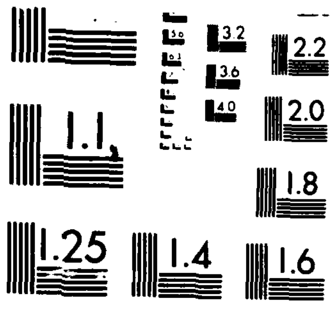
AD-A178 428   LISP ON A REDUCED-INSTRUCTION-SET-PROCESSOR(U) STANFORD   1/1
UNIV CA COMPUTER SYSTEMS LAB P STEENKISTE ET AL. 1986
MDA903-83-C-0335

UNCLASSIFIED   F/G 9/2   NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

LISP on a Reduced-Instruction-Set-Processor

Peter Steenkiste and John Hennessy
Computer Systems Laboratory
Stanford University

Contract MDA-903-83-C-0335

Abstract

We ported the Portable Standard Lisp compiler to MIPS-X, a reduced-instruction-set processor. In this paper we report on a number of profiling measurements made on a set of 11 LISP programs. The measurements give information on two levels. First, we look at what instructions LISP programs use at the assembly level. Because the instruction set of MIPS-X contains only very simple, basic instructions, the profiling information is at a very low level. In a second group of measurements, we determine how much time each program spends on the most common primitive LISP operations. Because MIPS-X executes all machine instructions in a single cycle, it is possible to get very accurate timing measurements by instruction counts.

1. Introduction

In the last couple of years a number of reduced-instruction-set processors have been designed: the IBM 801 [17], the Berkeley RISC machine [15], and the Stanford MIPS processor [10]. The design of these processors is based on the observation that during the execution of compiled programs, mostly simple operations are executed. Reduced-instruction-set processors exploit this feature by only implementing simple operations in hardware, and by handling the more complicated operations in software. This makes it possible to execute the more frequent, simple operations very fast. The set of most frequently executed operations is determined by studying a number of programs written in high-level languages, mostly Pascal and C for the projects to date.

We ported the Portable Standard Lisp (PSL) compiler to MIPS-X, a reduced-instruction-set processor. Our goal was two-fold. First, we wanted to determine whether MIPS-X, the design of which is based primarily on extensive measurements of programs written in Pascal and C, is a good host for LISP. Second, we wanted to get detailed information on the behavior of LISP programs, both at the machine level and at the source level. In this paper we report the results of the second part of this project.

A study of the Berkeley RISC processor as a host for LISP [16], suggests that reduced-instruction-set processors are potentially good hosts for LISP. However, the study is based on a single benchmark, and the project never resulted in a working compiler.

1.1. The MIPS-X architecture

MIPS-X [11] is the successor of MIPS [10]. It has a simple instruction set with single cycle execution for all instructions. The projected cycle time is 50 ns. The processor has a load-store architecture with 32 general purpose registers. A VLSI implementation of the architecture is in progress.

The instruction set contains 3 groups of instructions. The *compute instructions* take two operands from the registers and leave the result in a third. The compute operations include add, subtract, logical bit operations, arithmetic and logical shift, byte rotate, and one bit multiplication and division steps. The *memory instructions* transfer data between memory and the registers. They are the only instructions that access memory. The *branch and jump instructions* include a pc-relative unconditional branch, a full set of pc-relative conditional branches, and an absolute jump for procedure calls and returns.

All branches and jumps have two *delayed branch slots*.

The MIPS-X research project has been supported by the Defense Advanced Research Project Agency under contract # MDA903-83-C-0335

To appear in the 1986 ACM Conference on
LISP and Functional Programming
MIT, August 4-6, 1986

This means that the control transfer to the branch target occurs two cycles after the execution of the branch, and that the two machine instructions immediately following the branch are always executed, independent of whether the branch goes or not. Delayed branch slots are not visible at the assembly level. The *reorganizer*, a special pass in the assembler, creates the delayed slots, and tries to fill them with useful instructions from before the branch or from the predicted branch successor. All conditional branches come in two flavors. A first type always execute the instructions in the two delayed slots. If these instructions are taken from a branch target, then they should not overwrite any resources that might be needed if the branch goes the other way. The second type, called *squashed branches*, only executes the instructions in the delayed slots if the branch goes, otherwise two no-ops are executed [13].

1.2. The PSL compiler

Portable Standard Lisp [8, 7] is built around a portable compiler. It is a small, efficient LISP system and it has been ported to a large number of architectures.

Bringing up PSL on a new architecture requires a running PSL system. The backend of the compiler is changed to generate assembly code for the new processor, and this is used to compile the entire PSL kernel, which is written in PSL. The kernel can then be assembled and linked on the new machine. This was done for MIPS-X with a VAX PSL implementation as host. Because the MIPS-X hardware implementation is not finished, all programs have to be run on a simulator. For this reason we did not port the whole kernel, but only the parts that are necessary to run the test programs.

We chose the PSL dialect mainly because it was reputed to be fairly efficient, and easy to port. This proved to be correct although we added a few standard optimizations. An important question is whether the results presented here apply to other LISP dialects. We believe they do. Although PSL is a rather small dialect it supports all widely used LISP data types like numbers, lists, structures, arrays, and strings. Several other LISP dialects, for example Common LISP [18], support more complicated features like default parameters and closures. Although these features are important, we do not think that they will be used often enough to have a lot of influence on the behavior of the programs.

1.3. The programs

We now describe the 11 programs that were used to study the behavior of LISP programs on MIPS-X. The number between parenthesis is the number of MIPS-X machine instructions in the compiled program.

- *inter*: a simple interpreter for a subset of LISP is used to calculate the Fibbonachi number 10, and to sort a list of numbers. The interpreter is adapted from "Lisp in Lisp" [24]. (1533)
- *deduce*: a deductive information retriever for a database organized as a discrimination tree. A number of rules are added to the tree, and a number of deductions are made. Adapted from [2]. (3419)
- *ded-gc*: the same program as *deduce*, but the heap has been reduced so that the copying garbage collector is invoked a number of times. The program spends about 50% of its time in the garbage collector. (4112)
- *rat*: a rational function evaluator that comes with the PSL system. (6315)
- *comp*: the first pass of the frontend of the PSL compiler. (9466)
- *opt*: the optimizer that was added to the compiler. It uses lists, and vectors with and without type and range checking. (11121)
- *frl*: a simple inventory system using the *frame representation language*. (11802)
- *boyer*: the boyer benchmark; a rewrite-rule-based simplifier combined with a dumb tautology-checker; benchmark published by R. Gabriel in [6]. (1793)
- *brow*: a short version of the browse benchmark; creates and browses through an AI-like database of units; benchmark published by R.Gabriel in [6]. (2296)
- *trav*: a short version of the traverse benchmark; creates and traverses a tree structure; uses structures with full type and range checking; benchmark published by R. Gabriel in [6]. (1673)
- *travf*: same as *trav* but without type and range checking. (1547)

2. Low Level Profiles

In this section we look at the behavior of the above PSL programs on the assembly level (that is, before the creation of delayed branch slots). We first look at the execution frequency of the different groups of assembly instructions, and we compare these numbers with similar numbers for Pascal and C programs on the MIPS architecture [9], which has an almost identical assembly language. Then we try to link the low level assembly level information back to high level operations in the LISP program.

2.1. Assembly Instruction profiles

Table 2-1 gives the instruction frequencies for the following instruction groups: register to register operations (alu), local branches with the frequency for conditional branches following in parenthesis, non-local jumps (used mainly for procedure calls and returns), and memory to register (load) and register to memory transfers (store).

For a group of large, optimized Pascal programs we found on average, 55% ALU operations, 30% load and store operations, and 15% branches and jumps. For a group of large C programs these numbers were 44%, 37% and 19% respectively.

We see that the branch frequency in LISP is substantially higher. As we will discuss later, this is mainly a result of the high frequency of procedure calls. We also observe a slightly higher load and store frequency in our LISP programs than in our suite of Pascal and C programs. The load/store frequency depends strongly on the quality of the register allocation though, and PSL has a very simple register allocation strategy. The high procedure call frequency makes it difficult to effectively use of the 32 registers in MIPS-X.

2.2. More about ALU Instructions

In Table 2-2 we give the frequencies for groups of related ALU instructions, relative to the total number of ALU instructions executed. The instructions in the bit and shift groups are used almost exclusively to handle tags. They constitute almost 50% of the ALU instructions!

The ratio of arithmetic instructions to bit and shift instructions for our set of Pascal programs is 3:1. For the C programs, more than 20% of the ALU instructions are connected with character handling, and the ratio decreases to 1.1:1. For the above LISP programs the ratio of arithmetic to bit and shift instructions is 0.7:1, what clearly shows that LISP programs spend their time on different operations than Pascal and C programs. Tag handling is definitely one important difference, but the low frequency of arithmetic instructions also suggests that LISP programs execute fewer arithmetic operations.

Over 50% of the add/sub instructions are add immediate instructions that are used to adjust the stack pointer during procedure calls (two adds per call). Pascal and C programs also use add instructions for this purpose, but the higher procedure call frequency in LISP (see section 2.4) would normally generate more add immediate instructions in LISP programs. The fact that LISP numbers actually show fewer add instructions indicates that our set of LISP programs contain very few explicit arithmetic operations.

	alu	branch(cond)	jump	load	store
inter	31.38	15.06 (12.83)	6.75	34.10	12.71
deduce	35.00	13.43 (11.02)	9.66	28.17	13.74
ded-gc	33.23	14.82 (12.67)	8.56	28.40	14.98
rat	42.63	14.16 (13.75)	10.68	22.27	10.26
comp	35.91	15.08 (13.25)	9.02	26.72	13.27
opt	38.62	19.78 (15.44)	6.63	27.15	7.82
frl	33.34	17.01 (13.25)	9.17	26.77	13.71
boyer	33.96	12.24 (11.47)	10.60	28.02	15.17
brow	35.02	11.93 (9.35)	6.26	32.81	13.97
trav	40.57	15.25 (13.38)	9.18	25.13	9.87
travf	32.01	14.11 (10.43)	10.44	33.41	10.03
average	35.61	14.81 (12.44)	8.81	28.45	12.32

Table 2-1: Assembly instruction frequencies

	add/sub	mul/div	bit	shift	move
inter	23.45	0.00	49.59	18.64	8.29
deduce	27.91	0.00	26.86	21.51	23.71
ded-gc	41.50	0.00	22.39	19.86	16.25
rat	40.21	7.62	17.87	16.37	17.92
comp	31.44	0.06	26.40	21.08	21.05
opt	25.61	0.00	34.54	25.48	14.37
frl	51.05	0.00	20.86	15.28	12.81
boyer	29.51	0.00	38.66	17.46	14.34
brow	25.10	0.03	42.86	13.71	18.28
trav	35.20	1.38	13.93	23.05	26.45
travf	35.08	3.66	33.80	6.06	21.40
average	33.28	1.16	29.80	18.05	17.72

Table 2-2: ALU instruction frequencies

The high frequency of *mov* instructions might be a surprise, considering that all MIPS-X ALU instructions take three operands. In PSL, procedures get most of their parameters in registers, and they always return the result in register 1. Most moves are used to rearrange parameters and results between procedure calls. The constant NIL is kept permanently in a register, and 24.5% of the moves are used to move NIL into another register, either to be used as a parameter, or to be returned as a result. A small fraction of the moves could be eliminated. The PSL intermediate language uses 2 operand instructions, and the compiler front-end sometimes generates a move to duplicate a register, so that its value will not be overwritten by a later operation. This move is not necessary at the assembly level, but it is not removed during code generation.

2.3. The target of load and store instructions

In this section we take a closer at what parts of memory are accessed by the load and store instructions. We distinguish 3 memory areas: the *heap*, the *stack*, which is used for local variables and return addresses, and a *global/user* area with fluid (global) user variables, constants, and system variables (e.g.: the heap pointer). The first 3 rows of table 2-3 show what percentage of the loads, stores, and memory references (loads + stores) access each area. The last row gives the geometric mean over all programs, of the load to store ratio in each of the 3 memory areas. The mean load/store ratio for all memory accesses is 2.34.

	heap	stack	user global
loads	23.96	47.76	28.28
stores	14.21	73.21	12.07
all accesses	20.09	55.37	24.54
load/store ratio	4.25	1.51	7.11

Table 2-3: Target of load and store instructions

Most stores go to the stack, and both the heap and the area with global data have a high load/store ratio. In a later section we study the accesses to the heap in more detail.

2.4. Branches

In Tables 2-4 and 2-5 we break up the jump and branch frequencies according to the source level operation they implement. The difference between the call frequency (4.94%) and the return frequency (3.62%) in the first two columns of Table 2-4 shows what fraction of the procedure calls have been converted into tail-recursive jumps, or tail-transfers [19] (1.32%, that is one out of every 4 calls). Tail-recursive calls that call the calling procedure itself don't show up in the call frequency column, because they are transformed into local branches by the front-end of the compiler. One in every 11.7 instructions executed is a non-local jump, so the average number of instructions executed between two jumps, each of which is a procedure call or return, is less than eleven instructions.

	call	ret	other
inter	3.97	2.69	0.09
deduce	5.22	4.42	0.01
ded-gc	4.86	3.70	0.01
rat	6.66	3.38	0.64
comp	5.07	3.94	0.01
opt	3.80	2.49	0.34
frl	4.60	3.76	0.79
boyer	6.32	4.28	0.00
brow	3.14	3.11	0.01
trav	4.87	4.04	0.28
travf	5.81	4.06	0.58
average	4.94	3.62	0.25

Table 2-4: Use of non-local jumps

Table 2-5 gives more information on pc-relative branches. One very interesting result is that almost 5% of all instructions executed, that is 20% of all branches and jumps, are conditional branches related to tag checking. Note also the large difference in tag checking frequency between trav and travf. The fairly high frequency of comparison with NIL, and of copying NIL into a register (see section 2.2) indicate that the allocation of NIL to a register is an important optimization, especially for a machine without versatile long immediates.

	cmp nil	tag check	other cond	uncond
inter	1.53	4.43	6.87	2.23
deduce	2.27	6.81	1.94	2.41
ded-gc	1.72	5.36	5.59	2.15
rat	1.03	4.99	7.73	0.41
comp	3.64	6.60	3.01	1.83
opt	1.29	7.45	6.70	4.34
frl	1.39	4.12	7.74	3.76
boyer	4.48	4.33	2.66	0.77
brow	2.17	3.24	3.94	2.58
trav	1.85	4.53	7.00	1.87
travf	4.00	0.66	5.77	3.68
average	2.31	4.77	5.36	2.37

Table 2-5: Use of pc-relative branches

2.5. Comparison with earlier work

A few papers have been published with dynamic profiling data of LISP. In [22], Urmi describes the design of a compiler for InterLisp. The compiler uses an intermediate language that is very close to LISP, and an interpreter was written to interpret this interm. diate language. The profiling information generated by the interpreter is on a higher level, but the results are very similar. The most frequently executed intermediate language instructions were: *load* and *setq* (they would translate into loads and stores on MIPS-X), *listp* (shift followed by conditional branch), car and cdr (logical and followed by a load), and procedure calls.

Foderaro measured the Franz LISP implementation of the algebraic manipulation system Macsyma [5]. Dynamic profiling information shows that *movl* is the most executed VAX instruction (implemented by load, store, and move in MIPS-X), followed by *cmpl*, *bnequ*, and *beqlu* (these correspond to the MIPS-X conditional branches), and arithmetic shift. They found that 3% of all instructions were procedure calls, and there were very few arithmetic instructions.

The SPUR microprocessor is described in [20]. The SPUR architecture is very similar to the RISC-II, but hardware was added to support the fast execution of Common LISP. Type checking is done in parallel with arithmetic operations, and a trap occurs if both operands are not integers. Some primitive operators, like *car* and *cdr*, are implemented in hardware. Multiple overlapping register windows are used to avoid memory accesses for locals. The assembly instruction frequencies for Gabriel's benchmarks on SPUR are: 16.9% loads, 7.7% stores, 43.8% ALU, and 28.1% branches and jumps. We notice a lower load and store frequencies than in the assembly instruction frequencies of Table 2-1. To account for the presence of the register file on SPUR, we subtract from the assembly frequencies for the MIPS-X, the loads and stores that access the stack, using the results of Table 2-3. This gives the following adjusted frequencies for MIPS-X: 46% ALU, 31% branches, 19% loads, and 4% stores. These numbers are almost equal to the numbers for SPUR.

SOAR [21] is a RISC processor especially designed for Smalltalk-80. Smalltalk also uses tags to store the type of variables, and SOAR has hardware support for tag handling. About 9% of the instructions executed on SOAR are tagged arithmetic instructions, that do automatic typechecking in hardware. The mean of the ratio of add and subtract instructions to bit and shift instructions for 4 macro-benchmarks is 3.2. This is the same ratio that we found for Pascal programs. This is not a surprise, considering all tag manipulation is done in hardware.

An overview of the literature shows non-local jump frequencies that are consistently lower than what we found for LISP on MIPS-X: 5.3% and 6.5% for bliss on the VAX [12], an average of 4.9% for a set of compilers written in basic, pascal, bliss, and pl/1 on the VAX [23], and 5.3% for mesa on the Dorado [4]. Measurements of VAX Macsyma [5], written in Franz LISP, show a jump frequency of 6%, but Franz LISP has a kernel written in C, so only part of the executed code is really LISP code.

The high jump frequency on MIPS-X is certainly unexpected: programs on RISC architectures are usually less dense, so one would expect lower procedure call frequencies. This suggests that the high procedure call frequency is typical for LISP programs. A comparison with the branch frequency for Pascal and C on the MIPS [9] confirms this observation: for Pascal and C, 1 out of every 7 branches is a call or return. For LISP this ratio is 1 out of 3.

The call frequency, not including tail transfers, on SPUR [20], another RISC executing LISP, is 3.5%. This is slightly lower than for our set of programs on MIPS-X. On SOAR [21], a RISC executing Smalltalk, the frequency of non-local jumps for 4 macro-benchmarks is on average 13.3%. This is extremely high, so high call and return frequencies seem to be common in exploratory programming languages.

2.6. Summary of low level profiling information

Figure 2-1 summarizes the results of this section. The total surface of the square corresponds to all instructions executed (100%). The surface of each rectangle shows for some instruction type, e.g. loads to the stack, what fraction of the total instruction count are instructions of that type.

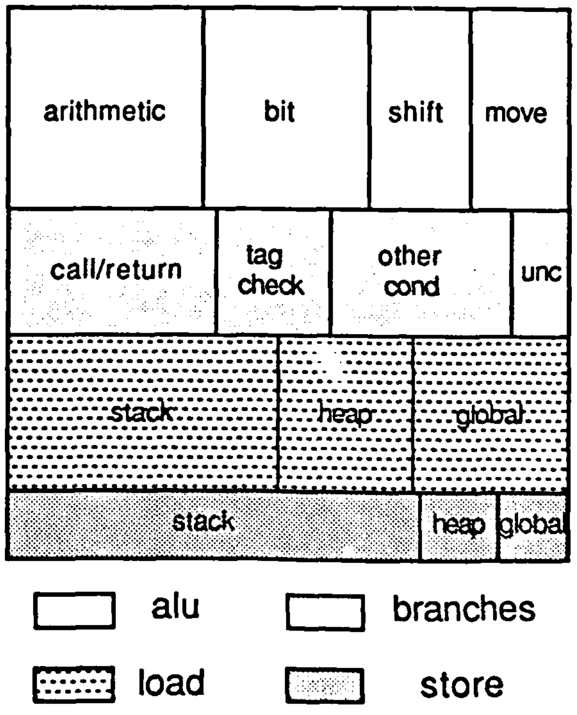


Figure 2-1: Overview of low level profiling

3. Cost of some important operations

In this section we look at the relative frequency of a number of source level operations, and at the fraction of the total execution time that is spent on these operations. All MIPS-X instructions are executed in a single cycle, but the cost of executing a load, store, or branch instruction can sometimes be higher. The result of a *load* instruction is only available in the second cycle after the load. The reorganizer tries to put an instruction that does not need the loaded value after the load, but if no such instruction can be found, a *no-op* (idle) instruction will be inserted. Similarly, no memory access instructions can be executed after a *store*, and an idle instruction may have to be inserted. The two instructions after *branches* and *jumps* are always executed before control is transferred to the branch target, and if no assembly instructions can be moved from before the branch or from the branch target, idle cycles are again inserted. In this section, *cost* always includes any idle cycles present immediately after load and store instructions and in any of the two cycles (statically) after branch instructions. The cost of an ALU instruction is always 1 cycle. We consider only cpu cycles in our cost measurements.

3.1. Time in the LISP system versus time in the user program

Table 3-1 gives the fraction of time that each program spends in procedures that are part of the LISP environment. Examples of such procedures are generic arithmetic routines, the cons function, vector operations with type checking, and list functions like append and reverse. We observe that LISP programs spend a substantial part of their time, 52% on average, in the underlying LISP system. Comparing trav with travf also shows that high *system time is* for a large part a result of the use of general operators, implemented as procedures that do type checking, in contrast with the use of fast, in-line operators.

inter	ded	de-gc	rat	comp	opt
46.8	36.1	69.8	49.7	61.1	67.1

frl	boyer	brow	trav	travf
74.1	29.9	64.3	51.5	21.2

Table 3-1: Time in LISP system procedures

3.2. Frequency and cost of list operations

Table 3-3 shows the number of list operations executed, normalized by the number of *cons* operations. The first line gives averages for these ratio's for the programs in our set. Because the amount of time each program spends on list operations varies strongly, the average has been weighted by the time spent on list operations. Clark [3] has done an elaborate study of the use of lists in LISP, and the average ratio's for his 3 programs are given on the second line. The results are very similar.

	cons	car	cdr	rplaca	rplacd
Our set	1	13.2	12.9	0.11	0.22
Clark's set	1	9.2	9.7	0.13	0.52

Table 3-3: Comparison of frequency of list accesses

In Table 3-2 we give the fraction of the total execution time that was spent on each individual list operation. The allocation of space is done with a procedure call, and requires 19 cycles. This cost does not include the cost of garbage collection. *Car* and *cdr* require 2 or 3 cycles: one cycle to mask out the tag, one for the load, and 75% of the time an idle cycle to wait for the result. *Rplaca* and *rplacd* require the masking of the tag, a store and, in 50% of the cases, an idle cycle. In the case of car and cdr, the cycle after the load can

	cons	car	cdr	rplaca	rplacd	total
inter	19.21	17.78	10.02	0.01	0.02	47.05
deduce	11.07	9.52	9.62	0.00	0.37	30.58
ded-gc	4.77	5.44	4.80	0.76	0.16	15.93
rat	15.13	8.91	5.04	0.01	0.01	29.10
comp	12.01	9.59	5.71	0.16	0.13	27.61
opt	3.79	14.56	13.52	0.10	0.09	32.07
frl	9.60	6.19	6.39	0.01	0.10	22.29
boyer	21.72	14.33	8.81	0.00	0.00	44.86
brow	21.92	13.22	14.03	0.00	2.04	51.21
trav	0.27	3.39	3.87	0.01	0.02	7.55
travf	0.55	6.96	7.94	0.03	0.06	15.54
average	10.91	9.99	8.16	0.11	0.27	29.44

Table 3-2: Cost of list operations in percent of all cycles

only be used 25% of the time because of car-cdr chains; when there are several car and cdr operations in a row, it is often hard to find an unrelated instruction to execute after all but the first load in the sequence.

In the program opt, a special effort has been made to avoid (unnecessary) allocation of cons cells. This shows up clearly in the cons column.

In PSL, the allocation of cons cells is done with a procedure, which results in a fairly large cost. Keeping copies of the heap pointer in a register, with a 'pair'-tag already inserted, and doing the allocation of pairs in-line would bring this cost down from 11% to about 3% of the execution time, with no increase in program size. The price would be a very small increase in the cost of allocating space for other data types, and the loss of 2 registers for user data. An additional advantage would be fewer procedure calls, which may allow further optimization.

PSL does not do any type checking on list operations. Adding type checking increases the execution time of our programs by 14.6% on average. Some of these tests could be eliminated by an optimizing compiler that uses information available in the program to determine the type of variables, or by type declarations.

3.3. Cost of vector operations

Table 3-4 gives the fraction of the execution time spent on vector operations. In opt, some vector operations are done with type and range checking (line 1), and some without (line 2), and trav has been executed once with and once without checking (travf).

	type check	alloc vector	vector read	vector write	total
opt	y	1.23	4.45	9.67	15.39
	n		0.03	0.01	
trav	y	0.02	33.26	4.75	38.03
travf	n	0.05	2.85	0.44	3.34

Table 3-4: Cost of vector operations in percent of all cycles

Comparing the entries for trav and travf clearly shows that runtime type and range checking can be very expensive, and the real cost difference is even larger. Because vector accesses with checking are done with procedure calls and accesses without checking are done in-line, there are more procedure calls in the user part of the program in trav than in travf. The high frequency of calls decreases the effectiveness of the register allocator and it inhibits many optimizations. In trav and travf, for example, several vector accesses occur in the same basic block. In travf, the compiler detects this, and the masking of the tag in the vector item is done only once at the beginning of the basic block. This is not possible in trav, because the vector operations are done with procedure calls.

3.4. Cost of some other operations

Table 3-5 summarizes the cost of a number of additional LISP operations. We see that on average almost 25% of all time is spent doing procedure calls and returns. This includes only the cost of the jump and of maintaining the very simple stack frame. The cost of putting the parameters in place is not included. The cost of procedure calls might very well be even higher in LISP dialects with a more complicated procedure call convention.

	call	ret	jcall	nilcmp	arith	tagch
inter	8.81	6.79	2.99	1.48	3.68	10.75
deduce	14.57	13.18	1.90	2.01	0.44	15.02
ded-gc	11.47	9.80	1.95	1.65	0.19	11.98
rat	10.17	8.18	6.28	0.82	24.82	13.17
comp	12.59	11.11	2.39	2.96	0.36	16.07
opt	7.94	6.74	2.46	1.04	11.91	15.69
frl	11.48	11.18	1.66	1.06	2.24	9.35
boyer	13.91	12.01	4.20	3.64	0.00	11.44
brow	10.62	8.12	0.06	1.85	0.28	7.68
trav	12.20	11.77	2.55	1.41	8.77	12.20
travf	11.89	11.92	3.17	2.89	18.01	2.02
average	11.42	10.07	2.69	1.89	6.43	11.40

Table 3-5: Cost of some important LISP operations

Generic arithmetic operations are very expensive: adding two numbers in a register costs only 1 cycle, but the cost increases to 59 cycles when type and overflow checking is done. This can become very expensive for arithmetic programs (e.g.: rat). One way to reduce this cost is to use the fast, unchecked operators that are provided by PSL. For a program like tak [6], this optimization speeds up the program by a factor of 6.3. Using unchecked integer operations is only possible if the programmer is (feels) sure that all numbers involved are small integers. This is not always possible, and errors introduced by an undetected overflow can be very hard to find. A more attractive solution is to speed up the generic operations for the most frequently used datatype (integer), by doing a fast, specific test for this type before going through the expensive general test.

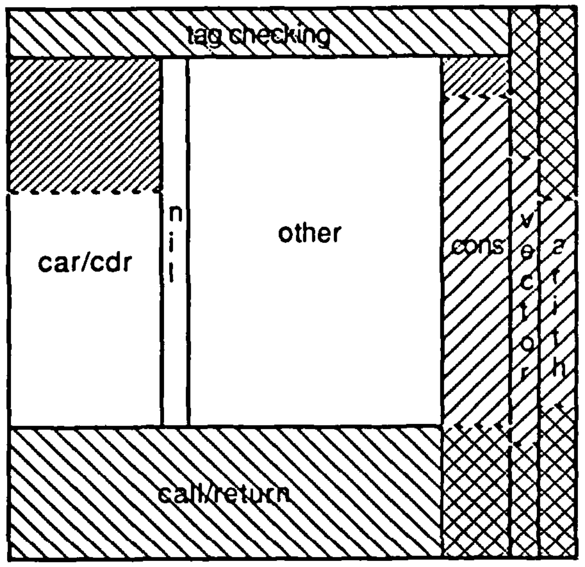
Our set of LISP programs spend an average of 11.4% of their time checking the tag of data items. This number includes both the cost of extracting the tag and the cost of the conditional branch with possibly empty delayed slots. To determine the cost of tag handling we should add to this the cost of inserting and removing the type tags. These operations are done using logical bit operations (see table 2-2) so the total cost of tag handling becomes about 22% of the total execution time. Type checking does not only involve tag handling, but also dispatching on the arguments and, for PSL, procedure calls, so the cost of doing runtime type checking is slightly higher. It is this cost that LISP machines try to reduce with special hardware or micro-code for tag handling [1, 14].

It is not really correct to count the full cost of tag handling as runtime type checking cost, because not all tagcheck operations are real type checking operations. An example is the implementation of the function append. To detect the end of the first list, it checks whether the cdr of the successive cells is a pair. Although this is technically a type checking operation (taking the car or cdr of an atom has to be avoided), it is different from the type checking required for the operands of the addition operator. Compile-time declarations can eliminate the type checking in arithmetic operations, but they cannot eliminate the test in the append function. If no runtime checking were available, this test would have to be replaced by some other test to detect the end of the list. This shows that even with compile time declarations, the efficient checking of tags will remain important.

3.5. Summary

Figure 3-1 gives an overview of the cost of primitive LISP operations, averaged over the 11 programs. The surface of the square represents the total execution time, and the surface of each rectangle shows how much time was spent on each operation. Certain areas are overlapping. For example, the right most columns shows the how much time is spent on generic arithmetic operations, and it also indicates how much of that time is used for procedure calls and tag checking.

"nil" stands for comparison with nil, plus all *mov nil* instructions. The contribution of he rplaca and rplacd operations is so small that it should be represented by a line. The exact form of the graph depends strongly on the program.



▨ fraction of time of car/cdr and cons that is spent on tag handling

▩ fraction of time of cons, vector and arith that is spent on call/return and on tag checking

Figure 3-1: Overview of the cost of LISP operation

4. More about procedure calls

LISP programs are reputed to be very procedure call intensive, and to contain a lot of small, recursive procedures. The first claim was supported by our benchmarks in section 2.4. To collect data to verify the second claim, we changed the VAX compiler for PSL to insert before each procedure call a statement that prints out trace information. We recompiled and rebuilt the PSL system, and we recompiled and ran our 10 programs (*dcd-gc* was excluded because it is not possible to rebuilt PSL with a small enough heap). The resulting trace files, together with static information about the programs, were then used to generate tables 4-1 and 4-2.

The procedure size in column 3 of Table 4-1, is expressed in lines of intermediate code. Each intermediate instruction expands on average to 1.1 assembly instructions. We see that procedures pass an average of less than 2 parameters, and allocate fewer than 2 words on the stack (plus one word for the return address). The average procedure size is about 28 assembly instructions.

	average parameter count	average frame size	average procedure size
inter	2.0	1.6	25.9
deduce	1.6	1.5	14.2
rat	2.0	1.3	26.9
comp	1.8	2.0	22.8
opt	2.1	1.4	32.3
frl	1.9	1.1	18.8
boyer	1.8	1.2	20.9
brow	1.6	1.2	16.3
trav	2.2	3.6	24.7
travf	1.9	2.2	50.8
average	1.9	1.7	25.4

Table 4-1: Procedure properties, weighted by call frequency

Table 4-2 shows that 23% of the calls are recursive, but this number is reduced to 14% if we exclude the four Gabriel benchmarks. A call from procedure A to procedure B is recursive, if procedure A *can* be reexecuted before the call returns. This is determined at compile time, so paths through functions like *eval* do not count. Only three of the programs use *apply* and *eval*, and of these three only *frl* uses *eval* extensively.

	recursion	apply/ eval
inter	26.7	-
deduce	16.3	-
rat	19.3	0.5
comp	12.0	1.1
opt	3.2	-
frl	4.7	2.4
boyer	51.8	-
brow	7.0	-
trav	17.3	-
travf	70.7	-
average	22.9	-

Table 4-2: Procedure call properties

5. Conclusion

In this paper we looked at how LISP programs behave, both at the assembly level, and at the source level. We compared dynamic profiling information for a number of PSL programs, with profiling information for Pascal and C programs, and we observed a larger branch and jump frequency for our set of PSL programs. We also noticed a substantial shift from arithmetic instructions towards logical bit operations and shifts. By linking our low level information to LISP level operations, we found that LISP programs spend about 25% of their time doing procedure calls, and that they spend another 22% of their time on operations related to tag handling and tag checking. We also noticed that the use of generic arithmetic and vector operations can be very expensive and that specific operations, without type checking should be used if possible.

Acknowledgments

We thank the members of the MIPS-X group for their help and suggestions. We are especially grateful to Scott McFarling and Steve Tjiang, whose work on the reorganizer, the linker and the loader made our work feasible. We also want to thank Thomas Gross for his many contributions and his never ending enthusiasm throughout this project.

References

1. Bawden, A., et al. LISP Machine Progress Report. Memo No 444, MIT Artificial Intelligence Laboratory, August, 1977.

2. Charniak, E., Riesbeck, C. K., and McDermott, D. V.. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1980.

3. Clark, D. "Measurements of Dynamic List Structure Use in Lisp". *IEEE Transactions on Software Engineering 5*, 1 (January 1979), 51.

4. Clark, D.W., and Levy, H.M. Measurements and Analysis of Instruction Set Use in the VAX-11/780. Proceedings of the 9th Symposium on Computer Architecture, April, 1982, pp. 9-17. Also published as SIGARCH Newsletter, 10(3), April 1982.

5. Foderaro, J. and Fateman, R. Characterization of VAX Macsyma. Proc. 1981 ACM Symposium on Symbolic and Algebraic Computation, Aug, 1981, pp. 14-19.

6. Gabriel, R.P.. *Computer Systems Series*. Volume : *Performance and Evaluation of Lisp Systems*. The MIT Press, 1985.

7. Griss, M.L, Benson, E., and Hearn, A. Current Status of a portable LISP Compiler. Proc. SIGPLAN '82 Symposiuum on Compiler Construction, SIGPLAN, Boston, June, 1982, pp. 276-283.

8. Griss, M.L., Benson, E., and Maguire, G.Q. PSL: A Portable LISP System. Proc. 1982 Symposium on LISP and Functional Programming, August, 1982, pp. 88-97.

9. Hennessy, J., Gross, T., Rowen, C., and Przybylski, S. Measurement and Evaluation of the MIPS architecture and Processor. Stanford University, February, 1986.

10. Hennessy, J.L., Jouppi, N., Baskett, F., and Gill, J. MIPS: A VLSI Processor Architecture. Proc. CMU Conference on VLSI Systems and Computations, October, 1981, pp. 337-346.

11. Horowitz, M., and Chow, P. The MIPS-X Microprocessor. Westcon 1985, November, 1985.

12. McDaniel, G. An Analysis of a Mesa Instruction Set Using Dynamic Instruction Set Frequencies. Symposium on Architectural Support for Programming Languages and Operating Systems, August, 1982, pp. 167-176.

13. McFarling, S. and Hennessy, J. Reducing the Cost of Branches. Proceedings of the Thirteenth Symposium on Computer Architecture, June, 1986.

14. Moon, D.A. "Architecture of the Symbolics 3600". *SIGARCH Newsletter 13*, 3 (June 1985). Also publushed in Proceedings of the 12th Annual Symposium on Computer Architecture, June 1985, Boston.

15. Patterson, D.A. and Sequin, C.H. "A VLSI RISC". *Computer 15*, 9 (September 1982), 8-22.

16. Ponder, C. ... but will RISC run LISP?? UCB/CSD 83/122, University of California, Berkeley, August, 1983.

17. Radin, G. The 801 Minicomputer. Proc. SIGARCH/SIG PLAN Symposium on Architectural Support for Programming Languages and Operating Systems, ACM, Palo Alto, March, 1982, pp. 39 - 47.

18. Steele, G. L. Jr.. *Common Lisp - The Language*. Digital Press, 1984.

19. Steele, G. L. Jr., and Sussman, G. J. LAMBDA : The Ultimate Imperative. AI Memo 353, MIT, Artificial Inteligence Laboratory, March, 1976.

20. Taylor, G.S., Hillfinger, P.N. et.al. Evaluation of the SPUR Lisp Architecture. Proceedings of the Thirteenth International Symposium on Computer Architecture, ACM, Tokyo, June, 1986.

21. Ungar, D. *The Design and Evaluation of A High Performance Smalltalk System*. Ph.D. Th., UC Berkeley, March 1986. Tech Rep UCB/CSD 86/287.

22. Urmi, J.. *Linkoping Studies in Science and Technology Dissertations*. Volume 22: *A machine independent Lisp compiler and its implications for ideal hardware*. Linkoping University, Linkoping, Sweden, 1978.

23. Wiecek, C.A. A Case Study of the VAX-11 Instruction Set Usage For Compiler Execution. Symposium on Architectural Support for Programming Languages and Operating Systems, August, 1982, pp. 177-184.

24. Winston, P. and Horn, B.. *Lisp*. Addison-Wesley Publishing Company, 1981.

END

4-87

DTIC