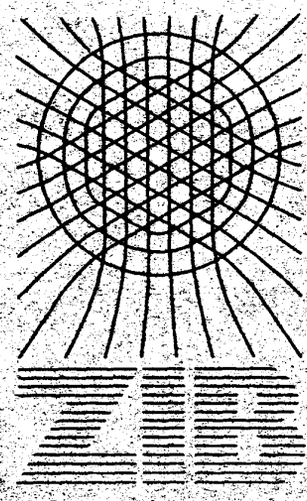

Konrad-Zuse-Zentrum
für Informationstechnik Berlin

J. Anderson W. Galway R. Kessler H. Melenk W. Neun

The Implementation and Optimization of Portable Standard Lisp for the Cray

Preprint SC-87-1

The Implementation and Optimization of Portable Standard Lisp for the Cray

January 29, 1987

A version of this paper appeared in the proceedings of the 20th annual Hawaii International Conference on Systems Sciences, January 1987. It has been revised to include the work of Melenk and Neun.

J. Wayne Anderson, Los Alamos National Laboratory[1]
William F. Galway, University of Utah[2]
Robert R. Kessler, University of Utah[3]
Herbert Melenk, Konrad-Zuse-Zentrum für Informationstechnik at Berlin[4]
Winfried Neun, Konrad-Zuse-Zentrum für Informationstechnik at Berlin[4]

[1]C-10, Computer User Services, Los Alamos National Laboratory, Los Alamos, New Mexico 87545
[2]current address: University of Bath, Claverton Downs, School of Mathematics, Bath England
[3]Department of Computer Science, University of Utah, Salt Lake City, Utah 84112
[4]Heilbronner Str. 10, D 1000 Berlin 31, Federal Republic of Germany

Abstract

Portable Standard Lisp (PSL), a dialect of Lisp developed at the University of Utah, has been implemented and optimized for the Cray 1 and Cray X-MP supercomputers. This version uses a new implementation technique that permits a step-by-step development of the PSL kernel. The initial Cray version was acceptable, although the execution speed of the PSL was not as fast as had been anticipated. Cray-specific optimizations were undertaken that in some cases provided a ten-fold speed improvement, resulting in a fast Lisp implementation.

1 INTRODUCTION

Research at the University of Utah toward developing a portable Lisp system received impetus in 1979 when a model for a standard Lisp subset [11] was developed to make the REDUCE [8] symbolic algebra package more portable. This research effort has since produced progressively larger and more portable dialects of Lisp [6], 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 has been distributed to over 700 sites world-wide on DECSystem-20s, VAXs running both UNIX^{tm}, and VMS, Apollos, Suns, IBM 370 class machines with CMS, Goulds, and the Apple Macintosh^{tm}. PSL has been ported to, but not currently distributed for, the HP IPC, HP 9000/320, and Silicon Graphics Iris. 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 transported.

There are currently no other implementations of Lisp on the Cray supercomputers, although there are several reasons to have Lisp available on such a machine. One is the ability to have symbolic programming environments on one of the fastest machines available. This would provide the capability of solving symbolic problems that would not be feasible to solve on smaller systems. There is also interest in the possibility of combining symbolic methods with some of the large numeric programs typical of supercomputers [2].

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 continue with a description of the process of instruction scheduling which provided further speed improvements. We then discuss some of the timing results, mention some initial work in vectorizing PSL, describe the implementation of a

Common Lisp subset on the Cray, and conclude with proposals for future work.

2 PORTING OF PSL

PSL is implemented in PSL itself. Most of the code is written as vanilla PSL functions, while some parts are written in SYSLisp [1], a version of PSL that permits the allocation of and access to untagged data structures, construction of explicit pointers, etc. Since PSL is written in terms of itself, it is ported to a new processor through the use of the PSL compiler (a version of the Portable Lisp Compiler [5]). A running PSL compiler is modified into a cross-compiler (see Figure 1), 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 cross machine is no longer required and the target machine can be used directly for further optimizations and enhancements.

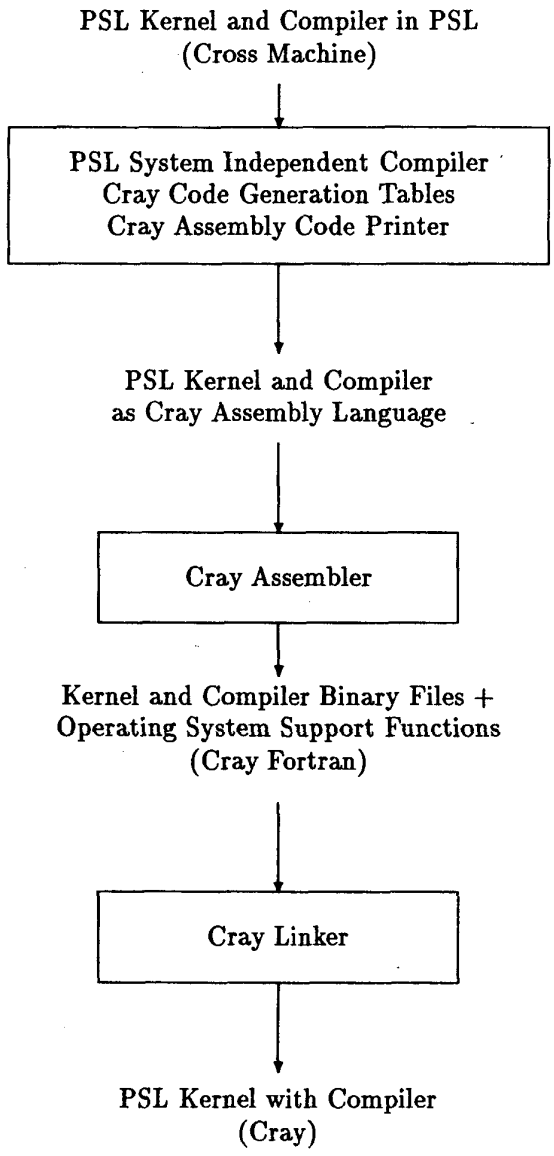

Figure 1: THE PSL PORTING PROCESS

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 [7]. 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 and local variables, along with the results of intermediate calculations.

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 (such as lambda binding for local variables).

5. A set of addressing modes that vary in complexity from simple immediate operands to `car` and `cdr`.

The ALM is a low-level instruction set closely related to register machines. In contrast, the Spice Lisp instruction set [18] is quite high-level, with primitives for direct manipulation of Lisp primitives and stack operations. The Symbolics Lisp machine also provides a high-level abstract machine with microcode support [16]. The low-level model of the ALM is better for transporting the Lisp system to conventional machines, because the compiler can do optimizations such as register allocation, and the mapping process from ALM to the Target Machine (TM) is fairly simple.

The process of translating from the ALM to the 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):

Target machine instruction to move indirect register 1 into register 2.
```
(MOVE (INDIRECT (REG 1)) (REG 2))
```

ALM instruction to add the Cdr of register 2 to register 1.
```
(*WPLUS2 (REG 1) (CDR (REG 2)))
```

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 files. 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 10K lines of code, which expands into about 125K bytes of assembled code space. This large amount of code makes implementing PSL for a new machine tedious and time consuming. Without a guarantee that the code being generated is correct, the debugging of this many lines of code can be a real nightmare. Also, once a bug fix has been determined, a complete regeneration of all of the files must be performed. Finally, the sheer number of bytes of assembly code being transported to the target machine can overwhelm many file transfer methods.

Our solution to this problem is the test series, which is a step-by-step approach to generating a PSL kernel. Each test embodies a small set of Lisp functionality which can be semi-independently tested and then combined into a full PSL kernel. Since each test is fairly small, the file transfer overhead is minimal. Also, each test is conceptually simple and makes debugging easier. When a test fails, generally the cross-compiler needs to be repaired, and the tests need to be performed again. The semi-independent nature of the testing is due to the "onion-skin" implementation. When the first test succeeds, its code is used as the kernel to test the next one. When that test succeeds, it is added to the kernel as a new layer and used in the next test and so on. This allows each test to use functions and features known to be correct from earlier tests and saves the effort required to implement new support functions for each test. Lisp is already a highly integrated environment and is quite amenable to layering functions in this way.

There are two test series, the first with eight 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 eight tests are:

1. Attempts to verify that simple input/output (I/O) of characters is working correctly, which can then be used in later tests to indicate the status of each test.

2. Defines and tests the various Lisp print functions.

3. Defines a mini-allocator to allocate Lisp structures, like lists, strings, and vectors.

4. Defines a mini-`read`, which does not use scan tables and only supports a few of the data types (for example, no floating point numbers).

5. Defines a mini-`eval` of most Lisp forms except it does not permit user defined functions.

6. Defines a more extensive set of primitives to support the mini-`eval` from the previous test. These include `lambda` expressions, and user defined `expr`, `fexpr`, `nexpr`, and `macro` functions. This is a complete model of PSL but has a restricted set of the PSL functions present.

7. Defines a set of routines to test a minimal file-io package.

8. Defines the full garbage collector.

Upon completion of the eighth test, a simple PSL kernel has been built. At this point we can be fairly confident that the 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 executable 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 a larger than necessary kernel since the compiler is a loadable module. Thus, the final step is to restore the kernel to the way it was at the end of the tenth test, and include the support from test eleven for reading binary files.

4 THE CRAY PORT

The Cray implementation began in June 1982, with the initial version running by July 1984. Much of the effort was part-time, requiring about 12 man-months, some of which was directed toward the design of the test series. The actual porting time was close to the six month estimate.

The implementation began with the first decision to choose which machine should be used as the cross machine. Due to the cross bootstrap nature of the porting process, the ability to rapidly transport files between the cross and host machines is very important. Originally we chose a DECSystem-20 at the University of Utah as the cross machine and the Cray was located at the National Magnetic Fusion Energy Computer Center in California. The large "electronic distance" between these machines made the file transfer process painful. Eventually we moved to the development effort to a host VAX 11/780 running BSD UNIX^{tm} at Los Alamos National Laboratory. This facilitated the effort as the VAX was connected directly to a Cray at Los Alamos, thus virtually eliminating the time required to ship files.

The next step was to determine 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, 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 and still is quite a debate about whether one or two Lisp items should be packed into a single word. We chose speed at the expense of space and used one item per word. Another factor was that there was no other PSL implementation where more than one Lisp item could fit into a word. Although this type of design decision is characterized in a few PSL system constants, all of the code would have to be checked to verify that it was written properly and would not be confused by 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 ALM instructions 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 instruction

```
S1 S2+S3
```

adds the contents of register S2 to that of register S3 and stores the result in register S1. This "pseudo" infix form is quite different from prefix LAP format and from the assembly format used by other machines on which PSL was implemented.

The solution of this problem required the introduction of an 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 the standard CAL format. This trick permitted a more natural debugging environment because we were able to look at the generated macros and did not have to deal with 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 with the EPIC compiler [10].

One final note that characterizes the Cray version of PSL from the previous versions is the extensive use of recursive ALM to TM macros. Previous PSL versions had used some recursion, but not as extensively as in Cray PSL. In many of the previous versions, 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` instruction (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. Cray arithmetic instructions require the operand to be in registers, thus code must be generated to move the arguments into registers. The solution for the Cray was to carefully code the `*MOVE` ALM instruction so that it could move any possible operand to any possible location. Once this was accomplished, the other ALM macro could recursively invoke the `*MOVE` ALM instruction to place the operands in the appropriate locations, perform the operation, and move the result to the appropriate destination. This made writing each ALM expansion much simpler. For example, using the old technique, the ALM macro expansion for `*WPLUS2` might appear as:

```
Defines the ALM macro expansion table for addition.  The first part
of each form 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.
(defcmacro *wplus2
  ((SRegP SRegP)      (add ARGONE ARGTWO))
  ((SRegP ARegP)      (move ARGTWO (reg s6))
                      (add ARGONE (reg S6)))
  ((ARegP SRegP)      (move ARGTWO (reg a6))
                      (adda ARGONE (reg a6)))
  ((SRegP 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 this makes the code easier

to modify since there is only one actual generation of the CAL instructions ADD - addition of S registers, and ADDA - addition of A registers):

```
Defines the *WPLUS2 ALM macro using recursive expansion.
The final clause without a predicates always succeeds.
(defcmacro *wplus2
  ((SRegP SRegP) (add ARGONE ARGTWO))
  ((ARegP ARegP) (adda ARGONE ARGTWO))
  ((SRegP AnyP)  (*move ARGTWO (reg s6))
                 (*wplus2 ARGONE (reg S6)))
  ((ARegP AnyP)  (*move ARGTWO (reg A6))
                 (*wplus2 ARGONE (reg a6)))
  ((AnyP SRegP)  (*move ARGONE (reg S6))
                 (*wplus2 (reg S6) ARGTWO)
                 (*move (reg S6) ARGONE))
  (              (*move ARGONE (reg S6))
                 (*move ARGTWO (reg S7))
                 (*wplus2 (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; such as, I/O routines. Since Fortran is the high-level language of choice on the Cray, we used it 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 coerce the data types between the two languages. Cray Fortran is fairly rich in its capabilities and made manipulation of the various data structures quite reasonable. The main goal of the first test is to verify that the Fortran code and the techniques for its interface are working. As the test series builds upon the previous tests, it is an absolute requirement that I/O work properly.

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 our attention to further optimizations.

While Cray PSL 3.2 was being optimized, a parallel effort was started in December 1985, at Konrad Zuse-Zentrum für Informationstechnik at Berlin to port the PSL 3.4 version to the Cray [13]. The ultimate goal was the creation of a

distributable Lisp version for the Cray Operating System (COS) and Cray's new UNIX^{tm} style operating system UNICOS. PSL 3.4 is an enhanced version of PSL developed at the Hewlett Packard Laboratories. One of the major differences between PSL 3.4 and the previous versions is the concept of the micro-kernel. The task of the micro-kernel is to allocate the various data spaces (heap, stack, etc.) and then load the remainder of the kernel. The original PSL kernel was changed to be a set of independently compilable modules. The micro-kernel (about 1500 lines of Lisp code) is the only part that must be cross compiled and processed by the system assembler. This drastically reduces the size of the assembly language files to be shipped to the target machine.

The first step in the process was to upgrade Cray PSL 3.2 with PSL 3.4 modules to produce a cross compiler for PSL 3.4. This enabled the use of the Cray itself as a cross compiling device. Soon afterwards PSL 3.4 was able to cross compile itself directly on a Cray. A complete rebuild of kernel currently takes less than 200 cpu seconds (as compared to the hours that it took on the Vax 780 using PSL 3.2). At the same time the operating system interface of PSL 3.4 was polished to better fit the Cray environment, which is characterized by a real address memory and a batch oriented command language. Added were libraries for load modules, a command language interpreter, catalog access and system specific diagnostic aids.

Another important addition was a completely dynamic memory management system. In a virtual addressing environment, data areas can be luxuriously allocated because unused parts do not cost physical memory. Real addressing systems like the Cray require careful tailoring of sizes of the data areas according to the problem. Therefore dynamic management of these areas is necessary. The PSL 3.4 version is based on Cray's HEAP manager common to both COS and UNICOS (this has nothing to do with the Lisp heap), which permits the expansion and contraction of the executing image. The Lisp structures that have variable size are BPS (binary program space, which includes compiled programs and static arrays), Heap, Stack and Binding Stack (used for special variables). These structures are allocated in one huge data area provided by the operating system. This data area can be enlarged or shrunk upon request, with a fixed lower end. Since the Cray binary instructions are not easily relocatable after linkage, BPS starts on the lower end and can be altered in size. The other data areas follow and can be moved in memory. The heap is moved via garbage collector techniques, using a compacting garbage collector. Care must be taken with the special pointers into data areas which reside in the T or B registers and therefore they are not "seen" by the garbage collector. The PSL 3.4 version automatically enlarges the code and heap areas when the amount of free memory is low; the stacks are not automatically enlarged.

5 TUNING THE IMPLEMENTATION

Once PSL was successfully implemented, we ran a set of Lisp timing benchmarks developed by Gabriel [4]. 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 of 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.

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 instead of memory locations.

Another interesting feature of the Cray architecture is that it is a very orthogonal machine. Many different versions of a few ALM macros were tried to see if a more efficient mapping could be determined (i.e., instructions like `*FIELD` which performs a bit field extraction). Nearly every different method tried resulted in exactly the same number of clock cycles. Although this inability to find "betterf" code sequences was disappointing, it demonstrates that the Cray architecture is highly orthogonal and really quite good.

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 or 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 system implementation variables in T registers instead of memory locations (for example, the stack and heap boundary pointers were moved into the T registers). This and the previous stack frame optimization 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). Cray 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 therefore the expensive 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 were implemented as in-line code.

Table 1 shows the improvements in the Gabriel benchmarks resulting from this first set of optimizations. The benchmark programs are briefly described below.

BOYER - a "theorem prover" emphasizing the use of "typical" Lisp structure manipulations;

Benchmark	Original	Optimized	$\frac{Original}{Optimized}$
	Real Time in Seconds		
BOYER	3.4	2.4	1.42
BROWSE	8.4	6.0	1.40
DESTRUCT	0.4	0.3	1.33
STAK	1.1	0.9	1.22
PUZZLE	1.0	0.8	1.25
TRIANG	14.4	12.7	1.13

Table 1: Optimization results for PSL 3.2.

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.

The authors at Konrad Zuse-Zentrum für Informationstechnik at Berlin incorporated these optimizations into their PSL 3.4 version and began a new round of optimizations. The first set began during the update of pattern tables to produce the 3.4 version. A number of enhancements were implemented, mostly by adding new open-coded functions. Open coding is an important aspect for machines with instruction pipelines (like the Cray). This is because any jump interrupts the instruction issue and requires an expensive load of instructions from memory, whereas the wasting of memory by additional inline instructions is acceptable with today's memory sizes. The most prominent example for open coding is the function cons, which is the workhorse of structural work. Open coded `cons` speeds up some program parts by more than a factor of 10 and costs only about 10% in code size (measurement of REDUCE resulted in a 10.8% larger size with open-coded cons). Other open coded optimizations included special variable binding, function calling, exit handling (`catch` and friends), and arithmetic. Some of these open-coded optimizations significantly improved the execution of the Gabriel benchmarks in areas

where the previous version was weak (for example, open coding the special variable access produced an eight fold improvement in execution time of the STAK test).

The PSL 3.2 version which allocated the stack frame to a block of T registers is much better than conventional stack management in memory. However, the loading and restoring of the stack frame between memory and the T registers is one of the main bottlenecks. This observation lead to another refinement in PSL 3.4, with the objective to avoid memory operations wherever possible. Three additional cases are distinguished:

1. If a function only uses the stack to save its return address, the T Registers are not used at all. The return address is saved by the function itself and the frame length is set to zero.

2. If a function uses a small frame and this frame must not be saved, then an extra area of T registers is reserved for this purpose, which is never saved.

3. In the case of an unusually large frame, the conventional stack technique is used.

These optimizations helped to improve many small functions.

The next set of optimizations required the addition of Cray specific passes to the PSL compiler. The fact that the PSL compiler is written in PSL and the internal structures used by the compiler are Lisp structures, made the implementation of these passes easy. The basis for the first pass optimization is the Cray's ability to overlap the execution of instructions. When one expensive instruction is executing (like a memory access instruction), many extra instructions may be executed in parallel while the expensive instruction is running. Thus, the extra instructions take no time to execute. The only caveat is that the extra instructions may not access any resources that are used in the expensive instruction (for example, if an instruction is reading into a register, none of the extra instructions may access the destination register). Another type of "instruction scheduling" (scheduling of instructions so some of their execution is "free") is mentioned in the next section. This first new pass processes the ALM instructions to rearrange the stack deallocation in order to find the best time to unload the stack frame. This required breaking some of the original PSL ALM instructions into smaller parts so they may be placed separately into the resulting function. The original PSL compiler deallocated the stack frame just prior to exit. Moving the deallocation back from the end as far as possible allows the memory operations to be overlapped with other instructions.

Three additional passes which operate upon the TM instructions were also added. These passes regard the program as a whole and are not limited by the boundaries

Benchmark	Original	Opt. Orig.	Opt. PSL 3.4	$\frac{Original}{Opt.PSL3.4}$
	Real Time in Seconds			
BOYER	3.4	2.4	0.9	3.56
BROWSE	8.4	6.0	1.3	6.46
DESTRUCT	0.4	0.3	0.1	4.00
STAK	1.1	0.9	0.1	11.00
PUZZLE	1.0	0.8	0.7	1.42
TRIANG	14.4	12.7	10.9	1.32

Table 2: Optimization results with PSL 3.4.

of ALM instructions. The types of optimizations that these passes perform include: loops are partially unrolled, multiple loading of the same item is suppressed (it performs a simple peephole optimization) and instructions are rescheduled to use overlapped processing wherever possible.

All optimizations were motivated by the results of simultaneously developed diagnostic aids. Most important of them is the PSL interface to Cray's SPY feature. SPY interrupts a running program at very short regular time intervals and counts the interrupts per place. An arbitrary Lisp program can be supervised by SPY and the summary shows a very exact distribution of cpu consumption per Lisp function. This was the starting point for optimization steps in many cases and resulted in a 32% to 1100% speed improvement in the benchmarks shown in Table 2.

6 INSTRUCTION SCHEDULING

The Cray processor architecture is based to a high degree on segmentation and pipelining. Scalar instructions using different data resources (registers, memory locations) can overlap their execution. Even if they use the same functional unit (e.g., doing integer add/subtract) their execution may be overlapped (one starting each clock period). Vector instructions can operate in parallel, if different functional units are used (they need their pipelining for the vector processing).

The objective of instruction scheduling is to rearrange a given sequence of instructions so that maximum parallelism is used. An early approach only regarded the scalar `LOAD` instructions. A load takes about 14 clock periods (cp) depending upon the type of processor, while simple `MOVE`s, `SHIFT`s, integer `ADD`s, etc. take one or two cps. This early approach was unsatisfactory for several reasons: it was not

very systematic; it did not handle several simultaneous LOADs and there are additional "medium" weight candidates that are viable for scheduling, e.g. floating point arithmetic (ADD: 6 cps and MULTiply: 7 cps).

A new approach now handles all instructions generated by the compiler. The algorithm works directly on the TM instructions as a preprocessor to the final assembly. It operates in several steps:

1. The basic blocks are initially determined. A basic block is a section of code limited by "non movable" instructions like jumps, labels, etc.

2. For each instruction in a basic block a node is built which describes the input and output resources of the instruction, its execution time and its functional unit requirements. Each node also includes links to other nodes and slots for local variables used during the selection process.

3. The nodes are scanned for data flow dependencies. Each node is doubly linked to those nodes creating its inputs or using its output resources. The result is a directed graph completely representing the dependencies between the instructions. The graph establishes a partial ordering among the instructions. Each rearrangement of instructions (linear sequence) is a full ordering; it is a functionally correct variant of the original program, if and only if the full ordering is a superset of the partial ordering.

4. During the selection process instructions are taken from the graph. A counter simulates the execution time of the target code on a clock period basis. The minimal elements of the graph, that is, the instructions with no (more) predecessors, are candidates for selection in each step. The criterion for selection is

   (a) the earliest possible execution time (availability of resources and units), and

   (b) the weight of the instruction (its execution time and the weights of its successors).

   The number of cps needed is added to the actual cp counter and this value is propagated to the directly dependent nodes as "earliest time to execute." Afterwards the selected instruction and its links are removed from the graph and the operation is repeated.

As an example, the following `test` function

```
(defun test (a b)(fun (caar a) (caar b)))
```

compiles the parameters for fun into the following code:

```
1: (move (reg a5) (reg 2))    % parameter "b" to address reg.
2: (load (reg a5) (reg a5))   % first car to address reg.
3: (load (reg 2)  (reg a5))   % second car to parameter reg.
4: (move (reg a2) (reg 1))    % parameter "a" to address reg.
5: (load (reg a2) (reg a2))   % first car to address reg.
6: (load (reg 1)  (reg a2))   % second car to parameter reg.
```

Analysis of this sequence reveals that the processor has to wait for memory before instructions 3 and 6 can begin. The dependency graph appears in Figure 2. Note, that there is a direct link from 1 to 3 (and 4 to 6) because 3 overwrites (Reg 2) which is input to 1.

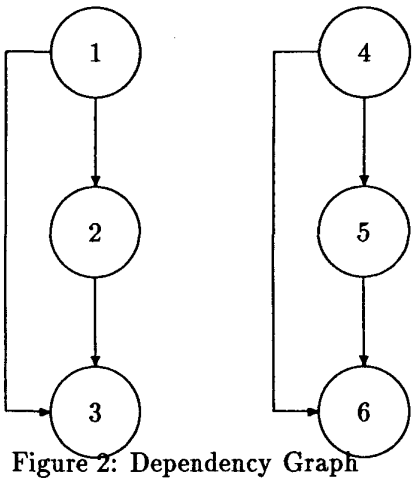

Figure 2: Dependency Graph

The scheduler can completely intermix the two independent subgraphs resulting in the following code sequence:

```
1: (move (reg a5) (reg 2))
2: (load (reg a5) (reg a5))
4: (move (reg a2) (reg 1))
5: (load (reg a2) (reg a2))
3: (load (reg 2)  (reg a5))
6: (load (reg 1)  (reg a2))
```

In this version, only instruction 3 has to wait. Instructions 2 and 5 load in parallel as do 3 and 6. In reality, some code preparing the linkage to "fun" is inserted after 5 to fill the 14 cps waiting time. The value of scheduling improves with larger functions, however in this example, if we replace `fun` with `cons`, the execution time with instruction scheduling is 75% of the time without scheduling.

The algorithm used by the instruction scheduler itself is independent of the Cray architecture. It describes the processor as a set of abstract resources (individual registers, the memory) and devices (e.g., functional units). The node constructing functions are attached to the symbolic operation codes. They are table-driven for generally applicable instruction patterns and individually coded for special cases. The algorithm is described completely in [15].

7 TIMINGS

Tables 3 and 4 illustrate the execution speed of PSL relative to that of other dialects of Lisp on the VAX 11/780. Since there are no other implementations of Lisp on the Cray, these tables provide some indication of the comparison of PSL with other dialects. These results are given for the same subset of benchmarks used in the previous section. 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.

Once PSL was successfully implemented on the Cray, Gabriel's benchmarks were executed and the results were compared to their execution on other machines running PSL. Tables 5 and 6 summarize these results.

Another interesting comparison might be to compare the Cray against a microcoded Lisp engine like the Symbolics 3600 with IFU. The numbers indicate that the Cray is between 10 and 23 times faster.

Benchmark	INTERLisp	VAX CL	FRANZ Lisp	PSL
BOYER	53.3	32.4	71.5	41.3
BROWSE	111.5	38.2	170.3	50.3
DESTRUCT	5.4	4.1	13.7	3.9
STAK	9.7	3.0	6.3	5.4
PUZZLE	110.3	23.4	-	16.3
TRIANG	1076.5	303.5	-	212.2

Table 3: Various Vax 780 Lisp implementations with real time in seconds.

Benchmark	INTERLisp	VAX CL	FRANZ Lisp	PSL
BOYER	1.6	1.0	2.2	1.3
BROWSE	2.9	1.0	4.5	1.3
DESTRUCT	1.4	1.1	3.5	1.0
STAK	3.2	1.0	2.1	1.8
PUZZLE	6.8	1.4	-	1.0
TRIANG	5.1	1.4	-	1.0

Table 4: Various Vax 780 Lisp implementations with normalized execution times.

Benchmark	CRAY X-MP	VAX 780	DEC-20	IBM 3081
BOYER	0.9	41.3	23.6	4.6
BROWSE	1.3	50.3	28.7	6.3
DESTRUCT	0.1	3.9	2.4	-
STAK	0.1	5.4	2.7	1.7
PUZZLE	0.7	16.3	15.9	1.5
TRIANG	10.9	212.2	86.9	25.4

Table 5: Various PSL implementations with real time in seconds.

Benchmark	CRAY X-MP	VAX 780	DEC-20	IBM 3081
BOYER	1.0	45.9	26.2	5.1
BROWSE	1.0	38.7	22.1	4.8
DESTRUCT	1.0	39.0	24.0	-
STAK	1.0	54.0	27.0	17.0
PUZZLE	1.0	23.3	22.7	2.1
TRIANG	1.0	19.5	8.0	2.3

Table 6: Various PSL implementations with normalized execution times.

CRAY X-MP PSL 3.4	0.96
S-810	2.80
Cray X-MP PSL 3.2	3.00
DEC-20	22.50
VAX 11/780	50.30
HP 9836	65.30
VAX 11/750	78.70
APOLLO DN 320	80.40

Table 7: REDUCE Timings in Seconds

The REDUCE algebra system distribution includes a standard timing benchmark. Table 7 presents the time required for its execution on several different machines. All but the Hitachi S-810 implementation are based upon PSL [12]. The PSL 3.4 improvements and development of the diagnostic feature were closely connected to the implementation of REDUCE on the Cray. The REDUCE test sequence was measured under SPY and gave hints for further optimizations to Cray PSL 3.4. The initial timing was about 1.7 seconds, while the optimized version now requires less than 1 second of cpu time.

8 VECTORIZING PSL

Vector instructions were introduced into PSL 3.4 in the summer of 1986. They were initially present only on the ALM instruction level. They were easy to add as ALM instructions because of the highly modular structure of PSL. The patterns

for generating the "new" instructions were written in Lisp and then collected in a separately loadable module.

A two step approach was then undertaken to make vector operations available to the user. In the first step the vector operations were transformed on a one-to-one basis as open-coded functions. This made the vector instructions and vector registers directly available from the Lisp language level; the compiler simply expands them during its code generation pass. This permits a mixture of assembly type language (for vectors) and Lisp high level language (for control structures and scalar values) to be available for system programming. These operations were used for some Cray specific modifications to the kernel of PSL, mostly for comparisons, presetting initial values and move operations.

For example a fast routine copying a storage area for the PSL kernel is now written as follows (note that Cray vectors have a maximum length of 64):

```
Copy length words from the first address to the second.  It
first computes the size of the copy, then issues a vector load
vector store instructions.
(defun vector-copy (from to length)
    (ifor (from i 0 (- length 1) 64)
          (do (vsetVL (if (> (- length i) 64)
                          64
                          (- length i)))
              (vload  (vreg 0) (+ from i) 1)
              (vstore (vreg 0) (+ to i) 1))))
```

Of course, this language level is not adequate for the casual Lisp user. Therefore, a second step was undertaken to provide direct high-level Lisp support of the vector operations. This can be accomplished for some of the Common Lisp sequence operations, which may be partially rewritten using the above style. An automatic transformation of user supplied code to vector instructions has been designed for the Common Lisp `map` function and for array references within `do` loops. These are accomplished with a special set of macros. Of course, this "autovectorization" is restricted to those operations which can be performed by vector hardware: arithmetic and binary logic for uniformly typed data. Explicit and detailed declarations are necessary prerequisites.

As an example of the automatic vector code generation, the following code computes a linear combination of two vectors of floating points values. These values must be stored as untagged floats in a word vector. Note the use of instructions combining a vector and a scalar (the products in this example) and instructions

combining vectors with vectors. The scalars are expected as tagged floats and the access to the actual float is accomplished inline with the **getmem** expressions.

```
(map '(vector float)
     #'(lambda (x1 x2)
                (+ (*  a1 x1)
                   (*  a2 x2)  ))  v1 v2)
```

The macro generated intermediate LISP, annotated with comments is:

```
(PROG (**L **D **I **R** **R G0147 G0148)
 (SETQ **L (INF (GETMEM V1 )))        take the vector length.
 (SETQ **R** (MKVEC (GTWRDS **L)))    construct result vector.
 (SETQ **R (+ (INF **R**) 1))         set output pointer,
 (SETQ G0147 (+ (INF V1) 1))          input pointers.
 (SETQ G0148 (+ (INF V2) 1))
 (IFOR (FROM **I 0 **L 64)            basic loop in 64-steps.
   (DO (PROGN
        (SETQ **D (+ (- **L **I) 1)) rest length of operation.
        (IF (> **D 64)
            (VSETVL (SETQ **D 64))    actual vector length.
            (VSETVL **D))
        (PROGN
         (VLOAD (VREG 0) G0147 1)     load operand.
         (VFTIMESS (VREG 1) (VREG 0) vector * scalar.
                   (GETMEM (+ A1 1)))
         (VLOAD (VREG 2) G0148 1)     load operand.
         (VFTIMESS (VREG 3) (VREG 2) vector * scalar.
                   (GETMEM (+ A2  1))
         (VFPLUSV (VREG 4) (VREG 1) (VREG 3)) vector + vector.
         (VSTORE (VREG 4) **R 1))     store result.
         (PROGN
           (SETQ **R (+ **R 64))      update pointers
           (SETQ G0147 (+ G0147 64))
           (SETQ G0148 (+ G0148 64)))))))
 (RETURN **R**))
```

A detailed description of this example is included in [14]. Initial experiments with the FFT from Gabriel's benchmarks are encouraging; they signal a speedup of about a factor of 10.

9 PCLS

One of the problems with Lisp has been the proliferation of numerous different Lisp dialects. The advent of Common Lisp (CL) [17] and the "band wagon" effect that it has started, has made it the Lisp dialect of the future. Our latest CL compatibility package is the Portable Common Lisp Subset (PCLS) [9] which implements approximately 80% of the Common Lisp language. It is essentially a large application program that runs on a version of PSL. Its transportation to a new machine requires the customization of a few files, along with some small enhancements to the PSL kernel.

The Cray version of PCLS was recently completed and required about one man-month of work, longer than the other versions of PCLS. This was due to the file name size restrictions of CTSS (8 characters) and the lack of a real directory structure. The CL pathname facility does not map well to this restricted structure and thus could not be used to solve the problem. PCLS consists of about 100 files scattered across six directories plus a number of scripts to help build the system. Many of the files explicitly load named modules, along with the build scripts and thus would require extensive rewriting to use the shortened names. We designed a file name mapping convention that would take the PCLS defined names and map them into the Cray names. This scheme worked well.

One of the original goals of PCLS was to be able to run CL code as fast as PSL code on any implementation. This was accomplished using a number of system independent compiler source-to-source transformations. The system independent nature of these optimizations meant that no special optimizations were required. This has resulted in the Cray PCLS being just as fast as Cray PSL.

10 SUMMARY AND FUTURE WORK

PSL and more recently PCLS have been successfully implemented under CTSS and COS 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 and helped to improve the implementation time on other machines. The idea of developing a working system as quickly as possible and then using it as the base to perform optimization experiments has been quite successful.

Although the initial performance was acceptable, it was not as fast as we had expected. The many optimizations from movement of the top stack frame into registers, to hand-coding certain Lisp operations, to instruction scheduling, have resulted in a very fast Lisp. Using the Gabriel benchmarks as a guide, this implementation provides the fastest Lisp environment currently available.

Work on compiler extensions for higher level vector operations is continuing. They will generate the previously mentioned vector code from Common Lisp style sequence operations and do loops with array references based upon macro techniques [13]. These will be eventually incorporated into Cray PCLS. The ultimate goal is to provide a full Common Lisp which takes complete advantage of the unique features of the Cray supercomputer.

11 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 optimizations and reviews of this paper. Dr. Martin Griss, referred to by many as the father of PSL, also worked on the early part of the development. Dr. Richard Gabriel for the many benchmarks and results he supplied and allowed us to cite in this paper. Walter van Roggen for providing his latest benchmark results for VAX LISP 2.0 on the 780. We appreciate Cray Research Incorporated (Mendota Heights) and Cray Research GmbH (Munich) for their support. Finally, we thank Stan Shebs and Sandra Loosemore for their implementation of PCLS and Cathy Elwell for her work on porting PCLS to the Cray.

12 REFERENCES

1. Benson, E. and Griss, M. L. SYSLISP: A Portable LISP Based Systems Implementation Language. Utah Symbolic Computation Group Report UCP-81 University of Utah, Department of Computer Science (February 1981).

2. Chalfan, K. M. A Knowledge System that Integrates Heterogeneous Software for a Design Application. The AI Magazine 7, 2 (Summer 1986) 80-84.

3. Cray X-MP Series Mainframe Reference Manual, Cray Research Inc., 1982, HR-0032.

4. Gabriel, R. P. Evaluation and Performance of Lisp Systems. MIT Press (1985).

5. Griss, M. L. and Hearn, A. C. A Portable Lisp Compiler. Software - Practice and Experience 11, 6 (June 1981) 541-605.

6. Griss, M. L., Benson, E., and Maguire Jr., G. Q. PSL, A Portable Lisp System. The Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, Carnegie-Mellon University, Pittsburgh (August 1982) 88-96.

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

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

9. Kessler, R. R., Shebs, S., and Loosemore, S. A Portable Common Lisp Subset with High Performance. Utah PASS Opnote, 86-01 (February 1986).

10. Kessler, R. R., Peterson, J. , Carr, H., Duggan, G., Knell, J., and Krohnfeldt, J. EPIC - A Retargettable, Highly Optimizing Lisp Compiler. Proceedings of the SIGPLAN '86: Symposium on Compiler Construction, (June 1986) 118-130.

11. Marti, J. B., Hearn, A. C., Griss, M. L., and Griss, C. Standard Lisp Report. SIGPLAN Notices 14, 10 (October 1979) 48-68.

12. Marti, J. B. and Hearn, A. C. REDUCE as a Lisp Benchmark. SIGSAM Bulletin, 19, 3 (August 1985) 8-16.

13. Melenk, H. and Neun, W. Portable Standard Lisp Implementation for Cray X-MP Computers. Konrad-Zuse-Zentrum für Informationstechnik Berlin Technical Report ZIB 86.2 (September 1986).

14. Melenk, H. and Neun, W. Usage of Vector Hardware for Lisp Processing. Konrad-Zuse-Zentrum für Informationstechnik Berlin Technical Report ZIB 86.3 (September 1986).

15. Melenk, H. and Neun, W. A LISP Program for Rearranging Instructions for a Pipelining Processor; Optimizing Pass for Cray PSL. Konrad-Zuse-Zentrum für Informationstechnik Berlin Technical Report (in preparation).

16. Moon, D. Architecture of the Symbolics 3600. Proceedings of the 12th Annual International Symposium on Computer Architecture (1985) 76-83.

17. Steele Jr., G. L. Common Lisp - The Language. Digital Press (1984).

18. Wholey, S., and Fahlman, S. F. The Design of an Instruction Set for Common Lisp. Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming (August 1984) 150-158.

13 Biographies

Dr. J. Wayne Anderson heads a research effort at Los Alamos National Laboratory that is investigating language issues in parallel processing and various aspects of artificial intelligence. Two of the projects include a graph reduction effort attempting to realize implicit parallelism on a variety of architectures including a network of Lisp machines and the Cray X-MP, and the design of expert systems for various applications. He served in various professorial positions at the University of Southwestern Louisiana, Corpus Christi State University and Montana State University. Dr. Anderson received a B.A. in math and a Ph.D. in computer science from the University of Texas at Austin. He has published papers on operating systems, architectures, and symbolic programming.

William F. Galway is currently a systems programmer with the School of Mathematics at the University of Bath, England. His interests are in area of Lisp implementations, pattern matching, and practical and theoretical aspects of factoring numbers. He graduated with his M.S. in 1985 from the University of Utah.

Dr. Robert R. Kessler is a Research Assistant Professor of Computer Science and head of the Portable A.I. Support Systems Project at the University of Utah. His research interests are in the area of portable Lisp systems and target architectural description driven Lisp compilers. He also has interests in expert systems, their application in Lisp compilers and as programming productivity tools. Dr. Kessler is currently writing a Lisp programming textbook with particular emphasis on object oriented style. He graduated with his Ph.D. in 1981 from the University of Utah and is a member of ACM and the IEEE Computer Society.

Herbert Melenk is currently head of the research group "Symbolik" at Konrad-Zuse-Zentrum für Informationstechnik at Berlin. The group works in the field of symbolic computation and application of expert systems. He received his diploma in mathematics and physics from the University of Marburg in 1969 and has worked since 1972 in the area of Lisp implementation.

Winfried Neun is currently with the "Symbolik" group at Konrad-Zuse Zentrum at Berlin. He received his diploma in mathematics from the Free University of Berlin in 1977 and has worked in the area of supercomputing since 1980.