LA-UR--86-831

DE86 008717

CONF-860733--1

Received by OSTI

APR 07 1986

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

TITLE: THE IMPLEMENTATION AND OPTIMIZATION OF PORTABLE STANDARD LISP FOR THE CRAY

AUTHOR(S): J. Wayne Anderson, C-10
Robert R. Kessler, University of Utah
William F. Galway, University of Utah

MASTER

SUBMITTED TO: The European Conference on Artificial Intelligence
Brighton, England
July 21 - 25, 1986

## DISCLAIMER



Los Alamos Los Alamos National Laboratory
Los Alamos, New Mexico 87545

FORM NO 836 R4
ST NO 2629 5/81

# The Implementation and Optimization of Portable Standard LISP for the Cray

**J. Wayne Anderson**
C-10, Computer User Services
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

**Robert R. Kessler and William F. Galway**
Utah Portable Artificial Intelligence Support Systems Project
Computer Science Department
University of Utah
Salt Lake City, Utah 84112

## ABSTRACT

Portable Standard LISP (PSL), a dialect of LISP developed at the University of Utah, has been implemented on the CRAY-1s and CRAY X-MPs at the Los Alamos National Laboratory and at the National Magnetic Fusion Energy Computer Center at Lawrence Livermore National Laboratory. This implementation was developed using a highly portable model and then tuned for the Cray architecture. The speed of the resulting system is quite impressive, and the environment is very good for symbolic processing.

Work supported in part by the Burroughs Corporation, the Hewlett Packard Corporation, the International Business Machines Corporation, the National Science Foundation under grant numbers MCS81-21750 and MCS82-04247, the Defense Advanced Research Projects Agency under contract number DAAK11-84-K-0017, and the U. S. Department of Energy under contract number W-7405-Eng.36.

## 1. Introduction

Research at the University of Utah toward developing a portable LISP system received impetus in 1966 [4] when a model for a standard LISP subset was developed to make the REDUCE [5] symbolic algebra package more portable. This research effort has since produced progressively larger and more portable subsets of LISP [2], the most recent of which is Portable Standard LISP (PSL)[1].

Among the goals of the designers of PSL were to provide a uniform LISP programming environment across a spectrum of machines, to produce a portable system comparable in execution speed to other non-portable LISP systems, and to effectively support REDUCE on different machines. PSL has met these goals and is currently being distributed for DECSystem-20s, VAXs running both UNIX and VMS, HP9836s, Apollos, Suns, IBM 370 class machines with CMS, Goulds, and a small version for the Macintosh. PSL is ready for distribution to Crays running the CTSS operating system and is being developed for Crays running COS. This wide range of machines demonstrates the ease with which PSL is ported.

There are several reasons for wanting LISP on Cray supercomputers. One is the interest in having symbolic programming environments on one of the most powerful machines available. This would provide the capability of solving symbolic problems that would not be feasible to solve on less powerful systems. There is also interest in the possibility of combining symbolic methods with some of the large numeric programs typical of large supercomputers.

In this paper we continue with a discussion of the porting process used to implement PSL on the Cray, followed by a discussion of the tuning that was performed. We then discuss some of the timing results, and conclude with proposals for future work.

## 2. Porting of PSL

Our Cray implementation began in June 1982 when a meeting was held at the University of Utah to outline the effort. By July of 1984 the PSL interpreter and compiler were available for use on all CRAY-1s and CRAY X-MPs at the Los Alamos National Laboratory. Soon thereafter, PSL was also available on the Crays at the National Magnetic Fusion Energy Computer Center of Lawrence Livermore National Laboratory. REDUCE was subsequently implemented at both sites. This process took much longer in actual elapsed time than the typical 6 man months required for most implementations of PSL. This is primarily because it was accomplished using part time efforts, equivalent to approximately 12 man months. Many of the early problems were related to getting reliable network access to machines, and to delays caused by the transfer of many large files between the development machines and the target Cray. The effort required to implement PSL on the Crays, while non-trivial, was much less than that required to implement a non-portable dialect.

PSL is transported using a half bootstrap technique. A cross compiler is built upon a running PSL. This compiler accepts PSL source code and produces assembly code for the target machine. The code is then transferred to the target machine, assembled, and linked with machine-dependent operating system routines to produce an executable system. This

[1] Work is under way at Utah to produce a Common LISP compatible subset built upon PSL.

system is tested, changes may be made to the compiler or machine-dependent source code, and the process is begun again. In order to more fully understand the implementation procedure, it is first necessary to take a look at the steps involved in PSL compilation [3].

The compiler first translates LISP code into instructions for an abstract LISP machine (ALM). The ALM is a general-purpose machine, with 15 general registers and a stack. The stack holds both return addresses and frames which are created upon function entry to hold saved registers and temporary values across calls. Arguments to functions are passed in the general registers, and values are returned in the first register. The ALM provides approximately 50 instructions varying in complexity from simple move instructions, to function calls, to lambda binding. Operands to instructions are either simple structures–like registers, stack frames, and memory–or they are higher level addressing modes like car and cdr. ALM instructions are expressed in LISP assembly program (LAP) format, which has the form:

```
(ALMopcode ALMoperand-1 ... ALMoperand-n)
```

After translation into ALM instructions, these instructions are then expanded, through the use of macros, into target machine instructions–expressed in the same LAP format. This is accomplished using a set of handwritten tables that describe each ALM instruction as a set of target machine instructions. Several expansions may be described–the expansion chosen depending upon the operands. The tables are called cmacro (compiler macro) definitions.

From here there are three separate paths that can be taken in the compilation process. The target machine instructions can be translated into:

1. machine code and placed into the memory of a running PSL system,
2. machine code and saved in a file for loading at a later time into a running PSL system, or
3. assembly language for later assembly and linkage on a target machine.

This third path is the one taken when bootstrapping PSL from a host system to a target machine. The original host for the Cray development was a DECSystem-20 at the University of Utah, while later at Los Alamos the host was a VAX 11/780 running UNIX.

The closer the target machine's assembly language is to that of the host, the easier the translation will be because the PSL compiler on the host can then be easily modified to generate assembly for the new target. Unfortunately, in the case of the Cray, the match was not close. For example, consider the following Cray assembly language (CAL) instruction:

```
S1 S2+S3
```

This adds the contents of register S2 to that of register S3 and stores the result in register S1. Thus the format of CAL instructions is along the lines of:

```
destination operand opcode operand
```

which is quite different from LAP format and from the assembly format used by other machines on which PSL was implemented.

To deal with this problem, we introduced one extra step in the translation process. The target machine instructions are written out as CAL macros that more closely match LAP format. These are then expanded by the CAL assembler into standard CAL format.

Some support code had to be written on the Cray to interface the cross-compiled code to Cray system functions–for example, input/output routines. The cross-compiled code is assembled on the Cray and then linked with this support code for execution.

There is a carefully graded set of tests that is used in the bootstrapping process. Each test provides an ever increasing subset of PSL to be cross-compiled, shipped to the Cray, assembled, and executed. A test that fails requires correction of the cmacro definitions or the support routines (or, rarely, correction of other parts of the implementation procedure). That portion of PSL that is successfully tested is then used as a basis for succeeding tests. After all tests are executed, the major portion of PSL has been implemented.

## 3. Tuning the Implementation

Once PSL was successfully implemented, we ran a set of LISP timing benchmarks developed by Gabriel [1]. The benchmarks were executed on the Cray, and the results compared to their execution in PSL on other machines. As expected, the benchmarks ran more quickly on the Cray. However, all the power of the Cray was not realized. For instance, translating from an ALM with 15 general-purpose registers to the Cray with its many special-purpose registers was a complicated task, one that the initial implementation did not do well. None of the temporary (T) registers were used, the arithmetic (S) register usage was not scheduled, and no vector registers were used.

At this point an optimization effort was undertaken at Los Alamos and the University of Utah. Several ideas were proposed, some of which were implemented, some rejected and, at this point, some are still being considered. These optimizations are detailed below.

A major feature of the Cray architecture when determining optimizations is the large ratio between memory and register access time. On the Cray the ratio is about 14 to 1, while on more conventional architectures the ratio is around 4 to 1. Since most of LISP's internal activity is accessing memory, as much information as possible must be maintained in registers. The Cray provides block move instructions that permit movement of multiple words to or from memory at a cost of only 1 extra clock for each additional word. Therefore, optimizations that combine accesses into block movement are advisable for the Cray. Using this concept, we found a number of potential optimizations that attempt to use the registers versus memory locations.

The first optimization involved moving the stack into registers. One thought was to move the entire stack into all of the vector registers (8 vectors, each with 64 elements, each 64 bits wide) providing a much faster stack. However, there are no instructions for accessing a variable vector register nor a variable register index; thus we could not implement a moveable top of stack pointer. An idea along similar lines was to move the stack into the T registers (64 registers, 64 bits wide), but they also do not permit variable access to a register. The final solution was to allocate the current stack frame to a set of the T registers. Since all accesses to frame locations are performed using compile time constants, registers could be used effectively For example, access to the first frame location could map into T20 and the second frame location would be T21. Using the T registers, access to each frame location is performed in 1 clock cycle, instead of the 14 before. Offsetting this advantage is that upon function entry and exit, the stack frame must be rolled to and from memory. However, this could be accomplished using fast block transfer. Another disadvantage is that the number of available T registers puts a limit on the size of a frame. This limit could be increased by using vector registers instead of T

registers, but we have not found this necessary.

A similar optimization was to keep heap pointers and other heavily used global variables in T registers instead of memory locations. These two sets of optimizations resulted in an improvement of approximately 25% in speed. However, because of the extra code required to move the stack frames to and from memory, the size of the code actually increased by about 10%.

An important optimization in the garbage collector takes advantage of the Cray's large word size. PSL on the Cray uses a mark and sweep compacting collector. One feature of this scheme is that the collector must compute the distance that each word must be relocated, and then store that distance. Generally a separate relocation table is used to store this relocation distance for each segment within memory. On the Cray, a 64-bit word represents each LISP item (a LISP cons cell requires two 64-bit words). PSL's tagging scheme allocates 8 tag bits and 24 pointer bits per item, leaving 32 bits left over. Since the maximum relocation distance can never exceed the addressing size, 24 of the 32 bits are used to store the relocation distance for each word. Eliminating the relocation table, and the extra memory references to it, doubled the garbage collection speed.

An optimization that we have considered, but have not yet implemented, is to use the vector registers while performing garbage collection. During the marking and pointer adjustment phases, each of the primary data structures are scanned to find active data. The stack and symbol table are scanned in sequential order, so we could block move them into a vector register (64 words at a time) and then scan from the vector registers instead of memory. Since a random memory access requires 14 clocks, while a block move to vector registers requires 2 clocks per access, we could reduce the the access time for these structures by a factor of 7.

Some operations on the Cray, such as integer division, are fairly difficult to implement directly in assembly language, and so were first implemented as calls to FORTRAN library routines. Some of these are now implemented as in-line code.

Generally, other implementations of PSL hand code critical parts of the system to improve speed. The original Cray implementation was the most portable implementation to date (which meant that less hand crafting was required to get the initial version functioning). The Gabriel benchmarks helped reveal areas that required tuning. Generally the timing ratios between the Cray and other PSL implementations should be fairly consistent. Ratios indicating poor Cray performance revealed areas that could be improved through hand coding. For example, it appears that one candidate is the lambda and fluid binding mechanism as illustrated by the relative performance of STak shown in Table 5 below. This optimization hasn't been accomplished yet, but should result in significant speed improvements in programs like REDUCE that make fairly heavy use of fluid binding. On the Cray, hand-coded routines should attempt to use register scheduling, as well as minimizing references to memory.

Table 1 shows the improvements in the Gabriel benchmarks resulting from those optimizations that we have currently implemented. The benchmark programs are briefly described below.

BOYER - a "theorem prover" emphasizing the use of "typical" LISP structure manipulations.

BROWSE - an "expert system" emphasizing the use of pattern matching and of frames for knowledge storage;

DESTRUCT - a program emphasizing the use of destructive list operations such as `rplaca` and `rplacd`;

STak - a program that times function calls using fluid (special) binding;

PUZZLE - a game implemented using many vector references; and

TRIANG - a board game benchmark.

Table 1.
Real Time in Milliseconds
PSL

| Benchmark | Old | New | New/Old |
|---|---|---|---|
| BOYER | 3.4 | 2.4 | 0.71 |
| BROWSE | 8.4 | 6.0 | 0.71 |
| DESTRUCT | 0.4 | 0.3 | 0.75 |
| STak | 1.1 | 0.9 | 0.81 |
| PUZZLE | 1.0 | 0.8 | 0.80 |
| TRIANG | 14.4 | 12.7 | 0.88 |

## 4. Timings

Tables 2 and 3 illustrate the execution speed of PSL relative to that of other dialects of LISP on the VAX 11/780. For the sake of brevity, results are given for just a few of Gabriel's benchmarks. These results, however, are typical. An entry of "-" means that a benchmark was not able to execute in that dialect of LISP at the time these figures were collected. These results seem to show that PSL is a very fast LISP on "conventional" architectures.

Table 2.
Real Time in Milliseconds
VAX 11/780

| | INTERLISP | VAX COMMONLISP | FRANZLISP | PSL |
|---|---|---|---|---|
| BOYER | 53.3 | 87.7 | 71.5 | 41.3 |
| BROWSE | 111.5 | 205.0 | 170.3 | 50.3 |
| DESTRUCT | 5.4 | 6.4 | 13.7 | 3.9 |
| STak | 9.7 | 4.1 | 6.3 | 5.4 |
| PUZZLE | 110.3 | 47.5 | - | 16.3 |
| TRIANG | 1076.5 | 360.9 | - | 212.2 |

Table 3.
Normalized Execution Times
(shortest execution time = 1.0)
VAX 11/780

| | INTERLISP | VAX COMMONLISP | FRANZLISP | PSL |
|---|---|---|---|---|
| BOYER | 1.3 | 2.1 | 1.7 | 1.0 |
| BROWSE | 2.2 | 4.0 | 3.4 | 1.0 |
| DESTRUCT | 1.4 | 1.6 | 3.5 | 1.0 |
| STak | 2.4 | 1.0 | 1.5 | 1.3 |
| PUZZLE | 6.8 | 2.9 | - | 1.0 |
| TRIANG | 5.1 | 1.7 | - | 1.0 |

Once PSL was successfully implemented on the Cray, Gabriel's benchmarks were executed and the results were compared to their execution on other machines. Tables 4 and 5 summarize these results.

Table 4.
Real Time in Milliseconds
PSL

| | Cray | VAX 11/780 | DEC-20 | IBM 3081 |
|---|---|---|---|---|
| BOYER | 2.4 | 41.3 | 23.6 | 4.6 |
| BROWSE | 6.0 | 50.3 | 28.7 | 6.3 |
| DESTRUCT | 0.3 | 3.9 | 2.4 | - |
| STak | 0.9 | 5.4 | 2.7 | 1.7 |
| PUZZLE | 0.8 | 16.3 | 15.9 | 1.5 |
| TRIANG | 12.7 | 212.2 | 86.9 | 25.4 |

Table 5.
Normalized Execution Times
(shortest execution time = 1.0)
PSL

| | Cray | VAX 11/780 | DEC-20 | IBM 3081 |
|---|---|---|---|---|
| BOYER | 1.0 | 17.2 | 9.8 | 1.9 |
| BROWSE | 1.0 | 8.4 | 4.8 | 1.1 |
| DESTRUCT | 1.0 | 13.0 | 8.0 | - |
| STak | 1.0 | 6.0 | 3.0 | 1.9 |
| PUZZLE | 1.0 | 20.4 | 19.9 | 1.9 |
| TRIANG | 1.0 | 16.7 | 6.8 | 2.0 |

The REDUCE distribution includes a standard timing benchmark. Table 6 presents the time required for its execution on several different machines. All but the S-810 implementation are based upon PSL.

Table 6.
REDUCE Timings in Seconds

| | |
|---|---|
| S-810 | 2.8 |
| Cray | 3.0 |
| DEC-20 | 25.0 |
| HP9836U | 55.0 |
| VAX 11/780 | 60.0 |
| APOLLO | 80.0 |
| VAX 11/750 | 90.0 |

## 5. Summary and Areas for Future Work

PSL has been successfully implemented under CTSS on the Cray. Sites currently running Cray PSL include Los Alamos National Laboratory, the National Magnetic Fusion Energy Computer Center at Lawrence Livermore National Laboratory, Kirtland Air Force Base, and the Center for Supercomputer Applications at the University of Illinois. Performance studies indicate that this implementation provides one of the fastest LISP environments currently available. However, all the power of the Cray has not been realized. In mapping from an ALM with 15 general-purpose registers, it was extremely difficult to make efficient use of the many special-purpose registers and vector processing capabilities of the Cray. This resulted in an implementation with many possible areas of optimization. Some areas under consideration now include scheduling of registers and using the vector registers during garbage collection.

## 6. Acknowledgments

We would like to acknowledge the contributions made to the implementation effort by Bruce Curtiss of the National Magnetic Fusion Energy Computer Center and Dana Dawson of Cray Research, Inc. We would also like to thank other members of the Utah Portable AI Support Systems Project for their discussions on potential optimizations. We would also like to thank Richard Gabriel for the many benchmarks and results he supplied and allowed us to cite in this paper.

## 7. REFERENCES

[1] R. P. Gabriel, *Evaluation and Performance of LISP Systems*, MIT Press, 1985.

[2] M. L. Griss, E. Benson, and G. Q. Maguire, Jr., "PSL, A Portable LISP System," *The Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, Carnegie-Mellon University, Pittsburgh, August 1982, pp. 88-96.

[3] M. L. Griss, E. Benson, R. Kessler, S. Lowder, G. Q. Maguire, Jr., and J. W. Peterson, *PSL Implementation Guide*, Utah Symbolic Computation Group, Computer Science Department, University of Utah, Salt Lake City, 1983.

[4] A. C. Hearn, "Standard LISP," *SIGPLAN Notices* 4,9 (September 1966).

[5] A. C. Hearn, *REDUCE 2 Users Manual*, Utah Symbolic Computation Group Report UCP-19, Comp' ter Science Department, University of Utah, Salt Lake City, 1973.