Inter-Office Memorandum

To CSL, SSL

Date June 12, 1274.

From Jim Mitchell

Location PARC-CSL

Subject What Mesa needs in an Alto Virtual Memory Scheme

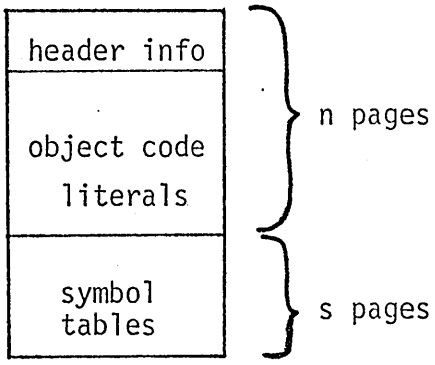
Organization

XEROX

Since there has been much discussion of Virtual Memory (VM) schemes in the context of Alto "Gold Coins", it seemed appropriate that the desires of the Mesa group be specified. This memo is a reasonably high level description of the characteristics which would be helpful to Mesa in a VM scheme.

Mesa Object Files

A Mesa Object Module (file) is composed of two main parts, each of which is a contiguous group of pages in the file:



Generally, s is larger than n, and sometimes s is twice as large as n.

The action of "loading" a Mesa module requires only that the code pages be mapped into memory. The code is never altered in any way - all external connections (generally procedures and ports) are in the data associated with a program. A Mesa routine (procedure or coroutine) is uniquely identified by a frame containing its state and local variables, and many routines may share the code in an object module. The information which allows Mesa to associate a symbol table and code with a routine is kept in a frame called a creator frame. The act of "declaring" an object file to Mesa causes a creator frame for that file to be made. Instances of that module can then be created simply by transferring control to its

creator; the result is a new routine. The cost of creating new instances is roughly comparable with the cost of a procedure call.

The header information in the code block of an object file contains sufficient information to enable the Mesa loader to manufacture a creator for it.

Abstractions for Managing (Overlaying) Code Blocks

A contiguous set of file pages is called a page group in Mesa, and is identified by three values:

filehandle: some handle on a file by which the pages of the file can be named (for example, a JFN in Tenex);

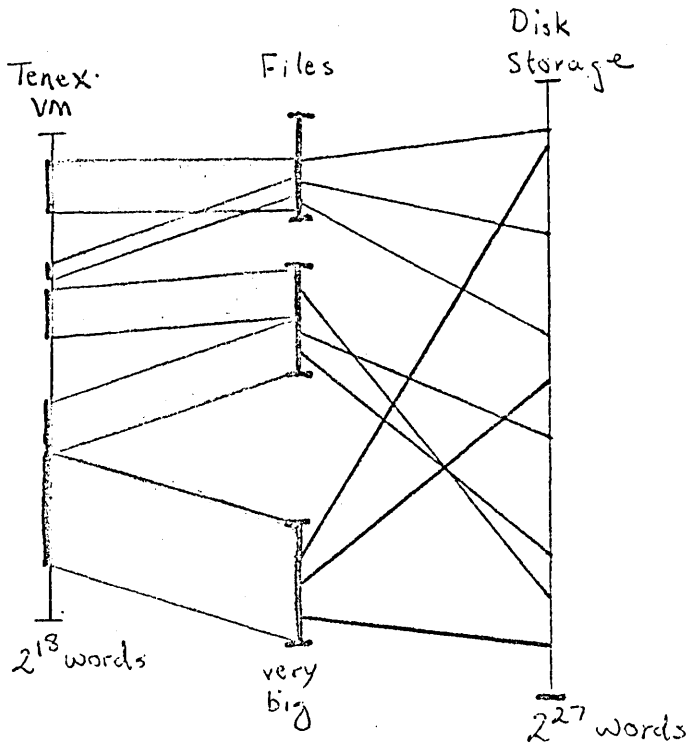
pagebase: the page in the file which corresponds to page zero in the page group;

size: the number of pages in the group.

A page group is accessed by a PG-handle (actually a protected [sealed] pointer) and page groups may be created, destroyed, and swapped into memory or out. More than one page group may, in principle, be associated with the same pages in a file. On Tenex this is accomplished by PMAPping the appropiate file pages into the Tenex VM. One special kind of page group, called a Window group, is provided: the pagebase and size attributes of such a group are alterable after it is initially set up, and it can thus "window" pages of the file to which it is attached.

The code and symbol table parts of a Mesa module are each modelled as a fixed page group by the Mesa loader, overlaying and debugging software. The main implication of this is that Mesa code is "swapped" (overlayed) in units corresponding to the code in a single module. Symbol tables, being separately swappable, only consume VM space when needed by debugging or dynamic (LISP-like) binding mechanisms.

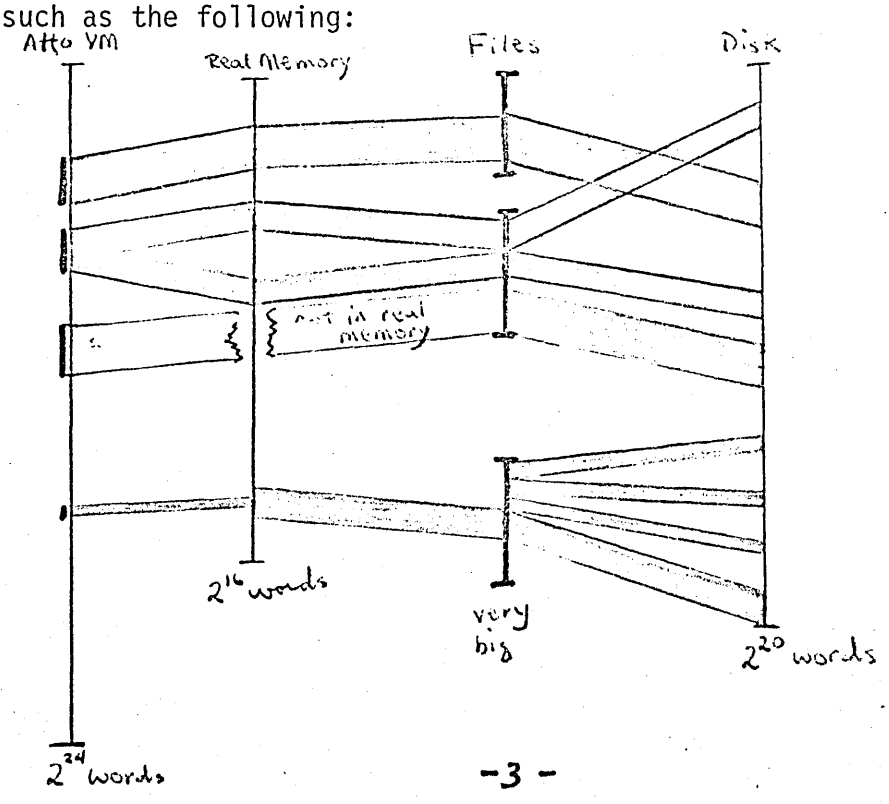
The following diagram portrays the relationships between the Tenex page-grouped VM as seen by Mesa, File memory, and Maxc disk storage (see next page, please):



One obvious conclusion from this diagram is that the logical grouping of file pages which Mesa considers useful is not at all maintained at the level of disk storage. A second observation is that the Tenex-provided VM is being treated as a real memory into which a much larger VM, composed of page groups, is mapped. For obvious efficiency reasons on Tenex, only Mesa code and symbol tables are considered swappable because pointers to them can be controlled by software (for not-in-memory traps, etc.).

Mesa/Alto

We would like to propose a VM scheme for Altos which supports a structure such as the following:



The main things to notice about this second diagram are the following:

(1) As well as logical page groups in VM, there are real page groups in files which correspond to real page groups on disk storage;

(2) A physical page group represents an indivisible disk-to-real memory unit of transfer;

(3) Logical page groups do not have to correspond to physical page groups, but there can be considerable performance payoffs when they do, or when they require a small number of physical page groups. This is because the definition of a physical page group includes the fact that reading it requires a minimal number of disk head seeks and disk rotations.

(4) If all physical page groups are size p, the scheme is very similar to Tenex's VM, with page size = (p*256) words.

Sundry Details

Disk Bad Spots

The disks on Altos do have bad spots. If a bad spot develops in a physical page group, one could either

(a) change the single page group into (at most) three page groups, one of which is a newly allocated single disk page to take over for the page which is unusable;

(b) move the page group as a whole to a new contiguous area and free the previous area, except for the bad page, which is put into a file of bad pages called BadSpots.

I favor solution (b) since it uses machinery which will have to exist in any case, and because there are very few bad spots. Its advantage is that physical page groups are never fragmented; this might be very valuable if someone is treating the VM as consisting of fixed size pages as mentioned previously.

Private Memory

The memory private to a VM should just reside in a PrivateMemory file, but without all the fiction maintained by Tenex to pretend that the JOBPMF for a job is really a file when it is not.

Sharing VM

Virtual memory is only sharable by sharing files; i.e., by mapping a file name string into a logical page group in one's own VM.

Inter-Office Memorandum

To: CSL/SSL

Date: June 15, 1974

From: Ed Satterthwaite

Loc.: Palo Alto

Subject: Mesa for the Alto

Org.: PARC/CSL

On June 12 and 13, the group actively involved in the implementation of Mesa (Geschke, Mitchell, Satterthwaite, Sweet) met to consider the problem of creating a version of Mesa for the Alto. Our conclusions about the major steps as well as time estimates for each are summarized below. An attachment summarizes the more important interdependencies of the steps and indicates a possible division of the work among the members of the group.

We believe that certain additions and changes to TENEX Mesa are essential before Mesa can be moved to the Alto. Most of these are well understood; we propose to implement them in parallel with design of an interpretive system for both MAXC and the Alto.

TENEX Mesa

(1) New version of the segmentation machinery (3 weeks)

(a) complete and test new SEGRUN and associated modules

(b) modify debugger, loader, and bootstrapper

(c) change the compiler to produce modules with new symbol table formats, expanded initialization code

(d) ALSO: extend the compiler to allow arbitrary named types

(2) Finish control structures (4 weeks)

(a) implement support for the control primitives

(b) change compiler's code generators

(c) modify debugger, binder, loader, error handling

(3) General cleanup (3 weeks)

(a) control structures cleanup

(b) implement INCLUDEd program modules, revised binding mechanism

(c) introduction of simple constructors and of sets as data types

(4) Documentation (indefinite)

Alto Mesa

(5) Design the interpretive machine (8 weeks)

(a) instruction set

(b) interpretive engine

(c) Alto microcode feasibility study

(6) Implement interpretive Mesa (i-Mesa) for TENEX (8 weeks)

(a) make an i-Mesa TENEX compiler

(b) make a TENEX i-Mesa interpretive engine

(c) allow i-Mesa and c-Mesa modules to interact

(d) make a complete i-Mesa system for TENEX

Note that this will not be quite identical to an i-Mesa system for Alto (e.g., 36 vs. 16 bit words, different operating system services, etc.)

(7) Move i-Mesa to the Alto (8 weeks)

(a) write a simulator of the Alto operating system interface for TENEX

(b) alter low-level routines of i-Mesa to match the Alto

(c) modify i-Mesa compiler to produce code for the Alto's interpretive engine

(d) modify the TENEX interpretive engine to accept Alto i-Mesa

(e) make a complete i-Mesa system for Alto (running under simulation by TENEX Mesa)

(f) write an interpretive engine in BCPL

(g) transfer i-Mesa system from TENEX to Alto

(8) Move i-Mesa interpreter to Alto microcode (indefinite)

External Constraints

Early in the design of the interpretive machine (step 5) we need to understand the basic facilities to be provided by the kernel operating system for the Alto.

Prior to steps 7a and 7b, we will need a precise definition of that operating system's behavior and interfaces.

Prior to step 7f, we will need one or more dedicated Altos with reliable and reasonably complete (but not highly tuned) utilities and operating system. Easy access to an Alto for familiarizing ourselves with the utilities and service routines would be helpful toward the end of step 5.

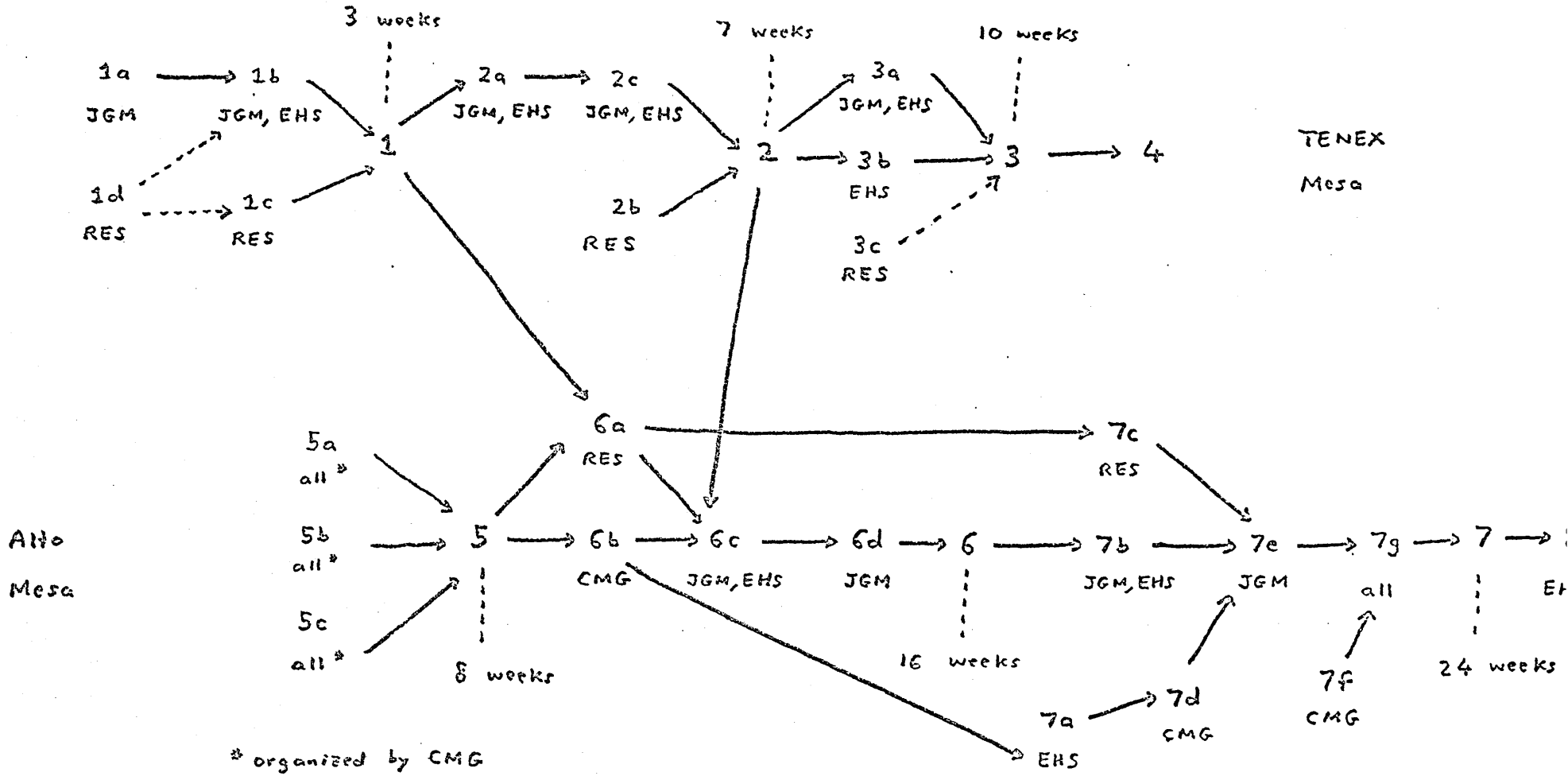
Notes

These time estimates, which are thought to be somewhere between realistic and optimistic, imply that a slow but usable version of Mesa could be running on Altos by the end of 1974 (with some luck and few diversions).

They also imply that, with present manpower commitments, further investigation of the substantive issues in the design of Mesa data structuring facilities as well as the implementation of any solutions will be pushed well into 1975.

During the remainder of 1974, we would like to encourage others to begin using TENEX Mesa to the extent that this is possible without a major effort to produce additional documentation.

Mesa Implementation Plan



Inter-Office Memorandum

To: Mesa and Alto Groups *Date:* August 21, 1974

From: Ben Wegbreit, Chuck Geschke *Loc.:* Palo Alto

Subject: The Implementation of Mesa on Alto *Org.:* PARC/CSL

XEROX

During the past six weeks, a number of measurements have been performed using Tenex Mesa with an eye toward designing and predicting the performance of a Mesa virtual machine emulated by Alto microcode. The purpose of this memo is to outline the method used in this analysis and explain some of the results. The measurements can be divided into two classes: dynamic and static. The static measurements were collected as a basis for deciding how to design a compact representation of Alto-Mesa object code. The results of this static analysis are presented in section VI at the end of this memo. The dynamic statistics were gathered to (1) analyze the compatability of the Mesa virtual machine and the virtual memory scheme proposed for Alto by Wegbreit et. al. and (2) to give a feeling for the performance degradation/improvement in moving from Maxc to an Alto. We begin by giving a brief overview of the essential components of the Mesa virtual machine.

I. The Mesa Virtual Machine

The Mesa machine consists of a number of system controlled registers used to point at a program module's global (own) data and code and to point at the frame of a currently active procedure. In addition the user is allotted some fixed number of registers for pointing at his privately managed data regions. These registers will be implemented via base registers of the virtual memory hardware. A Mesa program module consists of a collection of subroutines and (potentially) a main body. To facilitate subsequent discussion, let's define the following terms:

Greg: pointer to globals of entire program

Creg: pointer to code for main body and all procedures defined in a module

Dreg: pointer to the "own" data of a program module

Freg: pointer to the "local" data of the currently active procedure, i.e. the frame pointer

Uregs: pointers (user computed) to data.

Given the notion of pagegroups in the proposed Alto virtual memory scheme, it is desirable to minimize the number of times that various "base" registers must be reloaded in moving between pagegroups. More precisely, a base register will be said to "fault" when its bounds registers must be reloaded. Let's assume that the above-mentioned Mesa registers are implemented as base registers on the Alto. Further, let's assume that each Mesa module is treated as a pagegroup. Then it follows that as long as program control remains within a given module, Creg and Dreg will not fault. Uregs, of course, can fault at any time. What about Freg? Well, if one notes (as is true in the case of Tenex Mesa programs) that the total active frame space never grows very large, then a pagegroup can be allocated for frame usage so that Freg seldom, if ever, faults. Of course, when a procedure call occurs, Freg must be changed so that it bases the new frame. This requires reloading Freg; however, its bounds registers are unchanged and no fault occurs. Indeed, with the observation that module own variables also occupy a limited amount of storage in most cases, one can reasonably allocate them from the same pagegroup as frames. Thus Dreg also seldom, if ever, faults. Greg bases data common to the entire program; this includes the transfer vector and other similar data. Since this is reasonably small, it should be implemented as a single page group. Hence, Greg never faults.

II. Method of Dynamic Statistics Gathering

Fortunately Tenex Mesa is a reasonable vehicle for measuring the behavior of such a proposed model. It already exhibits an architecture consisting of Creg, Freg, and Dreg and allows acceptable analysis of Uregs. A PDP-10 emulator, created at Harvard and modified at BBN to measure Lisp, has been modified to be a suitable instrument for measuring Mesa. Every instruction fetch and non-instruction-fetch memory access is trapped in the emulator's interpretive loop and a call made to the statistics-gathering routine(s). All the various dynamic measures (while presented separately) were gathered in parallel.

One of the first observations noted was that roughly 40% of the instructions being executed on Maxc were in subroutine call and return sequences. Clearly that points to the necessity of micro-emulated call and return instructions for Alto-Mesa. Those sequences also introduced a great deal of noise into the data collected. Hence the emulator was re-structured to account for call and return sequences in a special manner which will be discussed in more detail later on.

The measuring of Uregs is confused somewhat by the fact that the present Tenex-Mesa compiler allocates user ac's in a stack-like fashion. Almost any other allocation discipline of these six ac's would produce better utilization for the purposes of this model. As a result the Ureg utilization is measured in two ways. The first assumes two ac's, one for reading and the other for writing (as in the Sturgis BCPL study). The second simply takes the six PDP-10 ac's as allocated in the Tenex object code for the applicable read or write operation. The experiments show

nearly no difference in the behavior of the two modes. A more reasonable allocation scheme like LRU could be presumed to perform better but there does not appear to be an easy method to measure its effect.

III. The Mesa machine and address translation

In order to explain the results of the study of Mesa/Alto-virtual-memory compatability, let's examine the test data gathered for two very different Mesa programs. Example A is the MPL compiler compiling a large (35k character) source file and example B is a formatting program (Ed Satterthwaite) handling a (107k character) text file.

(A) Compiling NEWMPLEXP.NLS

```
Instructions emulated: 14666840
Call inst: 6053898      Non-call inst: 8612942
Non-call memory refs: 6770596  Intra-call memory refs: 4441902

Module faults: 228633   Local calls: 155130     Total calls: 383763
Params: 499304  Ret vals: 290715
```

User:	R	W	Total	RFLT	WFLT	Faults
	1282887	450158	1733045	529626	157911	687537

AC#:	R	W	Total	RFLT	WFLT	Faults
0	27360	266859	294219	12594	12187	24781
1	778790	89722	868512	362662	73421	436083
2	279231	166488	445719	91163	54905	146068
3	205313	0	205313	74233	0	74233
4	36	0	36	1	0	1
5	0	0	0	0	0	0
6	0	0	0	0	0	0

```
Total mem refs: 20592966

Faults on AC's 0 thru 6 memrefs: 1138432    5.53
Faults on 2-cache memrefs: 1144803          5.60
```

(B) Formatting of ch7

```
Instructions emulated: 11398563
Call inst: 3286631      Non-call inst: 8111932
Non-call memory refs: 5803251  Intra-call memory refs: 2415508

Module faults: 141258   Local calls: 54737      Total calls: 195995
Params: 187543  Ret vals: 94557
```

User:	R	W	Total	RFLT	WFLT	Faults
	398745	274688	673433	99843	112275	212118

AC#:	R	W	Total	RFLT	WFLT	Faults
0	122851	178238	301089	110968	114648	225616
1	232525	1875	234400	18432	1564	19996
2	25607	95959	121566	2124	795	2919
3	17699	0	17699	741	0	741
4	162	0	162	1	0	1
5	0	0	0	0	0	0
6	0	0	0	0	0	0

Total mem refs: 16722681

Faults on AC's 0 thru 6 memrefs: 531789 3.18
Faults on 2-cache memrefs: 494634 2.96

Notes:

(1) The runs chosen were very long because shorter runs (250k instructions) were dominated by initialization and user interaction with the Mesa debugger. The compiler was run on several long sources and demonstrated very uniform behavior.

(2) Notice that instructions in call/return overhead change from 40% to 30% between the compiler and formatter. The compiler uses a Tree Meta parser which translates into long sequences of subroutine calls. The emulator distinguishes those memory references which occur during call/return from those which do not. The intra-call/return memory references are accounted for separately as will be described later.

(3) Procedure calls are divided into two classes: local calls and module faults (i.e. external calls). Parameters and returned values were counted but not used in the analysis.

(4) The non-call/return memory references which are user computed (as opposed to frame references, references to literals, and module own references) were analyzed in the two ways mentioned earlier. The first ("User:") assumes a two-pointer cache, one for reading and one for writing. The second ("AC#:") uses the PDP-10 AC's as allocated by MPL. A fault occurs when a pointer reference occurs to memory and the value which appears in the pointer falls in a different pagegroup from the preceding reference using that same pointer.

(5) "Total mem refs" are computed by summing the following quantities:

(a)non-call/return instructions
(b)non-call/return memory refs
(c)10*local calls
(d)16*external calls

In (c) the 10 comes from:

call	
allocate new frame	3 (best case)
save PC	1
save Freg	1
return	
restore Freg	1
restore PC	1
de-allocate frame	3

In (d) the 16 comes from the 10 in local call/return plus:

call	
save Creg,Dreg	2
load new Creg,Dreg	2
return	
restore Creg,Dreg	2.

The "best case" assumption in frame allocation of (c) is probably the most common case.

(6) Faults (in both cases) are computed by multiplying the module faults by 2 (one load of Creg for call and one for return) and adding that to the faults incurred by the relevant Ureg discipline. The fault rate is computed by dividing the faults by the total memory references.

(7) The above experiments were first run with the assumption that Mesa data modules (where the Uregs sometimes roam) were one page in size. The emulator was then modified to treat each user data module as a multi-page-pagegroup. The effect, however, was negligible.

IV. Cost of Base Register Faults

When a base register faults, i.e. the address computed through it does not fall in its page group, the register is reloaded by consulting a hash table as proposed by L. P. Deutsch. The cost of this per fault may be estimated as follows:

(1) Using Peter's current micro-code ("Second Try at Lisp Microcode", 6/20/74):

The first probe of the hash table requires 13 microinstructions; subsequent probes require 15 microinstructions. Once the right entry is found, the steps for reloading the base registers depend on new instructions and hardware, so this is somewhat less certain - 4 additional microinstructions is plausible. If the hash table is 1/2 full, then using double hashing and assuming random hash functions, we can expect an average of 1.3 probes for a successful search (since an unsuccessful search implies a disk seek, we won't consider that here). Hence, the mean number of microinstructions per fault is: 13+(.3*15)+4 = 21.5.

Note: If we use linear reprobing instead of double hashing, then probes subsequent to the first require only about 8 microinstructions, while the mean number of probes is 1.5. This gives: 13+(.5*8)+4 =21. That is, it really doesn't make much difference - the first probe and terminal computation are dominent. Hence, to simplify subsequent discussion, we'll assume linear reprobing and a mean number of probes of 1.5.

(2) Using special hardware:

(a) hardware hash computed in the memory interface board and supplied on the bus reduces each probe by 5 microinstructions (from 7 microinstructions needed to obtain a hash to 2).

(b) Putting the hash table in RAM - reduces each read of the table by 2 micro-instructions (from 4 of which one is certainly overlapped to 1).

(c) Putting everything on the memory interface. The best possible probe sequence would seem to be (i) compute initial probe (ii) start fetch from RAM (iii) get word from RAM (iv) compare against the virtual page number sought. A success would be followed by 2 cycles to reload the bounds and mapping registers.

For these four possible organizations, the number of micro-instructions needed to handle a fault are:

(a) 8+(.5*10)+4 = 15
(b) 11+(.5*8)+4 =19
(a&b) 6+(.5*6)+4 = 13
(c) 4+(.5*5)+2 = 8.5

From the fault rate and number of micro-instructions per fault, the performance degradation caused by faults can be computed. Simple instructions on the Alto in Nova emulation mode require about 1200 ns per memory reference (i.e. computation is about 1/3 non-overlapped with memory.) This is probably low over the entire mix, but in the absence of reliable data, we'll take this as typical. The time spent in processing faults per unit of computation time is: faults/memory ref * microinstruction/fault * 170 ns/microinstr * memory ref/1200 ns.

Thus the percentage degradation for each of the hardware organizations and each of the above fault rates are as follows:

	FAULT RATE	
Organization	5.6	3.0
all microcode	17.0%	9.1%
(a) hardware hash	11.8%	7.3%
(b) RAM table	15.0%	8.0%
(a&b)	10.3%	5.6%
(c) all hardware	6.7%	3.6%

Considering the estimates used in various steps, these numbers are best treated as accurate only to within a factor of 1.5 or so either way.

V. Page Faults

Statistics on the expected number of page faults were gathered using a model similar to H. Sturgis' ("Some Statistics for Virtual Memory Fans, Part 2", 7/3/74). In brief, this models a LRU page replacement algorithm for all possible core buffer sizes as follows: A queue of page numbers is maintained, with the i-th most recently referenced page in the i-th position. A vector C of integer counts in maintained in parallel. On each memory reference, the queue is searched for the referenced page. Suppose it is found to be the j-th page in the queue; then the j-th position of C is incremented by 1 and the page is moved to the front of the queue. If an LRU page replacement algorithm is used with a paging buffer of k pages, then the number of page faults is the sum C[K+1] + C[K+2] + C[K+3] + ...

Note: This model takes into account none of the following: choice of dirty vs. clean pages for replacement, page groups, types of references to memory.

Page fault statistics for the two runs discussed in Section III are as follows:

(A) Compiling NEWMPLEXP.NLS

Memory References Considered: 15,383,534

# pages in buffer	# page faults	fault rate
10	448,181	2.91 10↑-2
20	259,541	1.68 10↑-2
30	161,611	1.05 10↑-2
40	86,418	5.62 10↑-3
50	56,306	3.66 10↑-3
60	36,396	2.37 10↑-3
70	22,883	1.49 10↑-3
80	15,641	1.02 10↑-3
90	10,713	6.96 10↑-4
100	7,622	4.95 10↑-4
110	5,550	3.61 10↑-4
120	3,657	2.38 10↑-4
130	2,145	1.39 10↑-4
140	1,160	7.54 10↑-5
150	565	3.67 10↑-5
160	310	2.02 10↑-5
170	132	8.58 10↑-6
180	84	5.46 10↑-6
190	67	4.36 10↑-6
200	55	3.58 10↑-6
210	36	2.34 10↑-6
220	23	1.50 10↑-6
230	0	0

(B) Formatting of ch7

Memory References Considered: 13,915,180

# pages in buffer	# page faults	fault rate
10	15,404	1.11 10↑-3
20	6,037	4.34 10↑-4
30	3,042	2.19 10↑-4
40	1,335	9.56 10↑-5
50	925	6.69 10↑-5
60	680	4.89 10↑-5
70	392	2.82 10↑-5
80	170	1.22 10↑-5
90	30	2.16 10↑-6
100	14	1.01 10↑-6
110	0	0

Note: Memory References Considered in each case were the non-call memory references, i.e. the sum of the classes "Non-call instr" and "Non-call memory refs". This was done to maintain consistency with the data in Section III. However, it produces results with unduly high fault rates since call sequences will require memory references for moving data which are well-behaved with regard to paging. If, in fact, no faults were

caused by these memory references then the fault rates would be lowered by about 1/3 for case (A) and 1/5 in case (B).

To a first approximation, each Maxc page of 512 36-bit words corresponds roughly to one Alto page of 512 16-bit words, since integers fix on one word and it is anticipated that each instruction will fit in one word. (This neglects characters, real numbers, large pointers, etc.) With 60 k of main memory used for buffers, this gives 120 pages.

The effect of the fault rate on overall performance can be done in two ways:

(1) Elapsed time

If a portion of the disk is used for paging (like the Maxc "drum"), then the access time on the Model 44 disk is about 30 ms and about 50 on the Model 31. Consider case (A) as an example. Computation time is about (15.10↑6)*(1.2.10↑-6) = 18 seconds. Given 120 pages for buffers, the number of faults is 3,657 and the time spent in paging with a Model 44 is (3,657)*(30.10↑-3) = 110 seconds.

(2) Comparison with Tenex

A second way of assessing the effects of page faults in the proposed Mesa/Alto machine is in comparison with paging in the current Mesa implementation on Tenex. As the average time to access a "drum" page on Tenex is 42 ms, the effective device speeds may be regarded as essentially the same. Cases (A) and (B) were each rerun twice under different load averages and the actual number of page faults (PGSTAT + 1) were obtained.

	load < 2	load > 6
(A) compiler	432	2987
(B) formatter	487	1006

What conclusions can one draw? A medium-size well-behaved program such as the formatter, example (B), may be expected to fit in core and never page fault. Time waiting for the disk is only that required to load pages. For the example run of (B), we have

Alto upper bound of time to load pages	3.3 seconds
Alto compute time (estimate)	16.7 seconds
Tenex time for page faults	20.4 seconds
Tenex compute time (estimate)	16.7 seconds

That is, Alto Mesa might be expected to run about twice as fast for this run - due to its lower paging needs.

The compiler, example (A), is a large program for which there has been, to date, no major attention to obtaining good paging properties. As it stands, its page fault rate implies a performance of Tenex with a load average somewhat larger than 6. That is, with 120 pages, the number of faults is 3,657. Observe that with 150 pages, the fault rate drops to 565. Considering the ratio of compute time to paging time, there is a critical knee in the curve at around 150 pages. It is anticipated that with a moderate amount of effort, the compiler could be reconfigured to shift the curve and bring the knee down to 120. At a fault rate of 565, Alto Mesa would be comparable to Tenex Mesa with a load average of less than 2.

VI. Static measurements of Mesa Programs

A number of static measurements have been collected on Mesa programs and several more remain to be gathered. The measurements documented here were obtained (for efficiency reasons) by examining the PDP-10 object code for Mesa programs. More static measures are being collected by Dick Sweet by metering the compiler but those results are not yet available. The programs measured consisted of all the Mesa programs stored on the <MPS> directory.

(a) Frame References

The Mesa frame is the locus for parameters, local variables, and (rarely) temporaries (which in this analysis are grouped with locals). The purpose of this study is to determine how many bits of offset from Freg are needed to address variables in the frame.

Bits	%-of-frame-references
1	38.6%
2	60.1%
3	81.9%
4	95.4%

Of course, the most frequently referenced frame variable was not necessarily the one with smallest offset in the PDP-10 object code. Frame variables are simply allocated in declaration order. So, what happens if those most frequently accessed are allocated to the smallest offset position? The following table shows the results for sorted frame variables.

Bits	%-of-frame-references
1	50.3%
2	72.6%
3	89.6%
4	97.4%

The result of the sorting experiment shows that sorting is only worthwhile in the (unlikely) event that one allocates only one or two bits to frame offset addressing.

(b) Constant usage

The use of constants was measured in the object code. This approach had good and bad aspects. On the bad side, one has to be careful to account for constants which are implicit in certain PDP-10 opcodes (e.g. AOS, JUMPG, HRROI, etc.). On the good side, once object code has been produced, the compiler has already done a fair amount of compile-time evaluation of constant expressions. The measurement was done as follows. Constants in the interval [-15,14] were counted individually. Since many of the measured programs did character handling, constants in the range [15,127] were lumped into a separate bin (called CHARS). Finally all constants less than -15 or greater than 127 were thrown into NEG and POS bins, respectively. Here are the results:

Constant	#-of-occurrences
NEG	105
[-15,-2]	72
-1	264
0	4259
1	1705
2	422
3	227
4	143
5	122
[6,14]	627
CHAR	1259
POS	1532
Total	10737

Note that the range [0,1] accounts for 55.5% of the occurences, [-1,2] accounts for 61.9%, and that 81.6% fall in the range [0,127]. If the domain of constants is limited to the range [-15,14], then the interval [0,1] accounts for 75.4% of the occurrences, the interval [-1,2] accounts for 84.8%, and the interval [-1,6] accounts for 95.3%.

(c) Procedures

A couple of static measures were made for procedures that can be compared with dynamic measurements made earlier. Procedure calls were partitioned into local and external calls with the result:

Local calls:	1868	24.5%
External calls:	5746	75.5%

This compares with the dynamic local/external percentages of 40.4%/59.6% and 27.9%/72.1% in examples A and B in section III.

The number of parameters for each procedure call were also tabulated.

#-of-parameters	#-of-calls	percentage
0	1969	21.2%
1	4498	48.4%
2	2026	21.8%
3	563	6.0%
4	165	1.8%
5	70	0.7%
6	11	0%

In dynamic example A, procedures had an average of 1.3 parameters and in example B an average of .96 parameters.

A number of other rather PDP-10 specific static mesaurements were also made which are of little or no interest for Alto-Mesa. The purpose of gathering the static data is to provide information helpful in designing a compact representation for Alto-Mesa object code. For example, the high frequency of one-parameter procedure calls suggests that a two operand, single-parameter-function-call instruction might be profitable. Dick Sweet is gathering data from the compiler on more complex expressions than those which can be conveniently deduced from object code. In particular, if "f[a]" is so frequent, it may well be that instances of "b←f[a]" are so common that a three operand instruction is warranted.