LA-UR-86-2662

CONF-870108--1

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

LA-UR--86-2662

DE86 013818

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

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

SUBMITTED TO: The 20th Annual Hawaii International Conference on System Sciences
Honolulu, Hawaii
January 6-9, 1987

DISCLAIMER

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

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 I oughs 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 1979[1] when a model for a standard LISP subset was developed to make the REDUCE [2] symbolic algebra package more portable. This research effort has since produced progressively larger and more portable subsets of LISP [3], the most recent of which is Portable Standard LISP (PSL).

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 and COS operating systems. 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

PSL is actually implemented in PSL itself. Most of the code is written as plain PSL functions, while some parts are actually written in SYSLISP [2], an enhanced version of PSL that permits the allocation and access to untagged data structures, construction of explicit pointers, etc. Since the system is written in terms of itself, it is ported to a new processor through the use of the PSL compiler (originally known as the Portable LISP Compiler [3]). A running PSL compiler is modified into a cross-compiler, so that instead of generating code for the current machine, it generates code for the target machine. The code defining the PSL system is then sent through the cross-compiler to create a PSL system that runs on the target machine. The compiler itself is then sent through the cross-compiler and added to the system on the target machine. Once that is accomplished, the PSL on the target machine is capable of self-building and can be used for further optimizations and enhancements.

Although conceptually simple, the porting process generally takes about six man-months to complete for each new target machine. The early phases of the process are involved with creating the cross-compiler [4]. Much of the compiler is system independent; however, parts must be customized for each target machine. The initial part of the PSL compiler translates PSL code into instructions for the Abstract LISP Machine (ALM). The ALM is characterized as follows:

1. Fifteen general-purpose registers, which are used for local computations and the passing of arguments to functions. The first register is used to return

the value from the function.

2. Stack frames that are used for saving return addresses, temporary calculations, and temporary values between function calls.
3. Caller save model, in which each function saves any values to the stack frame that are needed after the call to another function.
4. A set of about 50 instructions that defines the various operations of the ALM. Many are standard data movement, arithmetic operations, and function calls, while others are LISP-specific (like lambda binding for binding lambda variables).
5. A set of addressing modes that vary in complexity from simple immediate operands to car and cdr.

The process of translating from the ALM to the target machine (TM) is performed through macro expansion. PSL uses the LISP Assembly Program (LAP) format for both the ALM and TM instructions, which consists of an operator followed by one or more operands. Typical LAP format instructions are (ALM instructions are indicated with a leading asterisk (*) on the operator symbol):

```
(MOVE (INDIRECT (REG 1)) (REG 2)) % TM - move indirect register
                                  % 1 into register 2.
(*WPLUS2 (REG 1) (CDR (REG 2)))   % ALM - Add the Cdr of register
                                  % 2 to register 1.
```

Once the TM instructions have been generated, the compiler has three final phases:

1. Assemble the TM instructions into binary code and save the code in memory for execution. This is used when compiling PSL code for immediate use.
2. Assemble the TM instructions into binary code and save the code in a FASL (fast load) binary file for execution in some future PSL system.
3. Directly translate the TM instructions into assembly language. The assembly language can then be assembled by the target machine and linked and loaded into the run-time system.

The last phase is used when transporting the PSL kernel to the target machine. Thus, a cross-compiler is generated when the following steps are taken:

1. Determine a mapping from the structure of the ALM into the TM (e.g., are the fifteen general-purpose registers represented, does the stack grow up or down, etc.).
2. Write the macros that translate from the ALM instructions into a sequence of TM instructions.
3. Translate from the TM LAP instructions into TM assembly instructions. This includes the generation of the preamble and postamble code that must be included with each assembly language file.

3. The Test Series

Before the Cray implementation, one of the problems with the transportation of PSL was that once the cross-compiler was built, the entire PSL kernel had to be compiled into target machine assembly code and assembled on the target machine. PSL without the compiler is approximately 10,000 lines of code, which expands into about 125K of assembled code space. This large amount of code makes implementing a PSL for a new machine tedious and time consuming. To solve this problem, we designed the test series as a step-by-step approach to generating a PSL kernel. The test series slowly builds the various parts of the PSL kernel, incrementally testing each part. This permits small files to be sent to the target machine where they are assembled and tested. When tests don't work, generally the cross-compiler needs to be repaired, and the tests need to be performed again. The first test attempts to verify that simple input/output (I/O) is working correctly, which can then be used in later tests to indicate the status of each test. The test series has been designed so that once each test has been completed, the next test uses the code from all of the previous tests, along with new code, to develop the next part of the system.

There are two test series, one with nine simple tests and the second with eleven tests, that eventually result in a full PSL kernel and compiler. The first test series has just enough parts of the PSL kernel to verify the code generation routines, the assembly language constructs, and the interface to the target machine operating system (like character I/O, file I/O, terminating the task, signaling an error, etc.). These tests are small enough that transportation of their code to the target machine and testing is relatively easy. On completion of the ninth test, a simple PSL kernel has been built, including file I/O, garbage collector, and EVAL. At this point we can be fairly confident that cross-compiler is generating good code and can move on to the second test series.

The second test series is more extensive in that it brings in the various parts of the PSL kernel in complete detail. When the tenth test is reached, a complete PSL kernel has been constructed. The eleventh test is then used to build the PSL compiler. This requires more customization for the target machine, because it must be able to assemble the TM LAP code into directly exectuable binary code. It must also be able to perform binary I/O and save generated binary code into loadable modules. Once accomplished, the compiler itself can be translated into a loadable module. At the end of the eleventh test, the compiler was included as a part of the PSL kernel. This results in the kernel being larger than necessary as we now have the compiler in a stand-alone form. We, therefore, restore the kernel to the way it was at the end of the tenth test.

4. The Cray Port

Our Cray implementation began in June 1982 when a meeting was held at the University of Utah to outline the effort. By July 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 than the typical six 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.

One of the first decisions in generating Cray PSL was deciding which machine should be used as the cross machine. The original host was a DECSystem-20 at the University of Utah. Early development efforts included the generation of the Cray assembly code on the DecSystem-20 and shipping it across four machines until it finally reached a Cray at the National Magnetic Fusion Energy Computer Center in California. Going through so many machines and networks was a laborious process. Later, we moved the development effort to a host VAX 11/780 running BSD UNIX at Los Alamos National Laboratory. This facilitated the effort as the VAX essentially was connected directly to a Cray at Los Alamos, thus virtually eliminating the time required to ship files.

The next step was determining the mapping of the architecture of the ALM into the Cray. If the vector register capabilities of the Cray are ignored, the Cray is very RISC-like. There are few addressing modes and few computational registers (ones in which arithmetic operations can be performed), but a large number of cache-like extra registers (sixty-four 64-bit registers and sixty-four 24-bit registers). The Cray word size is 64 bits, so we decided to represent a LISP item (tag and information part) in a single word. There was quite a debate (and there still is), about whether we should pack two LISP items into a single word or just use one word per item. We decided that it was better to go for speed of access (its cheaper to just retrieve a single word than retrieve a word and then mask off the appropriate part) than size of the heap. Another factor was that there was no other implementation where more than one LISP item could fit in a word; thus the code would have to be checked to make sure that it was written properly and would not be confused with this new representation.

The large number of registers made mapping the ALM registers into the Cray registers easy. Five of the eight S-registers (64-bit computational registers) were chosen to represent the first five ALM registers. The first S-register (S0) is special and mainly used for comparison operations; thus it was left alone. The remaining two registers were designated as temporaries and used by the macros that mapped from the ALM instruction into TM instructions. The remaining 10 ALM registers were allocated to the bank of T registers (the sixty-four 64-bit cache registers). These registers may not be directly involved in a computation, but may be moved to the S registers in one clock cycle. Another T register was permanently assigned the value of NIL because it is used in so many comparisons. The eight A registers (24-bit address registers) were allocated for temporary addressing calculations, and one was allocated as the stack pointer. All of the vector registers and their instructions were ignored because no direct relationship between them and the ALM instructions could be found.

One major problem with the Cray port was the significant difference between the Cray's assembly language (CAL) and the standard LAP format. Nearly all other computers use an operator followed by operand format for their assembler, but the Cray is significantly different. CAL uses a semi-infix notation for its instructions, where the destination operand is the first element and the source operands are next enumerated with infix operators. For example, the CAL instructio..

```
S1 S2+S3
```

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 the following:

```
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 were written out as CAL macros that more closely match LAP format. These were then expanded by the CAL assembler into standard CAL format. This trick permitted a more natural debugging environment because we were able to look at the macros that were generated and did not have to worry about the nonstandard CAL syntax.

The previous example of CAL code introduces another interesting characteristic of the Cray. It uses three-address instructions. The ALM instructions are all two-address instructions; since all two-address instructions are subsets of three-address instructions, they did not present any initial problems. However, this was indeed a restriction because more efficient code could be generated for a three-address machine than for a two-address machine. We are currently exploring ways to take advantage of the three-address code.[1]

One final note that characterizes the Cray version of PSL from the previous versions is the use of recursive ALM to TM macros. In the past, most of these macro tables were written independently, where each ALM instruction carefully determined the various operand locations and generated the appropriate code to perform the requested operation. Thus, a *wplus2 macro (which performs addition) would test to see if the arguments are in registers or in memory. In either case, appropriate but different code would be generated. If the arguments were not in registers, before generating the code to perform the addition, code may be generated to move the arguments into registers. The solution for the Cray was to carefully code the *move ALM macro so that it could move any possible operand to any possible location. Once this was accomplished, the other ALM macros could recursively invoke the *move ALM macro to place the operands in the appropriate locations, perform the operation, and move the result to the appropriate destination. This made writing each ALM macro much simpler. For example, using the old technique, the ALM macro for *wplus2 might appear as:

```
% Defines the ALM macro expansion table for addition.
(defcmacro *wplus2
  % First part tests the type of the operands, and the second
  % is the list of instructions.  ARGONE refers to the first
  % ALM operand, ARGTWO is the second, etc.
  ((SRegP SRegP) (add ARGONE ARGTWO))
  ((SRegP ARegP) (move ARGETWO (reg S6))
                 (add ARGONE (reg S6)))
```

[1] Researchers at the University of Utah are currently developing a new compiler, EPIC, which generates better code than the current PSL compiler and takes advantage of three-address instruction sets.

```
((ARegP SRegP) (move ARGONE (reg S6))
               (add (reg S6) ARGTWO)
      (move (reg S6) ARGONE))
((SReP SmallInumP) (move ARGTWO (reg S6))
               (add ARGONE (reg S6)))

 ... THERE ARE MANY MORE POSSIBLE OPERANDS
```

The example demonstrates that this is a tedious process. Using the recursive technique, this could be written as follows (notice that there is only one actual generation of the add CAL instruction, which makes the code easier to modify):

```
% Define the *wplus2 ALM macro using recursive expansion.
% Addition instructions must operate in the S registers.
(defcmacro *wplus2
  ((SRegP SRegP) (add ARGONE ARGTWO))
  % AnyP will match any possible operand.
  ((SRegP AnyP)  (*move ARGTWO (reg s6))
                 (*add ARGONE (reg S6)))
  ((AnyP SRegP)  (*move ARGONE (reg S6))
                 (*add (reg S6) ARGTWO)
 (*move (reg S6) ARGONE))
  % No predicate is the otherwise clause.
  (              (*move ARGONE (reg S6))
                 (*move ARGTWO (reg S7))
                 (*add (reg S6) (reg S7))
 (*move (reg S6) ARGONE))
```

Once the cross-compiler was successfully built, the next step was to try the various parts of the test series. Before we could perform the first test, some additional support code had to be written on the Cray to interface the cross-compiled code to Cray system functions, for example, I/O routines. Since Fortran is the high-level language of choice on the Cray, it was used to implement all of the operating system interface code. The only difficult part of this process was determining the appropriate calling mechanism so that the generated CAL code could call Fortran code and coercing of data types between the two languages. The Fortran provided on the Cray is fairly rich in its capabilities and made manipulation of the various data structures quite reasonable. The main goal of the first test was to verify that the Fortran code and the techniques for its interface were working (without working output capabilities, it is difficult to verify that various parts of the system are working).

The bootstrap process then continued through each of the tests until eventually a full PSL kernel was completed. Progress slowed at that point until the resident assembler

could be defined and an interface to binary files could be implemented. Once those were accomplished, the initial version of Cray PSL was released and we turned out attention to further optimizations.

5. Tuning the Implementation

Once PSL was successfully implemented, we ran a set of LISP timing benchmarks developed by Gabriel [5]. The benchmarks were executed on the Cray, and the results were 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. Few of the T registers were used, the 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 one 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), which would provide 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 movable 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. Because of the extra code required to move the stack frames to and from memory, the size of the code 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 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. 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 to minimize 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

6. 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	67.7	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

7. Summary and Areas for Future Work

PSL has been successfully implemented under CTSS on the Cray. The use of the test series proved to be valuable and permitted an incremental approach to the development of the PSL kernel. It has helped to make PSL even more portable. 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 realised. 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 optimisation. Some areas under consideration now include scheduling of registers and using the vector registers during garbage collection.

8. Acknowledgments

We 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 also thank other members of the Utah Portable AI Support Systems Project for their discussions on potential optimisations and Dr. Martin Griss, referred to by many as the father of PSL. We also thank Richard Gabriel for the many benchmarks and results he supplied and allowed us to cite in this paper.

9. REFERENCES

[1] J. B. Marti, A. C. Hearn, M. L. Griss, and C. Griss "Standard LISP Report," *SIGPLAN Notices* 14,10 (October 1979).

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

[3] 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.

[4] 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.

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