AN OPTIMIZING COMPILER FOR LISP FOR THE Z8Ø

Jed Marti

University of Oregon

ABSTRACT This paper describes an optimizing compiler for the Z8Ø. Described are the compilation mechanisms, optimization techniques, and performance statistics.

Introduction

UOLISP [1] is a subset of Standard LISP [2] implemented for the Z8Ø microprocessor. It runs in a minimum of sixteen thousand bytes of storage and most effectively in thirty-two thousand or more. The system is more than just a basic LISP interpreter. The entire facility consists of:

- a program to load precompiled "fast load" files.
- a parser for a subset of RLISP [3].
- a function trace and break facility.
- a LISP structure editor.
- an online help facility and text formatter.
- a pretty print facility.
- the Little META translator writing system [4].
- a compiler and optimizer.
- arbitrary precision integer package.

This paper addresses the mechanisms of the compiler and its optimizer.

The compilation process is divided into three passes: the first translates LISP into pseudo-assembly code called LAP (for Lisp Assembly Program), the second pass performs a peephole optimization on the LAP assembly code, the third pass translates this LAP into absolute machine code and places this in storage for execution or dumps it to a file for later restoration.

Overview

The LISP interpreter contains code for rea functions into the LISP system and executing tl interpretively much like other microprocessor based systems. Unfortunately interpreted funct require large amounts of storage and execute v slowly.

A more efficient scheme reads functions in interpretive form, and then compiles them to machine code to be executed directly by the mi processor. The interpreted version of the func disappears, its storage becomes available for at a later time.

For example, the function FACT, which comp the factorial of a number recursively, is defi in UOLISP as follows:

```
(DE FACT (N)
   (COND ((LESSP N 2) 1)
         (T (TIMES2 (FACT (SUB1 N)) N))))
```

In UOLISP, dotted-pairs, of which this functio composed, take 4 bytes each. 22 dotted-pairs a used to define FACT for a total of 88 bytes. UOLISP's compiler generates the following code FACT:

```
Loc.   Code        LAP                     INTEL

ØØØØ:             ENTRY FACT,EXPR ;FACT:
ØØØØ: CD96B3      CALL  ALLOC     ;     CALL  ALJ
ØØØ3: Ø2          DEFB  2         ;     .BYTE 2
ØØØ4: D7FE        STOX  HL,-1     ;    *STOX  HL
ØØØ6: 11Ø24Ø      LDI   DE,2      ;     LXI   DE
ØØØ9: E7          RST   LINK      ;     RST   LI
ØØØA: 172Ø        DEFW  LESSP     ;     .ADDR LE
ØØØC: 28Ø5        JREQ  $1        ;     JRZ   $1
ØØØE: 21Ø14Ø      LDI   HL,1      ;     LXI   HL
ØØ11: 18ØD        JR    $Ø        ;     JMPR  $Ø
ØØ13:          $1:                ;$1:
ØØ13: CFBF        LDX   HL,-1     ;    *LDX   HL
ØØ15: E7          RST   LINK      ;     RST   LI
ØØ16: 832Ø        DEFW  SUB1      ;     .ADDR SU
ØØ18: E7          RST   LINK      ;     RST   LI
ØØ19: A82Ø        DEFW  FACT      ;     .ADDR FA
ØØ1B: CF7F        LDX   DE,-1     ;    *LDX   DE,
ØØ1D: E7          RST   LINK      ;     RST   LI
ØØ1E: 1D21        DEFW  TIMES2    ;     .ADDR TI
ØØ2Ø:          $Ø:                ;$Ø:
ØØ2Ø: CDØ8B4      CALL  RDLLOC    ;     CALL  RDI
ØØ23: FE          DEFB  -2        ;     .BYTE -2
```

* means macro form.

Fifth SIGSMALL Symposium on Small Systems, August 2-3, 1982.
Colorado Springs, Colorado.

A total of 36 bytes are used, less than half the size of the interpreted version. The compiled version runs over 40 times as fast.

Compilation Mechanisms

Compiled programs move information between registers and call subroutines to perform most operations. In this section we describe how important LISP constructs are implemented in LAP and enumerate the various support functions required.

Parameter Passing

Zero to three parameters may be passed to a function. The first argument of a function (if it has any) will always be in the HL register pair, the second in DE, and the third in BC. Functions with more than three arguments cannot be compiled. This particular mode of execution is called the register model as opposed to the more common stack model. We believe that the register model is inherently more efficient than the stack model though perhaps more difficult to compile for.

Stacks

Function parameters and PROG type variables are kept in a stack frame, sometimes called an activation record, a contiguous block of locations pointed to by the IX index register. When a function is invoked it creates a new frame on the top of the stack by calling the ALLOC support routine. When a function terminates it calls the DALLOC routine which subtracts the number of locations used from IX, freeing the space for use by the next function.

Storing and retrieving values from the stack frame is accomplished by the two support routines LDX and STOX. Since these operations occur frequently in compiled code it is necessary that they use as little storage as possible. Therefore the LDX and STOX routines are called using the Z8Ø RST instruction with the following byte containing which register pair is to be stored (or loaded), and the displacement from the top of the stack frame. The LAP instructions generated by the compiler are also called LDX and STOX and contain the register pair name and what displacement is to be used.

Both LAMBDA expressions and PROG forms generate the ALLOC and DALLOC calls to handle stack frames. One of the optimizations performed is to substitute the appropriate number of increment or decrement IX instructions, or for larger frames, a sequence to add to IX. This has the disadvantage of not checking for stack overflow.

The Z8Ø internal stack is used for saving return addresses and intermediate values during function evaluation. A call to a function FUN3 with three arguments stores the results of evaluation of the first two arguments on the Z8Ø stack while the third is being computed. The values are popped into the appropriate registers just before the function is invoked.

```
(FUN3 (FUNA ...) (FUNB ...) (FUNC ...))
```

would generate the following code sequence:

```
... evaluate FUNA ...
    PUSH  HL          ;Save result of FUNA.
... evaluate FUNB ...
    PUSH  HL          ;Save result of FUNB.
... evaluate FUNC ...
    LDHL  BC          ;Move HL to BC.
    POP   DE          ;FUNB is second argument.
    POP   HL          ;FUNA is first argument.
    RST   LINK        ;Call FUN3.
    DEFW  FUN3
```

Function Invocation

The compiler will not always know the address of a function being called because it might not be defined yet. Even if the function is defined the compiler does not know whether it will be compiled or interpreted at run time. A special internal subroutine called LINK is used to transfer control at run time. Since both compile and interpreted functions can exist at the same time, LINK will perform either of two functions. If an interpreted function is being called from compiled code the LISP interpreter will be invoked for that function. If the function being called is compiled or is a system function the call to LINK will be replaced by a direct call t that function. The call to the LINK function mus be an RST type link so that the three byte Z8Ø CALL instruction will exactly replace the compil call. If the system global variable !*FLINK is N the substitution will not take place and the slo link form will remain. This is a useful debuggin tool as it allows you to compile functions and change their definitions (for tracing) without reloading the system.

Compiled as:	Changed by LINK to:
RST LINK DEFW function-name	CALL function-address

The two byte DEFW attached to the LINK contains the symbol table pointer of the function being called. At execution time the LINK routine looks for either a compiled or interpreted function attached to the name and either invokes EVAL, generates the CALL, or if the !*FLINK flag is on just transfers to the function. If no such function is defined, an error will occur and the name of the function will be displayed.

The LIST Function

The LIST function is compiled in a special w to take advantage of the Z8Ø internal stack. The arguments of the LIST function are compiled and the results of each are pushed onto the stack. When all have been computed the support function CLIST is called.

```
(LIST (F1 ...) ... (Fn ...))
```

compiles to:

```
  ... evaluate F1 ...
      PUSH  HL          ;Save result of F1 for CLIST.
        .
        .               ;Evaluate other arguments.
        .
  ... evaluate Fn ...
      PUSH  HL          ;Save result of Fn for CLIST.
      MVI   A,n         ;Number of values on stack.
      CALL  CLIST       ;Call to CLIST routine.
```

COND Compilation

The LISP COND function is compiled into a series of tests and conditional jumps. The CMPNIL support routine compares the result of a predicate to NIL and sets the Z8Ø NZ and Z flag bit which controls the conditional branch instructions generated. If the last predicate of the COND is T, the predicate and jump will not be compiled (the usual case).

(COND ($a_Ø$ $c_Ø$) ... (a_n c_n))

generates the following code:

```
  ... evaluate aØ ...
      RST   CMPNIL      ;Is aØ NIL?
      JPEQ  GØØØ1       ;Yes, jump to next antecedent
  ... evaluate cØ ...
      JP    GØØØ2       ;First consequent done, quit.
GØØØ1:                  ;Come here if aØ not T.
        .
        .               ;Evaluate other a - c pairs.
        .
GØØØx:                  ;Try last predicate.
*... evaluate an ...
*     RST   CMPNIL      ;Is last one NIL?
*     JPEQ  GØØØ2       ;Go return NIL if yes.
  ... evaluate cn ...
GØØØ2:                  ;Always come here when done.
```

Lines preceeded by an asterisk are not generated if the last predicate is T.

PROG, GO, and RETURN

The PROG function and the control constructs GO and RETURN are compiled by plugging lablels and values into a template.

```
  (PROG (X)
        .
   LBL ...
        .    ... (RETURN value)
        .
        .
        (GO LBL)
        .
        ... )
```

compiles to:

```
      CALL  ALLOC       ;Space to save X allocated.
      DEFB  2
      LDI   HL,NIL      ;PROG variable set to NIL.
      STOX  HL,-1
        .
        .
LBL:                    ;A PROG label generated.
        .
        .
  ... evaluate value ...
      JP    GØØØ1       ;Jump to the end of PROG.
        .
        .
        .
      JP    LBL         ;(GO LBL) generates JP.
        .
        .
        .
GØØØ1:                  ;RETURN's come here.
      CALL  DALLOC      ;Deallocate stack frame f
      DEFB  -2          ; storage of X.
```

AND and OR Compiled.

AND and OR are compiled identically except that the evaluation of the arguments of AND terminates if one is NIL, and the evaluation o terminates if one is non-NIL. The compilation AND generates JPEQ instructions after a compar to NIL, and the compilation of OR generates JP instructions.

(AND $a_Ø$ ... a_n)

compiles to:

```
  ... evaluate aØ ...
      RST   CMPNIL      ;Is result of aØ NIL?
      JPEQ  GØØØ1       ;Stop evaluation if yes.
        .
        .               ;Evaluate other arguments
        .
  ... evaluate an ...
GØØØ1:                  ;Always end up here.
```

Constants, Variables, and Quoted Values

These items are loaded directly into the correct register for the function to which the are to be passed. Local and global variables m have values assigned to them with the appropri store instruction.

Quoted items are saved on a list of compile quoted values so that the garbage collector wil not remove them. The value representing the quoted item is loaded into the appropriate register.

Compiling FEXPR Calls

When compiling calls to user or system defi FEXPR's the argument list is passed as a list t the function for evaluation. This interpreted form interacts poorly with compiled code for th following reason. All local variable names declared in a function are replaced with their stack frame locations by the compiler. When the FEXPR tries to evaluate its argument in the environment of the calling routine, the variabl name in the S-expression cannot be found. The solution is to declare any variables to be pass to an FEXPR for evaluation as GLOBAL. This need not be done for COND, PROGN, PROG, OR, and AND because these forms are compiled into object co rather than as calls to functions.

The Optimizer

The optimizing phase is divided into two passes and features two levels of optimization and a speed or space choice. The first phase is an extended peephole optimization, the second removes function prologs and epilogs from routines which do not need stack frames. The three levels of optimization include a "safe set", a set of speed optimizations which increase code size, and a "dangerous set" which removes some error checking.

The Closing Window

There has been considerable research on peephole optimization for retargetable compilers [5-7]. The version used in the UOLISP optimizer might be more aptly called a "closing window" optimizer. The hole examined by the optimizer initially includes the entire program. Each instruction is removed from the window in turn. The advantage of this mechanism is that the entire program may be scanned for each instruction examined. Most of the optimizations do not scan very far ahead.

Redundant Instruction Removal

This optimization removes several forms of instructions which replicate data already in registers. For example:

```
STOX  HL,-1       becomes    STOX  HL,-1
LDX   HL,-1
```

The closing window method permits any number of instructions between the STOX and LDX which do not modify the contents of HL (or whatever register is used).

A second optimization removes store instructions whose location is never referenced. This optimization is very important in small subroutines. If all store instructions are removed, the stack frame allocation prolog and epilog may also be removed. Many very small routines can be reduced in size by as much as 85%. Since a great deal of time is spent in small routines, this optimization can be very important.

Jump Instructions

Several optimizations of this type are performed. The simplest removes unreachable code.

JP $\quad label_a$

.
.
.

$label_b$: ...

All instructions between the JP instruction and the first label ($label_b$) following it are removed since they cannot be reached from anywhere. The same optimization is performed when a subroutine is called from which no return can be expected. Functions which always generate an error or use the THROW function have this feature.

Another jump optimization removes worthless forward jumps. Thus:

JP $\quad label_a$
$label_a$:

results in the jump instruction being removed completely.

Conditional expressions are examined for multiple inversions. Thus:

```
CALL  NOT          becomes    RST   CHPNIL
RST   CHPNIL                  JP-not-cond. labe
JPcond. label
```

The final jump optimization garners the most savings of all optimizations. It determines the distance jump instructions must travel and if i is less than 127 bytes in either direction the instruction is converted to its short form. Sin most LISP functions are very short, most jumps up in their short forms saving 1 byte. Unfortunately short jumps are usually 20% slower.

Stack Frame Optimizations

Many times the end of a PROG form is also t end of its corresponding LAMBDA expression and DALLOC calls will occur in a row. In this case optimizer combines the two calls into one by ad their sizes together. A further optimization occurs if the last CALL DALLOC is immediately followed by a RET instruction. The call to DALL is replaced by a call to the special routine RDLLOC which automatically does the extra retur The use of this routine saves 1 byte and about 5 microseconds (for the 4 mhz. Z8ØA) on each function exit.

Reduction in Strength

This class of optimizations replaces severa long form instructions (or sets of instructions with a simpler Z8Ø instruction. Thus moving HL DE has an XCHG instruction substituted, saving single byte. A 3 byte call to any of the CAAR, CADR, CDAR, and CDDR is replaced with two singl byte calls on CAR and CDR saving a single byte. This optimization is disabled on machines which not have the 1 byte calls on CAR and CDR. Final the 4 byte version of the LHLD instruction is replaced with its shorter and faster 3 byte version.

Fast Optimizations

The LDX and STOX stack frame referencing functions take two bytes for each use. The functions themselves take approximately 50 micr seconds to execute. Approximately 50% of the execution of compiled code is spent in these tw routines. By open coding them as indexed MOV instructions, the time is reduced to less than microseconds at the expense of 4 additional byt This particular optimization can be turned on a off by the user so that very important function are optimized and less important ones, slower b much smaller. In the factorial example, use of this optimization results in a 24% speed improv ment at a cost of a 38% increase in size.

Dangerous Optimizations

This set of optimizations removes a number of error checks to increase execution efficiency. With selective use they cause no problems. One such optimization replaces the stack frame allocation routine calls by a string of increment or decrement register IX instructions:

```
CALL  ALLOC      becomes     INX   X
DEFB  4                      INX   X
                             INX   X
                             INX   X
```

Larger stack frames use a DADX instruction rather than the increments.

```
CALL  ALLOC      becomes     EXX
DEFB  16                     LXI   HL,16
                             DADX  HL
                             EXX
```

The corresponding decrement forms are used for the stack frame deallocation calls. The deallocation is done as part of the fast optimization because it is never dangerous.

The second optimization is open coding of the ADD1 and SUB1 functions. These are replaced by INX HL, and DCX HL instructions. They are not dangerous as long as the sign of the number does not change. A sign change causes overflow into the tag field of a number changing it into a bad identifier or string pointer.

Second Optimization Pass

The second optimization pass removes the function prolog and epilog if no stack frame is used. Thus the function:

```
(DE CAAAAR (X) (CAAR (CAAR X)))
```

is compiled without optimization into:

```
ENTRY CAAAAR,EXPR
CALL  ALLOC
DEFB  2
STOX  HL,-1
LDX   HL,-1
CALL  CAAR
CALL  CAAR
CALL  DALLOC
DEFB  -2
RET
```

This version uses 19 bytes. After the first optimization pass the following code is produced:

```
ENTRY CAAAAR,EXPR
CALL  ALLOC
DEFB  2
RST   CAR
RST   CAR
RST   CAR
RST   CAR
CALL  RDLLOC
DEFB  -2
```

This version takes 12 bytes. The second pass notices that the stack frame is never used (there are no STOX or LDX instructions). The final pa produces:

```
ENTRY CAAAAR,EXPR
RST   CAR
RST   CAR
RST   CAR
RST   CAR
RET
```

The final version takes only 5 bytes, a saving of about 75%.

Execution Statistics

We now examine the effect of the optimizer on size and execution speed. A rough approximatio two different types of programs and their size execution statistics are given. The first prog is the factorial example. 6! was computed 1Ø,Ø times on a 4 megahertz, 64k CP/M system. The second test does a complete reversla to all le of a binary tree. It is also executed 1Ø,ØØØ t and experiences 6 garbage collections.

```
(DE SUPER!-REVERSE (A)
  (COND ((ATOM A) A)
        (T (CONS (SUPER!-REVERSE (CDR A))
                 (SUPER!-REVERSE (CAR A)) ))
```

The tree ((A . B) . (C . D)) was reversed to ((D . C) . (B . A)).

	Size A/B Bytes	Time Secon
No optimization	42 / 44	48 /
Safe optimization	37 / 38	45 /
Safe and fast	51 / 56	34 /
Fast and dangerous	49 / 56	27 /

At best the optimizer provides a 47% speed up the expense of a 20% space increase.

To get a view of the effectiveness of each the individual optimizations over a class of programs, 8 different programs were compiled a the number of bytes saved by each of the reduc tion in size optimizations were tallied.

Opt. No.	Program A	B	C	D	E	F	G	H	Total
1	44	16	16	12	Ø	24	12	12	136
2	52	43	2Ø	5	13	16	7	42	198
3	34	34	38	2	26	28	8	98	268
4	18	28	8	Ø	2	2	2	36	96
5	56	118	52	1Ø	3	4	Ø	Ø	233
6	Ø	6	Ø	Ø	6	Ø	Ø	15	27
7	12	54	Ø	3	Ø	6	6	33	114
8	47	8	26	4	1Ø	21	18	77	211
9	16	64	16	Ø	Ø	7	7	84	194
1Ø	22	Ø	36	2	8	8	2Ø	2	98
11	66	27	27	9	6	18	24	30	207
12	129	17Ø	75	2Ø	33	55	6Ø	135	677
13	12	1	31	Ø	4	2	5	8	63
14	33	21	Ø	9	4	12	Ø	Ø	79
%	12	12	13	1Ø	14	12	14	14	12.5

The most important space optimization by far is the short jump conversion, the second, the removal of redundant load register instructions, the third the conversion of 4 byte LHLD instructions to 3 bytes, the fourth, the conversion of 16 bit move HL to DE instruction (actually two instructions) to an XCHG instruction, the fifth, the inversion of conditional jumps, and the sixth the use of the RDLLOC stack deallocation routine. The least important is the removal of dead code after functions which do not return.

The average reduction in size achieved by the optimizer is a little over 12.5%. This compares very favorably with other peephole optimizers which gather about 15 % (one of these has over two hunderd separate optimizations).

A final test compares UOLISP generated compiled code with that produced by various compilers for mainframes.

Test	A	B	C	UOLISP
1	132	391	51	1450
2	135	3502	1037	4000
3	117	748	1173	15500
4	562	4692	2312	18500
5	2072	8313	2023	37000
6	1098	9231	12806	109000
7	1062	1972	1360	13700
8	1019	18326	6800	49000

The 8 different programs tested were designed to exercise various features of compiled LISP code. The tests for the first three LISP compilers were taken from [8] and have been subsequently improved. Machine A is a large DEC 2060 running LISP 1.6 with the Portable LISP Compiler [9], machine B is a VAX 11/750 running Franz LISP, machine C is a VAX 11/750 running Portable Standard LISP Version 2 [10], and UOLISP runs on a 64k Z80A system with CP/M 2.2. A few of the time tests reflect the relatively small amount of space available and a large number of garbage collections. The statistics show that compiled UOLISP code is on the average one fiftieth the speed of a DEC 2060 running LISP 1.6, one seventh the speed of Franz LISP, and one tenth the speed of Portable Standard LISP on the VAX 11/750.

Conclusions

The UOLISP compiler runs on almost any Z80 based machine with a minimum storage configuration of 32k bytes and a disk drive. The compiler and optimizer have been tested under both CP/M and the TRS-80 Model I and III with success. Turning on all of the optimizations slows down compilation by approximately 40 percent. The UOLISP compiler occupies 3750 bytes of storage and the optimizer with statistics collection another 3000 bytes. Standard Use has debugging done without the presence of the optimizer and the final run with the optimizer enabled.

List of References

1. J. Marti, 'UOLISP Users Manual', University Oregon Department of Computer and Informatic Science Technical Report CIS-TR-80-18.

2. J. Marti, A. C. Hearn, M. L. Griss, C. Gris 'The Standard LISP Report', SIGPLAN Notices, Vol. 14, No. 10, (October 1979), pp. 48-68.

3. A. C. Hearn, 'REDUCE 2 Users Manual', Utah Symbolic Computation Group UCP-19, Universit Utah, 1973.

4. J. Marti, 'A Session with the Little Meta Translator Writing System', University of Or Technical Report CIS-TR-82-01.

5. J. W. Davidson, C. W. Fraser, 'The Design a Application of a Retargetable Peephole Optim ACM TOPLAS, Vol. 2, No. 2, (April 1980), pp. 191-202.

6. A. S. Tannenbaum, H. van Staveren, J. W. Stevenson, 'Using Peephole Optimization on Intermediate Code', ACM TOPLAS, Vol. 4, No. (January 1982), pp. 21-36.

7. D. A. Lamb, 'Construction of a Peephole Opt izer', Software Practice and Experience, Vol No. 6, (June 1981), pp. 639-647.

8. M. L. Griss, PSL Interest Group Newsletter February 1982.

9. M. L. Griss, 'A Portable LISP Compiler', So ware Practice and Experience, Vol. 11, (June 1981), pp. 541-605.

10. 'The Portable Standard LISP Users Manual', The Utah Symbolic Computation Group, Univers. of Utah, January 1982.