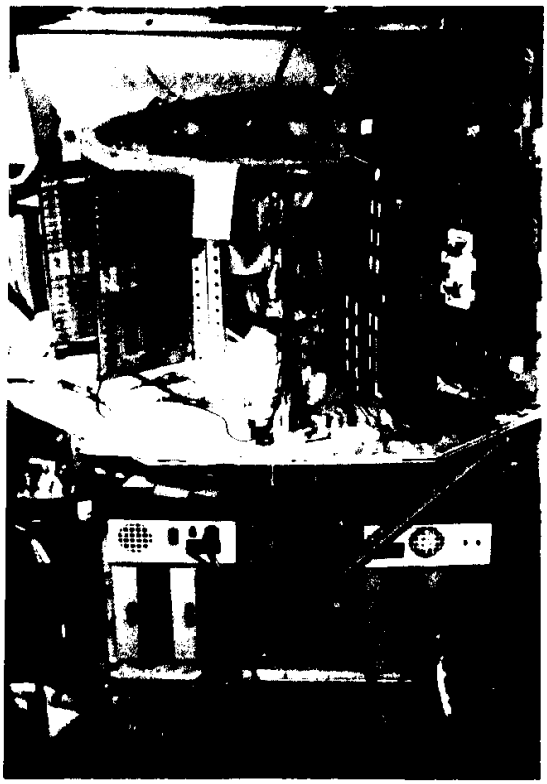

Figure 5: Front View of the AHR Machine
AHR is built as a circular structure. The top of this figur shows the Lisp processors, the different memories, the couplers, etc. The bottom left includes the host computer; to its right there is another Zilog Z-80, which is the distributor in its software version. The Lisp processors are not visible in this picture.

3.4. The Communication Media in AHR

The Lisp processors connect their mailboxes with the distributor of the grill through the high-speed bus. The AHR machine itself communicates with the host through a window. The *variable memory* and the *passive memory* each have a single port; thus, one of these memories can be accessed by a Lisp processor while the other memory is being accessed by another Lisp processor simultaneously.

3.4.1. The High-Speed Bus

The high-speed bus connects the *grill*, whenever the *distributor* decides to do so, to the *mailbox* of one of the Lisp processors requesting access to the *grill*. The bus transfers a node in 7 cycles of 55 nanoseconds each.

3.4.2. Channels to Variables Memory and Passive Memory

Each of these is a memory channel with an arbiter. The Lisp processors have a fixed priority in which the "closest" to the *distributor* has the highest priority. As soon as a Lisp processor requests access to a particular memory, it goes into a wait state; consequently, it can not request access to more than one memory at the same time.

3.4.3. The Coupler and the Mailbox

Each Lisp processor contains a *coupler*. As described in the preceding section and illustrated in Figure 7, this piece of hardware contains a *mailbox* and latches to indicate requests/grants from/to the Lisp processor.

3.4.4. The Distributor

The *distributor* monitors requests from the Lisp processors to access the *grill*. If requested, the *distributor* selects a processor and produces the desired access in accordance with the *mode* in which the request was made. Most frequently, the *distributor* is asked to do the following tasks:

- to take, from the *mailbox*, the previous result;
- store it in the parent;
- subtract one from the *nane* of the parent; and
- inscribe this parent into the *fifo* if its new *nane* becomes zero.

This entire process can be considered as a "take my previous result" request.

After this procedure is done, the *distributor* is usually requested to "give me new work." The *distributor* responds by transferring a node through the *high-speed* bus to the *mailbox* of the requester. At this point, the *distributor* signals the requester to proceed.

The first version of the AHR machine had a software *distributor* embodied inside another Z-80, as depicted in Figure 5. This software *distributor* was very useful for debugging purposes. Once the exact "code" for the *distributor* was known, a hardware *distributor* replaced it [7, 8].

3.4.5. The Low-Speed Bus

The *low-speed bus* is not really a part of the AHR machine. Its width is 16 bits, eight of which indicate which Lisp processor is addressed, and the others to carry data. This

low-speed bus is used in the following situations:

- to transmit to the Lisp processor, at initialization time, the code for the Lisp primitives;
- to gather statistics; and, when necessary,
- to broadcast to all Lisp processors the number of a program that needs to be killed [6].

3.4.6. The Window

Part of the *passive memory* of AHR maps into the private memory of the host computer through the use of a movable *window* of 4K addresses. From the viewpoint of the user, the Lisp programs are loaded into the host-machine memory but they actually go into AHR's *passive memory*.

3.5. The Host Computer

The host computer, as shown in Figure 6, is still another Z-80, although any other computer could be used. Though not actually part of the AHR design innovations, the host:

transfers data to/from AHR through the *window*,
signals to AHR to start working, and
waits for the "work has finished" signal from AHR.

The host accesses the result to be printed through the *window*. The host, which is also called the "I/O machine" due to input/output taking place there, uses its normal operating system, disks, etc. The host uses the AHR machine as a back-end processor whenever the host desires to execute Lisp programs. While the AHR machine is executing, the host can be doing other jobs, including non-Lisp-related ones.

4. How the AHR Machine Works

The section discusses the following topics: input, initialization, evaluation of programs, and output.

4.1. Input

Through the host, the user may develop a Lisp program, as, for example, in Figure 1. By loading the program from disk into the memory of the host, the user is really loading a Lisp list into the *passive memory* of AHR. After the program is loaded, the host signals AHR to begin execution and gives it the address in *passive memory* where the program to be evaluated resides. In a more realistic example, the user would have defined several functions, perhaps factorial, and then typed **(factorial 4)**.

4.2. Starting

At this point each Lisp processor is assumed to have had its programs loaded into its private memory. This memory contains Z-80 machine code for Lisp primitives, together with routines that handle the special hardware, such as the *mailbox* and the memory's access modes. All Lisp processors are idle and, consequently, requesting *work to do*. When the AHR machine receives the "start" signal, the *distributor* makes available a node, called the RUN node, to some Lisp processor. This node points to the program, stored in *passive memory*, that is to begin to be evaluated.

The program in *passive memory* is then copied into the *grill*; in the course of being "copied," the program is transformed from list notation to node notation. The RUN node does the initial copy/transformation, but soon more and more Lisp processors aid

Figure 6: Host Console with AHR
In the foreground (G) sits the console attached to the host; the AHR machine appears in the middle ground. At the top of the picture are partially visible two spies; these are television screens that display in ASCII the contents of selected blocks of private memories of the Lisp processors. These were very useful for debugging, but they are not referenced in the text. Here, two Lisp processors (H) are being debugged off-line.

in copying the program to the *grill*. The number of processors needed to copy a program is determined by the number of leaves or branches in that program. Copying is done in parallel: a processor that is copying (*foo x y z*), for example, can still copy *foo* while requesting some other processors to copy *x*, *y* and *z*, the latter of which, one can assume, are large S-expressions. This "request" on the part of the former processor takes the form of the creation of suitable nodes on the *grill*.

Nodes with *nane* = 0 are inserted, by the Lisp processors copying them, into the *fifo*, in order for other Lisp processors to execute them. At any given time, there are some Lisp processors copying the program while nodes with *nane* = 0 are being evaluated by other Lisp processors.

4.3. Evaluation

As explained in the section on the communication media and as shown in Figure 7, the *distributor* sends a node to each Lisp processor that requests more work.

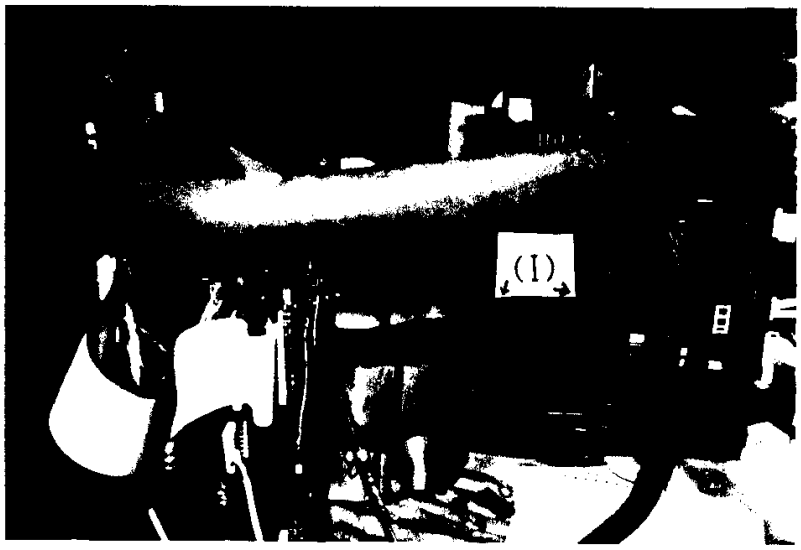

Figure 7: A Lisp Processor
Here a Lisp processor (I), to which a fat cable of wires from a development system is connected, is evident, complete with its coupler. The empty connectors in the circular structure of AHR are where additional (up to 5) Lisp processors are attached.

Before sending the node, the *distributor* extracts the previous result from the processor's *mailbox* and inserts the value into the parent. If no Lisp processor requests more work, the *distributor* sits idle, wasting *grill* cycles. If more than one Lisp processor requests work, a fixed priority arbiter choses one of them.

Thus, nodes with *nane* = 0 are consumed by the Lisp processors and converted into results. At the same time, the subtraction of 1 from the *nane* of the parent eventually creates more nodes with *nane* = 0. In addition, recursion—as indicated by the substitution of the word *factorial* by its definition (lambda (n) ...)—creates more nodes in the *grill*, some with *nane* = 0 and some with *nane* > 0.

4.4. Output

When the complete program has been converted into a single result, for instance a list, residing in *passive memory*, the AHR machine signals the host, giving it also the address where the result is stored. The host proceeds to read the result through the *window* and to output the data to the user's terminal.

5. What the IIMAS Team Learned from the AHR Project

The following analysis of the results from the AHR project includes a consideration of the successful premises as well as a discussion of the technical weaknesses to be avoided in similar experiments. In addition, a list enumerates certain choices that could be varied

in future experiments. Finally, the section concludes with an explanation of recommendations for further work.

5.1. Benefits of the AHR Project

Certain conclusions were far-reaching. The AHR project showed, by its very construction:

- that it is possible to build an entire parallel computer of a novel type in Mexico,
- that simulation methods are reliable,
- and that the AHR design is viable for other design projects.

The AHR design is very flexible: it offers concepts about interconnection and parallel work applicable to many homo- and heterogeneous systems. The architecture was not only proven to be a well-designed concept, but the prototype offered a myriad of practical checks during the course of development. The research effort also proved the validity of simulation methods, in this case, the use of Simula on a Burroughs B6700. The simulation methods developed by the team were checked in practice, and the simulation results had a sufficient correspondence to actual measurements. Finally, the project yielded practical data about the complexity of different parts of the computer [3] and about time delays in different parts of the computer [6].

5.2. Key Decisions for the AHR Project

Several key architectural considerations, as listed in the table below, contributed to the success of AHR.

Key Decisions for the AHR Project
To use pure Lisp.
To use a *grill* that was very fast relative to the slow Lisp processors.
To use simulation tools for the design of AHR.
Not to do garbage collection in the *grill*.
To use other modes besides *read* and *write* for memory access.
To use a host computer.

Table 2: Key Decisions for the AHR Project

The decision to use pure Lisp made the hardware design very simple, since there were no worries about side effects. The use of pure Lisp also freed the programmer from having to worry explicitly about parallelism, for special commands such as DOALL, FORK, JOIN, etc., were unnecessary.

The design of a very fast grill also had repercussions for the success of the experiment. One particularly valuable implementation was the procedure that allowed the *grill* to inject and extract work from the *mailbox*. That design made it possible to use a simple shared memory concept. In fact, if the time that it takes the *grill* to receive a previous result and emit a new work is **k** times faster than the average Lisp primitive, as executed in the Lisp processor, then up to **k** Lisp processors can be supported by the *grill* without any bottlenecks. A bigger **k** could be realized through several (not mutually exclusive) choices:

1) to have a hierarchy of memories,
2) to have cache memories for data in each Lisp processor, and
3) to increase the grain of parallelism.

The latter option of increasing the grain would be possible if larger Lisp primitives were

constructed that, in effect, would define a new Lisp-based language.

The small number of processors in Version 1 of AHR, as well as its memory constraints, prevented serious performance studies from being undertaken, yet the principles listed above could be the basis for new experiments.

As noted in Table 2, simulation was a key ingredient in the successful outcome of the project. The AHR project used simulation tools to detect the places and causes of non-effective work. By first simulating the operation of AHR, the design team found ways to simplify the design by migrating this non-effective work to the hardware. One can think of this type of work as "red tape," meaning non-effective, bureaucratic work. As a result, the *modes* of the *grill* were specially chosen to make the bureaucracy fast.

Another important factor was the elimination of the role of garbage collection from the *grill*. Because the *distributor* knows exactly when a node is no longer needed, it returns the node to the free-nodes list directly.

A key premise in the AHR design was the decision to use other modes in addition to *read* and *write* for memory access. This decision helped to blur the distinction between memory and processor. One example is the mode, *insert this node into the free-nodes list*.

The AHR experimental architecture would not have been possible, within the limits of the time and budget, without the use of a host machine. The use of a normal host allowed AHR to be designed as a back-end (memory-to-memory) processor for the host. In fact, the resources of the host, together with the use of the *window*, saved the team from having to write I/O routines, file management systems, editors, and so forth. The users had access to *unmodified* tools belonging to the host. Consequently, the team could concentrate on the novel parts of the AHR design.

5.3. Possible AHR Design Modifications

Some important decisions had definite alternatives, which were not, in all cases, thoroughly investigated at the time the machine was being constructed. In addition, there was a certain degree of arbitrariness in the chosen course of development. The AHR team does not know to what extent the following decisions would have improved the behavior of AHR:

- List vs. node forms for Lisp program storage.

  A Lisp program is kept in list form in the *passive memory*. Each time a program is copied to the *grill*, it then has to be converted to the node form, as shown in Figure 2. Perhaps it would have been better to have also kept the Lisp programs in node form in the *passive memory*.

- Fixed vs. variable size of nodes.

  As evident in Figure 3, the nodes are of a fixed size, and they can have as many as four arguments. Thus, Lisp on the AHR machine can not directly handle functions with more than four arguments. Using nodes of variable size would maximize the use of space in the *grill*—at the expense of a more difficult memory management.

- Serial vs. parallel garbage collection.

  The garbage collector on the AHR was a serial one. Since this factor caused all processors but one to wait, a parallel collector was clearly in order. However, the AHR team concentrated on other aspects of the AHR machine and did not pursue this topic.

- Immediate vs. delayed evaluation of *nanes* with a value of zero.

  When a Lisp processor, after subtracting 1 from the *nane* of a node, discovers it to be 0, it could go ahead and evaluate that node, instead of inscribing it in the *fifo*.

5.4. Weak Engineering Points

Some decisions had a negative impact either on the performance of AHR or on its design and construction. As such, these flaws need to be documented.

- Cards that were too large.

  The AHR cards were too large, and they tended to bend, causing printed-circuit connections to break or become loose. These flaws produced bugs and intermittent failures that were hard to diagnose.

- Card interfaces that initially were incompatible.

  The interfaces among the different cards were not precisely specified early in the construction phase. Consequently, slightly different—but incompatible—assumptions were made by the people building cards that were later interconnected. The decomposition of the computer into its constituent printed circuit cards was not done in such a way as to make each card simple and easy to test.

- Cables that were too long.

  The flat cables running among the different cards of AHR were just too long; there was no backplane. The length of the cables gave rise to pulse reflections and impedance mismatches, among other problems, as well as pulse distortions. There were too many lines going to each Lisp processor. The well-known techniques of multiplexing, time sharing, etc., were not used.

- Debugging tools that were insufficient.

  Too little detail was devoted to debugging tools, mainly due to scarcity of resources (people). Thus, debugging this parallel machine was difficult.

5.5. Recommendations for Further Work

Any additional work in parallel processing that considers the results gained from the AHR project will have a greater degree of success if the factors listed in Table 3 are observed.

Recommendations for Further Work
Introduce Impurities into Pure Lisp
Emphasize Hardware Improvements
Increase the Granularity of Parallelism
Provide Natural Parallel Constructs
Provide Natural Parallel Data Structures

Table 3: Recommendations for Further Work

5.5.1. Introduce Impurities into Pure Lisp

Most Lisp programs are written in impure Lisp, with setq's, rplacd's, goto's, etc. Learning pure Lisp in order to use a machine such as AHR may seem a very high price to pay. Moreover, there are some applications where there is not enough memory for continually

replicating data when only a small part changes. A program that modifies an array of a million pixels, for example, may have changes affecting only a few pixels.

Consequently, there is a need to introduce the assignment concept (setq), and similar arguments can be made for other impurities. At the time of the AHR project, the AHR team proposed to define *suitable* impurities, perhaps by inventing a few *new* types, to add to pure Lisp. These "impurities" could have been introduced only if the overall design maintained the following features of the architecture:

a) the simplicity of the AHR schema,
b) the suitability of the program for parallel systems, and
c) the transparent user interface for a parallel system.

A programmer should, ideally, have little or no awareness of programming a machine that is a parallel system. The AHR design team did not want to load (not heavily, at least) the programmer with special parallel constructs, with worries about synchronization, assignment of tasks to processors, etc.

5.5.2. Emphasize Hardware Improvements

Once tasks that need to be done efficiently are identified, it is worth the effort to develop hardware that does them quickly. For instance, the different modes of AHR memories, in addition to *write* and *read*, greatly simplified the design and made the system run faster. Simulations using the B6700 allowed the AHR designers to discover what parts were worth improving and speeding up. Along the same lines, actual machines now contain tag bits and forward pointers in hardware.

Monoprocessors with a program counter work better with a memory buffer register bigger than one word. If, for instance, the memory buffer register can hold two words, then **a** fetch issued to the memory location **a** can bring both the contents of **a** and **a+1** to the CPU, thereby cutting the effective fetch time in half. This trick works because the next instruction is normally executed right after a prior one instruction is executed. Why not, then, when the Lisp processor asks for cell **a**, bring also the cell pointed to by the **car** of **a** as well as that pointed to by the **cdr** of **a**?

5.5.3. Increase the Granularity of Parallelism

The AHR machine sends nodes for evaluation to the Lisp processors that are too small, perhaps, compared with the overhead involved in using the *distributor*, the *high-speed bus*, etc. Thus, there could be a concept such as *send bigger nodes*. Implementing this concept would necessitate the invention of higher primitives in Lisp. Lisp could be enriched with functions that "compute more" once they have all their arguments. To some extent, Lisp already does that, for it provides MEMBER as a primitive function, primarily for efficiency reasons, although MEMBER could easily be a user-defined function.

5.5.4. Provide Natural Parallel Constructs

A collection of Lisp functions—such as Mapcar, Forall, For-the-first, Parallel-and, etc.—would entice the user to "think parallel." The use of *future* [9] is natural in this sense, since (future (foo x y)) means the same as (foo x y), but the "future" version is intended to be *computed in another processor*. Nevertheless, asynchronous evaluation creates indeterminism in the presence of side effects and assignments of global variables; therefore, it has to be used carefully with impure code. One example of a language that provides natural parallel constructs is L, a language developed for image processing [10].

A language that emphasizes natural parallel constructs can be described [11] as follows:

- it would create primitives that operate on *all* processors;
- it can be interpreted as an order to the *whole* parallel computer, as opposed to CAR, which is an order to *one* of the Lisp processors;
- and it captures, in a MIMD machine, the "spirit of programming" a SIMD architecture.

5.5.5. Provide Natural Parallel Data Structures

The **cdr** of a list is generally bigger than the **car**. This "unbalanced" data structure predisposes the programmer to a sequential mode of computation: there is a tendency to think, "First I will handle now the **car**, and then later I will take care of the rest." Perhaps something like the xapping construct [12] is needed.

6. Conclusion

The history of the AHR project was not only a precursor of later work with parallel systems, but it also posed some questions that are still of intrinsic interest to computer research.

Acknowledgments

The AHR machine is the result of many hours of work by the AHR team. A special word of gratitude is due to the IIMAS director, Dr. Tomaś Garza, and the IIMAS administration, who all made their best efforts to help the AHR project.

The financial support from CONACYT (grant #1632) is acknowledged, as well as CONACYT's help in the interchange of visiting scientists between IIMAS-UNAM and the Institute for Control Sciences, USSR Academy of Sciences.

Thanks are due to the Parallel Processing program at the Microelectronics and Computer Technology Corporation (MCC), particularly Dr. Stephen Lundstrom, its Vice President and Program Director, for his continuous support. A special word of appreciation is also due to my colleagues of the Lisp group for their fruitful comments and help.

References

1. Guzman, A., and Segovia, R. *A Configurable Lisp Machine.* IIMAS-UNAM Technical Report Na 133 (AHR-76-1). National University of Mexico, P. O. Box 20-726, Mexico City, 1976.
   Hereinafter, all reports such as this one are signified by the note, "IIMAS Technical Report."
2. Guzman, A., and Segovia, R. "A Parallel Reconfigurable Lisp Machine." *Proceedings of the International Conference on Information Science and Systems.* August 1976. University of Patras, Greece, 207-211.
3. Guzman, A., and Norkin, K. "The Design and Construction of a Parallel Heterarchical Machine: Final report of Phase 1 of the AHR Project." IIMAS Technical Report NA 308 (AHR-82-21), 1982.
4. Guzman, A. "A Heterarchical Multi-microprocessor Lisp Machine." *Proceedings of the 1981 IEEE Workshop on Computer Architecture for Pattern analysis and Image Database Management.* Hot Springs, Va., 1981. IEEE Catalog 81CH-1697-2.

5. Guzman, A. "A Parallel Heterarchical Machine for High Level Language Processing." In *Languages and Architectures for Image Processing.* M. J. B. Duff and S. Levialdi, eds. Academic Press, 1981.
Also published in *Proceedings of the 1981 International Conference on Parallel Processing*, 64-71. IEEE Catalog 81CH-1634-5.

6. Guzman, A., *et al.* "The AHR Computer: Construction of a Multiprocessor with Lisp as Its Main Language." IIMAS Report Na 253 (AHR-80-10), 1980. (In Spanish.)

7. Gayosso, N. "The *Distributor* for the AHR Machine: The Microprogrammable Hardware Version." B. Sc. Thesis. ESIME-National Polytechnic Institute, Mexico City, 1981. (In Spanish.)

8. Peñarrieta, L., and Gayosso, N. "Alternatives for the AHR *Distributor.*" IIMAS Technical Report Na 302 (AHR-82-20), 1982.

9. McGehearty, P., and Krall, E. "Potentials for Parallel Execution of Common Lisp Programs". *Proceedings of the 1986 International Conference on Parallel Processing*, 696-702. IEEE Catalog 86CH2355-6.

10. Barrera, R., Guzman, A., *et al.* "Design of a High-level Language (L) for Image Processing" in *Languages and Architectures for Image Processing*, M. J. B. Duff and S. Levialdi, eds. Academic Press, 1981.

11. Lundstrom, Stephen. Personal communication.

12. Steele, G. L., and Hillis, W. D. "Connection Machine Lisp: Fine-grained Symbolic Processing." *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, 279-297. ACM Order No. 552860.

Index