H. Melenk W. Neun

Portable Standard LISP for Cray X-MP Computers

Preprint SC 86-2 (Dezember 1986)

Konrad-Zuse-Zentrum für Informationstechnik Berlin;
Heilbronner Straße 10; D-1000 Berlin 31

Portable Standard LISP for Cray X-MP Computers

Herbert Melenk
Winfried Neun

Konrad-Zuse-Zentrum für Informationstechnik Berlin

Cooperative Project of ZIB Berlin and Cray Research

10th Bochum Colloquium on Supercomputers and Applications
December 1986

Abstract

Portable Standard LISP (PSL) is a portable implementation of the programming language LISP constructed at the University of Utah. The version 3.4 of PSL was implemented for Cray X-MP computers by Konrad-Zuse-Zentrum Berlin; this implementation is based to an important part on the earlier implementation of PSL 3.2 at the University of Utah, Los Alamos National Laboratories and Cray Research Inc. at Mendota Heights.

During the work on implementing PSL the language LISP was investigated for areas which can be supported by vector hardware. One area was found in the COMMON LISP sequence functions and some typical application areas of LISP programming can be improved by vector processing too. A model for the implementation of vector instructions in LISP was developed. For arithmetic an experimental vectorizing extent of the PSL compiler was constructed. With this means full vector hardware capacity can become available for LISP applications.

The topic is discussed with full length in the paper /7/.

The work of implementing Portable Standard LISP (PSL) version 3.4 for Cray-1 and Cray X-MP computers started in December 1985 as a cooperative project of Konrad-Zuse-Zentrum in Berlin and Cray Research and has led to a first distributable release, named "Release 5" now. The present implementation is a continuation of the work started in 1981 at Salt Lake City /1/ and Los Alamos implementing PSL 3.2 for Cray computers running CTSS, and at Mendota Heights for COS systems. The COS version of PSL 3.2 was upgraded to a complete cross compiling system and was the basis for implementation of PSL 3.4.

1. Implementation of PSL 3.4

Apart from the general difference between PSL versions 3.2 and 3.4 the present version differs from its predecessor by the following aspects:

Security

Provisions were taken to **protect** the running system against **overflow** of one of the stacks. The **reprieve handling** was enlarged in order to continue LISP after system interrupts whenever possible. Many important **diagnostic features** giving information about problem cause and location and for analysing a running system on LISP user level are incorporated.

Usability

Since the PSL version 3.4 was built on smaller machines with virtual memory and a completely different operating system environment, the version 3.4 for COS had to be adapted to the Cray COS environment and the real memory hardware of CRAY-1 and X-MP machines. A **dynamic memory management** was incorporated into PSL; all relevant portions of LISP memory can be enlarged and shrinked on request; the target area for compiled code and the heap are **enlarged automatically** in case of lack. The **access path** scheme for files was extended: local files can be processed as well as permanent datasets residing under the tasks OWN or a foreign ownership. Because of the wide spread usage of batch operation in the Cray world a **batch adapted behaviour** was implemented for PSL if running in batch mode (e.g. generating standard answers for system questions, including a bracket counter into the automatic echo printing of input). Because of the absence of a directory hierarchy in COS and in order to increase speed and decrease the number of files when loading compiled modules a **library facility** based on word addressable files was integrated into PSL. The COS **command language statements** for dataset management and dataset staging control are available in PSL via the dataset management subprograms of the COS library and a LISP specific interface. Further COS library programs may be linked into the system. There is a general utility for the linkage of CFT subroutines to LISP functions but because of the lack of a dynamic loader for COS this cannot be done dynamically.

Completeness

All LISP language features and most utilities described in the PSL manual /2/ are available for Cray PSL. Some Cray or COS specific utilities (e.g. a disassembler) have been created to replace system dependent utilities of other implementations. The tools for enlarging a local PSL version are included. PSL is completely self bootstrapping on a Cray computer without need for foreign machine execution.

Execution Speed

One of the major design goals of Portable Standard LISP is to build LISP systems with high speed in execution. Benchmarks with different LISP implementations on the same hardware show that PSL is one of the fastest LISP systems. Therefore it seemed to be reasonable to build a Cray LISP based on PSL. By further optimizations with respect to the Cray-1 and X-MP hardware architecture the execution speed of the Cray PSL 3.4 version was augmented significantly. These optimizations have led to a very fast LISP implementation:
A. Hearn's REDUCE Test /3/ consumes 1.0 seconds CPU time on a Cray X-MP, which is one of the best results known for this test.

Some optimizations in detail

Applications running on Cray systems /5/ get significantly faster if one can decrease the number of memory accesses, especially the blockstore and blockload instructions for B and T registers, and the number of reloads of the instruction stack. This is different to other hardware which has a memory cache and doesnt have a pipelined instruction issue. To optimize the code produced by the compiler there are several techniques supported by PSL:

Compiler features for vectorization (see below) will be distributed with the Portable Common Lisp Subset (PCLS) from U of Utah working on top of PSL. On system level **vector instructions** are known to the compiler now and are used by handcoded routines (LAP level), which gain further operation speed in spite of very small vector length. Using vector instructions an unusual stack technique can be used for recursive functions such as equal, which avoids usage of "normal" stack in almost all cases.

The heavily used function **cons** (the LISP workhorse), some arithmetic functions, the string and hashtable manipulation functions and binding functions are compiled **inline.**

Many global **control variables** are held in **T or B registers;** their values are treated correctly in interpreted code too.

The **stack frame** technique was **refined,** to avoid unnecessary saving of the stack frame or to give up the frame as soon as possible.

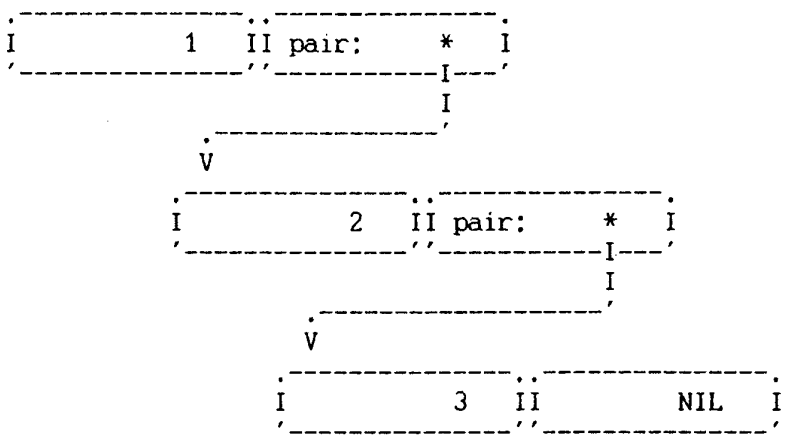
A **new compiler pass** called LapOpt was integrated into the compiler to reconfigure the produced code of the original compiler passes. This was done because in Cray code many valuable informations are residing in auxiliary registers (e.g. A registers) not known to the portable first passes of the compiler. This problem is unknown on other hardware. **Memory access** operations are **deleted** in many cases by this new compiler pass or at least the load operations are issued as early as possible.

FORTRAN calls were **removed** from arithmetic and string manipulation completely.

Some **system routines** heavily used were **reformulated** adapted to the code production strategy of the compiler or were **coded by hand** using the LAP language level.

2. LISP Processing

One major invention of LISP was the definition of the pointer to be the source of all luck. The standard data structure is the (pointered) binary tree with the important subcase of the linear list, which contains in each node one data element and one pointer to the successor.



Picture 1: Representation of the list (1 2 3) with LISP pointer technique; each box represents one word in memory and the links are tagged address references.

The processing of the linked structure requires the traversing of links. There is no simple formula which allows a quasi parallel (vectorizable) access to all elements of a structure because the address of node n+1 is contents of node n. Even worse, in general the type of data elements is not konwn in advance. LISP data carry their typing information with them and many algorithms operate "data driven": they determine their flow of control dependent of the data types or data values found in a structure. All these aspects **prevent a general vectorization approach** for LISP.

So vectorization remains a partial work within the LISP context. Of course, **arithmetic** and **binary logic** can take part. Especially for this area COMMON LISP offers with the function "**map**" in the **sequence** context a fascinating language interface for a very natural and transparent formulation of vectorized algebra. Target areas of vectorized LISP may be e.g. graphics (bit map handling, coordinate transformations), but even complex mathematical algorithms may be involved since this type of numeric calculation is no longer "very slow" compared to FORTRAN. An experimental macro compiler producing Cray vector instructions has been written. It opens the full calculating power of the vector computer for LISP applications.

3. Data Type SEQUENCE: The LISP Language Vector Interface

Already some early LISP implementations offered vectors or arrays additional to lists and trees. But there were no high level vector or matrix operations comparable to the well developed list and tree processing capabilities or these operations were too much implementation dependent. So from the historical (and today software) viewpoint vectors

play a minor role in the LISP world.

This situation may change rapidly when COMMON LISP is generally accepted. COMMON LISP has fixed the data type **sequence** which is a union of the classical vector or linear array and the linear processed list structure (cdr-surface of a tree). Both subtypes of the datatype sequence have in common, that their elements are ordered and a "numbered" access makes sense. Additional the data type vector can be further specified to hold data with a single elementary data type, e.g. integers, reals, bits, characters. This concept can be enlarged upward compatible for the subtype "list" too.

COMMON LISP has defined a common group of more or less complex operations for sequences which formerly were available for the linear lists only. These operations have in common that they process the elements of the data structure in a "one after the other" order or in parallel by a uniform elementary operation. The difference between the two structures, which lies in the memory representation and quality of accessability, is without importance to the LISP language level. It appears primarily as declarative context to the operation itself.

The sequence operations of COMMON LISP allow a wide range of typical vector operations and so they are a quasi "natural" entry point for usage of vector hardware. A LISP compiler can generate vector code from these operations if one uses the favorable declarative LISP style of COMMON LISP.

4. MAP: Combining Vectors by Arbitrary Operators

Especially the function "map" is of interest for vectorization: it allows the formulation of complicated operations over one or more sequences.

We want to discuss "map" in the context of a moderately complicated example, the calculation of a linear combination (sum of two vectors, each multiplied with one scalar value). For those people not familiar with LISP the FORTRAN equivalent would be

```
      REAL V1(nn),V2(nn),A1,A2
         . . .
      DO 100 I=1,nn
100   V3(I) = A1 * V1(I)  +  A2 * V2(I)
```

The COMMON LISP formulation based on the function "map" is

```
(declare ((vector float) v1 v2 v3))
(declare (float a1 a2))
  . . .
(setq v3 (map '(vector float)
              #'(lambda (x y) (+ (* a1 x) (* a2 y) ))
               v1 v2 ))
```

The lambda expression defines the basic arithmetic operation in standard LISP prefix form: the local variables x and y are placeholders for the values of the input sequences, x for the v1- and y for the v2-elements.

In this example all data structures are designed as vectors with float values. List structures can be used with the same formulation and there is no formal restriction regarding the usage and nesting of LISP operators. Of course, not all of them will be processable by vector hardware. Vector hardware will be usable for:

INTEGER and FLOAT arithmetic, binary LOGIC, comparing and searching.

The vectorizing compiler produces from the above example the following intermediate code:

```
(PROG (**L **D **I **R** **R G0147 G0148)
 (setq **L (INF (GETMEM V1 )))        ; initial setting of counters
 (setq **R** (MKVEC (GTVECT **L)))    ; and addresses
 (setq **R (+ (INF **R**) 1))
 (setq G0147 (+ (INF V1) 1))
 (setq G0148 (+ (INF V2) 1))
 (FOR (FROM **I 0 **L 64)             ; loop control with 64-steps
      (DO (PROGN
          (setq **D (+ (- **L **I) 1))
          (IF (> **D 64)                    ; true vector length
              (VSETVL (setq **D 64))        ; of step
              (VSETVL **D))
          (PROGN (VLOAD (VREG 0) G0147 1)          ; arithmetic
                 (VFTIMESS (VREG 1) (VREG 0)
                           (GETMEM (+ A1 1)))
                 (VLOAD (VREG 2) G0148 1)
                 (VFTIMESS (VREG 3) (VREG 2)
                           (GETMEM (+ A2) 1))
                 (VFPLUSV (VREG 4) (VREG 1) (VREG 3))
                 (VSTORE (VREG 4) **R 1)
                 (PROGN (setq **R (+ **R 64))      ; maintenance of
                        (setq G0147 (+ G0147 64))  ; counters
                        (setq G0148 (+ G0148 64))))))
   (RETURN **R**))
```

The expressions (Vreg n) in the above example represent the vector register Vn and the other V-functions represent one vector instruction each. In the final compiling phase these elements will be transformed to true Cray X-MP vector instructions.

Of course, the above patterns of control structures are dependent on the data structures taking part in the operation. If a list takes part, the simple load instruction is replaced by a loop reading the next 64 (or fewer) elements of this list into a vector register; if numbers have to be deboxed or if the numerical values have to be converted, this is done during the loading too. If a list controls the length of the total operation, the list length has to be evaluated in advance. If the types of the data structures taking part are not known at compile time, a generation of vector code is impossible. So declarations become very important for this area.

In a test sequence the timings of several generated operations were measured. The results are included in /7/. Compared to standard LISP arithmetic there is a dramatical speedup in many cases, and even if only

list structures take part in input and output, the speedup is important, if only the operations itself is complicated enough. Application areas for this type of operation will be e.g. graphics and image processing.

5. LISP System Programming with Vector Instructions

The intermediate vector code level shown above is more than the target level for a first compilation phase. It is at the same time an adequate programming language for LISP system programming. In this language the full structural power of "scalar" LISP is available, e.g. for scalar evaluations or for program control, and at the same time the vector part of the machine is present on a near to hardware level: vector registers and vector instructions can be used without further interface. This type of language combines the advantages of assembly language in direct access to hardware and of high level programming. Vectorized algorithms can be tested easily. Some cpu time critical parts of the current PSL implementation for Cray computers were constructed already using this level of programming.

6. Vectorizing LISP DO Loops

The vector compiler produces vector instructions from LISP functions including DO loops that work on arrays with the COMMON LISP functions (aref ... and (setf (aref ...) . The well known example from the FFT listed below in its COMMON LISP formulation runs approximately 10 times faster in the vectorized version.

```
(de butterfly (ar ai vtr vti ur ui wr wi n l le le1)
    (prog (tr ti)
        (Declare ((array float) ar ai vtr vti))
        (Declare (float ur ui))

      (DO ((J 1 (+ J 1))) ((> J LE1))    %Loop thru butterflies

         (DO ((I  J         (+ I LE))        % 1st loop variable
              (IP (+ J LE1) (+ IP LE)) )    % 2nd loop variable
              ((> I N))                     % end condition
           (setf (aref vtr i) (- (* (aref ar IP) UR)
                                 (* (aref AI IP) UI)))
           (setf (aref vti i) (+ (* (aref AR IP) UI)
                                 (* (aref AI IP) UR)))
           (setf (aref ar ip) (- (aref AR I) (aref vtr i)))
           (setf (aref ai ip) (- (aref AI I) (aref vti i)))
           (setf (aref ar i)  (+ (aref AR I) (aref vtr i)))
           (setf (aref ai i)  (+ (aref AI I) (aref vti i))))
          )              % do I

         (setq TR (- (* UR WR) (* UI WI)))
         (setq TI (+ (* UR WI) (* UI WR)))
         (setq UR TR)
         (setq UI TI)
      )                  % DO j
) )
```

Bibliography

/1/ M. L. Griss, A. C. Hearn, A Portable LISP Compiler, Software Practice and Experience, Vol 11, 1981

/2/ The Utah Symbolic Computation Group: The Portable Standard LISP Users Manual. Department of Computer Science, University of Utah, Version 3.2: March 1984

/3/ J. B. Marti & A. C. Hearn, REDUCE as a Lisp Benchmark, ACM SIGSAM Bulletin, Vol. 19 No. 3

/4/ G. L. Steele et al.: COMMON LISP: The Language. Digital Press, 1984

/5/ Cray-1 Computer Systems, M Series Mainframe Reference Manual, Cray Research Inc, Mendota Heights, 1983

/6/ H. Melenk, W. Neun, Portable Standard Lisp Implementation for Cray X-MP Computers, Konrad-Zuse-Zentrum Berlin Technical Report 2,1986

/7/ H. Melenk, W. Neun, Usage of Vector Hardware for LISP Processing Konrad-Zuse-Zentrum Berlin Technical Report 3, 1986