A MULTIPROCESSING SYSTEM
FOR THE DIRECT EXECUTION OF LISP

Rhon Williams
Department of Electrical Engineering
University of Illinois at Urbana-Champaign

Abstract

Current implementations were found to be impractical for airborne use due to LISP's incompatability with conventional computer architectures. Direct execution of LISP with tasks distributed between three processors, seemed to be a workable solution. The language was analyzed, and a special token was devised, using a descriptor with a single pointer. Through careful distribution of responsibilities, control and data flow between the processors was minimized. Significant memory savings resulted from ASCII storage and real time garbage collection. A simulation was used to help estimate execution times and showed a factor of 50 to 100 increase in speed. Thus, through the direct execution of LISP by a multiprocessing system, Computer-Aided Decision-Making could be implimented to enhance the safety of flight operations.

Introduction

Implementing Computer-Aided Decision-Making (CADM) in an airborne environment was found to be unfeasible due to current implimentations of LISP, and so a new approach was needed. The CADM program was an effort to develop the conceptual framework for using programmed intelligence to assist a pilot in various tasks such as navigation, fuel management, and degraded mode operations. It is impossible to use exhaustive techniques because of the inherent combinatorial complexity, so artificial intelligence techniques are used. Included are pattern-directed invocation of procedures, demons, and self-generated special purpose procedures. Also CADM must handle conditions of imprecise, imperfect, and possibly conflicting data [1]. A powerful general purpose language with these capabilites as well as allowance for future modification and expansion was needed, along with the full benefits of a high level language, such as dynamic types and structures and automatic memory management.

LISP became a logical choice because it is widely used in the artificial intelligence community and is one of the few languages meeting the above prerequisits. LISP contains built-in functions for building and dissecting list structures of objects which can be of any kind, including another list structure. No advance specification of type or structure is required and words may be of variable and dynamic length. Complete memory management is automatic and transparent to the user. Since the list structures are dynamically changing in size, memory is used up and abandoned randomly. It thus requires garbage collection to reclaim the unaccessable locations and is provided as needed. LISP is based on a number of stacks and thus allows complete recursion and makes possible context switching. User functions may be defined and can be used recursively, without undue overhead penalties, thus promoting efficient encodings. Since LISP both operates on and executes lists of words, using a simple control structure, self-generated code is possible, and compilers and systems can be written directly in the host language. FORTRAN, although commonly used in most mathematical applications, requires fixed types and lengths, has poor string manipulation, no recursion, and is not interactive or self-generating. PL/1 does allow recursion, has better string handling, and less structured types, but is large, bulky, and without self-generating code. ALGOL is somewhat better but, again falls short of LISP. Furthermore, these languages are all biased towards known machine architectures and not the way people think.

Unfortunately, these same advantages of LISP cause the difficulties in interpreting or compiling the language on conventional machines. In fact, even the sample CADM programs used a significant part of a DEC-10 system to interpret and execute. Furthermore, rewriting or converting CADM programs to another language would defeat many of its advantages and cause repeated problems whenever it was to be modified, thus limiting future applications. For example, dynamically varying data and data types, which give LISP its power, are computationally costly to convert to more efficient assemblers because of the nature of most machines. The self-generated code allowed by LISP is even worse in this respect. Thus even a one time conversion of LISP programs to assembler, having eliminated the inefficiencies of a compiler, still would not solve the basic mismatch between LISP and most machines. The need to make CADM realizable, the inherent incompatibility of LISP with conventional architectures, and the need for airborne equipment to be compact and reliable, led to the idea of a machine which directly implements LISP. Finding no existing guidelines to help design a HHL architecture, the project expanded to develop the general concepts useful in determining the achitecture best suited for directly implementing any high level language.

Analysis of the Language

Since the intent of a high level language machine is the efficient execution of a particular language, it is only reasonable that the nature of that language should shape the design of the

hardware. Thus, the first step in the design of the architecture is a careful analysis of the language to be implemented. This begins by looking at the purpose and common usages of the language, that is why is it useful and what kinds of problems are usually solved with it. Then the structure and form must be examined to initially determine the architecture, control structure, and word formats that will be required. Care must be taken to guarantee that parts of the language which give it particular power or special capabilities are not lost in a trade-off. Similarly, parts of the language that cause difficulties or inefficiencies in current implementations should be given careful consideration to insure that the same pitfalls are not carried along or amplified, but are alleviated even if special efforts are required. Finally, special abilities and acceptable limitations or simplificatons due to the particular application in mind, should be noted. LISP has a simple but powerful structure which allows complex list structures without formal type declarations, complete recursion, and even self-generated code. These attributes suggest a stack oriented machine which operates on tokens consisting of descriptor bits attached to a pointer.

Word format and memory size are heavily dependent on the number of bits used in this descriptor and pointer. Choice of the descriptor bits is one of the most important issues because it can simplify execution, routing, and testing considerably, but additional bits can significantly increase memory size. Similarly, the pointer size must be sufficient to address all memory (dependent on program size) unless special methods are employed to allow indexing or relative addressing. Most implementations of LISP use a CONS cell consisting of two pointers, the first called CAR, and the second called CDR, to build their structures. Analysis of the CONS cells used in typical list structures [2] has shown that the pointers often point very nearby, even the next cell. Thus considerable memory savings is possible with a better encoding, particularly for common linearized lists.

With this information in mind, a special cell can be defined for this system. It consists of the descriptor bits combined with a single pointer. Based on the concept of the CDR code used in MIT's LISP Machine [3], two of the descriptor bits are called NW (next word code) and keep the power of the cell in a more bit efficient manner, and in fact, make the difference transparant to the programmer.

NW=00 means that the CDR of this location is NIL

NW=10 means that the next word is the CDR

NW=11 means that the next word is a special node value, and the following word is the CDR

NW=01 means this location is a link pointer with no relation to the next word

Figure 1

For example, equivalent structures are shown in Figure 2.

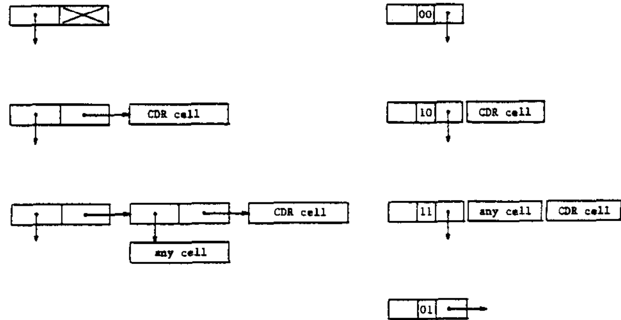

Figure 2

This allows a consistant size and format for all data forms, and gives a significant savings in memory for larger structures. For instance, assuming a 21 bit pointer, the structure

((A B C) (D E F) (G))

would require ten words of 42 bits or a total of 420 bits, using a normal LISP cell, whereas using the new cell requires ten words of 32 bits or a total of 320 bits a 25% savings even with the descriptor (See Figure 3).

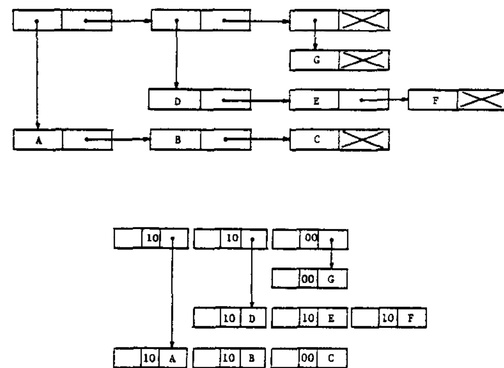

Figure 3

Choosing the pointer size is dependent on the application and program size expected. Current implementations such as the DEC-10 use an 18 bit address which has been found sufficient for all but the largest programs. On the other hand, with a faster more efficient implementation like the HLS, larger and more powerful LISP programs will become more practical and a larger address space will be needed. Choosing a 32 bit word as a first guess, and using eight bits for type information, two bits for the NW, and one for the indirect bit, leaves 21 bits for the pointer. This is a reasonable size since it gives eight times as much addressable space directly, but due to more efficient encodings, real time garbage collection, and reduced program space using high level ASCII storage, the actual gain is generally much larger.

Particularly in an application such as airborne use, where size and weight are important, the memory savings available with a high level

language architecture like the MLS is significant. The first savings occurs due to the semantic conciseness that is a benefit inherent in the high level language itself, in comparison to the code required for assembler languages. Next, since the program is stored and executed as source, the compiler's code and object code are eliminated entirely. Direct storage in ASCII and the use of descriptors saves space as shown using an expression which defines the factorial of N.

```
(DEFUN FACT (N)
       (COND ((EQ N 0) 1)
             (T (* N (FACT (- N 1))))))
```

These 61 characters would be packed five per word using 13 words or 226 bits, whereas typical LISP storage uses pointers which require 23 words of 42 bits or 966 bits total.

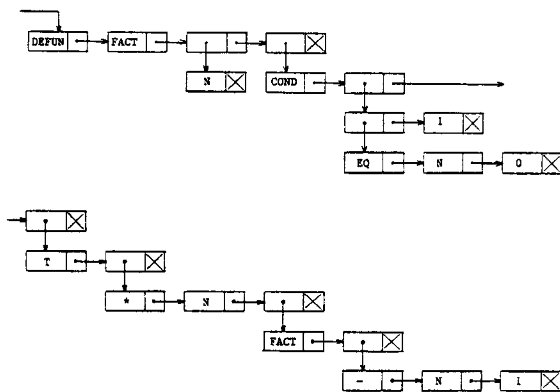

Figure 4

Finally, real time garbage collection requires less memory because of the lower overhead workspace needed and faster recycling of space containing garbage.

In order to define the descriptor bits, the language must be dissected to find to find the types and classes of data and operators. Including a special code for the raw ASCII, and another for processor control, seven catagories can be identified.

Code	Use
0,1	Raw ASCII program code - 5 six bit characters per word
2	Atomic literals - numbers and character strings
3	Symbols - atomic and non-atomic pointers
4	Special pointers - labels, arrays. etc.
5	Structure elements - property list elements
6	Functions - all operators
7	Control/Communication - stack markers, memory requests

Figure 5

Choice of Architecture

After the high level language has been carefully analyzed. an architecture must be found which will best implement the language's particular features in the appropriate environment. Design of the architecture is done by an iterative process which gradually blends in the necessary capabilities. In this design multiple processors were chosen to allow the functional division of tasks and memory accesses. Each section can be optimally structured and coded for its particular tasks, and thus the allocation of responsibilities is of particular importance. To help in this decision, typical LISP processes were broken into subunits which were then grouped on the basis of time, similarity of function, and memory use. Another consideration is that the control problems often associated with multiple processors can be minimized by making the division lines near points of least control and data flow. Typically, a natural division occurs giving a preprocessing section, an execution section, and a memory manager. These may also be subdivided with more processors to better distribute the workload, and often I/O operations are set aside and handled in a conventional manner. This combination allows a real time interpretation and an execution phase to be pipelined together, and makes it possible to do garbage collection concurrently.

The specific implementation of each section needs some consideration also. In view of the application and its environment, current trends highly suggest the use of microprocessors as the core of each section. In general they are lower cost, smaller, and lower power than equivalent random logic. and their limitations in speed are usually overcome by the use of multiprocessing and pipelining techniques. Bit slices in particular, allow great versatility in even permitting sections to use different word lengths and different CPU/memory/bus combinations. Overall their greatest value is the ability to microcode at a very low level, thus making each section optimum for its type of processing, while keeping development costs and interfacing minimized. Although the initial design should not be heavily biased towards a particular part or family, keeping in mind the types and capabilities currently available will make the final implementation more straightforward.

In the case of the MLS (Multiprocessing LISP System), examination of LISP gave a number of suggestions to the architecture. LISP is usually run interpretively and is easily divided into interpretive type tasks such as symbol translation and table maintaince; and execution type tasks such as function execution and conditional branches. Also, since LISP is memory intensive the expected interference between memory requests can be reduced by dividing memory among the processors. Thus functional divisions between sections of the MLS were on the basis of time order needed, compatibility with other tasks in the same group, the type of processing needed, and by the way memory was used. The first pass architecture chosen is shown in Figure 6.

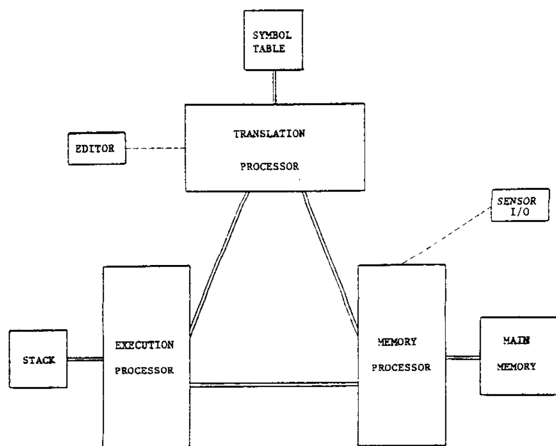

Figure 6

The translation processor (TP) is assigned the symbol oriented tasks needed to translate the raw ASCII into tokens to be executed. This begins with getting a word from main memory and sorting the delimiters and symbolic words. The spaces are dropped and the parentheses are made into control tokens signifying changes of lexigraphical level or list boundaries. The symbolic words are hashed into the locally maintained symbol table to find the appropriate token. This could be a pointer into main memory, a subroutine starting address, or special control information. Results of these different cases are always tokens to be passed to the Execution Processor (EP) via a FIFO buffer.

The Execution Processor (EP) takes the tokens from this buffer and examines their descriptor bits to determine what to do. Operands are generally placed onto the argument stack and operators placed on the function stack, possibly performing some actions first. Control tokens may cause a marker to be pushed on a stack or a function popped to begin execution. Both stacks are contained in the EP's local memory section, and does not interfere with other memory accesses. The EP requests data from the Memory Processor (MP) as needed and causes a branch if necessary, by sending out a new value for the PC (program counter). This signals the MP to ready a new word for the EP, clears the FIFO buffer, and restarts the EP process.

The third section consists of the Memory Processor (MP) and main memory. Main memory contains the stored program code and all list structures as well as data. The MP accesses the main memory upon request of another processor, controls information on the bus, and does the garbage collection as a background task. Additional capabilities such as associative lookups could also be added.

With the tasks assigned to the three sections as above, control and data flow at the interfaces should be examined. From the TP to the EP a FIFO buffer handles all cases in a simple consistant manner, with tokens carrying the information. In the special case of a branch, a signal from the EP is used to clear the buffer and interrupt the TP. Information to and from the MP is along a bidirectional bus connecting all processors. Since the MP always appears as a response. it watches for a request to be presented and switches the bus to the needed mode. The first word provides the control information with data in the next one or two if needed. Thus by dividing the tasks and memory appropriately, a minimum of control and data must be passed between processors.

Processors available at the start of the project were examined and Intel's bit slice CPU choosen for its versatile bus connections. The functionally divided processes, memories, and their interconnections were further developed and specified as shown in Figure 7.

Typical Operations Within the MLS

As discussed above, the MLS (Multiprocessing LISP System) consists of three loosely coupled processors operating asynchronously. Each has its own control program and is microcoded to execute its own specialized functions. Each also has its own distinct memory area which is structured to fit the section's particular needs. Communication between processors is via buffers and synchronized by a handshake flag when necessary.

More specifically, The MP (Memory Processor) is specialized for efficient memory accesses. Its associated memory is the main memory used to store program code, usually in packed ASCII, and data space, usually list structures. Both areas may actually be intermixed and used without distinction with the only difference being the format type found when accessed. In most cases accesses will occur due to a request and pointer from another processor, and the MP will access main memory and follow any indirect access as needed. The latter occur when a descriptor bit (INDIRECT) is found set in a word accessed, and indicates that this is a pointer to the wanted word. This capability gives great flexibility and easier operation of the garbage collection which is also a resident part of the MP. In this way the other processors need not know the actual location of a word but can let the MP store it wherever best. Since the MP is optimized for efficient address calculations and traces out indirects automatically, there is little cost to the overall processing rate.

Another type of request would be for a specific type of data, such as found in the property table, in which case the MP must index and follow lists until finding the requested data item. Other simple requests would be for the word pointed to by the current program counter, or for a pointer to a block of available space. Additional capabilities could be be added to allow contents addressing or associative lookups.

Since LISP requires garbage collection, it is assigned to the MP as a continually running background task. This can be implemented using

the built-in capability of the indirect bit, using a method such as suggested by Baker [4]. In an airborne system such as CADM, the MP would also be handling data base updates from an I/O processor in real time. During development, an editor processor could be attached to the memory bus to include the ability to store or recall at any memory location. If in some cases, virtual memory is required, it would also be supported by the MP.

Information exchanges are initiated by a processor other than the MP loading a control word and setting its buffer-full flag. The MP continuously polls these flags with a simple fixed priority, grants bus control to a caller, reads a word through its own bus buffer and branches to a routine appropriate for the request. On receipt of the word from the bus, the MP disables bus control, and clears the proper buffer-full flag, indicating completion of the handshake. The action to follow is then dependent on the request and may involve accepting one or two more words, or giving a word in response. Since the bus is bidirectional and tristate controlled by the MP, information can easily pass back across with a similar handshake.

The TP (Translation Processor) is specialized for character manipulations, symbol mapping, and type interpretations, and its associated memory is used exclusively for symbol tables. In normal operation, the TP requests the next word of program code from the MP (as pointed to by the program counter). This word contains five six bit ASCII characters, which must then be built into recognizable symbolic words or delimiters (spaces or parentheses). With the help of the byte rotate buffer, the TP would sort out (for example) a left parenthesis, then branch to a subroutine for additional action. In this case it would, among other things, increment the lexigraphical counter kept within a status register. construct the proper control token, and load that into the FIFO buffer to the EP (Execution Processor).

The TP would then return and scan characters until a delimiter occurs or the five charaters are exhausted. If a delimiter is found, the characters are assumed to form a known symbol and are sent to the hash code routine. This routine shifts and combines the ASCII bits to find a hash coded value which can be used as an entry point to the symbol table. If instead of a delimiter, the characters were exhausted, then the byte rotate buffer is reloaded from the MP and more characters are shifted in until again a delimiter or five characters have been assembled. In this latter case, the hash code is found and symbol table tenatively accessed. Then if more characters are found to belong to the same symbolic name, the hash is recalculated combining the new information. Thus with only slightly more overhead, any size symbol can be accomodated.

When a result is returned from the symbol table, one of its descriptor bits indicates if any special action is required. Otherwise the type is checked to determine if any preliminary main memory accesses should be made. The possibly amended value or a token representing a function, is then loaded into the FIFO output buffer. Thus spaces have been dropped, parentheses become simple control tokens, and functions and symbols are mapped into tokens. As a result except for some specially handled cases, and possible memory accesses, this part of the TP is nearly a simple

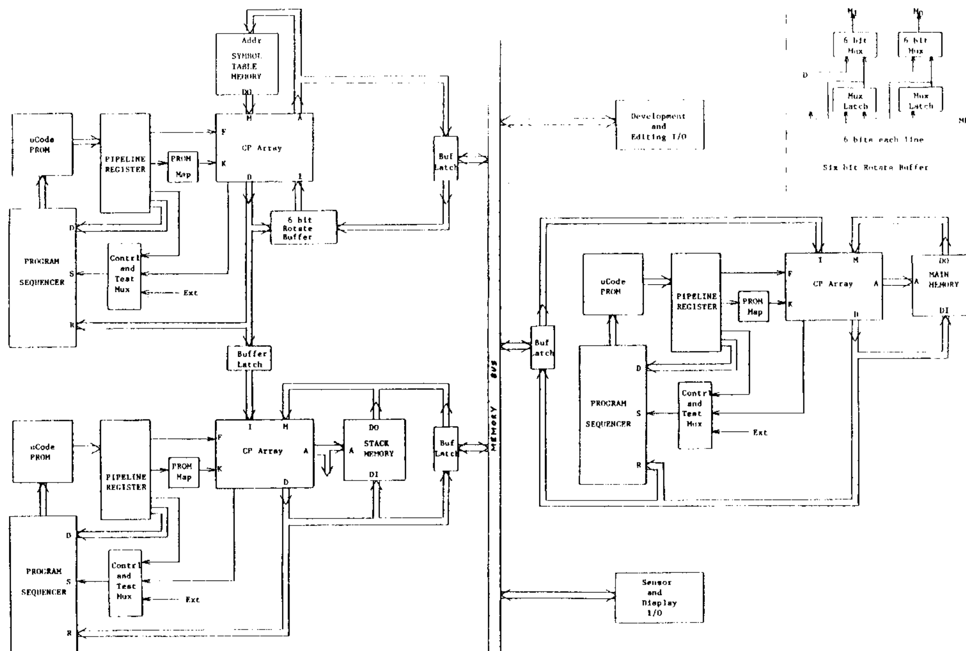

Figure 7

one to one mapping. The FIFO output buffer accumulates the tokens and allows for speed variations between the TP and EP, with its proper size determined through simulation.

The Execution Processor (EP) reads tokens from the FIFO buffer and branches to specialized microcode depending on the type of token. The memory associated with the EP is structured as two stacks. One is the function-stack, which saves the function type tokens until all of its arguments are ready. The other is the argument-stack, which takes the arguments as they are read and holds them until needed by a function. It also holds the results of a function execution, since they become the arguments for other functions. The argument-stack may also contain list markers which are used to delineate a variable length group of arguments which are to be operated upon together.

The top level of the control program in the EP is responsible for fetching a token from the buffer whenever needed and available. It then checks the type descriptor for a control token which would represent a left or right parenthesis. If a left parenthesis is found, the LEXCNT (lexigraphical level counter) is incremented and tenatively sets a LISTFLAG which indicates the beginning of a list. Both LEXCNT and LISTFLAG are kept as a part of the status word maintained in an internal register. If a right parenthesis is found, it normally indicates the end of a function's arguments, and thus the function-stack is popped, and a branch made to that specific subroutine. This subroutine would typically pop a number of arguments from the argument-stack, operate upon them, and return a result to the argument-stack with possible side effects along the way. If the top level type check finds a function, it clears the LISTFLAG and checks another descriptor bit INIT. When set, this bit indicates that the function should be called immediately for some preliminary actions. Then if the function is to be resumed again later, it pushes its own re-entry point onto the function-stack. When INIT has not been set, the function is simply pushed on the function-stack for later processing.

Any symbol found during the top level type check is pushed on to the argument-stack for later use. Note this causes a function's arguments to appear in the stack in reverse order to what may be needed during execution. Often the order does not matter, but when it would, they can be reordered by the choosy function just prior to execution by temporarily shifting them to the function stack.

A little more explanation is needed concerning the LISTFLAG. This is used because most lists of words enclosed by parentheses in LISP are of the form

(FUNCTION ARG1 ARG2 . . .)

but may also be just a list of words to be treated as a unit. This latter case needs to be marked on the argument-stack by a list-marker at the beginning and end. Thus as previously mentioned, the LISTFLAG is tenatively set whenever a left parenthesis is encountered. Then if a function follows, the LISTFLAG is cleared and the next associated right parenthesis is interpreted as an end of arguments. If instead, the LISTFLAG is still set when the right parenthesis is encountered, the end list marker is pushed on the argument-stack and control is returned to the top level.

To help derive more detail and evaluate the design, a simulation was run using GPSS. By writing some sample flowcharts and microcode, small timing blocks were identified and used in the model to determine an estimation of the execution rate of a few functions. The simulation also allows examination of processor interaction and determination of the approximate size of buffers needed.

Results and Conclusions

Execution speed is a crucial factor to the feasibility of CADM since the present version took nearly a half second to perform a limited demonstration and would not be fast enough when expanded. Comparison of speeds measured for MACLISP and the estimates for MLS are given for some basic functions in Figure 8.

Comparison Times (in micro-seconds)

Function	MACLISP	MLS	Ratio
CAR	200	1.8	111
CDR	200	2.6	77
CONS	380	7.0	54
COND	270	4.5	60
EQ	400	3.0	133
LIST	840	17.	49

Figure 8

Speeds as shown for the MLS are those determined through simulation, and based on admittedly small but representative blocks of microcode written for some functions. The numbers cannot be taken as exact with such a crude model, but their relative size is significant with as much as a factor of one hundred improvement over current implementations. The interpretive LISP timings were found by writing a LISP program which looped through ten thousand evaluations of the particular function, and then subtracted off the overhead of the loop itself.

The argument is often made that LISP can be compiled to achieve greater speeds, and an attempt was made to measure the difference. It was found that the MACLISP compiler only compiles certain sections of the code into LAP code, generating a list of subroutine calls for most functions. It is expected that the compiler would speed up some functions like DEFUN and DO, but otherwise just calls the appropriate interpretive subroutine. Thus there was no difference in speed for functions like CAR and CDR, and questionable differences for the others. Interpretive LISP

obviously does much more involved type and error checking, with the MLS doing almost none. On the other hand, for a real time airborne system, the development would be done on a ground based system, and the goal would be fast efficient execution of that debugged program in the airborne system. With a factor of fifty to one hundred improvement in speed and one third as much memory, the MLS overcomes LISP's apparent slowness, and could handle CADM on a real time basis.

The current CADM program contains approximately 128K characters and requires 220K of memory (36 bit words) on the DECsystem-10 to include the full list space and 18K for the interpreter. Allowing for a CADM program twice as large as the one already developed, MLS would require approximately 128K words of main memory, 16K words of symbol table, and 1K words for the stacks. Three processor systems plus this memory would cost less than thirty five thousand dollars for the hardware and could be packaged in less than a cubic foot including power supplies (See Figure 9).

Cost Breakdown for MLS

3 Processor Systems	$1000.
Support circuitry	200.
PROM (256K)	600.
RAM (2.2M)	22500.
Power supply, cards, and rack	8000.
TOTAL	$32500.

Figure 9

Estimates are based on the technologies and prices currently available, and about two-thirds of the cost is in memories which are decreasing in cost rapidly each year. Thus after the initial development costs, CADM implemented on the MLS would indeed be economically feasible for use in an airborne environment.

References

[1] Chien, R.T., et al, "Computer-Aided Decision-Making for Flight Operations, Tecnical Report No. 2", Coordinated Science Laboratory Report T-18, University of Illinois, June 1975.

[2] Clark, Douglas W., "List Structure: Measurements, Algorithms, and Encodings", Thesis, Department of Computer Science, Carnegie-Mellon University, August 1976.

[3] Greenblatt, Richard, "The LISP Machine", Working Paper No. 79, MIT Artificial Intelligence Laboratory, November 1974.

[4] Baker,Jr., Henry G., "List Processing in Real Time on a Serial Computer", Working Paper No. 139, MIT Artificial Intelligence Laboratory, February 1977.