Empirical Analysis of the Mesa Instruction Set

Richard E. Sweet
James G. Sandman, Jr.
Xerox Office Products Division
Palo Alto, California

1. Introduction

This paper describes recent work to refine the instruction set of the Mesa processor. Mesa [8] is a high level systems implementation language developed at Xerox PARC during the middle 1970's. Typical systems written in Mesa are large collections of programs running on single-user machines. For this reason, a major design goal of the project has been to generate compact object programs.

The computers that execute Mesa programs are implementations of a stack architecture [5]. The instructions of an object program are organized into a stream of eight bit bytes. The exact complement of instructions in the architecture has changed as the language and machine micro architecture have evolved.

In Sections 3 and 4, we give a short history of the Mesa instruction set and discuss the motivation for our most recent analysis of it. In Section 5, we discuss the tools and techniques used in this analysis. Section 6 shows the results of this analysis as applied to a large sample of approximately 2.5 million instruction bytes. Sections 7 and 8 give advice to others who might be contemplating similar analyses.

2. Language Oriented Instruction Sets

There has been a recent trend toward tailoring computer architecture to a given programming language. Availability of machines with writeable control stores has accelerated this trend. A recent *Computer* issue [2] contains several general discussions of the subject.

There are at least two reasons for choosing a language oriented architecture: space and time. We can get improved speed by assuring that operations done frequently have efficient implementations. We can get more compact object programs by using variable length opcodes, assigning short opcodes to common operations. The use of variable length encodings based on probabilities is, of course, not new; see the classical papers by Shannon [9] and Huffman [4].

Both space and time optimizations rely on knowledge of the statistical properties of programs. Static statistics are sufficient for code compaction, while dynamic statistics help in the area of execution speed. As most of today's computers have some sort of virtual memory, anything that makes programs smaller tends to speed them up by reducing the amount of swapping.

One of the first published empirical studies of programming language usage was by Knuth [6], where he studied FORTRAN programs. Several other studies have also been published, including [1], [11], and [13]. Similar studies have been made of Mesa programs before each change in the instruction set.

Basing an instruction set on statistical properties of programs leads to an asymmetric instruction set. For example, variables are read more often than they are assigned, so it makes sense to have more short load instructions than short store ones; certain short jump distances are more common than others, so variable length jump instructions can make address assignment a rather complicated operation. There is a misconception held by some that a language oriented architecture is one in which the compiler's code generators have a very easy task. Quite the contrary, in a production environment, we are willing to put considerable complexity into code generation in order to generate compact object programs.

There are trade-offs between code compaction and processor complexity. Encoding techniques such as variable bit length opcodes and conditional encoding add to the amount of microcode or hardware needed, and slow down decoding. The Mesa machines use a fixed size opcode (eight bits), and have instructions with zero, one, or two data bytes. A similar architecture was independently proposed by Tanenbaum [11].

The paper by Johnsson and Wick [5] describes the current Mesa architecture.

3. History of the Mesa Instruction Set

Each machine that runs Mesa provides a microcoded implementation of the Mesa architecture. Machines have spanned more than an order of magnitude in processing power, from the Alto [12] to the Dorado [7], with several machines in between. All have a 16 bit word size.

The overall concepts of the Mesa architecture have not changed since 1974, but the exact complement of instructions has changed several times. New language features, such as a larger address space, have required new instructions. New insights into the usage of these language features have allowed more compact encoding of common operations.

The first implementation of Mesa was done in 1974 for the Alto. Peter Deutsch's experience with Byte LISP [3] had shown the feasibility of a byte code interpreter to run on the Alto. A stack architecture was chosen to allow "addressless" instructions. Decisions on stack size and procedure parameter passing, etc. were partially based on statistics gathered on programs written in MPL, a precursor to Mesa that ran on Tenex (and partially forced by the limitations of the Alto hardware). The MPL study is described briefly in Sweet's thesis [10].

In 1976, a reasonable body of Mesa code existed and was analyzed. A study of source programs is described in [10]. There was also a study of the object code. These analyses lead to small changes in the instruction set; in particular to some two byte instructions where the second (operand) byte was divided into two four-bit fields.

It soon became clear that the small 16 bit address space of the original Alto implementation was too restrictive. There were several proposals for adding virtual memory to the Alto, but they were rejected in favor of designing a new machine whose microarchitecture was better suited for Mesa emulation. In 1978, we had a machine with virtual memory, and the type LONG POINTER (32 bits) was added to the language. This, of course, required instructions for dealing with the new pointers: loading, storing, dereferencing, etc. At the same time, 32 bit arithmetic was also added to the language (and Mesa architecture).

4. Experimental Sample

Today, Mesa has reached a significant level of maturity. Our programmers are working in a development environment written completely in Mesa; there are products in the field, such as the Xerox 8000 series, including the Star workstation, that are programmed entirely in Mesa. These are large programs that make extensive use of the virtual memory. Since the LONG POINTER instructions were added to the architecture before we had any body of code using long pointers to analyze, we were sure that there was room for improvement in their encoding. We did not have the resources at this time to completely redesign the instruction set, but we decided that it was worth our while to see if small changes to the instruction set could lead to more compact object programs.

We started with a sample of programs that was representative of all software running under Pilot [8], the Mesa operating system. We had to decide whether to analyze the source code or the object code generated by the then current compiler. We chose to do both, but this paper deals primarily with the object code analysis.

Some changes, such as increasing the stack depth, or adding new instructions for record construction, have significant effects on the code generating strategy in the compiler. These were studied by instrumenting the compiler or producing a new compiler that generated the expanded instruction set.

Most anticipated instruction set changes were sufficiently similar to the existing set that observing patterns in object code was a workable plan. This certainly included decisions about the proper mix of one, two, and three byte instructions for a given function. In fact, the compiler waits until the very last phase of code generation, the peephole optimizer, to choose the exact opcodes. This concentrates knowledge of the exact instruction set in a single place in the compiler.

5. Experimental Plan

The general plan of attack was as follows:

1. Normalize the object code.

   We converted the existing object code into a canonical form. This included breaking the code into straight line sequences, and undoing most peephole optimizations. The sample resulted in 2.5 million bytes of normalized instructions.

2. Collect statistics by pattern matching.

   Patterns took two general forms: compiled in patterns that looked at things like operator pair frequencies, and interactive patterns, where the user could type in a pattern and have the data base searched for that pattern.

3. Propose new instructions.

   Based upon the statistics gathered in step 2, we proposed new instructions.

4. Convert to new opcodes by peephole optimization.

   We wrote a general framework for peephole optimization that read and wrote files in a format compatible with the pattern matching utilities. This allowed us to write procedures that would convert sequences of simple instructions into new fancier instructions.

5. Repeat steps 2 through 4.

While the statistics from step 2 tell us how many of each new instruction we will get in step 4, the ability to partially convert the data file was helpful for questions of the form "What local variables are we loading when the load is not folded into another instruction?"

Normalization

The version of the Mesa instruction set under analysis used 240 of the possible 256 byte values. Moreover, many of the instructions are single byte encodings of what is logically an operation and an operand value, e.g. "Load Local 6" or "Jump 8." Other instructions replace two or three instruction sequences that are sufficiently common to warrant a more compact encoding. To simplify analysis, all code sequences were transformed into semantically equivalent sequences of a subset of the instructions, comprising slightly over 100 opcode values.

1. Expand out imbedded operand values.

   All instructions with embedded operand values were replaced by a corresponding two or three byte instructions where the operand is given explicitly. For example "Jump 8", a single byte opcode was replaced by the three byte sequence: the "Jump word" opcode, and a two byte operand with a value of 8.

2. Break apart multi-operation opcodes.

   Most complicated instructions were replaced by sequences of equivalent simpler instructions. For example, "Jump Not Zero" was replaced by the sequence "Load 0," "Jump Not Equal." Notable exceptions were the "Doubleword" instructions. These could often have been replaced by two single word instructions, but a major thrust of this analysis was finding out how doublewords were used in the language.

The procedure that did the normalization first made a pass over the code to find the targets of all jumps. These were then sorted so that the normalizing procedure could put a marker byte in the output file between each sequence of straight line code.

The analysis software was written so that the normalization routine could run as a coroutine with any of the pattern matchers, converting object files to a stream of normalized bytes. While not a complete waste of effort, this option was not used when the mass of data became large. The normal mode of operation was to convert a related set of object programs to a single output file, and then use that data file, or a collection of such files, as the input to pattern matching and peephole optimization.

When working with large amounts of data, you should plan for expansion. Consider the format of the code sequence data file. The normalization step reduces the opcodes to a set with approximately a hundred members. On the other hand, the peephole optimization (step 3 above) adds new opcodes. In fact, before we were done we had more than 256 logical opcodes (some of them became two or three byte sequences in the resulting instruction set using an escape sequence). As we desired to have the output of peephole acceptable to the pattern matchers, we used two bytes for each operation "byte" of the stream.

Pattern Matching

The collected files of normalized instructions may now be used to answer questions about language usage. One obvious question is "How many of each opcode do I have?" It is easy to write a routine that reads the data file and counts the opcodes. This was one of a class of generic patterns that we ran on our data file. The set of generic patterns waxed and waned throughout the several months of analysis, but at the end, we found the following patterns most interesting:

1. Static opcode frequency.

   Count the number of occurrences of each opcode.

2. Operands values.

   For each opcode, get a histogram of operand values.

3. Opcode successors.

   For each opcode, get a histogram of the set of next opcodes in the code sequences.

4. Opcode predecessors.

   For each opcode, get a histogram of the set of previous opcodes in the code sequences.

5. Popular opcode pairs.

   Consider the set of all pairs of adjacent opcodes; sort them by frequency.

The reader will doubtless observe that patterns 3, 4, and 5 all report the same information. Patterns 3 and 4 are valuable because, even when the frequency of an opcode pair is not especially high, the conditional probability of one based on the other might be high. Additionally, all three patterns provide information that can suggest additional areas of study, as described below.

We also wrote patterns for finding popular triples, and in fact popular n-tuples, where the search space is seeded with allowed (n-1)-tuple initial strings. These weren't as interesting as we had suspected; we got mountains of n-tuples that occurred only a few times, and we tended to run out of storage. Looking at pairs, along with a knowledge of the language and the compiler's code generation strategies, allowed us to generate patterns that gave us statistics on most interesting multibyte constructs.

User Specified Patterns

For matching of longer patterns, or answering specific questions about instruction use, we preferred not to have to recompile the matching program for every new pattern. We therefore wrote an interactive program where the user typed in a pattern which was parsed, and then matched against the data base. A pattern was a sequence of instructions; each instruction consisted of an operator and its operands. The operator/operands could be given explicitly in the pattern, or a certain amount of "wild carding" was allowed. For wild card slots, we provided the option of collecting statistics on the actual values.

Consider the pattern: `LLB * IN [0..16), RB $`. The instruction `LLB` is a two byte "load local variable" instruction where the second byte gives the offset of the variable in the frame (procedure activation record). Similarly, `RB` says "dereference the pointer on the stack, adding the offset specified by the operand byte." This pattern finds all occurrences of `LLB` followed by `RB` where one of the first sixteen local variables is a pointer being loaded. The $ is a wild card match like the *, except it tells the pattern matcher to gather statistics on the actual operand values for the `RB` instructions. The output of the pattern matcher looked something like this:

```
Total data: 1289310 inst, 2653970 bytes
------------------------
LLB * IN [0..16), RB $  total: 22813

   value    count    %    cum.%
       0     7575  33.20  33.20
       1     3638  15.94  49.15
       2     2838  12.44  61.59
       3     1700   7.45  69.04
       4     1291   5.65  74.70
       5      823   3.60  78.31
       6      746   3.27  81.58
       7      577   2.52  84.10
      13      344   1.50  85.61
      15      328   1.43  87.05
      10      315   1.38  88.43
      11      283   1.24  89.67
      14      277   1.21  90.89
      12      252   1.10  91.99
      16      220   0.96  92.96
      23      194   0.85  93.81
```

Figure 1. Sample Pattern Matcher Output

These data tell us that the vast majority of offsets are small. If the first "*" had been a "$", statistics would have been collected on which local variable was loaded as well. The statistics for this field are even more skewed over 90% of the matches are for locals at offset 0, 1, or 2.

Peephole Optimizer

Based on the statistics gathered by pattern matching, we proposed some new instructions. Some of these new instructions were single byte opcodes that encoded a common operand value of what was logically a two or three byte operation; other new instructions were combinations of operations that occurred frequently in code sequences.

Decisions about the two types of instructions were interrelated. The question "How many single byte load local' instructions should we have" is best answered by looking at the load local statistics after any loads have been combined into fancier instructions. We solved this problem by writing a peephole optimizer to convert normalized code sequences into sequences of new instructions. This simplified the patterns needed for decisions and also allowed us to look for patterns involving the new instructions. The actual peephole conversion was done by straightforward case analysis, but the framework that it was built upon is worthy of some discussion.

There are several problems with operating directly on the data files. Variable length instructions cannot be read backward, and some instructions have two operand bytes that are logically a single sixteen bit operand. For this reason, the file reading procedure produced fixed sized Mesa records containing the opcode and an array of parameters, correctly decoding multibyte operands. These were maintained in an array as shown in the figure below.

Figure 2. Peephole Optimization Framework

The optimizing procedures typically dealt with the element at index 0, based upon previous instructions (*i*) and following instructions (+*i*). The range of index values depends on how much history is required in the peephole procedure. For all of our routines, a range from −5 to +3 was more than adequate. The framework provided the following operations:

1. Delete *i*.

   Any instruction not already written to the output may be deleted.

2. Output new code.

   New instructions may be generated; they are buffered until the next shift, but will appear just to the right of index 0.

3. Shift left.

   The first new output, or the element at +1, is moved to index 0. Deleted cells are compacted. The buffered new code is moved into the array, possibly pushing some of the previous +*i* elements into a buffer at the right. Any instruction forced out the left is written to the output file. In the case of no change, this reduces to a write, a block transfer in memory, and a read; in the general case, the operation can be rather complicated.

One useful feature of the framework was a display facility that showed the entire array on the screen, with the instruction given as a mnemonic and the parameter array shown only to the extent that the given instruction had parameters. We had several stepping modes, allowing us to see the instructions streaming by, or allowing us to stop and display only when an optimization was to take place.

6. Results

There is certainly not room in this paper to show the complete results of our analysis. Instead, we will show some of the generally interesting results, and go into considerable detail for one class of jump instructions.

Statistics of the Normalized Instruction Data

Table 1 shows the most frequently occurring elements of the original normalized instruction set, together with their statistics.

Op	*count*	*%*	*cum.%*	
LI	208924	16.90	16.90	Load immediate
LL	156848	12.68	29.59	Load local variable
SL	81270	6.57	36.16	Store local variable
REC	64145	5.18	41.35	Recover previous top of stack
LLD	62950	5.09	46.44	Load local doubleword
EFC	55982	4.52	50.97	External function call
J	50726	4.10	55.08	Unconditional jump
R	42328	3.42	58.50	Dereference pointer on stack
SLD	37747	3.05	61.56	Store local doubleword
LA	29205	2.36	63.92	Address of local variable
ADD	28987	2.34	66.26	Add top two words of stack
JNE	25499	2.06	68.33	Jump not equal
RET	24176	1.95	70.28	Return
JE	23335	1.88	72.17	Jump equal
LG	21594	1.74	73.92	Load global variable
LFC	21450	1.73	75.65	Local function call
DADD	20652	1.67	77.32	Doubleword add
LGD	17895	1.44	78.77	Load global doubleword
LLK	16193	1.31	80.08	Load link

Table 1. Frequency of normalized instructions

Table 1 contains some interesting data about language usage. Note that the local variables of procedures are loaded twice as often as they are stored. Doubleword (32 bit) variables are loaded and stored almost half as often as single word ones. Over 6% of the instructions were procedure calls (EFC+LFC), and there were statically three times as many procedure calls as returns. Knowing that the compiler generates a single return from a procedure to facilitate setting breakpoints, we can conclude that procedures are called from an average of three places. Almost 17% of the instructions load constants (LI). Table 2 shows the most popular constants. Bear in mind that some of the loads of constants go away when then are combined into fancier instructions, as we will see in the section on conditional jumps.

Value	*count*	*%*	*cum.%*
0	96652	45.83	45.83
1	29546	14.01	59.84
2	8901	4.22	64.06
3	7094	3.36	67.42
4	5895	2.79	70.22
-1	5553	2.63	72.85
5	3411	1.61	74.47
6	3198	1.51	75.99
8	2220	1.05	77.04
13	2037	0.96	78.01
9	1853	0.87	78.88
7	1841	0.87	79.76

Table 2. Distribution of values for load immediate instructions

The distribution of local variables loaded is shown in Table 3. The reader should be aware that the compiler sorts the local variables by static usage before assigning addresses in the local frame.

Offset	*count*	*%*	*cum.%*
0	63152	40.29	40.29
1	23151	14.77	55.07
2	15125	9.65	64.72
3	10116	6.45	71.17
4	7886	5.03	76.21
5	5837	3.72	79.93
6	4323	2.75	82.69
7	3754	2.39	85.08
8	2718	1.73	86.82
9	2096	1.33	88.16

Table 3. Distribution of offsets of local variables loaded

Analysis of Conditional Jumps

We observe from Table 1 that approximately 4% of the instructions are testing the top two elements of the stack for equality (`JE` or `JNE`). It is instructive to describe in some detail the steps that we took in deciding upon what specific instructions to generate for the "Jump Not Equal" class of instructions (`JNE`).

In Tanenbaum's proposed architecture [11], he allocates 20 one byte instructions and one two byte instruction to each of "Jump Not Equal" and "Jump Not Zero." We would rather not use this much of our opcode space. We looked to see if some of the conditional jumps could be combined with other operations.

From the predecessor data, we observed that 84.7% of the `JNE` instructions are preceded by a load immediate. We next wrote a pattern that gave a distribution of the values being tested against. Table 4 shows the most frequent values.

Value	*count*	*%*	*cum.%*
0	11792	54.07	54.07
1	2181	10.00	64.07
3	1441	6.60	70.68
2	1032	4.73	75.41
4	390	1.78	77.20
5	314	1.43	78.64
6	238	1.09	79.73
7	232	1.06	80.80
-1	220	1.00	81.81
15	198	0.90	82.72

Table 4. Constants loaded before Jump Not Equal instructions

It comes as no surprise that 0 is the most common value, since 1% of the pre-normalization instructions were "Jump Not Zero," and they were normalized to the sequence `LI 0, JNE`. We clearly needed to put back in at least the two byte version of this instruction, "Jump Not Zero Byte" (`JNZB`), where the operand byte specifies the jump distance. The frequency of other small constants lead us to propose a new instrucion: "Jump Not Equal Pair," a two byte instruction where the operand byte is treated as two four bit fields, one a constant, and the other a jump distance. Since jump distances are measured from the first byte of a multibyte instruction, the first reasonable value to jump is 3 bytes jump over a single byte. When we looked at the jump distances for `JNE`, however, we saw that 3 byte jumps occur very seldom, and that 5 bytes is the winner, followed by 4 bytes. For this reason, we biased our distances by 4.

By using the data byte to hold a constant between 0 and 15, and a jump distance between 4 and 19, we found 4464 opportunities for the new `JNEP` instruction. This did not count the situations where the constant value was 0, since they could be encoded by the equally short `JNZB` instruction.

After the `JNZB` and `JNEP` instructions are removed from `JNE` statistics, there are still over 5000 cases of `LI *, JNE` left. In these, either the constant value or the jump distance was out of range. We decided to include a "Jump Not Equal Byte Byte" instruction one with two operand bytes: a value for comparison, and a signed jump distance. This took care of most of the remaining cases.

Now it was time to look at the operands of the remaining `JNEB` instructions to see if we should have any one byte `JNE` instructions. The distribution was fairly flat, with the most frequent occurring around 450 times. For this reason, we declined to include single byte `JNE` instructions.

We also looked at the operands of the `JNZB` instructions. There were two values, 4 and 5, that were frequent enough to warrant single byte instructions. We added the instructions `JNZ3` and `JNZ4` (remembering that the jump distance counts from the first byte of the instruction).

In summary, our Not Equal testing is now supported by the following instructions:

Opcode	*bytes*	*count*	*% of* JNE
JNEB	2	4501	18
Jump Not Equal Byte (all byte jumps are signed bytes)			
JNZB	2	8878	35
Jump Non-Zero Byte			
JNEP	2	4464	17
Jump Not Equal Pair (value in [0..15], dist in [4..19])			
JNEBB	3	4742	19
Jump Not Equal Byte Byte (value in [0..255], dist in [-128..127])			
JNZ3	1	1029	4
Jump Non-Zero 3			
JNZ4	1	1885	7
Jump Non-Zero 4.			

Table 5. Jump Not Equal in the new instruction set

The then current opcode set under analysis had a two byte JNZB instruction, a two byte JNEB instruction and eight single byte JNE instructions. The new instruction set has no single byte JNE instructions; most of them occurred in situations where we could combine the jump with the preceding instruction into a new two byte jump. The overall net change was a 13% decrease in code bytes used for not-equal testing compared to the previous instruction set, even though there are four fewer JNE instructions.

Statistics of the Final Instruction Set

From information theory, we know that the best encoding would have all single byte opcodes equally probable. While we do not meet this ideal, the distribution of opcode frequencies is a lot flatter than that of the normalized set. Table 6 shows the most frequently occurring instructions in the new instruction set. Note that of the twenty-two instructions shown in Table 6, fourteen are straightforward single operation opcodes with any operand values given explicitly as additional bytes, six are single byte instructions where operand values are encoded in the opcode, and two are compound operations combined into a single opcode.

`LI0 46956 4.57`
Load immediate 0

`LL0 35242 3.43`
Load local 0

`JB 25587 2.49`
Jump byte a relative, signed byte distance

`RET 24256 2.36`
Return

`LIB 19944 1.94`
Load immediate byte operand is literal value

`LL1 18951 1.84`
Load local 1

`EFCB 17074 1.66`
External function call byte operand specifies a link number

`LAB 16706 1.62`
Local address byte load address of a local variable

`LI1 16244 1.58`
Load immediate 1

`REC 15929 1.55`
Recover value just popped from stack

`SLB 13977 1.36`
Store local byte operand is offset in frame

`JZB 13618 1.32`
Jump zero byte pop stack, jump if value = 0

`LLD0 13553 1.32`
Load local doubleword 0

`LLB 13269 1.29`
Load local byte operand is offset in frame

`LL2 13132 1.27`
Load local 2

`ADD 12435 1.21`
Add adds the top two elements of the stack

`SLDB 12400 1.20`
Store local doubleword byte operand is offset in frame of first word

`LLDB 11222 1.09`
Load local doubleword byte operand is offset in frame of first word

`LIW 11205 1.09`
Load immediate word next two bytes are a 16 bit literal

`JW 10322 1.00`
Jump word next two bytes are a 16 bit relative jump distance

`LLKB 10306 1.00`
Load link byte operand specifies link number

`RLIP 9691 0.94`
Read local indirect pair operand has four bits to specify local variable pointer, four bits to specify offset of word relative to that pointer.

Table 6. Most frequent instruction of the new set.

It is interesting to compare the contents of Tables 1, 2, and 3 with that of Table 6. We see that over half of the `LI0` instructions have been folded into new instructions. Eighty percent of the `LL` instructions are either encoded as single byte instructions such as `LL0`, or folded into more complicated instructions such as `RLIP`. Several of the most common instructions are load immediate ones (`LI*`). In fact, the complete frequency data show that almost 13% of all new instructions are some form of load immediate. The most frequent instruction, weighted by instruction size, is `JB`, a two byte unconditional jump. The most frequent conditional jump is a test against zero, `JZB`; many of these arise from tests of Boolean variables. Table 7 shows the set of one and two byte load and store local instructions of the new instruction set.

Load instructions push local variable onto stack.

	bytes	*total*	*%*
`LLn`, for `n=0,1,2,3,4, 5,6,7,8,9,10,11`	1	103402	10.1
`LLB`	2	13269	1.3
`LLDn`, for `n=0,1,2,3,4, 5,6,7,8,9`	1	39989	3.9
`LLDB`	2	11222	1.1

Store instructions pop from stack into local variable.

`SLn`, for `n=0,1,2,3,4, 5,6,7,8,9,10`	1	44598	4.3
`SLB`	2	13977	1.4
`SLDn`, for `n=0,1,2,3,4, 5,6,8`	1	21829	2.1
`SLDB`	2	12400	1.2

Put instructions store from stack into local variable, don’t pop.

`PLn`, for `n=0,1,2`	1	10540	1.0
`PLB`	2	4195	0.4
`PLDn`, for `n=0`	1	2350	0.2
`PLDB`	2	5238	0.5

Variables outside the first 256 words of the frame are loaded and stored so infrequently that the compiler first generates their address on the stack and then uses the pointer dereferencing instructions. We considered a three byte "Load Local Word" instruction with a sixteen bit offset, but found that "Local Address Word," which loaded the address of a local variable, was more useful. The compiler needs to generate the address of large variables (larger than two words) in order to use the "Block Transfer" instruction; if a variable is at a large offset in the frame, it is probably a large variable as well.

We implemented fewer short instructions for storing local variables than for loading them. Note in Table 6 that four of the single byte load local instructions appear in the top fifteen instructions. Table 7 says that the most frequently referenced (and hence the first in the frame) locals are loaded over twice as often as stored. The variables that are loaded with the two byte `LLB` are loaded and stored at about the same frequency. The "put" instructions arise primarily at statement boundaries where a variable is stored in one statement and then immediately used in the next; such situations are found by the peephole optimizer of the compiler.

7. Analysis

The most useful patterns for finding sequences of instructions to combine are succeessors, predecessors, and popular pairs. A simple minded scheme for generating instructions is to start down the list of popular pairs and make a new instruction for each pair until the number of occurrences of that pair reaches some threshold. Of course, each new instruction potentially changes the frequencies of all other pairs containing one of the instructions.

Popular pairs will find many sequences but the data from the successors and predecessors patterns should not be overlooked. For example, the `WS` (Write Swapped) instruction writes a word in memory using a pointer and value popped from the stack. The `REC` (Recover) instruction recovers the value that was previously on the stack; after a `WS`, it recovers the pointer. The successor data showed that 91.4% of the `WS` instructions were followed by a `REC`. These two instructions were combined into the `PS` (Put Swapped) instruction which left the

pointer on the stack. We could then eliminate the `WS` instruction entirely and use the sequence `PS, DIS` (Discard) the remaining 8.6% of the time.

It helps to know what the compiler does when analyzing patterns. We were suprised to find no occurrences of the pattern `LI` 0, `LI` 0. We found them when we looked at popular pairs the compiler had changed that sequence into `LI` 0, `DUP` (Duplicate). This sequence was one of the more popular pairs, which lead us to include the new instruction `LID0` (Load Immediate Double Zero).

The pattern showing histograms of operand values is useful for deciding when to fold an operand value into a single byte opcode. Remember that combining instructions may change the operand distribution. For example, the initial operand data for `JNEB` showed very popular jump distances of 3 through 9 bytes. The original instruction set had single byte instructions for these jumps. After the analysis, most of these short jumps had been combined into the `JNEP` or `JNEBB` instructions. The operand data obtained after peephole optimization did not warrant putting the short `JNE` instructions back into the instruction set.

8. Implementation Issues

One cannot blindly apply the statistical results of the analysis to decide what instructions to have in the new instruction set. It is necessary to temper these data with knowledge of the compiler, history and expected future trends of language use, and details of the implementations of the instruction set.

There are some operations that are needed in the machine, even though they occur infrequently the divide operation is an example. Many such operations can be encoded as a single opcode, `ESC` (Escape), followed by an operand byte specifying the infrequently used operation. This makes available more single byte opcodes for more frequently occurring operations. Mathematically, it makes sense to move any operation to `ESC` if the available opcode can hold a new operation that gives a net savings in code size. On the other hand, each new opcode adds complexity to the implementation.

Suppose there are two potential new instructions with the same code size savings, one that combines two operations, and the other that combines an operand value with an operation. The latter often results in less complexity in the implementation of the instruction set. In particular, if you already have a LL6 instruction, it typically takes only a single microinstruction to add LL7.

There are many encoding tricks that can be used to save space. Some of these can be decoded at virtually no cost, others are more costly. In the analysis of JNE above, we ended up with an instruction, JNEP, where the operand byte was interpreted as two four bit fields, a literal value and a jump distance. The jump distance was biased, i.e. the microcode added 4 to the value before interpreting the jump. The literal value, on the other hand was unbiased, even though the compiler would not generate the instruction for one of the values. For one of the microprocessors implementing the instruction set, biasing the compared value would have significantly slowed down the execution of the instruction.

In an integrated system such as Mesa, global issues must be considered when making instruction set decisions. For example, many procedures return a value of zero. The statistics showed that an opcode that loads zero and returns would be cost effective. However, the source level debugger takes advantage of the fact that a procedure has a single RET instruction when setting exit breakpoints (all of the procedure's returns jump to this RET). We were unwilling at this time to add the complexity to the debugger of finding all possible return instructions (RET and the new RETZ) in order to set exit breakpoints. Therefore we declined to add this new instruction.

Finally, be careful when analyzing data obtained about an evolving system. Be aware that some common code sequences reflect attempts by older programs to cope with restrictions that are no longer in the architecture. For example, programs written to live in a small address space use different algorithms than those written to live in a large address space.

9. Conclusions

We began our analysis with limited goals: we had a short time in which to make recommendations about changes to the instruction set, we were generally happy with the old instruction set, and we didn't have the resources to handle the necessary rewriting of microcode and compiler that a massive change in the instruction set would require.

Our experience showed that our chosen method, analysis of existing object code, was a workable approach to the problem. Normalization of the code to a canonical form proved valuable for simplifying the subsequent pattern matching used.

We found that simple minded analysis of n-tuples becomes unworkable for n>2, but that informed study of opcode pairs allowed us to postulate longer patterns for study. An interactive pattern matching program was valuable for answering questions about longer patterns.

Our analysis predicted an overall reduction in code size of 12%. We converted the compiler to generate the new instructions and realized the expected savings on a large sample of programs.

10. Acknowledgments

The first opcode analysis of Mesa was done by Chuck Geschke, Richard Johnsson, Butler Lampson, and Dick Sweet. Loretta Guarino Reid helped to develop the current analysis tools, and LeRoy Nelson helped to produce the program sample. The analyses were run on a Dorado, whose processing power was invaluable for handling the large amount of data that we had.

Bibliography

[1] Alexander, W. G., and Wortman, D. B., "Static and Dynamic Characteristics of XPL Programs," *Computer*, vol 8. pp. 41-46, 1975.

[2] Chu, Yaohan, ed., Special issue on Higher-Level Architecture, *Computer*, vol. 14, no. 7, July 1981.

[3] Deutsch, L. Peter, "A LISP machine with very Compact Programs," *Third International Joint Conference on Artificial Intelligence,* Stanford University, 1973.

[4] Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the IRE,* vol 40, pp. 1098-1101, September, 1952

[5] Johnsson, Richard K., and Wick, John D., "An Overview of the Mesa Processor Architecture," *Symposium on Architectural Support for Prog. Lang. and Operating Sys.,* Palo Alto, Mar. 1982.

[6] Knuth, Donald E., "An Empirical Study of FORTRAN Programs," *Software Practice and Experience,* vol. 1, pp. 105-133, 1971

[7] Lampson, Butler W. *et. al., The Dorado: A High-Performance Personal Computer Three papers*. CSL-81-1, Xerox Palo Alto Research Center, Palo Alto, California, 1981.

[8] Mitchell, James G., Maybury, William, and Sweet, Richard E., *Mesa Language Manual.* Version 5.0. CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, California, 1979.

[9] Redell, David D. *et. al.,* "Pilot: An Operating System for a Personal Computer," *Communications of the ACM*, vol. 23, pp. 81-92, 1980.

[10] Shannon, C. E., "A Mathematical Theory of Communication," *Bell System Technical Journal,* vol 27, pp. 379-423, 623-656, 1948.

[11] Sweet, Richard E., *Empirical Estimates of Program Entropy*. CSL-78-3, Xerox Palo Alto Research Center, Palo Alto, California, 1978.

[12] Tanenbaum, Andrew S., "Implications of Structured Programming for Machine Architecture," *Communications of the ACM,* vol. 31, pp. 237-246, 1978.

[13] Thacker, C. P. *et. al.*, "Alto: A personal computer," in *Computer Structures: Readings and Examples*, Second edition, Sieworek, Bell and Newell, Eds., McGraw-Hill, 1981. Also in Technical Report CSL-79-11, Xerox Palo Alto Research Center, 1979.

[14] Wade, James F., and Stigall, Paul D., "Instruction Design to Minimize Program Size," *Proceedings of the Second Annual Symposium on Computer Architecture*, pp. 41-44, 1975.