A three-processor Lisp machine architecture based on statistical analysis of Common Lisp programs

Arno J. Klaassen Anton M. van Wezenbeek

Delft university of technology
faculty of electrical engineering
P.O. box 5031, 2600 GA Delft, the Netherlands
Phone: +31-15-786177 Telefax: +31-15-783622
Email: arno@duteca.UUCP

Abstract

A package has been written for statically analyzing Common Lisp source code programs. The analysis was performed on source code of the implementation of the Common Lisp dialect Spice Lisp and indicated which parts of Lisp are used often and therefore are important to be implemented such that they run fast. Based upon the identification of those parts we sketch the outlines of a three-processor Lisp machine architecture.

1 Introduction

Two main reasons for analyzing Lisp can be distinguished.

- Analyzing is primarily done for sake of efficiency. Efficiency falls apart into time efficiency and space efficiency. Time efficiency covers the wish to execute each instruction as fast as possible within a fixed maximum amount of time. Space efficiency covers the wish to keep program size as small as possible. Time and space efficiency are often in conflict. A common known example of such conflict is the dilemma between fast function calling and fast variable lookup [*Gabriel '82*] [*Pleszkun & Thazkuthaveetil '87*].
- The second reason is that Common Lisp is very large. Common Lisp has an exuberant amount of luxury accessories all of which can not be easily supported by the underlying Lisp machine. For example the use of keywords causes a substantial overhead on function calling either always or just never used.

In Section 2 important results are presented which lead to the architecture as described in Section 3. Finally in Section 4 conclusions and recommendations to further work are given.

2 Results

We used program source code for all our analysis. So also our data types come from program source code. This is possible in Lisp because program and data are represented in the same (list) form which results in the 'program-equals-data' paradigm. Comparable results can be found in [*Clark* '76,'79] [*Gabriel et. al.* '82] [*Gabriel* '85] [*Sansonnet et. al.* '82] [*Steenkiste & Hennessey* '86]. To verify this paradigm the following measurements on the Spice Lisp compiler were performed:

1. Determining the statically most often used primitives of the compiler, the interpreter and the editor.
2. Analyzing the dynamic use of twenty of these primitives, as well as the use of data types and the number of elements per list and sublist. Five dynamic measurements were performed:

- One measurement during loading the complete compiler.
- Four measurements during actually compiling four increasingly complex, but nevertheless very simple Lisp programs.

	static	*load*	*run-1*	*run-2*	*run-3*	*run-4*
car	777	1594	59	261	1211	4516
cdr	537	1713	59	172	942	3547
quote	9265	45317	745	2341	7066	26418
defun	764	449	0	0	0	0
if	775	8543	136	589	2226	9325
setq	864	2638	84	276	1829	6832
let	445	2083	50	177	564	2140
setf	1309	190	0	2	36	67
list	343	94	1	5	22	45
get	957	563	26	151	490	2278
progn	672	2850	89	230	1435	6122
eq	630	4311	12	57	295	1017
when	145	376	33	59	207	1238

Table 1: *Measured use of primitives*

	static	*load*	*run 1*	*run 2*	*run 3*	*run-4*
integer	4639	9533	321	642	2323	10140
ratio	0	0	0	0	0	0
float	0	0	0	0	0	0
complex	0	0	0	0	0	0
list	42245	50259	788	2522	10266	38206
char	24	0	2	2	0	14
string	16236	344	160	203	185	3703
vector	0	0	2	4	58	70
array	0	0	0	0	0	0
package	0	13	1	2	4	10
symbol	51005	124624	1877	5978	21485	86014

Table 2: *Measured use of data types*

As benchmarks the source code of Spice Lisp has been used. The following three modules are considered. First the *compiler* which adds up to 1,5M of source code and generates lispy assembler code. Second the *editor* which also adds up to about 1,5M of source code and at last the *interpreter* which serves as the core of the system and adds up to 2,7M of source code.

In table 1 the results of analyzing the sorted primitives can be found. In table 2 the results of analyzing data types are listed. Finally in table 3 the number of elements per list and sublist is presented. In each of these tables the column labeled *static* refers to the statically measurements, the column labeled *load* refers to the measurements during load phase, and the columns labeled *run-1* till *run-4* refer to consecutively performed dynamic measurements. Comparing the static and the dynamic results of table 1 2 and 3, a conclusion must be that our static analysis results are meaningful even for dynamic estimations. All analysis results have been bundled in one report [*Klaassen & van Wezenbeek '87*] [1]. Table 4 shows occurrences of all data types. On the premise that program equals data, it seems that the data types integer, string [2], list and symbol are used most often. Table 5 shows the absolute and relative occurrence of the most often used arithmetic operations. It's obvious that simple arithmetic is performed the most often.

[1] This report contains histograms and tables. The tables and figures in this paper are either directly got from the report or contain results from different tables we thought useful combining

[2] The high occurrence of strings is partly a result of the print name of symbols. For example '(a b c) will result in three implicitly present strings.

	static	*load*	*run-1*	*run-2*	*run-3*	*run-4*
0	211	19959	1044	17094	20625	61775
1	2087	23390	461	21771	12552	35184
2	10569	134674	1121	60492	52057	351751
3	10531	51166	1119	58262	45560	321863
4	4399	19012	156	3754	1980	20260
5	2188	5487	76	6855	6440	45522
6	880	1837	17	106	114	838
7	719	703	3	87	113	391
8	384	368	0	61	168	507
9	174	220	0	53	152	319
10	0	103	2	82	124	372
11	0	143	3	53	120	325
12	0	13	1	70	86	308
13	0	53	5	65	124	247
14	0	53	0	48	100	254
15	0	9	0	91	60	198
.						
.						
.						
48	0	4	0	21	15	51
49	0	4	0	20	22	52

Table 3: *Number of elements per list*

data type	*interpreter*		*editor*		*compiler*	
	abs	rel	abs	rel	abs	rel
integer	2216	1.84	1074	1.65	4639	4.06
ratio	2	0.00	0	0.00	0	0.00
float	88	0.07	0	0.00	0	0.00
complex	0	0.00	0	0.00	0	0.00
bitvector	0	0.00	0	0.00	0	0.00
vector	0	0.00	0	0.00	0	0.00
string	7378	6.12	3494	5.36	16236	14.22
list	44205	36.65	23576	36.17	42245	37.01
symbol	66257	54.94	36308	55.70	51005	44.68
character	446	0.37	606	0.93	24	0.02
array	0	0.00	0	0.00	0	0.00
structure	0	0.00	0	0.00	0	0.00
unreadable	15	0.01	122	0.19	3	0.00

Table 4: *Absolute and relative occurrence of data types*

3 Architecture proposal

Restating our primary results:

- The primary data types should be merely "list", "symbol" and "integer".
- Only simple integer arithmetic should be supported, although those integers have to have possibly any size, i.e. either fixnum or bignum.
- Both symbol and list implementation should be fast compared to the sequential manipulations on conventional (Lisp) machines, preferably by exploiting as much inherent parallelism as possible.

These considerations together with the work of [*Pleszkun & Thazhuthaveetil '86,'87*], [*Potter '83,'85*] and [*Keller '80*] led to the following suggestions:

- The architecture should contain at least two processors. One processor, henceforth called the "eval processor", handles program control and operates on virtual data. The other processor, henceforth called "data processor" does the function application and operates on real data [3].

[3] To exploit fully the possibilities of two processors, some lazy evaluation scheme should be used.

opeartion	*interpreter*		*editor*		*compiler*	
	abs	rel	abs	rel	abs	rel
= / = <						
> >= <=	751	22.22	333	29.54	465	33.00
+	310	9.17	170	15.08	208	14.76
−	363	10.74	236	20.94	129	9.16
*	320	9.47	59	5.24	46	3.26
/	96	2.84	4	0.35	35	2.48
1+	430	13.02	132	11.72	159	11.29
1−	211	6.24	122	10.83	74	5.25
Total	2481	73.70	1056	93.70	1116	79.20

Table 5: *Absolute and relative occurrence of arithmetic operations*

- The interface between the eval processor and the data processor should be of such a construction that various data type implementation models can be experimented with, while the eval processor remains unchanged. The first thing to experiment with are parallel list implementation models.
- System data such as the environment(s) and the oblist, should also be handled by the data processor, in the same way as ordinary user data. They should preferably be implemented as parallelized lists.

Name	Car	Cdr	
L0	L1	L2	≡ (+ 1 2)
L2	L3	L4	≡ (1 2)
L4	L5	nil	≡ (2)

Figure 1: *The virtual list table*

Parallel list implementation models form a corner stone of our architecture. However, further analysis of such models [*Klaassen & van Wezenbeek '87*] brings to the fore that the efficiency of such models decreases dramatically when list structures deviate from their optimal configuration [4], something which always happens when a run-time system manipulates lists.

Close observation of this phenomenon gives rise to a third processor, henceforth called "linearization processor", thereby indicating that the optimal configuration of a list most often is the most linear one. This linearization processor should work concurrently with the data processor. They should have simultaneous access to the heap. Since linearizing list structures most naturally implies copying, the linearization processor implicitly serves as concurrent garbage collector.

Now in principle we have recorded a tidy base for the architecture of a Lisp machine. However, in the beginning of the section the importance of fast list manipulation is stated. Recognition of the fact that once a list is accessed, often the access of its car and/or cdr is only a matter of time, an optimization can be made, as indicated by [*Pleszkun & Thazhuthaveetil '86*].

According to our virtual data principle, we introduce a virtual list table. This table contains tuples of virtual data (see figure 1). The first field indicates a list item and so does the third, being the cdr of the first field. The second field indicates the car of the first. In this way it is possible for the data processor to supply the eval processor three virtual data entities at once. This allows lazy evaluation and decreases the communication traffic between the eval processor and the data processor. All above given considerations lead to an architecture as given in figure 2.

[4] This optimal configuration depends on the specific model.

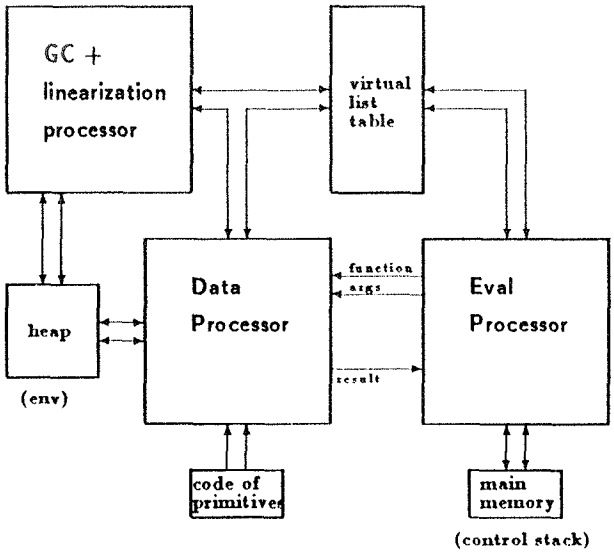

Figure 2: *Architecture of a kernel Lisp machine*

4 Conclusions and recommendations

The algorithm and consequences of concurrent garbage collection to the heap architecture have to be figured out. The semantic consequences of lazy evaluation and the overhead it may impose on the interprocessor communication are not clear. Further research has to be done on parallel list implementation models. Our preference goes to either structure codes such as the Cdar code [*Potter '83,'85*] or vector code such as the Conc code [*Keller '80*]. Some list manipulation primitives are tree based, rather than list based, due to the most common two pointer list implementation model, which is tree based. However, parallel list implementation models most often are really list based. This means that it is very inefficient to support tree based primitives such as rplacd and append with the same semantics as they have always had. We believe that their semantics should be changed into a more applicative direction [5].

5 References

- [*Bobrow & Clark '79*]
  D.G. Bobrow, D.W. Clark *"Compact encodings of list structure"*
  ACM Transactions on programming langguages and Systems, Vol. 1, No.2, October 1979, Pages 266-286
- [*Clark '76*]
  D.W. Clark *"List structure: measurements, algorithms and encodings"*
  department of computer science, Carnegie Mellon university, Pittsburgh august 1976
- [*Clark '79*]
  D.W. Clark
  "Measurements of dynamic list structure use in Lisp"
  IEEE, /79/0100-0051, January 1979
- [*Gabriel et al. '82*]
  R.P. Gabriel, L.M. Masinter *"Performance of Lisp systems"*
  ACM, 82/008/0123, 1982

[5] Side effects can *not* be parallelised

- [*Gabriel '85*]
  R.P. Gabriel *"Performance and evaluation of Lisp systems"*
  the MIT press, Cambridge, Massachusetts, 1985

- [*Keller '80*]
  R.M. Keller
  "Divide and CONCer: data structuring in applicative multiprocessing systems"
  ACM, Lisp conference 1980

- [*Klaassen & van Wezenbeek '87*]
  "Analysis result of Spice Lisp"
  Delft university of technology, department of electrical engineering, 1987

- [*Pleszkun & Thazhuthaveetil '87*]
  A.R. Pleszkun, M.J. Thazhuthaveetil *"The architecture of Lisp Machines"*
  Computer IEEE, /87/0300-0035, March 1987

- [*Potter '83*]
  J.L. Potter
  "Alternative data structures for lists in associative devices"
  IEEE, /83/0000/0486, August 1983

- [*Potter '85*]
  J.L. Potter
  "List based processing on the MPP"
  pages 124-140 from "The massively parallel processor",MIT press 1985

- [*Sansonnet et al. '82*]
  J.P. Sansonnet, M. Castan, C. Percebois, D. Botella, J. Perez *"Direct execution of lisp on a list-directed architecture"*
  ACM, 82/03/0132, 1982

- [*Steenkiste & Hennessey '86*]
  P. Steenkiste, J. Hennessy *"Lisp on a Reduced-Instruction-Set-Processor"*
  ACM, /86/0800-0192, 1986