UNIVERSITY OF ADELAIDE
23.2.83
LIBRARY

MTL : A VAX/VMS Compiler for a Multi-Tasking and Message Passing Language

by

Bevin Reginald Brett, B.Sc. (Hons.)

Department of Computing Science
The University of Adelaide

A thesis submitted for the degree of Master of Science

4th August 1982

ACKNOWLEGEMENTS

My thanks and gratitude go to Dr.C.J.Barter (my supervisor) for the assistance he has given me during the development of this thesis. Also to the many members, both students and staff, of the Computing Science Department for the many interesting conversations we have had discussing various aspects of computing science.

DECLARATION

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university and, to the best of my knowlege and belief, contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

B.R.Brett

4th August,1982

TABLE OF CONTENTS

Page

Page

Page

SUMMARY

The last decade has seen increasing awareness of the usefulness of concurrency in programming languages. Various proposals, including Message Passing and Remote Call, have been put forward as mechanisms for handling the concurrency in a structured way. Yet no implementations of any of the proposals has found wide acceptance. Many of the proposals have either remained un-implemented, implemented on a specialized operating system, or interpreted.

This thesis describes the development of MTL, a compiler for a language with both multi-tasking and message passing that generates executable code for a Digital Equipment Corporation VAX-11/VMS system. The code generated by the compiler is capable of interfacing with all other VMS languages, and is almost as efficient as any procedure-based language implementation on VAX/VMS.

Various suggestions are made about how the few remaining inefficiencies could be removed, and about what features hardware and software require to support such languages.

An editor was designed and implemented using MTL that took advantage of the of concurrency to provide a more versatile user interface. The implementation of the editor was to be a test of the adequacy and efficiency of MTL but it was found that it also lead to new view of the user's environment and the relationship of the role of the editor in relation to the operating system and other utilities.

The application of the concepts of concurrency and message passing to editing unified the whole user's environment in a system that is extremely productive and free of limitations, thus showing the fundamental power of these concepts.

UNIVERSITY OF ADELAIDE LIBRARY

Chapter 1
INTRODUCTION AND RATIONALE

Over the last decade the search for comprehensive and powerful ways of structuring programs has lead to considerable research into concurrency and message passing. It has been shown that these mechanisms can lead to natural and simple solutions that reflect the structure of many problems. A large number of different proposals for languages that incorporate concurrency and message passing have been put forward; however there has been a tendency for these proposals to be given simple interpreter implementations or not to be implemented at all.

Research has reached a point where serious implementation efforts are necessary to discover and solve the remaining difficulties that are hindering the widespread use of these languages. Such an implementation must address the problems of efficient task creation, context switching and message passing. Feedback from these implementations can be used to fine-tune the language proposals, as well as suggest further areas of research. The recent development of Ada is an example of this working in practice.

Some of the implementation efforts should be based on conventional operating systems and architectures to see if these impose any inherent difficulties. This thesis is primarily a description of one such implementation, a language called MTL (Messages and Tasking Language). The implementation was done on a VAX-11/780 running the manufacturer supplied VAX/VMS operating system.

The architecture of the VAX-11 series is modern but conventional, with a strong emphasis on support for such languages as Pascal, Fortran, and Cobol. Similarly VMS is an efficient conventional operating system that supports multi-tasking but was not designed with message passing languages as a strong consideration. Because of the obvious impact of of the VAX-11/780 and VMS on this project the reader is given introduction to both in the third chapter.

It was decided that a significant piece of software would be designed and written in MTL to check on both its sufficiency and efficiency. Several projects were considered, including a compiler (using concurrent tasks for the various lexical, syntatic, semantic, and listing phases) or a series of mathematical exercises (such as a Sieve of Eratosthenes, sorting, etc) that would use concurrency in unusual ways.

The eventual choice was a screen editor. This project stressed the I/O aspects of multi-tasking as well as allowing the author to express some ideas about editing that have come out the University of Adelaide's development of the Ludwig screen editor[BA80]. It was also hoped that this unusual design and implementation of an editor would suggest some interesting and novel views of the how an editor could be used and the sort of mechanisms it could provide for the user. This was indeed the case, and Chapter Six describes the resulting design with an emphasis on the use of concurrency at a user interface level in a screen editor.

Interactive computing, with the programmer or other user on-line to the computer, has also become increasingly well recognised as the most friendly and most productive approach to using a computer. However this has happened so fast that the software, unlike the hardware, has been unable to keep pace. At any given time when there is a reasonable number (say ten or more) users logged in to one of the university's VAX-11/780 computers, more than 75% of them will be editing text. The main editor used, Ludwig, is a powerful screen editor that provides many facilities for manipulating and correcting text. However these features are not available for correcting interactive inputs to the many other facets of the VMS system. More over it is very difficult, if not impossible, to directly interface any editor with any other utility (such as a compiler) on the system. All this leads one to suspect that in the future the problem of connecting various components of the system together is going to look less hierachical and more network-like. It also suggests that the primary user focus in this network is going to be the VDU and the editor (not necessarily just text but also pictures) that controls it. Because of this role for message passing languages (the network-like appearance will demand this) and the importance of editing, a natural union results - that of a screen editor written in a language that supports message passing.

There are available, under VAX/VMS, the manufacturer's VAX-11 Pascal[DEC79] compiler and implementations of two languages that support concurrency and or message passing, Maciunas' Mercury interpreter[MA81] and Roper's Cospol interpreter[RO81]. By using these for comparing the speed of execution of various algorithms a rough estimate of the efficiency of MTL could be obtained. In particular I was interested in the relative efficiency of message passing and procedure calling. Given the rapid decrease in the price of memory it is apparent that for this construct, which would use little memory no matter how badly implemented, efficiency corresponds to speed of execution. From the outset it seemed plausible that message passing could, in some circumstances, be at least as fast as procedure calling. This encouraged me to place as much emphasis as possible on fast message passing as it seemed an important phenomona to observe and document. A surprising result of this thesis is that this effect can indeed be observed on an architecture such as that of the VAX-11 which does not have special machine instructions for message passing but does have them for procedure calling.

Chapter 2
HISTORICAL OVERVIEW OF MESSAGE PASSING AND MULTI-TASKING SYSTEMS

2.1 Introduction

There has been a growing awareness of the inadequacies of purely sequential, procedure based languages for implementing natural solutions to simple and complex problems. The classical Odd-Word-Reversal Problem, posed by Dijkstra[DI72], is an example of a problem that has, as its solution, a multi-phase algorithm whose implementation is not obvious in an Algol-like language.

The problems of running several sequential algorithms simultaneously in one computer-system were initially addressed at the operating system level. Because operating systems had to be both reliable and efficient, a theory of shared data structures was developed. Programmers became aware that they could gain more flexibility by using a system on which several communicating jobs could be run.

This discovery is leading to the introduction of more features to support this mode of programming into both programming languages and operating systems.

Both because of the cost of computers (a factor rapidly diminishing in importance but historically the most significant) and because it seems a natural way of solving some problems, it is desirable to have several algorithms executing "concurrently" on a computer system. This thesis uses the words "TASK" and "PROCESS" to refer to a single one of these algorithms. These tasks may belong to users who don't even want to know of the other's existence, or the tasks may be co-operating in solving a problem.

It was soon realized that the complexity of computer systems could only be managed through a theoretical basis for secure multi-tasking. Ideally this theory should provide a calculus that could be applied to a system to prove that only legitimate actions could be performed by tasks. In practice this ideal has only been partly reached because of the complexity of the systems. In spite of this difficulty the research has provided tools and insights that simplify the design and implementation of secure systems.

2.1.1 <u>Shared Variables</u>

The implementation of multi-tasking may require the use of shared variables to store the state of some of the resources being managed by the system. The system data base that describes the state of the various tasks is a shared variable, as is the data base describing the state of each device. The devices themselves, such as disks and tapes, provide a storage area that may be shared between several tasks and hence can be viewed as shared variables.

Many of the advances in system design and implementation have resulted from the need to maintain the consistency and security of these shared variables because their accidental or malicious modification can have a severe impact upon the usability of the system. These advances are reflected in all parts of the system, from having indivisible instructions in the instruction set through to facilities such as semaphores in high level languages.

Dijkstra's semaphores[DI68] provide a simple yet effective mechanism for the various users of a resource to voluntarily synchronise their access to it, but do not enforce their usage. This enforcement could be gained in a high level language by only allowing access to the shared variables inside blocks of code that are surrounded by the appropriate P and V operations, but this does not prevent their uncontrolled modification by either run-away programs or malicious or ignorant users writing in more primitive languages. Furthermore "solutions to apparently simple communications problems are sometimes disproportionately complex"[LI77] when using semaphores to synchronise the tasks.

The addition of processor modes, which impose restrictions on instructions (eg. HALT is only allowed in a privileged mode) and the use of hardware memory protection solved this problem at a fundamental level. The only way to change from a lesser to a more privileged mode is via a system call or machine instruction that goes through a very small, hence manageable, interface.

A theoretical underpinning for this approach was provided by the Monitor concept. The system provides Monitors that manipulate resources, and which have to be called at a privileged processor mode. These Monitors validate the request and perform it on behalf of the task, sometimes returning information to it.

An alternative approach to calling the Monitors, which has an implied delay until the Monitor returns to the task, is to have one separate task controlling the resource. The separate task has a queue of requests from tasks that wish to access the resource which it controls. These requests are processed sequentially as they are removed from the queue. This approach works well for resources where the request may take some time to satisfy, such as an I/O device. A queue of requests can be implemented as a data structure with the Monitor providing services for en-queueing and de-queueing the messages.

2.1.2 Message Passing

Such an approach is called Message Passing. Various forms of message passing have been proposed and used since the concept was first introduced. Choi has classified the various alternatives by splitting the communication of the message from the method of synchronization. He classifies the possible behaviours of a user task as (A) Request a service, (B) Wait until the server accepts the request, and (C) Wait for a reply from the server. The server has two actions, (D) Wait for a request, and (E) Reply to a request. Various restrictions are placed on the ordering of these events to enforce their intended semantics.

Message passing also offers hope of a theory that supports distributed computing, with the N tasks spread, perhaps dynamically, over M processors. In this environment, with N and M reasonably large, the only communication method currently viable is message passing over some transmission media.

2.1.3 Correctness

The need for software that is transportable between operating systems is a problem that may be solved by consistent implementations of programming languages. This is turn requires a precise definition of both the syntax and semantics of the language so that it can be decided whether the behaviour of translators (compilers, interpreters, and even source-text to source-text translators) is correct or erroneous.

It may be possible to use this precise definition of the language's semantics to prove that a program is "correct" in a mathematical sense. Unfortunately for many programs the current proof techniques lead to proofs many times larger than the text of the program, and for other programs (eg.

interactive text editors) the precise definition of what the program should do is often as large as the text of the program, and therefore subject to "bugs" in the same way as the software.

In practice a proof of some aspects of a program's behaviour is often adequate, or a complete proof for small, critical, portions of the program. For instance, such proofs can show that a data base is correctly interlocked without consideration of its contents. The proof techniques are also useful in debugging programs because they provide tools for deducing which part of the algorithm is incorrect, given the behaviour it is exhibiting, and also by providing reassurance about some components of the system.

A method for showing the correctness of tasks using semaphores to implement a critical section that modifies a shared variable has been provided by Habermann[HA72]. He applies his technique to a bounded buffer, a problem we shall be returning to in section 5.2.5, where a bounded buffer is used as the implementation of message passing in MTL.

Dijkstra[DI75] addressed the problem of non-determinism by introducing a language with non-deterministic constructs, and then providing a calculus for both formally deriving such programs, and for showing them correct. There was no concurrency in the language. However it did show that the exact order of execution of the program was not only unnecessary for a proof, but that often a simpler proof could be constructed that ignored the precise history.

Hoare[HO78] combined message passing and non-deterministic commands in the language CSP, using an informal definition of the language's semantics. This spurred on efforts to provide a suitable formal model of such mechanisms for proof techniques. One such attempt is [FR79].

2.2 Implementations

It is considerably easier to develop interpreters or complete operating systems for message passing and multi-tasking languages than to develop a compiler and runtime system. Both interpreters and complete systems allow the implementor considerably more flexibility in where and how to solve the concurrency and efficiency problems.

2.2.1 The B5700/B6700 Computers

In the early 1960's the Burroughs Corporation [OR73,BU71] started producing a series of machines that supported multi-tasking with shared memory between the tasks, (the B5700 and B6700 range). Both the architecture and the operating system were designed to support Algol-like structured languages with the addition of multi-tasking.

Each task has a stack of its own, within which the activation records of each procedure or function are stored. Also it can have references to the stack belonging to its parent task, or any other ancestor. The stack in turn can contain descriptors to other memory segments. These descriptors are protected by a combination of the hardware and the system software, and specify either the location in physical memory or the location in the backing store of the segment.

Tasks are created by specifying a procedure, possibly with parameters, and a Task_Variable in a Process_Statement. By assigning values to fields in the Task_Variable before using it in a Process_Statement various attributes of the task (eg priority, stacksize) can be specified.

A Continue_Statement, with either an implicit or an explicit naming of the destination task, causes the current task to be suspended and the other to continue from where it last suspended at its Continue_Statement. This allows co-routining relationships to be set up easily.

The task dies when either it exits from its main level procedure or when its parent exits from any block which the task is capable of referencing. By this means the deallocation by the parent of variables accessed by the task can not damage the environment of the task, because the task itself dies. It is possible to set up independent tasks, but in this case parameters must be passed by value.

Control and communication between tasks is based on shared variables, on event flags, and on software interrupts. Event flags have two attributes, Happened and Available. A Wait_Statement causes the current task to wait until the specified Event Happens. A Cause_Statement makes an event Happen. A Procure_Statement waits until an event flag is Available and then procures it, later to release it with a Liberate_Statement. Interrupts are sections of code to be executed when the event is Caused.

This architecture and operating system was one of the earliest supporting multi-tasking inside a single address space. It has hardware support for the inter-task memory protection and for the cactus-stacks required for the nesting of tasks. Some decisions (eg. the decision to kill sub-tasks when their parent exited a critical block unlike Ada which suspends the parent until the sub-tasks terminate) have been found to be non-optimal but the influence of design of these machines on MTL can be clearly seen. This may partly be due to the fact that my initial introduction to computing was on a B6700.

2.2.2 RC 4000

Between 1967 and 1969 Brinch Hansen [PBH73] and others designed and implemented a multiprogramming system for the RC 4000 machine built by Regnecentralen in Denmark. The Monitor was designed as a bridge between the hardware and a virtual multiprogramming machine, not as an operating system. Indeed one of the aims of the RC 4000 system was to allow operating systems to be changed on the fly, even running several operating systems simultaneously.

The RC 4000 system very carefully defines the concept of a process and the mechanism for interprocess communication in a way that was not dependent on the particular operating system. The Processes are hierarchically organised and are able to create sub-processes dynamically, providing the resources required by the sub-process out of their own available resources.

The Monitor maintains a pool of small (eight machine words) message buffers and maintains a single queue of messages for each process. Message operations are Send-Message, Wait-Message, Send-Answer, and Wait-Answer. Each process has a quota of message buffers, thus preventing it using up the whole message buffer pool. To ensure that none of the answering process's quota is required for the transaction the same buffer is used for answer as was used for the message.

I/O devices are treated as processes, performing I/O operations as requested by messages.

Brinch Hansen lists several advantages and disadvantages of the RC 4000 system. Of these, the ones that seem directly related to message passing are:

Advantages...

() It implements a nucleus which was successfully extended to a spooling environment and several real-time systems,
() it was small and simple to implement,
() it was reliable and quickly made almost error-free,
() it was adequately documented.

It is probable that the use of message passing as the inter-process communication mechanism greatly contributed to these advantages being attained.

Disadvantages...

() the system does not make it easy to debug time-dependent errors,
() the high cost of verifying monitor calls and doing other protection checks at run time,
() the only way of achieving mutual exclusion between multiple processes to a data base is to have a single process managing the data, and sending it requests; at a maximum throughput of 500 messages per second this was too slow,
() the use of cooperating processes is too expensive on resources, hence designers of such features as multi-terminal support have tended to use one complex process rather than a group of cooperating processes,
() an artificial restriction - the size of the message buffer pool,
() an artificial data restriction - a fixed length message size,
() an inefficient implementation - physical copying of messages.

Apart from the first, none of these problems are inherently due to message passing, reflecting instead other aspects of RC 4000's design.

These problems are all significant, and should be kept in mind during the design of any message passing or multi-tasking system. The problem of debugging concurrent programs is one that is going to require considerable further research and experience. MTL does not do run-time checks of monitor calls, or other protection checks because the compiler does the usual Pascal checks on parameter lists, and since MTL is a single user environment it does not matter if the user tries to subvert the mechanism -

he can only damage himself. MTL does have shared variables, and it is possible to implement a conventional monitor protecting those variables. Large numbers of cooperating processes can be maintained at a very small cost in resources, the overhead per-process being around 100 bytes of memory. Neither of the artificial restrictions apply because MTL uses an indefinitely extensible heap for its messages and these can be any Pascal type. Messages are passed by pointer switching, taking advantage of the common address space being used by the tasks.

2.2.3 Concurrent Pascal

Following his experience with RC 4000, Brinch Hansen[PBH75] designed the Concurrent Pascal programming language which extended Pascal to include the concepts of Monitors, Processes, and Classes (a module that encapsulates variables that belong to one process or monitor).

Processes have private data, a sequential program, and a set of access rights. The access rights list the shared data that the process can operate on. Processes do not operate directly on the shared data, but use the appropriate Monitor calls. Monitors provide exclusive access to the resource they control, except at points where the process must be suspended. If the process is suspended, the same Monitor must resume it.

Concurrent Pascal has been used to develop the SOLO[PBH76], TRIO[PBH80], and MULTI[KR82] operating systems for the PDP-11 minicomputer, and has been implemented on such machines as the UNIVAC 1106[DU82]. The compiler generates an abstract instruction set which may be implemented as threaded code (Solo and Trio) or compiled [DU82] (although due to memory requirements this was later changed to use a modified form of threaded code).

Kruijer[KR82] found the lack of dynamic process creation a drawback in Concurrent Pascal when implementing MULTI, but draws an a priori conclusion that system software written in Concurrent Pascal will have a high reliability, adaptability, and portability.

These experiences with Concurrent Pascal show that it is possible for such high level languages to be used to implement reliable operating systems, and that such implementations are considerably easier than the more traditional approach of using assembly languages. However the extensive use of a kernel and threaded code on all these systems fails to release the full computing potential of the host machine. Concurrent Pascal does not support message passing or the dynamic creation of processes and thus these implementations do not address the main implementation problems of languages such as MTL.

2.2.4 Thoth

Thoth[CH79] is a portable real-time operating system that uses messages as the method of interprocess communication. The processes are grouped together into teams which share a common address space and set of resources. Processes in the same team can therefore also share memory between themselves, but not with other teams. Message passing provides the only method of inter-team communication.

The primitives implemented by the Thoth kernel are available to the processes as system calls. Amongst others there are

() .Create(function,stack_size),
() .Send(msg,id),
() .Receive(msg) and .Receive(msg,id),
() .Reply(msg,id),
() .Forward(msg,id).

Processes are created with a specified stack size, organized into a dynamic tree, based on a creator/creation ordering. Each process has a priority associated with it, and pre-emptive scheduling based on this priority and on machine interrupts is provided. However the priority of a process in a dynamically created team is only with respect to other members of the team. Thoth blocks .Sending processes until the .Receiving process .Reply's, the .Reply using the same buffer as the .Send did (as in RC 4000).

The grouping of tasks in a single address space into teams is similar to the B5700/B6700 design. However Thoth also supports the use of messages to communicate between different teams which greatly increases the flexibility of the system by allowing controlled communication between objects that can not access each other's address space. This means the same mechanism can be used between team members, who trust each other implicitly, and with other teams which are not so trustworthy.

2.2.5 VAX/VMS

VAX/VMS[DEC80] processes have separate address spaces, but there are system services that enable processes to share sections of memory. Processes may also share event-flags with others, and these event flags may then be used to synchronise them. Event flag are not semaphores because all processes waiting for an event flag are released when the event flag becomes set. Devices called Mailboxes allow processes to communicate with each other, treating the device as a normal file.

The structure of VAX/VMS with respect to multi-tasking and message passing is covered in more detail in Chapter 3.

2.2.6 Simula

One of the best known, and most commonly available, languages that supports a non-procedural flow of control is Simula[BI73]. Marlin[MA79-2] gives a formal definition of the flow of control in Simula. For our purposes it is sufficient to mention that Simula provides dynamic creation of co-routining classes, that the flow of control between the various class instances is explicitly defined by the language. The class instances each have algorithms and private data structures, as well as access to the data structures of other instances.

In Simula the flow of control is fully specified by the program and is thus completely deterministic. This differs from such languages as Concurrent Pascal and Ada where the flow of control is not specified. It is a rare example of a widely used routine-based system that needs more than a simple stack for the storage of the activation records, and as such it indicates an area (simulation) where a need for concurrency has long been recognised, with Simula satisfying most of the requirements.

If concurrency can be implemented in languages as efficiently as co-routining etc. has been in Simula (and there is no obvious reason why this should not be possible) it can be expected that many of the applications for Simula will also be valid candidates as applications requiring concurrency. Furthermore it may prove to be easier to dynamically gather data from such models because the flow of control is not so important.

2.2.7 COSPOL

COSPOL[RO81] is a compiler generating code for a virtual machine that is implemented as an interpreter. The language supports message passing, and all I/O is regarded as message passing to special tasks called Reader and Writer. A parallel-command executes a list of tasks concurrently, terminating only when all of the tasks terminates. Because there is no procedure calling the number of tasks possibly active at any given time can be determined by the compiler.

The interpreter uses a round-robin scheduling algorithm to control the execution of tasks, and terminates when no task is capable of execution. It inherits a problem from the Pascal runtime system. If a task waits for a message from Reader (corresponding to a Pascal READ statement) the interpreter would read from the file, thus the other tasks are not able to continue execution. The problem was partly alleviated by delaying the read from the file until no other tasks were capable of execution.

COSPOL has no procedure calling mechanism, and rescheduling is only done when the stack used to evaluate expressions is empty, hence there is no need to maintain multiple stacks in the interpreter. COSPOL only reschedules as a consequence of message passing.

Messages are stored in an interpreter implemented heap because the Pascal heap proved too awkward to use. The work of Marlin[MA79-1] on implementing heaps in Pascal interpreters was used in the design. Message queues were maintained for each task and each type of message on a strictly FIFO system. Message variables are pointers to heap objects. Transmission is done by copying the heap object and enqueuing a pointer to the copy because COSPOL semantics allow access to a variable which has been sent to another task, but a compiler optimization detects the case where the original object can not be accessed and in such instances avoids the copy. Message reception is done by copying this pointer into the message variable. There is no limit on the size of the queues other than that imposed by the implementation.

COSPOL illustrates both the advantages and disadvantages of using an interpreter for implementing a language. In particular there is the ease of implementation and modification of an interpreter versus the factor of about 20 or 30 between the cpu time used to interpret a program using COSPOL and the time required to execute it via compiled code.

2.2.8 Ada

In 1978 Per Brinch Hansen[PBH78] introduced the Distributed Processes concept as a suggested solution to the problem of implementing real-time algorithms on distributed processors. In DP he invisaged one process per processor, with the inter-process communication being performed by a procedure-call mechanism.

The suggestion has been embodied in the Ada[MI80] rendezvous because Ada has been designed to cope with a closely related problem, that of embedded systems. The remote-call appears to the caller as a simple procedure call, with both input and output parameters. The called task accepts the call by executing an Accept_Statement for the appropriate Entry_Declaration. The Accept_Statement includes a sequence_of_statements which are executed before the caller is released from the rendezvous, and during these statements the input parameters are obtained and the output parameters returned.

The rendezvous concept is similar to synchronous message passing, differing only by the addition of output parameters. This addition makes remote-call an excellent mechanism for situations in which an immediate reply is expected because it provides an extremely efficient mechanism for returning results. The calling task waits until the acceptor has released it from the rendezvous, thus there is no buffering between the calling task and the acceptor.

On the other hand message passing is preferable for pipe-line style communication, where each component is massaging the information and then passing it on. It also enables the caller to receive a delayed result some time after the request, thus allowing it to perform useful work while awaiting the reply from a request that will take some time to complete.

Message passing packages can be easily written, including bounded buffers or synchronous mechanisms, but experience with MTL indicates that the implementation of message passing packages will cause markedly slower run-time performance than the direct inclusion of message passing into the language (see appendix B). Ada would allow implementations to do this, as the Send and Receive implementations of MTL can be regarded as efficient replacements of a Buffer task.

Since message passing and remote-call are best suited to different roles they should be regarded as complementary mechanisms rather than competitive proposals.

2.3 Summary

Proof techniques for many programming language concepts, such as those of Habermann[HA72] and Dijkstra[DI75], have always lagged behind the implementation and usage of those concepts. They are still incapable of providing correctness proofs for most large programs. Nevertheless such efforts have successfully indicated which programming techniques should be used and which should be avoided in the construction of reliable software.

Concurrency has been shown to be a useful programming construct but it is only just appearing in the major programming languages (except for Burroughs Extended Algol). Its availability has been increasing but for the most part implementations are dependent on either interpreters or specialised operating systems.

The problems caused by the use of shared variables have proven manageable. A variety of different levels of control can be enforced upon them, from being completely unprotected, to being guarded by either a monitor or a task, the level of protection being chosen to match the requirements of the problem being solved.

Message passing has been used as as an effective and flexible mechanism that enables the programmer to exploit the benefits of concurrency. It is not necessary to establish an arbitrary master-slave relationship between two activities that are inherently parallel (the problem with procedures), nor is it necessary to explicitly transfer control between the various activities (the problem with co-routines), but rather each activity is executed when it is required.

The MTL compiler and runtime system implements concurrency and message passing on a machine with a non-specialized operating system and conventional architecture.

Chapter 3
The VAX-11/VMS Environment

The VAX-11 series machines are a recent product of the Digital Equipment Corporation, and VMS is the operating system they have developed for these. This chapter is intended to give the reader an understanding of both this machine's architecture and the operating system sufficient for the rest of this thesis. It is not intended to be a complete description of either the VAX-11 architecture[DEC79-A], or the VMS operating system[DEC80-SS] but only those parts that affect the implementation of multi-tasking or message-passing languages are covered in any detail.

3.1 VAX-11 Architecture

3.1.1 Virtual Addresses

VAX is an acronym for Virtual Address eXtension, which is a reference to the relationship between the VAX-11 series of computers, and DEC's PDP-11 range. The virtual memory is addressed via a 32-bit value. This value is translated to give either a physical memory address or a page fault by the memory management hardware. However there are some restrictions and characteristics that are important.

(1) The virtual memory is granularised into 512 byte pages. A single page is the smallest item that the memory protection hardware recognises, and must be aligned on a 512 byte boundary. The lowest 9 bits in the 32 bit address specify the byte offset of the byte(s) being accessed within the page. This mechanism makes it impossible to protect an arbitrary area of memory. The machine has four processor modes (called USER, SUPERVISOR, EXECUTIVE, and KERNEL respectively, and in increasing order of power) and pages are protected from either read or write access at each level. Only a restricted set of protections are available, as read/write access at a level implies the same access at the higher levels. For example, a page that is specified as USER:READ, SUPERVISOR:WRITE can be read by an instruction operating in any mode, and written by instructions running in Supervisor, Executive, or Kernel mode. Special PROBE instructions are available to determine the readability and writeability of areas of memory without actually causing a page fault or access violation (trying to access an area of memory that is protected against the access being attempted in the current access mode).

(2) The top 2 bits of the 32 bit address specify the region of the page. Specifically if these top bits are 10 or 11 (binary) then the address is said to be "in system space", whereas if they are 00 or 01 (binary) the

address is said to be "in process space". Only the first (S0) area of system space is used in VMS. Process space is broken into two regions, called "P0 space" and "P1 space" respectively. Translations of addresses in P0 and P1 space are done via a page table which is also located in virtual memory in system space. When VMS is discussed it will be explained how these four regions are used by the VMS operating system.

3.1.2 Operands, especially Queues

The VAX-11 architecture supports a wide variety of operands and operand sizes. Typically operands are integers or floating-point numbers and are stored in 1, 2, 4, or 8 byte long areas of memory, or general purpose registers. These sizes are referred to as BYTE, WORD, LONGWORD or QUADWORD items respectively. A 32 bit address would therefore require a LONGWORD to store it in. The architecture is slightly biased towards LONGWORD quantities.

Also supported are operations on strings of characters and on packed decimal strings. Of importance to the design of both VMS and MTL, is the architectural support for a doubly-linked queue as a basic data type with such instructions as INSQUE (insert an entry into a queue), and REMQUE (remove an entry from a queue). The operations can be performed in an non-interruptable manner at either end of a doubly-linked queue of items, and are so used in the implementation of the MTL heap and message passing mechanism (chapter 5).

A absolute-queue entry starts with two longword addresses, the forward and the backward link. The contents of the rest of the entry are irrelevant to this discussion. Each absolute-queue (there are also self-relative-queues) starts with a queue header, which is a pair of longwords, the first pointing to the first item in the queue, and the second to the last item. If the queue is empty both longwords point to the queue header.

The INSQUE instruction format is

```
INSQUE  entry,predecessor
```

which inserts the entry into the queue after the specified predecessor.

The corresponding REMQUE format is

```
        REMQUE  entry,pointer
```

which removes the entry from the queue, and places a reference to the removed entry in the pointer. Both these instructions set processor condition flags to indicate the state of the queue before/after the operation.

3.1.3 Instructions, Registers, and Addressing Modes

Instructions on a VAX-11 are orthogonal to the addressing mechanism used for the operands. For example the ADDL2 instruction can add a value to a register, to the top of stack, or to memory. The instructions take anything between 0 and 7 operands, where some of the operands may be large tables or other large areas of memory.

The VAX-11 architecture has 16 32-bit registers, named R0 through R11, AP, FP, SP, and PC respectively. R0 and R1 have, by convention, the role of carrying the result of a function call back to the caller unless the type of the result is too large in which case the caller is expected to provide an OUTPUT parameter to accept the result of the function. R1 also has the job of carrying the static link into a called routine. R0 through R5 are used by the string manipulation instructions. R6 through R11 have no special use although some of the VMS compilers generate code that point R11 into the static storage area and then use relative offsets from it to access the various static variables.

The SP is the stack pointer. The stack starts somewhere in memory and grows towards low memory. There are actually 5 separate SP registers, one for each processor mode (USER, SUPERVISOR, etc) and one for the INTERRUPT stack. Only the USER stack pointer is of concern to this thesis.

AP, FP, are a necessary component of the VAX-11 routine calling mechanism, and are discussed in that light in the next section. PC is the program counter of a typical Von-Neumann machine.

Operands are either (1) in a register or (2) accessed via an address that is formed by operations based on one or two registers and some constants that form part of the instruction stream. An example of the first case would be ADDL2 R0,R0 which adds the 32 bit quantity in R0 to itself and places the result back in R0 again. There are several possible ways of forming the address mentioned in the second, an example is relative addressing such as MOVL2 4(R0),R1 which moves the longword found at address 4+contents_of_R0 into register R1. Indirect referencing is also possible. Exact details of these are not critical to the understanding of this thesis.

3.1.4 Procedure Calling, especially the CALLS and CALLG instructions

One objective in the design of the architecture, the VMS operating system and the various language implementations was a standard calling mechanism that would enable any language to generate calls to routines written in any other language. While the wide range of entities that occur in the various modern computer languages make this difficult, the goal was fairly well attained and is not one to be sacrificed lightly. In fact any language compiler that does not generate code that follows this calling standard can not be regarded as a true VMS compiler.

The CALLS and CALLG instructions are the architectural basis of this standard. These instructions build a structure known as a call frame on the stack. This call frame contains

(1) the saved values of AP, FP, and PC.

(2) sufficient information to restore SP.

(3) any registers that the called routine wishes saved, this is done by having the first word of the called routine act as a mask where the bits set correspond to a register to be saved.

(4) various other small pieces of house-keeping information.

The FP (Frame Pointer) and SP (Stack Pointer) are pointed at the low end of the new call frame which is built on the stack corresponding to the current machine mode (User, Supervisor etc.). The AP (Argument Pointer) is pointed to the parameter list being passed to the routine, and then the routine is executed. The RET instruction restores all the saved registers, removes the frame from the stack, and returns to the caller.

The difference between the CALLS (call-stack) and CALLG (call-general) instructions is that the CALLS instruction assumes the parameter list is on the stack, whereas the CALLG instruction has as one of its operands the address of the parameter list which may then be located anywhere in memory. If the instruction was a CALLS, the RET instruction also removes the parameter list from the stack. The CALLG instruction is used by MTL for procedure calling as this allows MTL to place the parameter list in the activation record of the calling procedure which is stored in the heap.

The saved FP's form the dynamic chain for the routine, linking back through all the various stack frames. This chain is used during condition handling, a subject outside the scope of this thesis. During the execution of a routine the FP is not changed (except when another routine is called) but the SP oscillates up and down below FP as items are pushed onto or popped off the stack.

3.2 <u>The VMS Operating System</u>

3.2.1 <u>Address space layout</u>

VAX/VMS resides mainly in virtual memory at and above 80000000 (hex) and this area (system space) is shared between all the processes of the system. The scheduling between these processes is caused by pre-emptive and voluntary releases of the CPU. The area of memory below 80000000 (hex) is designated "process space" and is mainly for the exclusive use of this process. Processes can share areas of memory, and the shared areas may have different virtual addresses within the different processes. Some system information that is pertinent only to the process is maintained in P1 space and is protected against illegal access by means of the access checks described in section 3.1.1.

3.2.1.1 <u>P1 Space and the USER Stack</u>

Each process has (under VMS version 2) one USER-mode stack which is indicated only by the SP. This stack is normally found at the low end of P1 space, and grows towards P0 space. The area of P1 that is accessible grows downward from 7FFFFFFF (hex) and is limited in size by a quota imposed by the system and P0 space grows upward from 00000000 (hex) and is restricted by the same quota. This quota sets an upper limit on the total amount of space that is available in P0 and P1. This limit effectively blocks any attempt to have more than one USER-mode stack because there is no mechanism for preventing the stacks from colliding other than widely separating them and there is insufficient memory available for such wide separations. Various software interrupts use indefinite amounts of storage on the stack beyond the area in use by the main flow of program execution, and thus the danger of stacks colliding, either because of these events or because of deep recursion, is very real.

This forms a limitation if one attempts to introduce concurrency for such languages as Ada, Simula, and MTL inside a single VMS process. If this can be circumvented then P1 space, with its approximately 1,000 Mbytes of addressable memory should prove adequate, in the immediate future, for the allocation of stacks.

3.2.1.2 P0 Space, the Concept of an Image, and the Heap

The instructions and static data for a program are usually loaded into P0 space. The first page (addresses 00000000 (hex) to 000001FF (hex)) is usually left as a no-access page so that access violations catch most attempts to use (in Pascal terminology) NIL pointers (which point to location 0). The combined instruction and data area is loaded into virtual memory by a system routine called the IMAGE ACTIVATOR from one or more disk files. The term for this admittedly loose concept is an IMAGE. In practice often two and (rarely) even more images are loaded into virtual memory simultaneously. The main use of this practice is to load the VAX-11 Symbolic Debugger[DEC80-SD] into memory along with the program to be debugged. However the Debugger is loaded into a reserved area of P1 space so that is does not affect the behaviour of the user's program in P0 space.

This area can be expanded upwards to provide a large Heap area under the control of a multi-layered heap management system. The most basic rule is 'if you want more storage - extend P0 space', but there is a resource manager called LIBGET_VM/LIBFREE_VM for allocating this area in a more heap-like manner. Built on top of these are some routines for allocating/deallocating small quantities of dynamic storage fairly efficiently.

While the stack is in P1 growing downward, the Heap is in P0 space and grows upward. A layered approach to Heap management is implemented with the most fundamental routine being the $EXPREG Expand Region system service. This service is used to expand P0 space when the Heap is unable to satisfy a request. The next layer is the LIB$GET_VM and LIB$FREE_VM routines for allocating and deallocating areas of virtual memory. Built on top of these are other library routines for efficiently allocating and deallocating strings in a fairly efficient manner. Section 5.2.3 discusses MTL's heap management system that does not use this last layer of library routines because the costs involved in calling them are too high.

3.2.1.3 Summary and Address Space Layout Diagram

00000000	P0 Space
	Current Image
	Heap grows toward P1 space
3FFFFFFF	
40000000	P1 Space
	User-mode stack grows toward P0
	Command Language Interpreter
	Some kernel tables
	VAX-11 Symbolic Debugger
7FFFFFFF	
80000000	S0 Space
	VMS kernel and data base
BFFFFFFF	
C0000000	S1 Space
	reserved for future use
FFFFFFFF	

VMS defines a rigid convention of memory usage, partly caused by the VAX architecture and partly because of the design aim that all languages be capable of calling routines written in the other languages.

The language implementor must be aware of the restrictions that such a convention must cause, and find methods of providing the facilities required by the language within the bounds that they set.

As VMS matures and more languages are implemented for it some of these restrictions may be removed but they are all representative of a conventional modern operating system.

3.2.2 VMS Processes

A VMS process has its own process virtual address space, a collection of current resources, a list of quotas restricting the amount of futher resources it can gain, a set of privileges required for some activities, and a variety of miscellaneous other attributes. As was mentioned in 3.2.1 all processes share the system space, and it is possible for two processes to voluntarily share part of their process address spaces.

The process may be either detached, having an existence of its own, or a sub-process which is dependent on its parent process. Such a parent may be either a detached or sub-process itself, hence the process dependency structure is a tree. When a parent dies all its sub-processes are killed also. Sub-processes share the resources of the parent, but may have more restrictive quotas than the parent.

Typically each user of the system has a process in which context the various images requested by him are run. As each User-mode image terminates the user's CLI (Command Language Interpreter) regains control and interacts with him to determine the next image to be executed. The system is also capable of running any image as a process, without the presence of the CLI.

Processes can share memory, event flags, files, and devices between themselves. The system provides a "virtual" device called a Mailbox which provides a relatively simple method of piping data between processes without the overheads of shared disk files, and without the complexity (or efficiency) of shared memory.

The system uses several detached processes and sub-processes to manage various activities. There is a process called "JOB_CONTROL" which manages the basic level of the print and batch queues, as well as monitoring the terminals for interactive users wishing to log in. It uses Print Symbiont sub-processes to manage the printing of files. The disks are controlled by detached processes called Ancillary Control Processes which maintain their format and arrange the interlocks etc. required for sharing the files stored on them.

Processes are created via a system call which includes, amongst its other parameters, the name of the image file that the process is to run. This places a severe restriction on the number of processes that can be created per unit time because the creation of processes involves the reading the image file from disk. See Appendix A for a comparison of VMS sub-process creation rate, and MTL task creation rate.

3.2.3 VMS Event Flags and Asynchronous System Traps (ASTs)

Event flags are boolean variables maintained by VMS for either a process or a collection of processes. The primitive operations on the variables are setting, clearing, and reading. It is possible to cause a process to wait for a particular event flag to be set by using the $WAITFR system call. This call does not return until the event flag is set. Other services allow the process to wait until one or all of a set of event flags become set.

An AST is software interrupt that is delivered to process after the occurrence of a specified event. The AST specifies the routine within the process that is to be called as the interrupt's servicer, as well as the processor mode (USER, SUPERVISOR, etc.) at which it is to be called. A system service ($DCLAST, Declare AST) is the simplest way of causing an almost instantaneous AST to be delivered to the current process (it is not possible for a user program to deliver ASTs to other processes, although VMS does it internally). ASTs can be locked out for a particular processor mode (USER, SUPERVISOR, etc.) by means of the $SETAST system service. They are also locked out during the execution of the AST-called routine at the same processor mode or when the current processor mode is more privileged than that of the AST. For instance if a call to $DCLAST is made from SUPERVISOR mode and the specified AST is to be delivered in USER mode the interrupt will remain pending until the processor mode drops to USER then it will be delivered. ASTs may be delivered when a process is waiting for an event flag, in which case the process continues to wait after the AST routine exits.

For example a Chess program could request an AST to be delivered after five minutes before commencing to compute the next move. When the AST is delivered the program aborts the best-move search and plays the best move found so far, thus limiting the time spent searching for a move. However by far the most frequent use of ASTs and event flags is in the control of I/O operations.

3.2.4 VMS I/O

VMS uses a multi-layered approach to providing I/O [DEC80-IO] facilities for the user. The fundamental level is to connect the process directly to the interrupts being generated by the device. Immediately above that is an interface to the VMS device driver via the $QIO (Queue I/O request) system call. The request has several parameters controlling its behaviour. The required ones are ...

() a software channel number, which has already been associated with a device by means of an $ASSIGN (Assign I/O channel) system call.

() a function such as Read, Write, Change Characteristics.

() an event flag number. This event flag is set when the I/O completes.

The optional ones are

() an AST-routine address. An AST is delivered to this routine when the I/O completes.

() an I/O Status Block address. This block is filled in with information about the operation and the device.

There are also up to 6 function specific parameters. During a read request these would specify the size and location of the memory area to be read into, the numbers of seconds to time-out the request after, and a prompt to be issued if the input device is a terminal.

Although the $QIO mechanism is very powerful it is also device dependent. For this reason VMS provides the Record Management System which translates device independent requests into $QIO's as well as imposing an internal file structure on file-structured devices such as disks. RMS also makes non-file-structured devices such as terminals appear file structured. There are two basic modes of operation, synchronous and asynchronous. In synchronous mode a call to an RMS routine does not return until the operation is complete. In asynchronous mode the call returns immediately, and RMS delivers an AST to a completion routine once the operation is finished.

The language specific I/O support routines for the various high level languages (Pascal, Fortran, etc.), use RMS to provide the services they want. For instance these routines buffer the output from Pascal WRITE statements until a WRITELN is found at which point they call RMS to output the completed line as a record. RMS in turn buffers these records into blocks and uses $QIO calls to write them to disk as required. When an image is terminated RMS flushes all its buffers, so there is no problem with partially completed outputs.

Chapter 4
THE MTL LANGUAGE

4.1 Goals

The main aims of MTL's development were

(1) as a research tool for investigating problems associated with message passing implementations, and

(2) to provide a VAX/VMS compiler for a language that has message passing and concurrency.

The syntax of MTL, as well as many language features, was adopted from Pascal, with the differences described in section 4.2. The MTL language and run-time support routines provide some very low level facilities for supporting concurrency, such as shared variables and explicit scheduling statements. These are sufficient to implement message passing between tasks and other proposals for the control of concurrency, such as monitors, as user written routines. It is recognised that such low level mechanisms may not be desirable in a non-research environment.

There are a few areas where the implementation differs from Pascal, usually because the extra code required in the compiler was not warranted. Most of these differences and restrictions could easily be removed but no more would be learnt about implementing message passing or concurrency by doing so and the differences do not have a significant impact on MTL's use as a language.

To satisfy the second goal it was important that its compilation speed and the quality of the code produced should compare favourably with other VAX/VMS compilers such as Digital's VAX-11 Pascal V1.3. This goal was attained even though the compiler does not do extensive optimisations.

Some primitive facilities have been implemented for modular development and independent compilation to help the user develop large programs.

4.2 Restrictions to Pascal

4.2.1 Files

MTL only supports external text files. These files must be declared in the program header and not re-declared in the VAR declaration. They must be RESET and REWRITEn as in standard Pascal. They have no file buffers so the only method of doing I/O is by using the READ(LN) and WRITE(LN) statements. Reading and writing of enumerated types as character strings are supported (an extension to standard Pascal) as in the VAX-11 Pascal 1.3 compiler.

4.2.2 GOTO's

Only GOTO's within the current procedure or function are allowed. The compiler does not check for GOTO's into structured statements.

4.2.3 Operators and Predeclared Routines

Most Pascal operators are supported but no automatic coersion of real to integer or vice versa is done. FLOAT, ROUND, and TRUNC are available for converting between INTEGER and REAL type. Exponentiation (i**j) is not implemented. Some of the standard Pascal predeclared routines have not been implemented because there is a large library of mathematical and other routines available as part of VAX/VMS. MARK and RELEASE are not implemented because they would significantly degrade the performance of the heap, which is critical for MTL.

The following standard or VAX-11 Pascal 1.3 routines are predeclared.

ORD, CHR, ROUND, TRUNC
NEW, DISPOSE
EOF, EOLN, PAGE, LINELIM, READ, READLN, RESET, REWRITE, WRITE, WRITELN

4.2.4 Assignment and Type Checking

Because MTL differs from many current implementations of Pascal in its type checking, the type checking is described here as a restriction. Actually it is close to the draft ISO standard[ISO] for Pascal.

MTL uses name- rather than structural-equivalence type checking. In practise the user is unlikely to notice except when trying to assign between two arrays that are structurally equivalent but declared separately. The word PACKED is ignored. Constant strings are compatible with any ARRAY [1..LENGTH] OF CHAR where LENGTH is the number of characters in the string. NIL is compatible with any POINTER type.

Eg: Illegal.

```
var a1 : array [1..10] of char;
    a2 : array [1..10] of char;
a1 := a2;
```

Legal.

```
var a1,a2 : array [1..10] of char;
a1 := a2;
```

4.2.5 Scope Rules

4.2.5.1 Scope of Identifiers

MTL implements the draft ISO standard[ISO] for scope rules except for field identifiers (see 4.2.5.2). Identifiers have a scope that extends from the start of the body in which they are declared, to the end. They may not be used before they are defined except in the construction of a pointer type. When building a type, the type identifier is not defined until the right hand side of the definition is complete.

4.2.5.2 Record Field Names

Unlike Pascal, MTL regards the names of fields in records as field selectors rather than identifiers. This means that they do not get entered into the scope of the record, hence the following is valid MTL but not valid Pascal.

```
type stack_item = record
                    case nature : simple_type of
                      bool : (boolean : boolean);
                      intg : (integer : integer);
                      reel : (real    : real);
                    end;
```

The field selectors do become identifiers inside WITH statements, as one would expect.

4.2.6 Complexity Restrictions

MTL uses a simple-minded register allocation strategy and does not cope with the problem of insufficient registers. This is because there are 10 general purpose registers (R2-R11) available, and in practice MTL does not run out of registers very often. Registers are allocated for (1) holding the addresses of the activation record of each routine in static scope, (2) holding the address of the record inside a WITH statement, (3) holding the upper-bound of a FOR loop when that bound is not a compile-time constant, and (4) holding temporary results during expression evaluation. If MTL generates the TOOCOMPLEX error message during compilation it may be necessary to reduce any one of these.

4.2.7 Bugs

It is to be expected that a compiler developed as MTL was will have bugs in it, both at compile time and in the code produced. The compiler's error recovery after syntax and semantic errors is generally reasonable but sometimes poor. There are no known code generation bugs.

Currently the compiler does not detect the following error.

```
Program Bug01;
  procedure P;
    var v : boolean;
    begin
    end;
  procedure Boolean;    { Erroneous since "boolean" has already }
    begin               { been used in this level by the "V" in }
    end;                { procedure "P".                        }
  begin
  end.
```

4.3 General Extensions

A few extensions were added to Pascal to make it a more comfortable language to use, and to provide a separate compilation facility. There is no type checking or definition checking performed between separately compiled modules.

The following terms are used in the following discussion with a specific meaning.

() TASK - often called a Process in the literature. TASK is used instead because Process has a special meaning already in VAX/VMS.

() ROUTINE - a collective term referring to any Procedure, Function, or Task. This is used because these three have a common syntatic structure.

4.3.1 %INCLUDE directive

At any point where a space, comment, or lexical token is legal a %INCLUDE directive may be used. The format is

```
%INCLUDE 'file-name'
```

and this causes the compiler to behave as if the sequence of characters and end-of-lines encountered in the %INCLUDEd file had been encountered in the current file at that point. %INCLUDE directives may occur inside %INCLUDEd files. The main intended use of this feature is as files of definitions which contain those parts of modules that must be constant for all the modules making up a program.

4.3.2 Modules

The Module facility is implemented as in VAX-11 Pascal 1.3. A Module has almost the same syntax as a Program, the only difference being that a Program has a Compound_Statement and a Module does not.

```
Program      ::= PROGRAM Identifier [ File_List ] ; Block .
Module       ::= MODULE  Identifier [ File_List ] ; Module_Block .
Block        ::= Declarations Compound_Statement
Module_Block ::= Declarations END
Declarations ::= { [ Label_Declaration ]
                 [ Constant_Declaration ]
                 [ Type_Declaration ]
                 [ Var_Declaration ]
                 [ Procedure_or_Function_or_Task_Declaration ]
                 }
```

The following requirements and recommendations should be followed when writing Modules to be sure that inconsistences will not develop between the various components.

(1) The file and variable declarations in the outer most level of the Program and the Module must be the same, and it is strongly recommended that they be placed in an %INCLUDE file (see 4.3.1).

(2) It is strongly recommended that the declarations of the exportable routines in the Module be written in such a way that a text editor can easily extract the declarations and construct %INCLUDE files containing FORWARD or EXTERNAL declarations. The module itself can then %INCLUDE the FORWARD declarations, while other Modules wishing to import the declarations can %INCLUDE the EXTERNAL declarations.

4.3.3 Relaxed Declaration Ordering

To enable %INCLUDE to be used to build a Program out of several files (each of which may wish to contribute some constants, types, variables, and routines) MTL allows these various declarations to be mixed in any order. This allows the programmer to place the various constants, types, and variables related to a particular object or class of objects textually together in the source code of the program.

For instance one aspect of a program may be related to implementing an abstract data type called "string of characters". The constants (such as "maximum_string_length"), types (such as "string"), and routines (such as "concatenate") can be placed together in one place in the program, separate from another section of code that implements "complex numbers".

4.3.4 EXITLOOP and RETURN statements

To avoid many of the instances where Pascal requires labels and GOTO's, MTL provide EXITLOOP and RETURN statements. EXITLOOP is equivalent to a GOTO to a label just beyond the end of the current inner-most FOR, REPEAT, or WHILE loop. RETURN is equivalent to a GOTO to just before the END of the routine's compound statement.

4.4 Extensions Supporting TASKS

4.4.1 Goals

MTL tasks are not truly concurrent. There is only one current task, all other tasks are either in the list of executable tasks or suspended for one of a variety of reasons. It was decided that this was acceptable, given that MTL was designed to run on a single CPU system and provided that the user was given some methods of rescheduling. The message passing semantics (4.5) provide such rescheduling, as do some primitive statements (4.4.4).

MTL tasks provide the concurrency needed for a message passing system and they are flexible enough to allow the user to write his own scheduling mechanisms. Shared variables are supported. It was decided not to use pre-emptive scheduling because of problems with critical sections of code, and because of run-time efficiency. The tasks have to be capable of calling external routines (see 4.2.5) and this either could only be done by avoiding pre-emptive scheduling or by adding significant overheads to the run-time system. Furthermore by not using pre-emptive scheduling the user can experiment with indivisible operations without need for special interlocks. For example COUNT := COUNT+1 is a problem with pre-emptive scheduling because the value may change between its usage on the right hand side of the assignment and the assignment being made. This problem does not occur with MTL.

The creation and scheduling of MTL tasks had to be simple, fast, and flexible. The need for flexibility led naturally to dynamic rather than static creation of tasks so networks of tasks could be assembled as desired.

4.4.2 TASK routines

The declaration of a Task has the same syntax as that of a Procedure, except all parameters must be passed by value. A Task may access any variable in scope, not only main level variables but also those in the routines that enclose the task. Tasks have associated with them a unique value, of predeclared type TASK_ID, which can be used to refer to them for control purposes. This value is assigned at Task Creation time, and is guaranteed not to be used again until an indefinite but long time after the task's death.

A task dies when it has finished executing its Compound_Statement.

Since tasks can access variables in any surrounding scope, an attempt by a routine to exit (hence deallocate an activation record) causes the exit to be delayed until all tasks that are sharing the activation record have died. This guarantees that all the variables accessable by a task exist throughout its life time. If the task could have VAR parameters then either the task's creator or the owner of the deepest variable the task could reference would have to be suspended until the task died. To prevent this complication it was decided to restrict Tasks to value parameters. This leads to clearer programs and does not seriously restrict the usage of tasks or shared variables.

MTL tasks thus form a tree, with the main program being the root. When it dies the program terminates, but this can not occur before all the created tasks have died because all the created tasks are sharing its activation record. This differs from COSPOL[RO81] where initially there may be several tasks in existence and there is no sharing of variables between these tasks. Execution of a COSPOL program continues until all these tasks have died, with there being no restrictions on the order of their death. A Parallel_Command, which executes a list of tasks in parallel, does block until all the tasks have finished, and this corresponds to the delaying of routine exits in MTL.

If no tasks are executable, and none are waiting on a future event (see 4.2.7) then the program is dead-locked and the runtime system generates an error message. This normally results in image termination.

4.4.3 CREATE statement

Tasks are dynamic objects. They are created by the Create_Statement whose syntax is

```
Create_Statement ::=
     CREATE Task_Id_Variable := Task_Name [ Parameter_List ]
```

The task is created, any parameters are copied into its activation record, its task_id (see 4.4.2) is assigned to the variable, and it is placed in the list of executable tasks. However in keeping with MTL's philosophy of scheduling the task's creator is kept as the currently executing task.

4.4.4 RESCHEDULE, SLEEP, and AWAKEN procedures

Because MTL does not do pre-emptive scheduling three predeclared procedures are available for user-controlled scheduling, besides the implicit scheduling involved in message passing. They are

```
RESCHEDULE,
SLEEP, and
AWAKEN(sleeping_task:task_id)
```

Rescheduling places the current task at the end of the list of executable tasks, and picks another to replace it.

Sleeping suspends the current task's execution until another task awakens it.

AWAKEN(sleeping_task:task_id) is used for awakening such tasks. If the task specified in the call to Awaken is not sleeping nothing happens.

4.5 Extensions Supporting MESSAGES

4.5.1 Goals

Many and various proposals have been put forward concerning message passing. Amongst the decisions facing the language designer are

(1) Whether to transmit a copy of the message, or the original.

(2) Is message transmission Asynchronous, with some degree of buffering, or Synchronous?

(3) Whether or not the receiving task can name the required source of the message.

(4) Whether or not the sending task can name the task that is to receive the message.

In the design of MTL these choices were made using criteria of speed and flexibility. The choices made for MTL in each case leaves the alternative possibility available to the programmer as a subclass of the chosen possibility, with no loss of speed. Furthermore the source code of the program can be easily checked to verify that only the desired mechanism has been used.

4.5.2 MESSAGE variables

Transmission of the ORIGINAL was decided upon because that seemed to be the most common usage. MTL messages are objects allocated in the heap, and thus copies of them can be easily made using NEW and an assignment statement. It is possible to verify that the transmit-copy mechanism has been used by checking that every SEND statement (4.5.5) is immediately preceded by a copy being taken of the message, and that it is this copy that is sent. Such a check could be made by a simple mechanical program verifier.

Message types are formed using the reserved word MESSAGE. The syntax of a message type declaration is

Message_Type ::= MESSAGE [Field_List] END

This declaration is similar in effect to declaring a pointer to a record whose type is not explicitly available. Hence a variable of message_type must be treated as though it were a pointer to a record, and the fields of the message accessed accordingly. This is so because message passing is then done by passing this pointer (see 4.5.5).

4.5.3 BUFFER variables

ASYNCHRONOUS communication, with a bounded buffer, was implemented. By specifying the size of the buffer as 0 a SYNCHRONOUS protocol can be enforced.

Section 7.4.2 examines the performance implications of selecting asynchronous rather than synchronous communication, showing that asynchronous communication is inherently faster when bounded buffers are actually required.

Buffers are shared variables used during message passing to buffer the messages and to synchronise their transmission.

The syntax of a buffer type declaration is

Buffer_Type ::= BUFFER OF Message_Type_Identifier

Buffers must be initialized before their first use. There is a predeclared routine BUFFER_INIT that takes any variable of any buffer type and initializes it. It takes two parameters, the first being the variable, and the second being the maximum number of messages that can be buffered.

When the second parameter is zero, communication via the buffer is synchronous; thus it is possible to verify that the program only uses the synchronous mechanism by examining all the calls to BUFFER_INIT.

For example

```
      BUFFER_INIT(input_buffer,10)
```

4.5.4 BUFFER READERS, BUFFER WRITERS, BUFFER MESSAGES

Each buffer has three queues associated with it, a queue of readers trying to read messages from the buffer, a queue of messages being buffered, and a queue of writers trying to write messages to the buffer. The size of any of these queues can be obtained via the predeclared integer valued functions BUFFER_READERS, BUFFER_MESSAGES, and BUFFER_WRITERS. These functions have a single input parameter, a buffer variable.

This mechanism is slightly more flexible than the PENDING function in COSPOL[RO81] because it allows a program to determine how many messages are being buffered rather than just the existence of some pending messages. This may be useful in simulation programs which use the buffers to model queues.

The PENDING function is equivalent to

```
  FUNCTION PENDING(B:BUF) : BOOLEAN;
    begin
    PENDING := (BUFFER_WRITERS(B)+BUFFER_MESSAGES(B)) > 0;
    end.
```

On the other hand it is slower (in cpu-time) than PENDING because it requires some mechanism for keeping track of the number of objects in each queue. It remains to be seen which of the two is more desirable.

4.5.5 SEND and RECEIVE statements

Rather than naming sources and destinations MTL uses the buffers as mailboxes, with any task (within scope) being allowed to both SEND and RECEIVE messages via them. Static checks can often be made to verify that only one TASK is capable of receiving messages via the buffer if that is the desired behaviour, or that all messages of a particular type are received via one particular buffer.

In COSPOL the name of the receiving task and the type of the message are used to select an anonymous buffer for the transmission. Each task has a collection of buffers, one for each type it is capable of receiving. This is a subset of MTL's mechanism that has the advantage of less code being required for specifying the connections and the Send and Receive operations, but it is less flexible.

Buffers can be created dynamically, using NEW, and the pointer passed as a parameter to one or more created tasks. This allows the run-time creation of arbitrary networks of tasks with any desired pattern of message flow.

They also allow a task to have several input streams of messages of an identical type. For instance a task can be receiving messages from two buffers of the same message type, one being designated "high-priority" and the other "low-priority". Senders can place their message in either buffer according to the urgency of the message. A similar situation arises when using tasks to simulate nodes on a network (for example a intersection of several roads). The messages (cars) coming in from the different input buffers (roads) are clearly of the same type yet each buffer should be distinct.

Message transmission is done by use of SEND and RECEIVE statements whose syntax are

```
Send_Statement    ::= SEND Message_Variable VIA Buffer_Variable

Receive_Statement ::= RECEIVE Message_Variable VIA Buffer_Variable
```

The semantics of a Send_Statement are

(1) The task is suspended in the buffer's sender queue until either
(i) there is room in the buffer for the message, or
(ii) there is a pending Receive for a message from the buffer and the Send_Statement's message is the next message to be sent.

The second alternative allows the use of buffers of zero size for synchronous communication.

(2) The Message_Variable is set to NIL. All further attempts to access the message (by retaining copies of the Message_Variable) are invalid. This is the same situation as using DISPOSE on a pointer and then trying to access the disposed object by retaining other copies of the pointer. It is not expected that this problem with messages will

cause the programmer any more serious problems than DISPOSE currently does.

The semantics of a Receive_Statement are

(1) The task is suspended in the buffer's receiver queue until there is a message available for it to receive.

(2) Set the message variable to the message received. This is similar to NEW and, as would be expected, the task should either DISPOSE the message, pass it on to another task, or retain it. Re-using the message variable would have the same affect as re-using a pointer variable, leaving an in-accessible item in the heap.

4.6 Summary

The MTL language is an extension of Pascal that provides for concurrency and message passing, as well as some other additions to simplify the use of the language.

Tasking may be used either at a very low level, with explicit Sleep, Awaken, and Reschedule control, or the user can ignore the issue of scheduling by relying on the message passing primitives for the rescheduling required for pseudo-concurrency.

The use of buffers rather than task-directed messages allows all combinations and permutations of data flow, such as multi-server queues and dynamic redirection of message streams.

MTL message passing provides a collection of high-level primitives sufficiently powerful to implement most message passing proposals. The language designer can use MTL to experiment with the various mechanisms in the context of experimental programs, and be able to have these compiled and run rather than just looking at them on paper. This in turn will provide experience in debugging such programs - experience which is very hard to gain in any other way.

Alternatively the structures are sufficiently convenient for a user who merely wishes to implement a concurrent algorithm. This means MTL is usable by both naive and experienced programmers, and in a wide variety of roles.

Chapter 5
DETAILS OF THE MTL IMPLEMENTATION

5.1 The MTL Compiler

5.1.1 Goals

The major design goals of the MTL compiler and implementation were

(1) It should be a standard VMS compiler, generating standard object modules for the VMS Linker.

(2) It should support the VMS procedure-calling standard.

(3) It should be flexible, allowing further features to be easily added to the language.

(4) It should be acceptably fast.

(5) The code produced should be of a comparable quality to the Pascal V1.3 compiler. That compiler is not an optimizing compiler, but does produce reasonably good code. All simple optimizations should be done, including using the correct branch code optimizations, and peephole optimizations.

In view of these design goals a fairly simple, multi-phase compiler was implemented. Initially only a small subset of Pascal was implemented, then this set was incrementally extended until almost full Pascal was available.

5.1.2 An Overview of the Compiler

The compiler consists of a collection of modules written in BLISS32, a system programming language and compiler writing language for VAX/VMS. Since there were no sources of any VMS compilers available, nor any true compiler writing tools, the entire compiler was written from scratch. Some aspects of the design of the compiler reflect the fact that BLISS32 is a typeless language and hence various tricks are possible in the compiler that are not possible with a typed language.

A table generator was written in Pascal to translate a textual representation of the syntax and references to the required action routines into a MACRO-32 assembly language form that, when assembled and linked into the compiler, drives the LL(1) parser.

The scanner {LEX} is hand-coded and breaks the input stream into tokens. It also handles %INCLUDE files and removes comments. The parser {PARSER} is driven by the table to recognise the syntax and calls the appropriate action routines. It handles recovery from all syntactic errors. The result of the action routines is either the maintenance of the symbol table or the production of a code-tree for the current routine. This code-tree is fed through {CODE} to produce an instruction stream which is then fed through {OBJ} which produces an object module and a machine-code listing after applying branch code optimizations and peephole optimizations.

{HEAP} manages the compiler's heap, and {TXTIO} does the I/O of the source and listing files.

5.1.3 Specification of the Grammar and Semantic Actions

The input to the program that builds the Parser tables is a file of lines with some additional provisos. Lines with

(1) '!' in column 1 are ignored. (Comments)
(2) '-' in column 1 are appended to the previous line. (Continuations)

Blank lines are ignored.

The syntax of the input is specified by the following rules. Note that {...} means "zero-or-more occurences of", ¦ means "alternatively", and (...¦...¦...) means "one-of the specified possibilities.

```
  file  ::=  { atom-line } '$'-line { prod-line } '$'-line

  atom  ::=  lexical-token-identifier

  prod  ::=  identifier [ '=' rule { '¦' rule } ]

  rule  ::=  item { item }

  item  ::=  { '#' action-routine-name } unit { '#' action-routine-name }

  unit  ::=  identifier
          ¦  '[' rule ']'
          ¦  '{' rule '}'
          ¦  '(' identifier { '¦' identifier } ')'
```

For example an extract from the actual MTL tables is ...

```
!
! Declare the basic tokens that the scanner {LEX} knows about.
! {LEX} yields these pseudo-productions as it recognizes the
! individual tokens.  They are declared here so that the program
! that builds the table knows that they are not true productions,
! and hence will not complain about them not being declared in the
! list of productions below.
!

LEX_K_UNDEFINED
LEX_K_ENDOFFILE
LEX_K_IDENT
LEX_K_INT
LEX_K_PROGRAM
LEX_K_BEGIN
LEX_K_WHILE

$
!
! Part of the syntax specification of MTL, showing the syntax of a
! Program or Module, as well as calls to some of the action routines.
!

COMPIL_UNIT     = # ACT_INITIALIZE
-                   ( PROGRAM | MODULE )
-                 # ACT_WINDUP
-                   LEX_K_DOT

PROGRAM         = # SYM_NEWTOPLAYER
-                   LEX_K_PROGRAM
-                   NEW_MODULE # DECL_ROUTINE
-                   [ LEX_K_LPAREN HEAD_PARAM LEX_K_RPAREN ] LEX_K_SEMI
-                   PROGBODY
-                 # SYM_OLDTOPLAYER

MODULE          = # SYM_NEWTOPLAYER
-                   LEX_K_MODULE
-                   NEW_MODULE # DECL_ROUTINE
-                   [ LEX_K_LPAREN HEAD_PARAM LEX_K_RPAREN ] LEX_K_SEMI
-                   MODUBODY
-                 # SYM_OLDTOPLAYER
```

These are used to generate the table that is interpreted by {PARSER}. The table is a list of commands that control the flow of the compilation by directing {PARSER} to accept an input token, attempt to recognize a sub-production, call an action routine, etc.

The possible commands are...

CALL entry-point,
ENTER production-name,
RETURN

These implement the parsing of sub-productions. CALL directs {PARSER} to recursively call itself, with the entry-point specifying the first command to be executed. This is always an ENTER command which specifies the name of the production being recognized, a useful piece of information when tracing the behaviour of {PARSER} and when generating error messages. RETURN causes the current call to {PARSER} to return.

{PARSER} creates a record on the stack for the action routines called by the production to use as common storage. There are three special components in this record called BACKLINK, RETURN, and RESULT. BACKLINK is a pointer to the similar record of the calling production. The RETURN field is copied into the RESULT field of the calling production as part of the RETURN instruction. Thus action routines can locate and use information passed back by sub-productions or can access information stored higher up the parse tree. Typically this field passes back a pointer to a node that will be linked (by action-routines higher up the parse tree) into the code-tree.

ACTION action-routine

This command is the only way action routines are called from the Parser. They are passed, by reference, the current production's record and are capable of examining the current token and any other global variables. Action routines perform such functions as type-checking, symbol table maintenance, and the actual building of the code-tree.

LEXEME token,
OPTIONAL token-set alternative-command,
GOTO command
BAD

The LEXEME command accepts the specified token, and calls the scanner to read the next one. If the current token is not the specified token, an error message is generated and the limited error recovery mechanism goes into effect. OPTIONAL branches to the alternative-command if the current token is not in the specified token-set. The current token is not changed. The GOTO branches unconditionally to the specified command. BAD is used to indicate that there is no other possibility, the parse has run into a dead-end caused by a syntax error.

For example:

Consider the production

```
EXTERNAL_FILE_LIST = EXTERNAL_FILE { LEX_K_COMMA EXTERNAL_FILE }
```

This would translate into the following list of commands.

```
EXTERNAL_FILE_LIST: ENTER     'EXTERNAL_FILE_LIST'
                    CALL      EXTERNAL_FILE
LOOP:               OPTIONAL  [LEX_K_COMMA], FINISH
                    LEXEME    LEX_K_COMMA
                    CALL      EXTERNAL_FILE
                    GOTO      LOOP
FINISH:             RETURN
```

5.1.4 The Symbol Table

The Symbol Table is maintained by the module {SYM}. Unlike many Pascal compilers the identifier table is maintained using a hashing technique rather than a binary tree. Each entry in the hash table corresponds to a particular identifier spelling. All instances of this identifier currently in scope hash to this entry in the table, and are stacked (in a chain) in it. Later definitions of the identifier are pushed on top of this stack. When a scope is exited all identifiers that were declared in the scope are popped off their stacks. This mechanism was used because it is potentially faster (although no tests have been done) and I was interested to see if it was workable (a preliminary investigation failed to show up the one place where it would not work).

There is only one place where it does not work for standard Pascal. When an enumerated type is enumerated as part of a Record declaration the enumerated-type-identifiers must be declared at the main scope of the procedure or function, whereas the field-identifiers are only declared within the scope of the record.

For example:

```
  Type
    enum = (alpha,beta,gamma),
    rec  = Record
             enum : (alpha);  { ENUM doesn't clash, ALPHA does       }
           End;               { because ALPHA gets declared outside }
                              { the scope of the record.            }
```

Rather than hack a solution to this into the compiler, a decision was made that in MTL field-identifiers are not entered into the symbol table, but are an attribute of the record-type. Hence in MTL, but not in Pascal, the following is valid ...

```
  Record integer : integer; End
```

5.1.5 HEAP Management

The compiler's heap is managed as a stack with calls to NEW allocating the specified object on top of the stack. MARK and RELEASE operations are used to peel layers off the stack when entering and exiting scopes. There is also a true Heap available for use in places (such as the I/O routines and some aspects of code generation) where it is required because the pattern of allocation and deallocation does not follow the LIFO (Last-IN, First Out) behaviour of the rest of the objects.

5.1.6 The Semantics Routines

Each semantics routine (elsewhere called an action routine) perform one simple operation. They either build parts of the code-tree, performing any required checks such as type compatibility, or do such house-keeping as symbol table maintenance. The semantic routine that ends a routine declaration calls the code-generator passing it the code-tree for the routine.

The code-tree closely resembles a parse tree of the executable part of the routine. Each node is a variant record (in the Pascal sense) with a field specifying which variant the node is. This code tree contains pointers back into the symbol table for the various identifiers, types, etc. The nodes can be broken into two distinct groups, the STATEMENT nodes and the EXPRESSION nodes. The statement nodes have a pointer indicating their successor, whereas the expression nodes have a pointer indicating the type of the expression. Expression nodes also indicate which component of an expression they are, some possibilities being a variable, a binary operator (in which case the node will contain two pointers, one to the right-operand-expression-node and one to the left, as well as the name of the binary operator), or an array indexing operation.

For example during the processing of I := J the production for assignment statements would be used by the Parser. This production is ...

```
Assignment = # CHECK_VAR Var # CMP_ASSIGN Lex_K_Assign Exp # CMP_ASSIGNEXP
```

The semantic routine CHECK_VAR is necessary because a valid alternative would be Function_Identifier. The parser then parses a Var, and the RETURN from it would copy a pointer to a code-tree-node for a Variable into the result field of the current {PARSER} record (see 5.1.3). This result field is copied into the code-tree-node for an assignment expression that is built by CMP_ASSIGN. CMP_ASSIGN places a pointer to this node in the RETURN field of the current record. CMP_ASSIGNEXP gets a pointer to an expression code-tree-node from the result field, does a type-compatibility check between the Expression and the Variable, and copies the pointer to the code-tree-node into the assignment code-tree-node.

5.1.7 Generating Code from the Code Tree

The module {CODE} converts the code-tree into an Intermediate Code which is stored in an array. {CODE} is a recursive-descent tree traversing algorithm, but note that because a correct parse has already been achieved at this stage there is no need for error detection or recovery. During the traversal of the tree {CODE} does code-selection, register allocation, and some optimizations that are easier to perform on a tree than they are on the Intermediate Code.

One such optimization that {CODE} performs is the folding of any constant array indexes back into the address of the array. For instance A[i,4,j] is changed to A+constant+f(i,j) where f(i,j) must be computed at run-time.

Because {CODE} works on a code-tree it is relatively easy to do optimizations involving re-ordering the code. The simplest such optimization is the shifting tests to the end of loops which has the effect of removing one branch from inside the loop to outside.

For example:

```
  while condition do                          goto test
    begin                           start:  statement
    statement;...                           ...
    end;                            test:   if condition then goto start
                                    fini:
```

It is also possible to generate code that short circuits the evaluation of boolean expressions because the full context surrounding the boolean expression is available. Rather than evaluating the expression then testing the result, the code produced can use the evaluation to effect the flow of control. This is achieved by optionally passing a TRUE_LABEL and a FALSE_LABEL to the {CODE} routine that processes binary operators. When these labels are provided, the routine generates Test_And_Branch rather than Evaluate_And_Store style instructions. (This same technique optimizes such statements as "IF (NOT condition) THEN" by doing a recursive call with the TRUE_LABEL and the FALSE_LABEL transposed, rather than generating any code for the NOT.) For example ...

```
  if (a<b) or (b<c) then                    if (a<b) then goto true
                                            if (c<=b) then goto false
    begin                           true:   statement
    statement;...                           ...
    end;                            false:
```

Each call to a {CODE} routine that is to generate code for an expression optionally indicates where the caller wants the answer to be placed. If this is not specified the routine returns the place where the value is to be found. Thus statements such as "A := B+C" generate "ADD C to B giving A" rather than a more complex form that would then need improvement later. Computed results (as opposed to simple variables or constants) that must be returned are returned in registers unless a place has been specified for them.

{CODE} does not use a complicated register allocation scheme, although one could be added without affecting the rest of the compiler. Registers are only used for temporary results, limits of FOR loops, WITH statements, and similar constructs, and as pointers to each of the static scopes.

After completing the conversion to intermediate code, {CODE} calls {OBJ} to produce the object module.

5.1.8 Producing the Object Module from the Intermediate Code

{OBJ} does single-instruction peephole optimizations, and then branch code resolution. Where possible shorter branch code sequences are used, but if necessary the longer sequences are inserted.

The peephole optimization phase is essential because of the wide range of equivalent, but shorter, instructions that the VAX architecture supports.
For example the code produced by {CODE} from "A := A+1" would be

```
      ADDL3   #00000001,0000001C(R11),0000001C(R11)     ; 16 bytes
```

This is shortened by {OBJ} successively to

```
      ADDL2   #00000001,0000001C(R11)                   ; 11 bytes
      INCL    0000001C(R11)                             ;  6 bytes
      INCL    1C(R11)                                   ;  3 bytes
```

Branch code optimization is necessary because the VAX architecture has 3 different ranges of branch instructions; Conditional branches that can branch up to about 128 bytes forward and back, Unconditional branches that can branch up to 32767 bytes forward and back, and Jumps that can jump anywhere. If this optimization was not performed then unconditional branches or jumps would have to be used in all cases.

For instance the code for "IF A < B THEN" would be

```
this                          instead of


      CMPL    a,b                   CMPL    a,b
      BLSS    true                  BGEQ    false
      BRW     false           true: ...
true: ...                     false:
false:
```

Finally the data for object module is produced and a symbolic listing of the machine code and the actual byte stream is placed in the listing file.

5.1.9 Summary

The MTL compiler has proven fairly straight-forward to implement. The use of a table driven parser has meant there were almost no problems with the syntax recognition of Pascal, and the separation of code generation from the syntax/semantic analysis has simplified the action routines as well as the inclusion of optimizations. Another important advantage of the separation occurs after the first error has been detected. From then on no further attempts to generate code are made, and hence the compilation procedes at a faster rate, although syntax analysis and type checking proceed as before.

It has proven very easy to extend and modify MTL because of the extreme modularity of the design. The steps involved are in adding a new feature to the language are typically ...

(1) add any reserved words to the reserved word tables in {LEX}.

(2) add the construct to the SYNTAX.DAT file and rebuild the Parser table.

(3) add any needed action routines,

(4) add any needed support in {CODE}.

Usually there is no reason to change {OBJ}.

The compiler takes up 182 blocks (512 bytes each) of disk space, compared to the VAX-11 Pascal 1.3 compiler which is about 540 blocks. The difference in size is due to three causes.

(1) MTL does not support a complete implementation of Pascal, and does not do as good a job at error recovery as the Pascal compiler. However this would not account for such a large discrepency.

(2) The Pascal compiler is written in Pascal but MTL is written in BLISS-32, and the BLISS-32 compiler produces shorter code than the Pascal compiler.

(3) MTL uses a table driven LL(1) parser, but the Pascal compiler is a recursive descent compiler. This requires a lot of machine code compared to the compactly represented LL(1) table in MTL's {PARSER}.

5.2 The MTL Run-time System

5.2.1 Goals

Because MTL was to be used for speed comparisons with conventional languages the MTL run-time system had to impose minimal cpu-time overheads on the execution of programs.

Where possible standard VMS run-time support routines were to be used. Because MTL is an extension of Pascal most of the Pascal support routines were available, particularly for the support of I/O. This helped keep the size of the MTL-specific run-time system down to about 500 machine instructions, coded in assembler.

It was desirable that the normal VAX/VMS debugging tools, especially the VAX/VMS Symbolic Debugger, were available for use in debugging programs. This is only partially attainable because there is no document available to the author on the format of the data that is transferred from the compiler to the Debugger, hence a lot of the information the the manufacturer's compilers transfer is not transferred by MTL.

5.2.2 An Overview of the Run-time System and Environment

The MTL run-time system provides support for the multi-tasking and the message passing aspects of MTL. The normal Pascal run-time system is used to do I/O. Routines written in languages other than MTL can be called because MTL generates the standard VMS calling mechanism for procedure calls. However the main program must itself be written in MTL because initialization of some data structures is currently carried out as part of the main program.

The primary problem facing an implementor of a concurrent block-structured language on VMS is where to place the activation records (in VMS jargon these are called CALL FRAMES) of the routines. There are three basic alternatives.

(1) the various tasks are run as different VMS processes, and hence have their own address spaces. This makes interactive debugging difficult because VMS does not have any convenient way of managing multiple processes simultaneously. The processes, which may number thousands, would consume huge amounts of critical system resources (eg. such as the nonpaged-pool). VMS is very slow at creating such processes and this would impact their use

UNIVERSITY OF ADELAIDE LIBRARY

in very dynamic configurations.

(2) the stacks for the different tasks are placed in widely separated areas of the process address space. This is viable provided there is operating system support, such as stack limit registers. The discussion in 3.2.1.1 explains why this option was not used under VMS Version 2.

(3) some form of heap is maintained with the activation records spread through it in a tree structure. Since the only other two alternatives were unsuitable this one had to be used. The disadvantages with this approach are

i. the costs of copying CALL FRAMES from the one true stack (where they are placed by the CALL instructions) into the heap, and that of copying it back to the stack where the RETURN instruction expects it.

ii. because the call frames are not on the stack, but linked together in the heap, Debugger and Traceback utilities are not capable of providing a display of the dynamic scope. (The utilities should therefore be generalised to cope with call frames in other locations than the stack. Implementing such modifications to VMS were beyond the scope of this thesis.)

iii. also because the call frames are not on the stack the VMS condition handling mechanism can not be used. This is a serious disadvantage for the language, and one that seems impossible to avoid given the current behaviour of VMS. Condition handling is not discussed further in this thesis, although it is an issue that implementors of Ada under VAX/VMS are going to need to address.

As appendix C shows, MTL does badly in comparison with the conventional stack-based CALL on the VAX-11 because of the costs mentioned in (i). Several steps can be taken to alleviate the problem. A relatively simple optimization for the compiler would be to detect routines that can not be interrupted by rescheduling and run these on the stack, never copying the CALL FRAME into the heap. In practise this is what happens with all of the routines that are not written in MTL. For routines that are internal to a module (so that the compiler knows all instances of it is called) a different call mechanism could be used. A simple mechanism would be to use the JSB instruction to call the routine so that only the return address is placed on the stack. The called routine could then allocate the heap space and build the frame directly in the heap. For all routines the return mechanism could be sped up by emulating the RET instruction based on the heap, rather than copying the heap item back to the stack as is currently done. It is estimated that a 2- or 3-fold improvement in procedure calling rates could be achieved in this way.

Recent developments in computer architecture, for example the Mesa Processor[JO82,LA82], show that heap based procedure calling mechanisms can be implemented in hardware. There is no reason why such machines should run any slower than others using conventional stack architecture.

If VMS did support multiple USERMODE stacks then the difference in procedure calling could be made to disappear completely by allocating a stack per task. Most conventional operating systems would already meet this requirement because they do not have the restrictions that the VMS software imposes on the usage of the USERMODE stack.

5.2.3 Heap Management

It is immediately apparent that the allocation and deallocation of items from the heap is going to significantly affect the performance of procedure calling. As we shall see in 5.2.5 it also significantly impacts on the speed of message passing.

In the past memory has been relatively expensive and a lot of research has been done in the area of heap management. With the advent of relatively cheap memory and large virtual address spaces many of the original constraints are no longer binding. Instead it would appear that the most important criteria for a heap management scheme is cpu-time and page-fault behaviour, not the careful usage of every available portion of memory. This being the case any cpu-time spent in coalescing the heap is a waste of resources, unless fragmentation is a serious problem.

MTL uses an extremely simple heap management system. An array of queue headers, one queue for each size of object between 8 and 512 bytes in size. No object may be less than 8 bytes. When an object of a larger size than this is to be allocated, or the queue of free objects of the correct size is empty, the standard VMS heap management utility is used. However if the queue is not empty a single REMQUE instruction removes the first free object. When an object is deallocated it is placed in the free queue for the appropriate size unless it is too large, in which case it is deallocated again using the VMS heap manager. Experiments with the VAX-11 Pascal 1.3 implementation have shown a similar mechanism to be able to reduce page-faults in pathological situations by factors in excess of 100 because the list of free space is not being searched in futile attempts to coalesce the items with a neighboring free area.

In this way the queues effectively cache the heap. The queues of objects tend to grow to an optimal size, and allocations are satisfied via very short instruction sequences. Currently the following code would be executed as the result of a call to the Pascal predeclared procedure NEW(pointer).

```
        MOVL    S^#8,R0             ; R0 := the size of the object.
        MOVAL   B^-12(R13),R1       ; R1 := the address of the pointer.
        JSB     MTL.NEW_R1          ; Jump-subroutine to the MTL Heap manager.
        ...

MTL.NEW_R1::
        MOVQ    R0,-(SP)                      ; Save the two registers.
        CMPL    R0,#SHORT_HEAP_ITEM           ; See if short enough for our
        BGEQ    10$                           ; super-fast heap.
        MOVAQ   HEAP_Q_HEADERS[R0],R0         ; Get pointer to queue header.
        REMQUE  @(R0),(R1)                    ; Try to unlink item from queue.
        BVS     10$                           ; Br if queue was empty.
5$:     ADDL2   #8,SP                         ; Pop the registers.
        RSB                                   ; Return from subroutine.
        ...
```

If this sequence is not fast enough, the size comparison could easily be performed by the compiler, and the REMQUE/BVS pair of instructions inserted as in-line code. However this sequence was adequate for MTL. Deallocation follows a similar path. Appendix D compares this very simple strategy with that of VAX-11 Pascal 1.3.

There are two non-obvious advantages to this mechanism. Firstly the size of each queue tends to grow until they are just enough to cover peak demands. Secondly traditional heap management schemes that do coalescing during deallocation are susceptible to poor cpu-time performance due to strange orders of deallocation. Such behaviour can be expected in a message passing system. The reader is invited to try out the following example on an available Pascal system.

```
Program Heap(output);
  type ptr = ^integer;
  var i : integer;  a : array[1..1000] of ptr;  start : integer;
  begin
  for i := 1 to 1000 do new(a[i]);
  start := clock;
  for i := 1 to 500  do dispose(a[2*i-1]);
  for i := 1 to 500  do dispose(a[2*i]);
  writeln(clock-start:1);                      {COMPARE THIS FIGURE}
  for i := 1 to 1000 do new(a[i]);
  start := clock;
  for i := 1 to 500  do dispose(a[i]);
  for i := 1 to 500  do dispose(a[i+500]);
  writeln(clock-start:1);                      {AND THIS FIGURE}
  end.
```

5.2.4 The Multi-Tasking Subsystem

Multi-tasking is accomplished by context switching the single processor between the currently active tasks, thus is only pseudo-concurrency. It is implemented by saving the entire context of one task in a task control block and loading the context out of another block. The only aspects of the machine state that must be saved in this way are the 16 registers. All other aspects of the task are kept either in the TCB (task control block) or are in the activation records in the heap.

Each task can be in one of a variety of states. Each is discussed in turn below, in the order that they tend to cycle through. For each state there is a queue of tasks, and each task control block records which queue the task is currently in. This is very similar to the manner in which the VMS scheduler maintains information about processes, and indeed the design of VMS did suggest the design of the MTL scheduler.

Initially all TCB's are in the IDLE queue. There are finitely many TCB's and they are stored in an array. During the execution of a Create_Statement the Task is called as a normal function. The task copies the parameters (which may only be passed-by-value) into local variables in the activation record and then calls the MTL run-time routine MTL.FORK. FORK allocates a TCB from the IDLE queue, fills in the context, and places the TCB at the end of the COMPUTABLE queue. It then emulates a return back to the creator of the task, returning the Task Identifier (TID) of the created task.

When a task releases the cpu the task at the head of the COMPUTABLE queue is removed and placed into the CURRENT queue (which only ever has one entry). If the COMPUTABLE queue is empty, and no tasks are waiting for events flags (see 5.2.6) to happen the program is deadlocked and is therefore aborted. If tasks are waiting for event flags the whole VMS process is paused until one of the event flags being waited for becomes set.

The current task executes as long as it has something to do. Eventually it calls MTL.SCHEDULE either indirectly as a result of a message operation or wait-for-event-flag operation, or directly as a Reschedule_Statement. There are a variety of possible resulting states for the task.

The MESSAGE state is used when a task is either trying to send or receive a message but is temporarily blocked. The MESSAGE state is only used by the message passing subsystem, and the task is only put into this state by that subsytem, and is removed from it by that subsystem when the task is no longer blocked.

A task is placed in the SLEEP state by the Sleep procedure(see 4.4.4). The AWAKEN procedure removes tasks from this state and returns them to the COMPUTABLE queue.

The WAITING_FOR_EVENT queue is used for tasks waiting on an event flag. The task is returned to the COMPUTABLE queue some time after the event flag becomes set.

The TRYING_TO_RETURN queue is for routines that are trying to return to their callers, but can not do so because there are still sub-tasks capable of referencing variables declared by this routine. During task creation the task increments the reference count in the activation record belonging to its static parent. It decrements this count again when it dies. The parent routine is therefore unable to exit while this reference count is positive, and hence the task's entire dynamic scope remains accessible throughout its lifetime. When decrementing a reference count yields a result of zero, all tasks in the TRYING_TO_RETURN queue are placed at the end of the COMPUTABLE queue. However they all check their reference count again and any that were not affected by this change get returned to the TRYING_TO_RETURN queue.

Appendix E gives performance figures for MTL context switching.

5.2.5 The Message Passing Subsystem

Send_statements and Receive_statements specify both message and buffer variables. The MTL run-time system provides Monitor-style routines that manipulate these to perform the message transmission and reception. Because these routines are critical to the performance of message passing they are written in assembler, however the techniques of Habermann[HA72] can be applied to proving their correctness, as we will show.

The memory layout of a buffer type is

```
BUFFER_TYPE =
 RECORD                     +--------------------------------+
   FREE      : INTEGER;     | count of free slots in buffer |   4 bytes
                            +--------------------------------+
   RECEIVERS: QUEUE;        | queue header for queue of      |   8 bytes
                            |   blocked receiver tasks.      |
                            +--------------------------------+
   SENDERS   : QUEUE;       | queue header for queue of      |   8 bytes
                            |   blocked sender tasks.        |
                            +--------------------------------+
   MESSAGES  : QUEUE;       | queue header for queue of      |   8 bytes
                            |   messages in buffer.          |
 END;                       +--------------------------------+
```

The memory layout of a message type is

```
MESSAGE_TYPE =
 RECORD                     +--------------------------------+
   MESSAGES  : QUEUE;       | queue entry for message queue  |   8 bytes
                            |  (forward and backward link)   |
                            +--------------------------------+
   REST      : RECORD       |                                |   0 or more
                            |   type-specific fields.        |     bytes
                END;        |                                |
 END;                       +--------------------------------+
```

Using these record structures, the algorithms for Send and Receive can be written...

```
  PROCEDURE MTL_SEND(VAR BUF : BUFFER_TYPE; VAR MSG : MESSAGE_TYPE);
    BEGIN
    if not queue_empty(buf.senders) then wait(buf.senders,TAIL_OF_QUEUE);
      if buf.free <= 0 then wait(buf.senders,HEAD_OF_QUEUE);
        buf.free := buf.free-1;
        insert_in_queue(msg.messages,buf.messages);
    awaken_first(buf.receivers);
    if buf.free > 0 then awaken_first(buf.senders);
    END;


  PROCEDURE MTL_RECEIVE(VAR BUF : BUFFER_TYPE; VAR MSG : MESSAGE_TYPE);
    BEGIN
    if not queue_empty(buf.receivers) then
                    wait(buf.receivers,TAIL_OF_QUEUE);
        buf.free := buf.free+1;
        if queue_empty(buf.messages) then
          begin
          awaken_first(buf.senders);
          wait(buf.receivers,HEAD_OF_QUEUE);
          end;
        remove_from_queue(msg.messages,buf.messages);
    awaken_first(buf.receivers);
    if buf.free > 0 then awaken_first(buf.senders);
    END;
```

These algorithms are a modified version of those proven correct by Habermann[HA72]. The modifications (which do not affect the correctness proof) are...

(1) SIGNAL and WAIT are implemented via a queue of tasks, with a Wait(queue,location) and an Awaken(queue).

(2) Rather than waking a Sending task by the first Wait only to fall into the second, writers are only awakened when there is a free slot in the buffer for the message to be placed in.

(3) The free:=free+1, which corresponds to Habermann's SIGNAL(FRAME) is done before the message is removed. Since MTL does not do pre-emptive scheduling this is possible because either

i. the message queue will be empty in which case the free:=free+1 allows for the case when the buffer size is set to zero and the presence of the receiver has increased the size of the buffer, or

ii. the message queue will be non-empty in which case the removal of the entry and the incrementing of free are commutative.

(4) For RECEIVE the final "awaken(reader); if free > 0 ..." are the opposite way round to Habermann. This is purely because they are commutative (since MTL does not do pre-emptive scheduling) and this gives both routines the same ending, allowing them to share the same machine code.

Appendix F gives message passing rates for MTL under various circumstances, while appendix B shows the importance of supporting message passing in the language rather than having the application program implement it via some other technique.

The Odd-Word-Reversal Problem (Dijkstra[DI68]) was coded as a procedure calling and then as a message passing algorithm and cpu-times compared (appendix G). Several comments need be made about the results, which indicated that message passing could be of a similar speed to procedure calling. The MTL heap was apparently still too slow to keep up with procedure calling. By pre-allocating all the message buffers the cpu-times can be made almost equal. This pre-allocation is not an artificial trick, but corresponds to making the improvements mentioned in 5.2.3 to the MTL heap allocation/deallocation mechanism.

5.2.6 The I/O Subsystem

Pascal-like I/O is done using the VAX-11 Pascal runtime library routines, although the MTL compiler does not support all the standard Pascal I/O. Because of this, any input by a task holds up the entire VMS process while the operation is performed so no other task may execute while the input is being done. A similar situation holds for tasks doing output. In practice this usually only matters for terminals, because the VMS Record Management System does read-ahead and write-behind for disk files. MTL is thus incapable of doing asynchronous I/O on separate terminals using the READ(LN) and WRITE(LN) statements (however it can be done using calls to either RMS or $QIOW (see 3.2.4)).

The VMS I/O System Services are capable of operating asynchronously and notifying the process when the I/O is completed. This notification is carried out in one of two ways, either by the setting of an event flag, or by the delivery of an AST (Asynchronous System Trap - see section 3.2.3 for details).

MTL provides a run-time support routine that suspends the current task until an event flag is set. The event flag must have been obtained from MTL's event flag resource manager which is compatible with the VMS event flag resource manager, but only uses some of the possible event flags (namely 1-23, the omitted ones being 0 and 32-63. Flags 24-31 are reserved for VMS).

The following brief program illustrates the usage of MTL routines for managing an event flag.

```
Program EF_DEMO;
Type efn = integer;
Function  MTL_GET_EF : efn; extern;
Procedure MTL_FREE_EF(%immed ef : efn); extern;
Procedure MTL_WAIT_EF(%immed ef : efn); extern;
Procedure SYS$SETEF  (%immed ef : efn); extern;
var ef : efn;
begin
ef := MTL_GET_EF;        {Allocate an event flag}
sys$setef(ef);           {Set it}
MTL_WAIT_EF(ef);         {Wait for it}
MTL_FREE_EF(ef);         {Free it}
end.
```

There is a high cpu-time cost involved in having the scheduler check whether the event flags that tasks are waiting on are set, and VMS does not provide a mechanism for having an AST delivered when the flag becomes set. Rather than incur this expense the current implementation checks the event flag on the call to MTL_WAIT_EF and if it is set returns immediately. If it is not set then the task is entered into the WAITING_FOR_EVENT queue, and when no other tasks are capable of execution the whole process is paused until one of the event flags being waited on becomes set. At that time all tasks whose event flags have become set are returned to the COMPUTABLE queue.

5.2.7 A Problem with Asynchronous System Traps

The alternative method that VMS uses to notify a process of I/O completion (and for other purposes) is the AST (see section 3.2.3). VMS provides a System Service for disabling AST's, and this is typically used to provide a critical section. For example if both the main algorithm and an AST-driven routine (for example an interval-timer) access the same data base, by disabling AST's the main algorithm can temporarily gain exclusive access to the data base.

This presents a (minor?) problem with multi-tasking, because a task that wishes to block its own AST's also blocks all those for the other tasks. Furthermore it is quite likely that the AST will get delivered during the time when another task, other than the one that requested it, is current.

It is unclear without experience whether either of these difficulties will give trouble in practice. It was judged that the solution for this problem was too (1) complex, (2) cpu-expensive, and (3) VMS specific, to be fully explored in this thesis. It would appear that any solution would require modifications to VMS itself.

5.2.8 Summary

An implementation of message passing that is almost as efficient as proceduring calling was implemented under VAX/VMS version 2. Minor enhancements to both VMS, the MTL compiler, and the MTL runtime system would completely alleviate the added over-head of procedure calling, while the generation of inline code for NEW and DISPOSE would make the costs of message passing as efficient as procedure calling.

Increased Hardware and Software support for heap management and for multi-tasking inside a single address space (in particular instructions for loading and saving the complete register set) would eliminate completely any differences in efficiency between implementations of MTL-like languages and those of conventional block structured languages.

The interfacing of the multi-tasking with the asynchronous system services presents the greatest software difficulty in the implementation, and some of the problems were not solvable without modifications to the VMS operating system.

Chapter 6
EDITING AS A MESSAGE PASSING AND MULTI-TASKING PROBLEM

6.1 The Application of Concurrency to Editing

The use of a message passing and multi-tasking concepts in the design and implemention of an interactive text editor was found to yield new insights into the nature of the editing process as well as giving a deeper awareness of strengths and weakness of the particular implementation language (MTL).

Text editors stress both the character-manipulation and the sequence control aspects of a language to the limit. A powerful editor is itself an implementation of a programming language with the programs being interpreted in some form to manipulate the text and other aspects of the user's environment. Inside the editor boundary conditions are rife: lines become too long, heaps overflow, searches fail to find their target. The user can often request the abort of the current command. Yet in response to any of these occurrences the situation must be neatly tidied up and control returned to the user. The implementation of these 'neat tidy-ups' is a difficult problem in an editor written in a conventional language and presents new and interesting problems in a concurrent implementation.

Using a language that supports concurrency to implement an editor has several potential advantages. If the rest of the operating system is written in a concurrent language the editor will be better able to interface with it. For example if the operating system defines I/O in terms of message passing then the implementation will be easier if the editor uses the same concept. This will enable the overlapping of the reading and writing of data files, both to the terminal and to disk files, to be overlapped with other operations.

This overlapping of the reading of a file and the execution of commands entered by a user has proven advantageous in practice. The University of Adelaide has a utility for interactively examining the contents of a disk file from a VDU. These files are frequently hundreds or thousands of blocks long yet the delay involved in reading of these files is hidden from the user by just such an overlap of disk I/O and the execution of commands supplied via the keyboard. The user is not aware of exactly how much of the file has been read because the first few lines appear as soon as they are available and commands entered from the keyboard are executed as soon as it is possible to do so.

A concurrent implementation language would also enable a multi-processor implementation, with a portion of the editing being done in the user's terminal or work-station, taking a considerable load of the main CPU. With the arrival of steadily cheaper personal computers and networking this may soon be a totally obsolete 'advantage'.

In most conventional computing systems the power of the editor is unavailable whenever the user is entering input to any other program, indeed in most editors this power can not even be used to correct the commands being entered. The only way to solve this problem is to make the editor the only interface between the user and the rest of the system. To enable the editor to completely control the VDU all terminal I/O would have to be directed through it, and this is mostly naturally viewed as a message passing system. The steady introduction of high-resolution visual display units also leads to the possibility of having more than one active Task, with Task using separate areas of the screen for their terminal I/O. The combined impact of these requirements on the editor is that it should be implemented in a language that supports message passing and concurrency.

It is natural to propagate the concurrency of the rest of the system into the user's editing environment. Text editing includes such activities as sorting, typesetting, and other reformatting of the layout of the characters. It also involves pre- and post- processing of the files (such as text formatting and control character interpretation, eg underlining), and the splitting and merging of streams of data. The paradigm of text editing that we propose supports all these activities in a general and flexible manner by exploiting message passing and concurrency at the user environment level.

A primitive version of such an editor was designed using message passing and multi-tasking but only partially implemented (in MTL). The deep integration of this editor into VMS was not possible but it was apparent that a more sophisticated version would be a powerful programmer's work-bench with such tools as sorting and other reformatting algorithms, an editor, a compiler, etc., all available as Tasks which could be instantiated and connected as desired. It was also apparent that MTL would be capable of fully implementing this work-bench under VAX/VMS.

6.2 The Basic User Environment

The user's environment is a collection of Tasks which process input and generate output. Some of these Tasks are Scenes, which are special Tasks designed to edit text. Other Tasks could be any user program, or system utility, such as a Clock or a Mail system.

Each Scene contains a two-dimensional array of characters (which may also be regarded as a list of lines), and has input and output paths. These paths are used to connect the Scene to other Scenes, to disk files, and also to any other current Task. The paths are buffers for messages of lines of text. Any message arriving in the primary input path is automatically appended to the end of the rest of the text in the Scene, messages arriving on other paths have to be explicitly read and dealt with. Scenes provide the multiple editing environments needed in a powerful editor. They can be purely passive, storing text until it is required later, or they can be linked together as a series of programmable filters performing various operations on the text stream, such as formatting it.

Any Scene can be watched on the user's VDU, and several Scenes may be visible at once, thus allowing the user to monitor the behaviour of any group of Scenes at any time. This facility is essential, not only for debugging, but also for monitoring background activities such as Tasks that run for long periods of time.

Each Scene executes editing commands that it receives via its command path (note that these commands are actually programs in their own right). This execution is asynchronous, with each Scene executing its current program then accepting the next from its command path for execution. By connecting the output path of one Scene with the input path of another the application of reformatting algorithms to data can be done in parallel as stages in a pipeline.

A typical use of this facility would be three Scenes set up as a pre-processor, a Scene under the moment-by-moment control of the user, and a post-processor. If this configuration was being used to edit a Pascal program the post-processor could be directing its output to a disk file, and sending a duplicate of it directly to a Task executing the Pascal compiler. A listing of the errors detected by the compiler would be directed to another visible Scene to aid the user in their correction.

6.2.1 File Sub-system

Because the Scenes are not heirachically organised it would not be wise to equivalence them to files. Instead a file can be attached to an input path of a Scene, in which case its contents are appended into the Scene, or it can be attached to an output path, in which case all outputs from the Scene are appended to the file.

As an implementation detail this would require the instantiation of a Task to perform the actual I/O and to send or receive the messages via the appropriate paths. However this is a consistent way of viewing all Tasks, such as compilers, the editor, etc., and thus is a unifying factor rather than an awkward implementation issue.

6.2.2 Scene Programs

Since Scenes are used to perform considerably more powerful functions than just interactive editing a more sophisticated editing language would be required than is typical for text editors. The full power of a language such as Pascal[ISO] or SNOBOL4[GR68] would be desirable, with mechanisms available for interfacing the language directly to a Scene.

The instances of the Scene-programs are self-contained rather than relying on a particular Scene to provide storage for their variables, etc., but they only execute in the context of a Scene. This enables an instance to move from one Scene to another, carrying with it the values of all variables etc., but not the Scene's attributes of text and paths (although it can take references to the paths to be used for later directing input to or from the Scene).

This ability to change Scenes allows programs to use networks of Scenes to perform complex roles. For example two Scenes could be set up side-by-side with a merging program shifting from one to the other outputing lines to a common destination in a particular order.

Another use of this ability would be a program that searches through all the Scenes for occurrences of a particular textual pattern. Because this program does not modify any of the Scenes attributes it can be used "on-the-fly" without regard to the state of the Scenes (although it may get blocked by a Scene that is executing a program, this would be a situation where a transmit-with-timeout feature would be desirable).

6.2.3 VDU Sub-system, the Human Interface

Although the user is capable of seeing many Scenes simultaneously via the screen, all the inputs from the keyboard are directed to a single selected Scene. The user can select a new Scene at any time as the one to which the input is to be directed.

The keystrokes entered by the user are compiled into programs and directed to the selected Scene's command buffer for it to execute. Notice that these are presented to the Scene as a series of programs to be executed the same as any other program that is sent along its command path. This means that there is no need to wait for the command to finish execution before selecting a different Scene and directing commands to it.

The normal mode of compiling the keystrokes is to assume the characters typed are to be overtyped (or inserted depending on a mode-switch) into the Scene and the appropriate programs are produced to do this. Control characters, such as TAB, and Escape sequences, and other single keystroke commands are also compiled into the appropriate programs.

However if a complex command is to be entered the current selection is remembered and the Scene "COMMAND" is selected. Editing carries on as before but with this new Scene being the focus of attention. When the command is fully entered a simple directive is entered that means "Compile the completed command, direct it to the remembered Scene for execution, and make that Scene the selected one again". In this way the full power of the Editor is available for correcting the complex commands being entered.

Because the remembered Scene is still visible while entering the complex command the user need not retain a mental image of the text that the command is to modify. This should considerably reduce the number of incorrect complex commands entered since the command may be very dependent on the exact details of the text.

Another Scene, also usually left visible, is called "MESSAGES". All messages generated by the editor to the user are simply directed to the primary input path of this Scene, and hence become visible to the user. On a fully integrated system all messages, including those from other users, would be treated in this manner.

This design of the user interface does not insist that there is only one VDU in use. By connecting several VDU's into the system several user's could be using it simultaneously. This would allow the editing equivalent of a conference, with the minutes being automatically maintained. Either every user would have the same Scene selected, or alternatively they would each have different Scene selected but all the Scenes visible.

The introduction of several people into the system would then lead to the need for controlling the degree of interaction between them, purely to stop mistakes from badly affecting other users. Such constraints could also protect parts of one's own area from accidental damage.

6.2.4 Subsuming "the system"

The Scene model of editing provides a consistent framework for user interaction, concurrent activities, and all standard utilities. All terminal input uses the full power of the editor, and any input or output can be directed to or from any other output or input respectively. The power of the standard utilities (such as SORT) is available in the editor, and these utilities are fully interfaced with each other.

The Scene model is thus a time-sharing operating system with a fully integrated user environment rather than as just an editor, with the traditional concept of an editor disappearing into a much more general concept. This desirable goal is fully realised by the general application of message passing and concurrency to design, thus demonstrating the power of these concepts.

6.3 Implementation Issues

Only the basis of the Scene editor was actually implemented due to lack of time, the full implementation would be an extremely large project. It had the creation of Scenes and the execution of simple Scene programs entered via the keyboard, as well as the multiple visible Scenes, but neither the ability to execute Tasks other than Scenes nor the multi-terminal support.

6.3.1 Interactive I/O management

One serious problem with this design of an editor on a multi-user system is the large amount of cpu overhead associated with each keystroke. This problem can largely be controlled, as was done in the Ludwig Screen Editor, by making the Scene that the user is currently interacting with aware of its involvement with the user. When the user is overtyping text the truly asynchronous version would generate a new program for each character typed and place it on the selected Scene's command path. In practise it is much cheaper to have the Scene to accept the text from the terminal, and use the VMS operating system to do the echoing of the characters.

By a similar mechanism other frequently repeated keystrokes could be handled cheaply. Statistics gathered for Ludwig give the following distribution of commands, after overtyping has been ignored.

CARRIAGE RETURN	24%
CURSOR UP	14%
CURSOR LEFT	13%
CURSOR RIGHT	13%
CURSOR DOWN	12%

Total	76%

These figures are badly biased because repeated usage of any of these commands is only counted as one. For example if the user types the LINEFEED three times, only 1 would be added to the count of CURSOR DOWN, so we do not claim these figures reflect their true proportion of usage, but they do show quite clearly the gains to be made by optimizing the behaviour of a select subset of the commands.

6.3.2 Human Limitations

Humans may find the asynchrony of such an editor confusing. Unfortunately the implementation was not complete enough to fully investigate this but several problems did appear. The most severe problem encountered was associating the error messages with the event that caused them. Messages are generated by syntax errors, as well as other execution time failures, and a clear mechanism for linking the error with the cause is required. This problem is exactly the same one as with the debugging of concurrent algorithms in general.

The other problem, which occurs with conventional screen editors but is exaggerated by concurrent editing, is the need to view large quantities of text simultaneously. The terminal the editor was implemented on was only 24 lines high by 80 characters wide. It would be preferable to have a terminal that is 132 characters wide by at least 70 lines high to allow the display of adequately many Scenes simultaneously. Such terminals are now becoming available.

6.3.3 Termination

Many editors provide a facility for building and executing complex instructions from simple ones, and most provide an ABORT mechanism whereby, by typing a key, the editor aborts the execution of the command and returns control to the user.

Inside the Scene editor, with its many asynchronous activities, it is not obvious which Scene(s) the user wants aborted. However since the behaviour of an ABORT request can only be well-defined at those points where a program is interacting with the user in a dialogue, the ABORT statement can be construed as aborting the selected Scene's program.

The other point where ABORT is used is for run-away programs. In this case it is reasonable that the user should specify which Scene or which program (since programs can move between Scenes) is to be aborted.

Because the activities truly are asynchronous there is no problem in communicating with the user, unless due to the lack of pre-emptive scheduling in MTL. This lack can be avoided by automatically rescheduling at the end of each (n-th) step in the execution of a Scene-program.

6.3.4 Correctness

The correctness of any powerful editor is difficult to establish because of the wide variety of boundary conditions. A successful approach seems to be to have a list of assertions about the editor's data structure and to regard this as a list of transformation-invariants, ie. assertions that must be true before and after various operations are performed on the data structure. By writing a routine that evaluates these assertions, and calling this routine from various points in the code, errors can be very quickly located. The extensive usage of subrange checking in the Pascal implementation of Ludwig demonstrates this point very well.

6.3.5 Error Management

Because there are so many boundary conditions in an editor, it is a truism that some algorithmic errors will not manifest themself even after fairly exhaustive tests. When these errors finally do occur it is undesirable behaviour for the editor to crash, losing all the user's current session's work and the environment that has been established. Ludwig solves this by providing a VMS condition handler which merely forces the current routine to return to its caller returning a value of FALSE or NIL. By being aware of this possibility throughout the whole source code, the implementors were able to design their code to minimise the impact of the error. Errors that could not be contained cause Ludwig to abort and to write out the contents of the various Frames into their respective output files before the editor is completely exited. These measures have proven adequate, and it is extremely rare for the editor to abort, even after a fairly severe internal corruption.

Unfortunately condition handling is not implementable in MTL (see section 5.2.2) because of restrictions caused by VAX/VMS version 2.

It would appear that in an asynchronous environment the first step may be possible, but the second (the winding out of the various frames) may not be adequate because many Scenes will contain useful text, but will not have output files. A better approach would be to write out all the various Scenes to a dump file, and let the user pick his way through the ruins recovering those portions he needs.

Other approaches such as journalling (writing each command to a journal file as it is executed) and checkpointing (writing the current state to a file at regular intervals) may be tried. Ludwig experience indicates that journalling is unable to cope with editors that have ABORT facilities and the ability to build and execute programs, because the combination of the two implies that journalling must be performed at a fairly low level in the editor. This implies every command executed must be journalled which causes the journal to grow extremely large.

Checkpointing appears to be adequate. Since the Scene editor has a more complex user appearance than Ludwig, and since it is envisaged that the user would rarely exit from the editor, it would be a wise precaution to checkpoint regularly. This would protect the user from himself, flaws in the editor, and hardware failures in the computing system.

The implementation of checkpointing in a completely asynchronous editor presents several difficulties. In MTL the contents of the message buffers are not accessible to any Task without reading them, a situation directly analogous to messages currently in transmission on a network. This makes it impossible to checkpoint all possible states of the Scene editor. Any code to flush these buffers would be spread throughout the editor in a clumsy and awkward to maintain manner. The use of a central registry of all buffers, Scenes etc. may make it possible to keep track of all the data structures in a manner that allows a static picture to be taken of them, but the asynchronous interface between the editor and VMS could not be treated in this manner. It would be necessary to wait for this interface to enter a quiescent state.

6.4 Language Demands

The implementation of an editor places severe demands on the character manipulation facilities of a language, an area where Pascal, hence MTL, is particularly poor. It also stresses the I/O facilities, particularly those related to terminals, because of peculiar needs. Input is often "character-by-character : not echoed", or alternatively "up to n characters but stop before the first occurrence of a character in this set". Very often these requirements can not be met by the language, and there is a need to call routines written in some other system-programming-oriented language. Sophisticated editors tend to be large programs and the need for modular compilation is a natural consequence.

Editing does not make many unusual demands on the message passing or concurrency aspects of the language. If the user can see the activities of several pseudo-concurrent tasks the behaviour of the scheduler may become quite apparent. "Fair" scheduling is a difficult thing to define, but it will be very obvious to the user if a particular (visible) task is being starved or overfeed resources.

The source code of the Scene editor does not clearly indicate the expected flow of messages. Instead the code for establishing the various links and transmitting the messages tends to be scattered throughout the text of the various modules. This partly invalidates the self-documenting nature of languages such as Pascal. The work of Barter[BA82] on communication protocols is an attempt to clarify in the source code this aspect of the algorithm.

With multiple tasks capable of accessing the same heap item (which may or may not be a desirable feature of MTL) leads to problems in deallocation of these items. A solution to this may be some form of garbage collection, provided it does not have a severe impact on cpu utilisation. Some architectures, such as the iAPX-432[PO82], have considerable hardware support for this facility, and some languages, such as Ada and Lisp, seem to demand it. With large physical and virtual address spaces a steady build up of garbage in the address space may not be important. A reference-count mechanism may be an adequate comprise, as it only presents problems when there are loops in the data structure, which is fairly unusual.

6.5 Summary

A concurrent view of editing has been extended to a system that encompasses all the interaction between the human and the machine(s). This extension allows complete editing of all visible data, and the easy manipulation of data through a variety of filtering tasks, such as sorting algorithms and compilers.

Such systems will provide a more flexible, consistent, and friendly environment for users. Message passing is a viable and efficient basis for their implementation.

Chapter 7
RESULTS AND CONCLUSIONS

7.1 Language Implementation

7.1.1 The Run-time Heap

The performance of the run-time heap is the single most crucial feature in an efficient implementation of MTL-like languages. A simple design, as described in section 5.2.3, performs extremely well and imposes almost minimal overhead. The use of inline code rather than procedure calls for the allocation and deallocation of heap items is possible, with between one and three instructions required to allocate the object (depending on whether the hardware generates a trap when an attempt is made to remove an item from an empty list or if it sets the condition codes), and one or two instructions required to place the object back on the appropriate free list.

If the heap is managed in this fashion the cpu-time involved in the allocation and deallocation of storage can be ignored when comparing the performance of heap based and stack based systems, since it is of the same order of magnitude as stack maintenance.

7.1.2 Procedures and Functions

The design of the routine calling mechanism will depend on the operating system and machine architecture that the implementation is aimed at. Even in the worst case, such as the current MTL implementation, when the activation records have to be stored in a heap, the calls are only about nine times slower than the stack based mechanism, as is shown in APPENDIX C. Furthermore there will be fewer calls because of the use of message passing.

An optimising compiler that does global flow analysis is able to significantly reduce the number of calls that require the use of the heap PROVIDED that pre-emptive scheduling is not implemented, and use of specialised instructions can make procedure calls that use a heap to store the activation records just as efficient as those that use a stack.

Thus the apparent overheads in the MTL implementation of procedure calls is not a consequence of the multi-tasking per se, but rather a consequence of some limitations in the compiler and in the VAX-11 architecture.

7.1.3 The Performance of Message Passing

The VAX-11 architecture is capable of supporting message passing at speeds comparable to the CALL instructions but not at rates comparable to the JSB (jump-to-subroutine, which pushes the current program counter onto the stack and then jumps to the subroutine) instructions. This is to be expected with any conventional architecture because the JSB instruction represents the minimum amount of information that need be preserved. Hardware support for message passing would change this situation. Such hardware would completely implement the MTL buffer as a single instruction. Even faster JSB-style instructions where the return address is placed into a register rather than memory will probably remain the fastest way of calling trivial routines (such as random number generators). Of course nothing will beat in-line code for sheer speed!

Message passing has the following potential advantages, some of which are only realised when message passing is implemented with hardware support...

(1) No processor context (such as registers, etc) need be saved. Procedure calling using the CALL instructions involves writing between 20 and 60 bytes of memory from registers during the CALL, and reading this back into the registers for the RETURN.

(2) Parameters to a procedure are passed by two basic mechanisms, either pass-by-reference or pass-by-local-copy. In the pass-by-local-copy mechanism, such as Pascal requires, the calling routine provides a reference to the value of the parameter and the called routine copies this value into a local variable. If this value is passed on to another (nested) routine it is copied again. This repeated copying of values can be inefficient if the value requires a large amount of storage. Message passing is basically the passing-by-reference of a series of heap items and so it is efficient to pass around large structures as messages (the Sender los~~s~~es access to the message and the Receiver gains it).

The use of broadcasted messages can impact on this performance by requiring copies to be taken, but it appears that this is not the usual usage of messages. Where efficient broadcasting of large structures is required it may be possible to broadcast a pointer to a heap item, and by use of a reference count mechanism have this item destroyed when the last access is released. All the Receivers would have to be aware that they were sharing this heap item with other tasks.

Implementations of message passing on distributed systems are not able to get the full advantage of pass-by-reference (unless the systems have a shared memory) but this is another advantage. Since the sending task does not have access to the message after the call to the SEND procedure it does not need to be blocked until the message has been transmitted on the communication medium. If the communication medium was slow and if the sender was able to access the message after the SEND it would be necessary to either delay the sender or copy the message before transmission.

(3) In a three-phase operation (eg. read/massage/write) there would be three procedure calls but only two message passing operations to transfer information through the phases.

(4) Tasks tend to have loop-until-finished style behaviours. These loops cause an increase in code-locality and perhaps in memory access locality effects, which should lead to improved cache and paging performance during the execution of larger algorithms.

7.1.4 Context Switching

Context switching requires the saving of the current context, usually just the registers, in a control block and selecting and loading the context of the next task. As such its performance is very similar to an expensive procedure call, where all registers must be saved and restored, but only a few small parameters are passed. The use of asynchronous message passing significantly reduces the number of context switches required, but the size of the buffers used to implement the asynchrony need not exceed 10 items to get most of the enhanced performance.

Context switching in MTL is greatly simplified and sped up by the fact that all the tasks run inside the same VAX/VMS process. If this were not so context switching would also require the changing of the virtual-memory to physical-memory address translation tables and the invalidation of both the hardware memory caches and address translation caches.

7.1.5 I/O

I/O causes insuperable difficulties unless the operating system supports asynchronous I/O. Many current runtime support systems, such as that for VAX-11 Pascal 1.3 do not support asynchronous I/O even though VMS does. This is unfortunate because it means that such runtime systems can not be used without sacrificing the overlap of I/O and computation. In practice only terminal I/O (and possibly communication over slow media) cause problems, because disk I/O is overlapped by VMS anyway.

7.1.6 Shared Variables

The only efficient implementation of any language which supports shared variables is going to require the tasks to have a (partially) shared address space. This need not prohibit a desirable (but under VMS not easily attainable) goal of having each task also have a private address space for such things as stacks. MTL requires that all the activation records textually surrounding the task be available to it, precluding such an implementation. An alternative language design may require shared objects to be in the heap which could then be maintained in the shared address space while the activation records are maintained in the private address space.

7.2 Operating System Support

7.2.1 Heap Management

Heap management does not require much operating system support, other than the careful specification of a standard for all language implementations to follow. The heap should be specified so that simple in-line code can do all allocations and deallocations, with operating system support only required when an object of the required size is not available.

7.2.2 Process Creation and Deletion

Process creation in VAX/VMS could be made considerably faster if an area of the creator's memory could be designated as shared with the created process rather than specifying a disk file for the created process to run. This probably the most radical change to VAX/VMS that was observed during the researching of this thesis.

It would appear, however, that the large overheads in maintaining in separate processes for separate tasks (separate address space, accounting information, quotas, etc.) make such a use extremely expensive. A characteristic of programs in message passing systems is that the tasks are typically small, often only a few hundred machine instructions long, and it does not seem sensible to waste a complete process for such a small object.

7.2.3 Multiple Address Spaces

If it can be obtained simply and efficiently in the hardware it may be desireable for a process to be able to switch between several address spaces very quickly, allowing such implementation possibilities as 7.1.6.

7.2.4 I/O and other Asynchronous System Services

A minor annoyance in the design of MTL is the inability to tell VMS to deliver an AST whenever an event flag is set. Of course all operating systems should support asynchronous I/O on all devices for which it is sensible.

This leads to a general principle for designing operating systems to allow the full implemention of multi-tasking inside a process :

All events that involve waiting should be designed in such a way that other, unrelated, activities can be done during the delay and that notification is delivered to the runtime system of the occurrence of the awaited event.

7.2.5 Impact on Other Utilities

All utilities should be capable of accepting input and generating output as messages so that they can be joined in arbitrary ways. It should be possible, for instance, to invoke any compiler or sorting algorithm from any other utility. In particular it should be possible to edit any text entered via the keyboard, or any other visible text, using the editor. It should also be possible to apply any compiler or other utility to any text being currently edited without having to first write the text onto a disk file.

This can be easily done be separating the aspects of a utility concerned with obtaining input and returning output from those aspects that process this data. The I/O portion can then be either replaced with a different but functionally equivalent (from the utility's point of view) module that is capable of either being replaced, or of getting data from a variety of sources in a variety of ways, and likewise returning output in the form that is most convenient in the particular instance.

7.3 Hardware Support

7.3.1 Heap Management

Special hardware support for heap management is required in the form of instructions for manipulating queues without the need for software interlocks. This hardware support is available in the VAX architecture as the INSQUE and REMQUE instructions.

The absence of preemptive scheduling does not completely remove the need for these instructions because they are also essential if the language allows asynchronous interrupts, such as the AST mechanism in VAX/VMS (section 3.2.3). MTL falls into this category.

7.3.2 Procedure and Function Calling

The use of messages is not going to replace procedures and functions since they are complementary rather than opposing methods of structuring algorithms. However the introduction of the multi-tasking required does cause some difficulties with the allocation of activation records.

There are two basic possibilities. If the tasks each have their own address spaces (either as a private portion of a single address space, or as separate address spaces) there may be room for each to have its own stack. There is sufficient room in P1 space under VAX/VMS, as indicated in section 3.2.1.1, for the tasks to each be allocated an area there. Alternatively the activation records may be allocated from the heap, as is the case with the Mesa processor[JO82].

If the activation records are to be allocated from the heap several choices are possible. The location of the Queue_Headers for each of the queues may be known to the hardware with the first few bytes of the called routine indicating the size of the required activation record, or the caller could allocate the activation record and then specify it as part of the call, or a very simple calling mechanism could be used (such as the JSB instruction) with the called routine allocating and constructing the activation record.

This last choice would require no changes to the VAX-11 hardware but would require changes to the VAX/VMS calling standard which is probably not acceptable.

The simplest modification would be the addition of a CALLH instruction to the architecture which was only useable on CALLH'able routines. This CALLH instruction would get its activation record from the heap, using a size specified in the 4 bytes preceding the routine's entry mask (see the VAX-11 Architecture Handbook[DEC79-A] for details on VAX-11 procedure calling). In order to do this there would need to be a heap management system available that the hardware could easily access.

7.3.3 Context Switching between Tasks

Tasks in a single address space only require the registers to be saved and restored to do a context switch. The VAX architecture has instructions for pushing and popping registers onto/off the stack, but none for unloading them or loading them from memory. This restriction seems unnecessary and annoying. A single instruction, that takes as its arguments two areas in memory, for unloading all the registers and then loading them from the other area, would be useful. Much more useful than the HALT instruction that takes up an opcode on most architectures!

7.4 Language Design

7.4.1 Message Passing Mechanism

The choice of transmission-by-copy versus transmission-of-original is one that has often been made on grounds of "its being sent over a wire, so lets keep the original". For implementations of message passing on a single processor, or a group of processors with shared memory, transmitting a pointer to the original is more efficient. It would appear that the retained message is not usually wanted by the Sender anyway, and if it was the Sender can take an explicit copy.

Message passing has several advantages over more conventional methods of expressing algorithms.

(1) Tasks tend to be written as loop-until-finished constructs with local variables maintaining the context. Procedures on the other hand need global variables or parameters to maintain the context from one call to the next.

(2) The initialization of this context tends to be near its usage in the task, but in a separate procedure textually apart from its usage in the case of procedures.

(3) It is an extension to a language that can also have procedure calls, and as such increases the power of the language.

7.4.2 Asynchronous Versus Synchronous Communication

The cpu-time involved in context switching between tasks is larger than the cpu-time involved in a single message passing operation. Context switching requires the saving and loading of considerable quantities of information (the machine's registers) and furthermore the operation must be done four times for a single message passing operation when a bounded buffer is required! It is used to transfer control from the sender to the Bounded_Buffer, back again, and then later on to transfer control from the receiver to the bounded_buffer, and back again.

Appendix B shows some timing comparisons demonstrating this conclusion. Since the MTL implementation of message passing costs about half that of a context switch an overall performance of about 1:8 between the run-time system implementation of a bounded buffer and a task implementation of a Bounded_Buffer was observed.

This would indicate it is cheaper in terms of cpu-time to incorporate asynchronous communication into the run-time system rather than have synchronous communication and user provided buffers. In MTL the asynchronous communication algorithms force synchronous behaviour anyway when the buffer sizes are specified as 0.

7.4.3 Communication Paths

COSPOL[RO81] and Mercury[MA81] both use a communication path distinguished solely by the type of the message and the identity of the destination. This makes for difficulties in modelling such things as multi-server queues and multi-queue objects (for example the intersection of four roads). Curiously enough it also goes against such real-life models as a single letterbox serving all members of a family. Since one of the virtues often claimed for message passing is that it models the real world it would appear that buffers are a better choice.

However buffers are also less structured, the flow of control and messages far from obvious, and there is an extra layer of notational clumsiness involved. Language designers should look into methods of collecting such information in one place in the program source text and look for less clumsy methods of specifying the communication paths and activities.

7.4.4 The Effect of a Mono-Processor Implementation on the Language

MTL programs explicitly specify points at which rescheduling may occur. This is simple and provides the programmer with critical regions without the use of any other synchronisation primitives, but was viable only because MTL was aimed at a mono-processor implementation. In particular MTL does not address the issue of the fairness of CPU scheduling algorithm, as each Task holds the CPU until it is prepared to let it go.

This shows that the design of a multi-tasking languages is affected by the intended implementation, and that mono-processor languages and implementations are simpler than those aimed at systems with multiple processors.

7.5 Overall Conclusions

I was surprised at the adequacy of the VAX architecture and the VMS operating system for implementing MTL. In the areas where there was obvious need for improvements these would not be difficult to make, and were compatible with the rest of the system.

Procedures have been the main structuring tools in use for some time. As a consequence of this we tend to regard as 'natural', solutions to problems that use it. There should be continuing efforts to apply message passing in solutions to problems that have been traditionally been solved by procedural techniques (sorting, compiling, mathematical algorithms, etc.). These efforts will yield different insights into the nature of the individual problems and may lead to neater solutions. They will also teach us more about the ways of using message passing. The design of the Scene editor was sparked off purely by the desire to apply concurrency to the problem of implementing an editor, but the editor rapidly grew to encompass the user's entire environment.

The continuing development of the Scene environment is the most promising research project to come out of this thesis. It not only shows great potential for integrating all aspects of the user's computing environment but also demonstrates the unifying power of the concepts of message passing and multi-tasking.

Appendix A. MTL task creation rates V. VMS sub-process creation rates

Both MTL and VMS dynamically create concurrent objects (Tasks and Processes respectively) but they have very different performance figures as the following experiment shows.

MTL

This MTL program creates 10000 tasks, each of which sends a message to the creator and dies. Elapsed times and CPU times are printed out every 1000 tasks.

```
Program Creation;
Procedure Lib$Init_Timer; extern;
Procedure Lib$Show_Timer; extern;
Type termination = message end;
var  trm_mbx : buffer of termination;
Task Son;
  var t : termination;
  begin
  new(t);
  send t via trm_mbx;
  end;
var i : integer; tsk : task_id; t : termination;
Begin
Lib$Init_Timer;
buffer_init(trm_mbx,0);
for i := 9999 downto 0 do
  begin
  create tsk := son;
  receive t via trm_mbx;
  dispose(t);
  if (i mod 1000) = 0 then Lib$Show_Timer;
  end;
Lib$Show_Timer;
end.
```

The LIB$SHOW_TIMER routine displays the elapsed time and cpu time, in the form hours:minutes:seconds.centiseconds, since the call to LIB$INIT_TIMER. The first outputs from LIB$SHOW_TIMER were

```
ELAPSED_TIME = 00:00:00.75,  CPU_TIME = 0:00:00.74
ELAPSED_TIME = 00:00:01.53,  CPU_TIME = 0:00:01.51
ELAPSED_TIME = 00:00:02.27,  CPU_TIME = 0:00:02.25
```

which gives 0.78 seconds elapsed time and 0.77 seconds cpu time to create the corresponding 1000 tasks. The cpu time does include that used by the tasks. Hence an approximate rate for MTL is 1300 tasks per second cpu time, and the cpu time is about the same as the elapsed time.

VMS

The following two BLISS-32 programs perform a similar behaviour under VMS, and were executed on an otherwise idle system. Note that the calls to LIB$SHOW_TIMER are now done every 100 processes (rather than every 1000 tasks as above) because of the much slower performance.

```
Module Process(Main=main,debug) =
Begin
...                           ! Declarations omitted
Routine Main =
  begin
  $crembx(chan = chan);  $getchn(chan =.chan,pribuf=ddsc);
  lib$init_timer();
  decr i from 9999 to 0 do
    begin
    $creprc(image  = idsc  ! Create the sub process
           ...
           ,mbxunt = .dblk[dib$w_unit]
           );
    $qiow  (chan   = .chan ! Wait for the system to send us
           ,func   = io$_readvblk ! a message saying the subprc
           ...                    ! has died.
           );
    if (.i mod 100) eql 0 then Lib$Show_Timer();
    end
  end;
End
Eludom

Module SubPrc(Main=main,debug) =
Begin
Routine Main = 1; ! Simply die (by returning from
End               ! the main routine).
Eludom
```

The first few calls to Lib$Show_Timer yield these results ...

```
ELAPSED_TIME = 00:00:30.62,  CPU_TIME = 0:00:00.34
ELAPSED_TIME = 00:01:01.86,  CPU_TIME = 0:00:00.65
ELAPSED_TIME = 00:01:32.60,  CPU_TIME = 0:00:00.98
```

which, by subtraction, gives 31 seconds elapsed time and 0.31 seconds cpu time to create the 100 sub-processes between the two samples. THE CPU TIME DOES NOT INCLUDE THAT REQUIRED FOR THE SUB-PROCESSES. Hence an approximate rate for VMS is 3 sub-processes per second elapsed time, but no estimate can be made of the cpu costs because these are largely hidden in the subprocess, other than to comment that it too would be large.

CONCLUSIONS

MTL creates tasks at about 1300 tasks per second (cpu and elapsed time) whereas VMS creates them at about 3 tasks per second (elapsed time).

The huge difference in elapsed times is largely due to the VMS requirement that a sub-process must execute an image off disk, where-as for MTL the code for a task is memory resident as part of the current image.

MTL also uses vastly less cpu time because the creation of the VMS process and the loading of the image from disk are extremely complex and long operations.

Appendix B. User written Bounded Buffer cpu-times V. Runtime System

To illustrate the point about the cpu-time involved in user written bounded_buffers and those provided by the MTL runtime system the following two programs were compiled and run. The results were ...

```
runtime system   - CPU_TIME = 00:00:01.11 seconds
user written     - CPU_TIME = 00:00:09.47 seconds
```

```
Program Bounded_Buffer; { MTL Runtime system buffer }
Procedure Lib$Init_Timer; extern;
Procedure Lib$Show_Timer; extern;
const
  buf_siz =    10;
  msg_cnt = 10000;
type
  msg = message end;
var
  buf     : buffer of msg;

Task Tap;
  var i : integer; m : msg;
  begin
  for i := 1 to msg_cnt do
    begin
    new(m);
    send m via buf;
    end;
  end;

var i : integer; m : msg; tap_tsk : task_id;
begin
buffer_init(buf,buf_siz);
create tap_tsk := tap;
Lib$Init_Timer;
for i := 1 to msg_cnt do
  begin
  receive m via buf;
  dispose(m);
  end;
Lib$Show_Timer;
end.
```

```
Program Bounded_Buffer; { User written Bounded_Buffer }
 Procedure Lib$Init_Timer; extern;
 Procedure Lib$Show_Timer; extern;
 const buf_siz = 10; msg_cnt = 10000;
 type  msg = message end;
 var   inp,out : buffer of msg; bb_tsk  : task_id;

 Task BB;
   var buf : array [1..buf_siz] of msg;
       fst,lst : 1..buf_siz; siz : 0..buf_siz;
   begin siz := 0; fst := 1; lst := 1;
   repeat
     if (siz < buf_siz) and (buffer_writers(inp) > 0) then
       begin
       receive buf[fst] via inp;
       if fst < buf_siz then fst := fst+1 else fst := 1;
       siz := siz+1;
       end
     else
     if (siz > 0) and (buffer_readers(out) > 0) then
       begin
       send buf[lst] via out;
       if lst < buf_siz then lst := lst+1 else lst := 1;
       siz := siz-1;
       end
     else
       begin
       sleep;
       end;
   until false;
   end;

 Task Tap;
   var i : integer; m : msg;
   begin
   for i := 1 to msg_cnt do
     begin new(m); awaken(bb_tsk);
     send m via inp;
     end;
   end;

var i : integer; m : msg; tap_tsk : task_id;
begin
buffer_init(inp,0);
buffer_init(out,0);
create bb_tsk  := bb;
create tap_tsk := tap;
Lib$Init_Timer;
for i := 1 to msg_cnt do
  begin
  awaken(bb_tsk);
  receive m via out; dispose(m);
  end;
Lib$Show_Timer;
end.
```

Appendix C. MTL Procedure Calling Rates.

The following program times 100000 procedure calls. It was compiled and run using both the MTL compiler and the VAX-11 Pascal 1.3 compiler. The results were ...

	With the call.	Without the call.
MTL	26.9 cpu-seconds	0.24 cpu-seconds
VAX-11 Pascal 1.3	3.1 cpu-seconds	0.29 cpu-seconds

Clearly the current implementation of MTL performs this test very badly.

```
Program Calls;
var i,j : integer;
procedure lib$init_timer; extern;
procedure lib$show_timer; extern;
procedure dummy(var d1,d2:integer); begin end;
begin
lib$init_timer;
for i := 1 to 100000 do DUMMY(J,J);
lib$show_timer;
end.
```

Appendix D. <u>MTL and Pascal New/Dispose Rates</u>

This modified version of the program given in 5.2.3 was compiled and run using MTL and VAX-11 Pascal 1.3. It illustrates the performance difference between MTL and the conventional heap manager of VAX-11 Pascal 1.3, especially for pathologically bad disposal orderings. The time for the loops without the NEW/DISPOSE has been subtracted from the cpu-times. The results were for the two separate loops in the program ...

MTL	0.35 cpu-seconds	483 page faults.
	0.18 cpu-seconds	59 page faults.
VAX-11 Pascal 1.3	54.94 cpu-seconds	1052 page faults.
	0.67 cpu_seconds	129 page faults.

```
Program Heap(output);
type ptr = ^integer;
var i : integer;  a : array[1..10000] of ptr;
procedure lib$init_timer; extern;
procedure lib$show_timer; extern;
begin
for i := 1 to 10000 do new(a[i]);
lib$init_timer;
for i := 1 to 5000  do dispose(a[2*i-1]);
for i := 1 to 5000  do dispose(a[2*i]);
lib$show_timer;
for i := 1 to 10000 do new(a[i]);
lib$init_timer;
for i := 1 to 5000  do dispose(a[i]);
for i := 1 to 5000  do dispose(a[i+5000]);
lib$show_timer;
end.
```

Appendix E. MTL Context Switching Rates

This program switches context between a pair of tasks 100,000 times, with a cpu-time of 8.55 seconds, which yields a context switching rate of about 11,700 context switches per second. Curiously this is considerably faster than an MTL procedure call, and about half the speed of a VAX-11 Pascal 1.3 procedure call (appendix C).

```
Program Switch;
const k = 100000;
Procedure Lib$Init_Timer; extern;
Procedure Lib$Show_Timer; extern;
type msg = message end;
var  b : buffer of msg;
     i : integer;

Task s;
  var m1 : msg;
  begin
  while i > 0 do begin reschedule; i := i-1; end;
  new(m1); send m1 via b;
  end;

var t1,t2 : task_id; m : msg;

begin
buffer_init(b,2);
i := k;                  { initialize the counter }
create t1 := s;          { create the two tasks   }
create t2 := s;
lib$init_timer;          { start the timing       }
receive m via b;         { wait until both finish }
receive m via b;
lib$show_timer;          { show the timing        }
end.
```

Appendix F. MTL Message Transmission Rates

Message transmission times depend to a certain extent on the size of the buffers used. This phenomona is only observed between distinct tasks because it is caused by having to context switch whenever the buffer fills. The following program measures purely the time taken to pass the message, because there is no context switching required.

The performance figure was 100,000 messages in 8.28 seconds, or about 12,000 messages per second. However this includes time to NEW and DISPOSE the messages. When the NEW and DISPOSE were shifted out of the loop, the figures changed to 5.03 seconds, or about 20,000 messages per second. In practise performance is going to vary somewhere between these two figures depending on the number of messages passed compared to the number created.

```
Program Send_Receive;
const k = 100000;
Procedure Lib$Init_Timer; extern;
Procedure Lib$Show_Timer; extern;
type msg = message end;
var b : buffer of msg;  m : msg;  i : integer;
begin
buffer_init(b,1);
lib$init_timer;
for i := 1 to k do
  begin
  new(m);
  send m via b;
  receive m via b;
  dispose(m);
  end;
lib$show_timer;
end.
```

Appendix G. The ODDWORDS Problem

The following two programs implement the primary portion of Dijkstra's odd-word-reversal problem. They run in approximately equal time of 0.2 cpu- seconds, when compiled with MTL and VAX-11 Pascal 1.3 respectively. If the messages are not pre-allocated for the message passing version it takes about 0.3 cpu seconds. These results are more thoroughly discussed in 5.2.5.

MTL

```
Program ODDWORDS;
const message_count = 1200;
type  word     = array [1..5] of char;
      word_msg = message wd : word; len : integer; end;
      word_buf = buffer of word_msg;
var   msgs     : array[1..message_count] of word_msg;
      item_buf : word_buf;
      revs_buf : word_buf;

TASK ITEM;
  var  w : word_msg;  i : integer;
  begin
  for i := 1 to message_count do
    begin
    W := MSGS[I];                        { Get pre-allocated message }
    with w^ do begin len := 5; wd := 'ABCDE'; end;
    send w via item_buf;
    end;
  new(w); with w^ do begin len := 1; wd  := '.    '; end;
  send w via item_buf;
  end;

TASK REVERSE;
  var w : word_msg;  flip : boolean;
  begin
  flip := false;
  repeat receive w via item_buf;
    if w^.wd[1] = '.' then exitloop;
    if flip then {...reverse w^.wd...}
    flip := not flip;
    send w via revs_buf;
  until false;
  send w via revs_buf;
  end;

TASK METI;
  var w : word_msg;
  begin
  repeat receive w via revs_buf;
    if w^.wd[1] = '.' then exitloop;
  until false;
  LIB$SHOW_TIMER;
  end;
```

{Continued on next page...}

```
var t : task_id; i : integer;
 BEGIN
 for i := 1 to message_count do   { pre-allocate messages }
   new(msgs[i]);
 LIB$INIT_TIMER;
 buffer_init(item_buf,100); create t := item;
 buffer_init(revs_buf,100); create t := reverse;
                            create t := meti;
 END.
```

VAX-11 Pascal 1.3

```
Program ODDWORDS(input,output);
const  message_count = 1200;
type   word      = array [1..5] of char;
       word_msg = record wd : word; len : integer; end;
var    w1,w2      : word_msg;
       flip       : boolean;
       rnc_count : integer;

PROCEDURE ITEM(VAR W1: WORD_MSG);
  begin
  with w1 do
    begin
    if rnc_count <> message_count then
      begin len := 5; wd  := 'ABCDE'; end
    else
      begin len := 1; wd[1] := '.';   end;
    end;
  rnc_count := rnc_count+1;
  end;

PROCEDURE REVERSE(W1 : WORD_MSG; VAR W2 : WORD_MSG);
  label 99;
  var i : integer; len : integer;
  begin
  if flip then {...reverse w1 into w2 } else { w2 := w1 } ;
  flip := not flip;
  99:
  end;

PROCEDURE METI(W : WORD_MSG);
  begin
  end;

begin
LIB$INIT_TIMER;
rnc_count := 0;
flip      := false;
repeat
  item(w1); reverse(w1,w2); meti(w2);
until w2.wd[1] = '.';
LIB$SHOW_TIMER;
end.
```

REFERENCES

[BA82] C.J.Barter,
"Communicating Policy for Composite Processes" Technical Report 82-7, Dept. of Computing Science, University of Adelaide, Australia, 1982

[BI73] G.M.Birtwistle, O.-J.Dahl, B.Myhrhaug, and K.Nygaard,
SIMULA BEGIN, (Studentlitteratur, Sweden.) AUERBACH Publishers Inc. Philadelphia, Pa., 1973

[BU71] The Burroughs Corp.,
B6700 Extended Algol Reference Manual, The Burroughs Corp., Detroit, Michigan, Form No. 5000128, 1971

[CH79] D.R.Cheriton et al.,
"Thoth, a Portable Real-time Operating System" Comm. A.C.M. 22(2) 105-115(1979)

[CH81] Y.J.Choi,
"Process Interaction in Modular Processes", Technical Report 81-6, Dept. of Computing Science, University of Adelaide, Australia, 1981

[DEC79] Digital Equipment Corp.,
VAX-11 PASCAL Language Reference Manual, Software Distribution Center, Digital Equipment Corp., Maynard, Massachusetts, Order No. AA-H484A-TE, 1979

[DEC79-A] Digital Equipment Corp.,
VAX11 Architecture Handbook, Sales Support Literature Group, Digital Equipment Corp., Maynard, Massachusetts, 1979

[DEC80-SS] Digital Equipment Corp.,
VAX/VMS System Services Reference Manual, Software Distribution Center, Digital Equipment Corp., Maynard, Massachusetts, Order No. AA-D018B-TE, 1980

[DEC80-SD] Digital Equipment Corp.,
VAX/VMS Symbolic Debugger Reference Manual, Software Distribution Center, Digital Equipment Corp., Maynard, Massachusetts, Order No. AA-D026B-TE, 1980

[DEC80-IO] Digital Equipment Corp.,
VAX/VMS I/O User's Guide, Software Distribution Center, Digital Equipment Corp., Maynard, Massachusetts, Order No. AA-D026B-TE, 1980

[DI68] E.W.Dijkstra,
"Co-operating Sequential Processes", Page 43-112, In Programming Languages, (ed. F. Genuys), Academic Press, London and New York, 1968

[DI72] E.W.Dijkstra,
"Notes on Structured Programming", Page 1-82, In Structured Programming, A.P.I.C. Studies in Data Processing, No. 8, O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare Academic Press, London, 1972

[DI75] E.W.Dijkstra,
"Guarded Commands, Nondeterminacy and Formal Derivation of Programs", Comm. A.C.M. 18(8) 453-457(1975)

[DU82] B.R.Dunman, S.R.Schach, and P.T.Wood,
"A Mainframe Implementation of Concurrent Pascal" Software - Practice and Experience Vol 12(No. 1) 85-89(1982)

[FR79] N.Francez, C.A.R.Hoare, D.J.Lehmann, W.P.De Roever,
"Semantics of Nondeterminism, Concurrency, and Communication" Journal of Computer and System Sciences 19, 290-308(1979)

[JO82] R.Johnsson, and J.D.Wick,
"An Overview of the Mesa Processor Architecture" ACM Proceedings, Symposium on Architectural Support for Programming Languages and Operating Systems. Palo Alto, March 1982

[GR68] R.E.Griswold,J.F.Poage, and I.P.Polonsky,
The SNOBOL4 Programming Language, Prentice-Hall Inc., Englewood Cliffs, New Jersey 1968

[HA72] A.N.Habermann,
"Synchronization of Communicating Processes", Comm. A.C.M. 15(3) 171-176(1972)

[HO78] C.A.R.Hoare,
"Communicating Sequential Processes", Comm. A.C.M. 21(8) 666-677(1978)

[ISO] "Specification for Computer Programming Language Pascal"
Draft International Standard, DP 7185, 1981

[KR82] H.S.M.Kruijer,
"A Multi-user Operating System for Transaction Processing, Written in Concurrent Pascal" Software - Practice and Experience Vol 12(No. 5) 445-454(1982)

[LA82] B.W.Lampson,
"Fast Procedure Calls", ACM Proceedings, Symposium on Architectural Support for Programming Languages and Operating Systems. Palo Alto, March 1982

[LI77] A.M.Lister,
"Inter-Process Communication Mechanisms", Technical Report, 1977, Dept. of Computer Science, University of Queensland, Australia

[MA81] K.J.Maciunas,
"Mercury, a Communicating Sequential Process Language", Technical Report 81-04, Dept. of Computing Science, University of Adelaide, Australia, 1981

[MA79-1] C.D.Marlin,
"A Heap Based Implementation of the Programming Language Pascal", Software - Practice and Experience 9,2 101-119(Feb. 1979)

[MA79-2] C.D.Marlin
COROUTINES: A Programming Methodology, A Language Design, and an Implementation, Ph.D.Thesis, University of Adelaide, 1979

[MI80] MIL-STD-1815
MILITARY STANDARD Ada PROGRAMMING LANGUAGE, 10-Dec-1980

[OR73] E.I.ORGANICK,
Computer System Organization, Academic Press, New York, 1973

[PBH73] Per Brinch Hansen,
Operating System Principles, Prentice-Hall, INC., Englewood Cliffs, New Jersey, 1973

[PBH76] Per Brinch Hansen,
"The SOLO operating system" Software - Practice and Experience Vol 6(No. 2) 141-205(1976)

[PBH78] Per Brinch Hansen,
"Distributed Processes, a concurrent programming concept" Comm. A.C.M. 21(11), 934-941(1978)

[PBH80] Per Brinch Hansen and J.Fellows,
"The TRIO operating system" Software - Practice and Experience Vol 10(No.11) 943-948(1980)

[PO82] F.J.Pollack et al,
"Supporting Ada Memory Management in the iAPX-432", ACM Proceedings, Symposium on Architectural Support for Programming Languages and Operating Systems. Palo Alto, March 1982

[RO81] T.J.Roper and C.J.Barter,
"A Communicating Sequential Process Language and Its Implementation" Software - Practice and Experience (11) 1215-1234(1981)