An evaluation of Major Lisp Compilers

Seika Abe
Knowledge Engineering Division, MSI

October 8, 2009

Abstract

Lisp had not been good at numerical computations for their exotic design. This situation is however said to have been changed recently, because by giving appropriate declarations, modern Lisp compilers can generate good code for numerical computation competitive with traditional compilers of Fortran and C. Nevertheless, there is still a significant difference between those generated codes in the sense of execution efficiency. In this report, we investigate what is the sources of the significant difference for the three major Common Lisp implementations, ACL (Allegro Common Lisp 8.1), SBCL (Steel Bank Common Lisp 1.0.23), and LispWorks (6.0).

Contents

1 Introduction

To clarify our point of view on this Lisp compiler evaluations, we briefly summarize typical processes of traditional optimizing compilers.

1. Basic block and flow graph

   The instruction sequence of a program can be divided into blocks where each block is a subsequence of the sequence including just one branch instruction at the end. This blocks are called basic blocks. And then the program can be regarded as a directed graph consisting of the basic blocks as nodes and links corresponding to branches between source and destination basic blocks. This graph is called flow graph.

2. Data flow analysis

   Typical optimizing compilers firstly translate a given program into the flow graph of it, and then implement processes of optimizations and code generation as transformations on the flow graph. To generate better object codes, the processes need various static analysis. Those analyses are generically called data flow analysis. Flow graph is one of the most suitable expression for such analysis.

3. Code optimization

   Instructions in the flow graph of a given program are usually independent from target machine instructions and are thus called intermediate codes. Translating the intermediate codes into more efficient ones without changing the semantics of the program is called code optimization. We will investigate which Lisp compiler does which code optimization. The mentioned code optimizations are following very basic ones.

   (a) constant folding
   Replacing compile-time computable subexpressions with their values

   (b) constant propagation
   Replacing variables of constant values with their values

   (c) common-subexpression elimination
   Reducing redundant evaluations for subexpressions that evaluate to the same value

   (d) loop-invariant hoisting
   Hoisting expressions in a loop that evaluate to the same value each time out of the loop

   Code optimizations done within a basic block, such as constant folding, are called local optimizations, while done for entire flow graph are called global optimizations. Constant propagation can be local (local constant propagation) or global (global constant propagation). The former is easy but the latter requires a data flow analysis to find a variable whose value is always a constant at a point. Common-subexpression elimination also can be local or global and the global version requires another data flow analysis.

4. Instruction selection

   After code optimizations for a given flow graph, instruction selection phase rewrite the intermediate codes of the flow graph in the real target machine instructions. This is one of the most important part of code generators.

   Instead of writing this part by hand for a given target machine, compilers with a code generator generator that generates the instruction selection part of the compiler for the target machine from its machine description is called retargetable compilers. For example, GCC is one of the most widely used powerful C compiler.

   The machine description for a target machine in fact includes not only the definition of the instructions but various target machine dependent information, such as the structure of registers and their usage, the bit width of integers, etc. And the description is translated into instruction selection module and the other modules depending on the target machine. Finally, these generated modules are linked together with target machine independent modules to make the compiler for the target machine.

$$\text{Machine description} \xrightarrow{\text{Machine description compiler}} \text{Instruction selector, etc}$$

Since the instruction selector of a compiler is a complex program, writing it by hand is a messy work and thus may make serious bugs. This is very true when the compiler tries to make use of the instructions skillfully for a eccentric CISC machine, such as X86. On the other hand, recent retargetable compiler technology is mature and can provide methods to generate almost optimal instruction selector form a given target machine description. In particular, employing a dynamic programming based code generator with automatic code generator generator is the current trend.

5. Register allocation

   At the final stage of code generation, variables of intermediate langrage, called virtual registers, are replaced with real registers of target machine. This process is called register allocation. This is one of the most important part of code generators along with instruction selection. Therefore, various approachs have been proposed, studied, and implemented. And a method called graph coloring allocator is the current trend.

   For a given flow graph, this method firstly collects constraint conditions imposed on possible register assignment using a data flow analysis, called liveness analysis, and builds a graph expressing the constraint conditions. By this graph, register allocation problem can be regarded as a graph coloring problem that assigns colors (real registers) to nodes (virtual registers) under the condition that each adjacent nodes have different colors. Finally, graph coloring allocator solves this coloring problem by employing graph theoretic techniques.

   Although the graphs usually requires huge memory and the data flow analysis has high computational cost, this method achieves very good allocations. Thus this method is the current standard for optimizing compilers.

2 Circumstances of Lisp

In this section, we describe why Lisp is not good at numerical computations. The following code is written by a C programmer to add two integers.

```
(defun int-add-1 (x y)
  (+ x y))
```

ACL Lisp compiler generates the following long object code (another compiler should generate similar long code). The C programmer will feel faint.

```
 0: 55          pushl    ebp
 1: 8b ec       movl     ebp,esp
 3: 83 ec 28 subl     esp,$40
 6: 89 75 fc movl     [ebp-4],esi
 9: 89 5d e4 movl     [ebp-28],ebx
12: 39 a3 be 00 cmpl  [ebx+190],esp
    00 00
18: 76 03       jbe      23
20: ff 57 43 call     *[edi+67]   ; SYS::TRAP-STACK-OVFL
23: 83 f9 02 cmpl     ecx,$2
26: 74 03       jz       31
28: ff 57 8b call     *[edi-117]         ; SYS::TRAP-WNAERR
31: 80 7f cb 00 cmpb  [edi-53],$0        ; SYS::C_INTERRUPT-PENDING
35: 74 03       jz       40
37: ff 57 87 call     *[edi-121]         ; SYS::TRAP-SIGNAL-HIT
40: 8b d8       movl     ebx,eax
42: 0b da       orl      ebx,edx
44: f6 c3 03 testb    bl,$3
47: 75 0e       jnz      63
```

```
 49: 8b d8      movl     ebx,eax
 51: 03 da      addl     ebx,edx
 53: 70 08      jo       63
 55: 8b c3      movl     eax,ebx
 57: f8           clc
 58: c9           leave
 59: 8b 75 fc movl     esi,[ebp-4]
 62: c3           ret
 63: 8b 5f 8f movl     ebx,[edi-113]     ; EXCL::+_2OP
 66: ff 57 27 call     *[edi+39]  ; SYS::TRAMP-TWO
 69: eb f3      jmp      58
 71: 90           nop
```

SYS::TRAP-STACK-OVFL , SYS::C_INTERRUPT-PENDING , and EXCL::+_2OP are overflow checking function, interrupt checking function, and addition function, respectively.

2.1 Interactive environment

Good interactive environment is an attractive point of Lisp. To keep this, compiler must insert some additional code, such as stack overflow checking and interrupt checking, in object code. But we want to remove such codes from the final product. This can be done by the following declaration.

```
(defun int-add-2 (x y)
  (declare (optimize (speed 3) (safety 0) (debug 0)))
  (+ x y))
```

Even after removing such codes, generated code is still long.

```
  0: 8b d8    movl     ebx,eax
  2: 0b da    orl      ebx,edx
  4: f6 c3 03 testb    bl,$3
  7: 75 0d    jnz      22
  9: 8b d8    movl     ebx,eax
 11: 03 da    addl     ebx,edx
 13: 70 07    jo       22
 15: 8b c3    movl     eax,ebx
 17: f8         clc
 18: 8b 75 fc movl     esi,[ebp-4]
 21: c3         ret
 22: 8b 5f 8f movl     ebx,[edi-113]     ; EXCL::+_2OP
 25: ff 67 27 jmp      *[edi+39]  ; SYS::TRAMP-TWO
```

The C programmer should expect just addl of Location 11.

2.2 Polymorphism

Lisp is a dynamic typing language; the types of varialbles may change at runtime (aka polymorphism). Most of built-in functions do coercion (aka ad-hoc polymorphism) at execution time. For example, evaluating the form (+ x y) executes addition of integral, rational, floating, or complex depending on the arguments.

To implement such polymorphism, Lisp variables holds pointers to objects instead of objects itself. When C language simply add two integers, Lisp firstly needs to get objects by referencing pointers, then extract two integer values from the objects, add the two values, and finally the sum must be stored into a lisp object of type integer, and the pointer of the object is returned as the result. This complected processes are thus implemented as the function EXCL::+_2OP.

To solve this terrible situation, Lisp employs special representation for small integers called fixnum type. By this representation, fixnum integers are stored directory into Lisp variables. If the pointer value of a Lisp variable is a multiple of 4, it is regarded as a fixnum value (with 4 times), instead of a pointer to a object. For each Lisp variable, its low 2 bits are, in this way, used as a tag bits. This fixnum tag expression is used in ACL, SBCL, and LispWorks.

In the generated code of `int-add-2` at Location 2, incoming arguments (via eax and edx) are firstly checked their tag bits, and both of them are fixnum, they can be added by `addl` instruction, otherwise call universal procedure `EXCL::+_2OP`.

What the C programmer really wanted is just addition of simple integer, not of complex numbers. For this purpose, additional declaration that allow compiler to assume the type of specified variable is always fixnum is as follows.

```
(defun int-add-1 (x y)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type fixnum x y))
  (+ x y))
```

By doing this, the following code the C programmer wanted is generated (prologue and epilogue codes are ommited).

SBCL and LispWorks also accept similar type declaration, but there is a subtle difference. Unlike ACL, SBCL and LispWorks do not assume the resulting type of fixnum operation to be fixnum. Thus, the last line of the above example shuld be `(the fixnum (+ x y))`. From now on, we take the ACL's simpler assumption for brevity. And when a program in this report is compiled with SBCL or LispWorks, each fixnum operations are previously enclosed with `(the fixnum ...)`.

```
0: 03 c2    addl     eax,edx
```

Of course, the C programmer should understand that fixnum type is not equal to int of C because 2 bits are used for tag of fixnum. Unlike addition and multiplication, some operations, such as division, cannot directly be applied. They need a care of the tags.

3 Evaluation of code optimization

In this section, we will investigate which Lisp compiler does which code optimization.

3.1 Constant folding

We begin with a simple constant folding. The object code of the following program will give us whether a Lisp compiler do or do not the optimization. In this case is very simple since the arguments of `+` are all constants. In fact, this is just constant computation at compile time.

```
(defun test-const-fold-1 ()
  (declare (optimize (speed 3) (safety 0) (debug 0)))
  (+ 1 2 3))
```

ACL compiler generates the following object code. Constant compuations are apparently performed. The register `eax` used for return value is set to 24 instad of 6. This is because the needed conversion to fixnum tag expression.

Generated code actually includes miscellaneous code, such as prologue and epilogue code, constant table setting-up code, etc. We will ommit them from now on for brevity.

```
0: b8 18 00 00 movl  eax,$24      ; 6
```

SBCL compiler also performs the constant computation as the following generated code shows. SBCL use `edx` as the return value register.

```
0AE0E03A:       BA18000000       MOV EDX, 24
```

LispWorks compiler also performs as the following shows. Return value register is `eax` like ACL compiler.

```
0:      B818000000       move  eax, 18
```

In general, constant folding attempt for a given expression to gather up constant parts of it by using algebraic identities, and fold the gathered parts. Thus, it is not so simple. For example, the constant folding module of GCC (GNU C compiler) has over 10000 lines. For example, GCC can perform the following constant folding.

```
(1+n)+(n+2)*2  →  3*n + 5
```

One might think that such optimization is not serious because skillful programmers can do this by coding time. But these redundant expressions often arise a result of various optimizations. Thus optimizing compilers often call constant folding after those optimizations. And the folding is important to improve the quality of the next code optimizations.

Therefore a compiler performing good code optimizations has a good constant folder. in other words, poor constant folder reveals poor code optimizations. The following is a program for testing simple constant folding; the expression should be folded to 0 (fixnum).

```
(defun test-const-fold-2 (n)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type fixnum n))
  (+ 1 n -1 (- n)))
```

ACL does not perform this constant folding as the following generated code shows.

```
 2: 83 c3 04 addl     ebx,$4
 5: 83 c3 fc addl     ebx,$-4
 8: 33 d2     xorl    edx,edx
10: 2b d0     subl    edx,eax
12: 8b c3     movl    eax,ebx
14: 03 c2     addl    eax,edx
```

The policy of arithmetics of fixnum-type declared values of SBCL and LispWorks is different from that of ACL, and enclosing sub-expressions of `(+ 1 n -1 (- n))` with `(the fixnum ...)` may block the folding of them. Thus in this case, we give the program as-is to SBCL and LispWorks. After all, however, they do not perform fold for this case. SBCL generates the following code, which shows the fact.

```
 0A950DDD:       8D4201          LEA EAX, [EDX+1]
       E0:       83C0FF          ADD EAX, -1
       E3:       F7DA            NEG EDX
       E5:       01D0            ADD EAX, EDX
       E7:       6BD004          IMUL EDX, EAX, 4

      ... snip ...
```

And the generated code by LispWorks is as follows, which also shows the fact.

```
       0:      55              push  ebp
       1:      89E5            move  ebp, esp
       3:      50              push  eax
       4:      6A04            pushb 4
       6:      B502            moveb ch, 2
       8:      8B45FC          move  eax, [ebp-4]
      11:      FF15BCF90320    call  [2003F9BC]       ; SYSTEM:+$FIXNUM

      ... snip ...
```

3.2 Constant propagation

Next, we investigate constant propagation by inspecting the generated code of the following program. If a compiler do the optimization, the generated code should simply return the constant 3.

```
(defun test-const-propagation-1 ()
  (declare (optimize (speed 3) (safety 0) (debug 0)))
  (let ((a 1)
        (b 2))
    (+ a b)))
```

ACL generates the following code.

```
 0: bb 04 00 00 movl        ebx,$4       ; 1
    00
 5: ba 08 00 00 movl        edx,$8       ; 2
    00
10: 8b c3      movl eax,ebx
12: 03 c2      addl eax,edx
```

As this shows, constant propagation is not performed in ACL. Macro feature of Lisp is very powerful. Incorporating work of constant propagation and constant folding may drastically simplify macros. Thus not only for numerical computations, this optimization should be essential for extensive use of macros.

SBCL and LispWorks, on the other hand, performs constant propagation for the above simple case that let binded constant value variables are simply used. The following code is by SBCL (LispWorks also generates similar code).

```
BA0C000000       MOV EDX, 12
```

But the compilers give up to propagate when a subject let binded variable is assigned some value because the following program that can apparently propagate constants shows.

```
(defun test-const-propagation-2 ()
  (declare (optimize (speed 3) (safety 0) (debug 0)))
  (let ((a 1)
        (b 2))
    (declare (type fixnum a b))
    (setq a (the fixnum (+ a b)))
    a))
```

SBCL fails to propagete the constants.

```
0AADC782:       B804000000       MOV EAX, 4
      87:       83C008           ADD EAX, 8
```

LispWorks also fails.

```
0:      83C004           add   eax, 4
3:      83C00C           add   eax, C
```

Although Lisp is a functional programming language, actual programs include many assignment. To use macros at ease, these optimization should be essential.

3.3 Common-subexpression elimination

It is rare a complex common-subexpression appears in a source program directly because such program is less readable and also hard to maintain. But repeating small expressions, such as `i+1` is not unusual.

More serious things are the existence of 'hidden' common-subexpressions. Calculations of indices of an array is a typical example. Even `A[i]+B[i]` includes two common-subexpressions, namly, multiplications of `i` and the size of array elements. Thus, this optimization is essential for numerical computations that use many arrays. The following test program has the three common-subexpressions, `(- x y)`.

```
(defun test-common-subexpression-elimination (x y)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type fixnum x y))
  (+ (- x y) (- x y) (- x y)))
```

ACL generates the following code.

```
12: 8b d8       movl ebx,eax   ; eax=x
14: 2b da       subl ebx,edx   ; edx=y  <<
16: 89 45 dc    movl [ebp-36],eax
19: 29 55 dc    subl [ebp-36],edx       <<
22: 03 5d dc    addl ebx,[ebp-36]
25: 8b c8       movl ecx,eax
27: 2b ca       subl ecx,edx            <<
29: 8b d1       movl edx,ecx
31: 8b c3       movl eax,ebx
33: 03 c2       addl eax,edx
```

ACL does not perform this optimization because there is three subtractions here. There is also a register spill here as the work on frame [ebg-36] is used. If the optimization is performed, the following concise code would be possible.

```
subl  eax,edx
movl  edx,eax
addl  edx,eax
addl  edx,eax
movl  eax,edx
```

SBCL and LispWorks also do not performe this optimization and the subtraction is repeated three times. See appendix for their generated codes.

3.4 Loop optimization

Finally, as a basic loop optimization, we take loop invariant hoisting. Like common-subexpressions elimination, complex loop invariant expressions do not usually appear directory in souce programs. But there are still 'hidden' loop invariants, such as index calculations and loops generated by macros. This optimization is essential for numerical computations that use many loops. The following program has the loop invariant expression (+ n 10).

```
(defun test-loop-invariant-hoisting (n)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type fixnum n))
  (let ((s 0))
    (declare (type fixnum s))
    (dotimes (i n)
      (incf s (+ n 10)))
    s))
```

ACL generates the following code.

```
21: 89 45 d8      movl     [ebp-40],eax
24: 83 45 d8 28   addl     [ebp-40],$40 ; 10
28: 03 5d d8      addl     ebx,[ebp-40]
31: 83 c2 04      addl     edx,$4
34: 3b 55 dc      cmpl     edx,[ebp-36]
37: 7c ee         jl       21
```

ACL does not perform this optimization because the loop invariant (+ n 10) is still in loop. SBCL and LispWorks also do not perform this optimization. See appendix for their generated codes.

3.5 Conclusion of this section

Each Lisp compiler does not perform any traditional code optimizations. Although there are more traditional optimizations, but those also should not be performed since optimizations mentioned above are very basic ones.

4 Evaluation of register allocation

One might think that compilers performing code optimizations are called optimizing compiler. It is not false, but instruction selection and register allocation is more important than the code optimizations. If these part is poor, effort to improve code optimizatoin is nonsence.

Unlike evaluations of code optimizations, it is hard to decide what techniques are used in instruction selection and register allocation by just inspecting generated object codes. Thus, we mainly investigate the real registers that can be assigned to virtual registers. We call here the set general registers simply.

Usually, a compiler does not use all of the real registers of the target machine. For example, sepcial purpose registers, such as program counter, stack pointer and frame pointer cannot be used as a general register. Or, compiler may use some registers for special purpose. Such design restricts possible general registers.

On X86 CPU, possible general registers are, by excluding program counter, stack pointer, frame pointer, and condition code from all registers, only 6 registers, namely `eax`, `ebx`, `ecx`, `edx`, `esi`, and `edi`. For each compiler, we investigate its general registers by inspecting programs with high register pressure (program with high possibility of register spills).

4.1 ACL

We use the following program pattern to tune register pressure.

```
(defun test-regalloc-nreg-3 (x)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type fixnum x))
  (let ((y1 (+ x 1))
        (y2 (+ x 2))
        (y3 (+ x 3)))
    (+ y1 y2 y3)))
```

ACL generates the following beautiful code that each variable is assigned to a general register.

```
 0: 8b d8       movl  ebx,eax
 2: 83 c3 04    addl  ebx,$4
 5: 8b d0       movl  edx,eax
 7: 83 c2 08    addl  edx,$8
10: 83 c0 0c    addl  eax,$12
13: 03 da       addl  ebx,edx
15: 03 c3       addl  eax,ebx
```

Next, we increase register pressure.

```
(defun test-regalloc-nreg-4 (x)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type fixnum x))
  (let ((y1 (+ x 1))
        (y2 (+ x 2))
        (y3 (+ x 3))
        (y4 (+ x 4)))
    (+ y1 y2 y3 y4)))
```

At this point, a register spill arise.

```
12: 8b d8       movl  ebx,eax
14: 83 c3 04    addl  ebx,$4
17: 8b d0       movl  edx,eax
19: 83 c2 08    addl  edx,$8
22: 89 45 dc    movl  [ebp-36],eax
25: 83 45 dc 0c addl  [ebp-36],$12
29: 83 c0 10    addl  eax,$16
32: 03 da       addl  ebx,edx
34: 03 5d dc    addl  ebx,[ebp-36]
37: 03 c3       addl  eax,ebx
```

After all, ACL compiler uses only the three registers `eax`, `ebx`, and `edx` as general registers. We could not find the usage policy of `ecx`. It is sometimes used as a loop index variable and sometimes as a temporary. And in ACL, `esi` and `edi` are used to hold basis of constant table and built-in function table, respectively. For example, generated code of a program including `(cons (+ x 1) 1.23)` includes the following code fragment.

```
 0: 8b 56 12    movl  edx,[esi+18]    ; 1.23
 3: 8b 5f 8f    movl  ebx,[edi-113]   ; EXCL::+_2OP
                ...............
```

Real registers `esi` and `edi` are used only for this purposes. Lisp must implement a dynamic linking facility. This fixed usage may due to the situation. Such approach should be natural for RISC machines with rich registers, but not for poor register machines, such as X86.

For a reference, we also investigate ACL for SPARC. SPARC has 8 global reigsters, 8 local reigsters and 6 IO registers. These registers can be used freely under the condition of the SPARC calling convention. The following is the object code for program of increased register pressure `test-regalloc-nreg-4`. There are no spills here. See also appendix.

```
  4: 98062004     add  %i0, #x4, %o4
  8: 96062008     add  %i0, #x8, %o3
 12: 9406200c     add  %i0, #xc, %o2
 16: 92062010     add  %i0, #x10, %o1
 20: 90062014     add  %i0, #x14, %o0
 24: a0062018     add  %i0, #x18, %l0
 28: a206201c     add  %i0, #x1c, %l1
 32: a4062020     add  %i0, #x20, %l2
 36: a6062024     add  %i0, #x24, %l3
 40: a8062028     add  %i0, #x28, %l4
 44: 9803000b     add  %o4, %o3, %o4
 48: 9803000a     add  %o4, %o2, %o4
 52: 98030009     add  %o4, %o1, %o4
   ...
```

4.2 SBCL and LispWorks

For ACL, `test-regalloc-nreg-4` causes spills but SBCL does not as the following generated code shows. LispWorks also does not.

```
0AF62E75:       8D4201           LEA EAX, [EDX+1]
      78:       8D4A02           LEA ECX, [EDX+2]
      7B:       8D5A03           LEA EBX, [EDX+3]
      7E:       8D7204           LEA ESI, [EDX+4]
      81:       8D1408           LEA EDX, [EAX+ECX]
      84:       01DA             ADD EDX, EBX
      86:       01F2             ADD EDX, ESI
      88:       C1E202           SHL EDX, 2
```

We investigate the spill threshold by increasing register pressure, like in the case of ACL, SBCL and LispWorks use all of the possible registers `eax`, `ebx`, `ecx`, `edx`, `esi`, and `edi` as their general registers. The source of this advantage is implementations of their dynamic linkers. Unlike ACL, dynamic linkers of SBCL and LispWorks stores absolute addresses into the generated codes as follows and thus do not use special base registers.

```
MOV EDI, [#xB0F24F0] ; 1.23
CALL #x1000140       ; GENERIC-+
```

4.3 Conclusion of this section

We guess that ACL firstly imported on a rich register machine. Thus the design of its dynamic linker is based on the machine. But for X86, only 3 registers are apparently insufficient. ACL (for x86) needs to modify the design of the dynamic linker to free the base registers `esi` and `edi`. This is not a problem of the register allocator.

LispWorks say in a document that their register allocator is a graph coloring allocator. We do not know what algorithms are used for register allocators of ACL and SBCL.

Finally, we mention instruction selection of SBCL. SBCL often generates `LEA` instruction as follows.

```
81:       8D1408           LEA EDX, [EAX+ECX]
```

This code skillfully performs `EDX = EAX + ECX` by using addressing mode computation and `LEA`. On the other hand, we never see such code in generated code by ACL. We think that instruction selector of SBCL is better than that of ACL and LispWorks, but we do not know whether SBCL has a retargetable code generator.

5 Evaluation of floating-point arithmetics

Floating-point arithmetics are very important for numerical computations. For floating point numbers, it is impossible to keep them in variables directly by tagging like fixnum, and thus all floating point numbers are boxed. This is a serious reason why Lisp is not good at numerical computations.

But within a function, it is possible to use unboxed (raw) floating values. At the entry point of the function, floating values could be unboxed, and the only need to box the result at the function return.

Since X86 only supports 80 bits floating point arithmetics, this optimization is applied to double-float type (64 bits). X86 has 8 floating point registers `st(0)` .. `st(7)`. The structure of them is very different from conventional real registers because the first operand of a operation must be stored in `st(0)` (abbreviated as `st`), and after the operation, `st(0)` is replaced with the result of the operation, and `st(1)` is replaced with `st(2)`, `st(2)` with `st(3)` and so on. That is, so to speak, a stack structure.

Therefore, standard register allocation method cannot be applied, but ACL, SBCL and LispWorks implement this optimization. The following program adds two double-float numbers.

```
(defun test-float-add (a b)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type double-float a b))
  (+ a b))
```

ACL generates the following code. (unboxing and boxing codes are omitted)

```
 0: dd 42 f6 fldq [edx-10]
 3: dd da     fstp st(2)
 5: dd 40 f6 fldq [eax-10]
 8: dd db     fstp st(3)
10: d9 c1     fld st,st(1)
12: d8 c3     fadd st,st(3)
14: dd da     fstp st(2)
16: d9 c1     fld st,st(1)
18: dd d9     fstp st(1)
```

In the function body, raw (unboxed) floating-point numbers are directly processed with real instructoins. SBCL also generates simular concise code.

To enable this optimization, LispWorks requires additional declaration `(float 0)` in `(declare (optimize` .... The following LispWorks's generated code is redundant compared to that of ACL or SBCL.

```
11:     8B7D08          move  edi, [ebp+8]
14:     DD4705          fldl  [edi+5]
17:     DD4005          fldl  [eax+5]
20:     DEC1            faddp st(1), st
22:     DD5DF8          fstpl [ebp-8]
25:     83EC08          sub   esp, 8
28:     8B75F8          move  esi, [ebp-8]
31:     8975F0          move  [ebp-10], esi
34:     8B75FC          move  esi, [ebp-4]
37:     8975F4          move  [ebp-C], esi
40:     8B7500          move  esi, [ebp]
43:     8975F8          move  [ebp-8], esi
46:     83ED08          sub   ebp, 8
49:     8B750C          move  esi, [ebp+C]
52:     897504          move  [ebp+4], esi
55:     DD45F8          fldl  [ebp-8]
58:     DD5D0C          fstpl [ebp+C]
```

6 Bench marks

In this section, we compare execution time for each generated object code by ACL, SBCL and LispWorks.

6.1 Integer arithmetics and spill

The following program has only fixnum arithmetics. By the type declaration, there are no boxed integer and all operations are inlined. Thus, in fact, this test is intended to evaluate the efficiency of register allocators.

```
(defun test-intop-spill (n)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type fixnum n))
  (let ((sum 0))
    (declare (type fixnum sum))
    (dotimes (i n)
      (dotimes (j n)
        (dotimes (k n)
          (incf sum))))
    sum))
```

ACL generates the following code. As ACL utilize only 3 registers as general registers, some spills arise here.

```
 6: 89 75 fc  movl     [ebp-4],esi
 9: 89 5d e4  movl     [ebp-28],ebx
12: 33 db     xorl    ebx,ebx
14: 33 d2     xorl    edx,edx
16: 89 45 d8  movl     [ebp-40],eax        ; EXCL::LOCAL-1
19: 3b 55 d8  cmpl     edx,[ebp-40]        ; EXCL::LOCAL-1
22: 7c 08     jl      32
24: 8b c3     movl    eax,ebx
26: f8        clc
27: c9        leave
28: 8b 75 fc  movl     esi,[ebp-4]
31: c3        ret
32: 89 5d dc  movl     [ebp-36],ebx        ; EXCL::LOCAL-0
35: 33 db     xorl    ebx,ebx
37: 89 45 d4  movl     [ebp-44],eax        ; EXCL::LOCAL-2
40: 3b 5d d4  cmpl     ebx,[ebp-44]        ; EXCL::LOCAL-2
43: 7c 08     jl      53
45: 83 c2 04  addl     edx,$4
48: 8b 5d dc  movl     ebx,[ebp-36]        ; EXCL::LOCAL-0
51: eb de     jmp     19
53: 89 55 d0  movl     [ebp-48],edx        ; EXCL::LOCAL-3
56: 33 d2     xorl    edx,edx
58: 89 45 cc  movl     [ebp-52],eax        ; EXCL::LOCAL-4
61: eb 07     jmp     70
63: 83 45 dc 04 addl  [ebp-36],$4 ; EXCL::LOCAL-0
67: 83 c2 04  addl     edx,$4
70: 3b 55 cc  cmpl     edx,[ebp-52]        ; EXCL::LOCAL-4
73: 7c f4     jl      63
75: 83 c3 04  addl     ebx,$4
78: 8b 55 d0  movl     edx,[ebp-48]        ; EXCL::LOCAL-3
81: eb d5     jmp     40
```

SBCL generates the following code that has no spills and concise.

```
0B0D20AA:       31C0            XOR EAX, EAX
      AC:       31FF            XOR EDI, EDI
      AE:       EB20            JMP L5
      B0: L0:   31C9            XOR ECX, ECX
      B2:       EB13            JMP L4
      B4: L1:   31DB            XOR EBX, EBX
      B6:       EB06            JMP L3
      B8: L2:   83C004          ADD EAX, 4
      BB:       83C304          ADD EBX, 4
      BE: L3:   8BF2            MOV ESI, EDX
      C0:       39F3            CMP EBX, ESI
      C2:       7CF4            JL L2
      C4:       83C104          ADD ECX, 4
      C7: L4:   8BDA            MOV EBX, EDX
      C9:       39D9            CMP ECX, EBX
      CB:       7CE7            JL L1
      CD:       83C704          ADD EDI, 4
      D0: L5:   39D7            CMP EDI, EDX
      D2:       7CDC            JL L0
      D4:       8BD0            MOV EDX, EAX
```

```
    D6:       8D65F8          LEA ESP, [EBP-8]
    D9:       F8              CLC
    DA:       8B6DFC          MOV EBP, [EBP-4]
    DD:       C20400          RET 4
```

LispWorks generates the following code which has some spills and somewhat redundant.

```
      20:     33FF            xor   edi, edi
      22:     837DF400        cmp   [ebp-C], 0
      26:     7E3E            jle   L5
      28:     33DB            xor   ebx, ebx
      30:     8B75F4          move  esi, [ebp-C]
      33:     83EE04          sub   esi, 4
      36:     8975F8          move  [ebp-8], esi
L1:   39:     837DF400        cmp   [ebp-C], 0
      43:     7E32            jle   L6
      45:     33C9            xor   ecx, ecx
      47:     8B75F4          move  esi, [ebp-C]
      50:     83EE04          sub   esi, 4
      53:     8975FC          move  [ebp-4], esi
L2:   56:     837DF400        cmp   [ebp-C], 0
      60:     7E2D            jle   L7
      62:     33D2            xor   edx, edx
      64:     8B45F4          move  eax, [ebp-C]
      67:     83E804          sub   eax, 4
L3:   70:     83C704          add   edi, 4
      73:     3BD0            cmp   edx, eax
      75:     7D1E            jge   L7
      77:     83C204          add   edx, 4
      80:     8955F0          move  [ebp-10], edx
      83:     EBF1            jmp   L3
L4:   85:     E8A6A90700      call  20114962
L5:   90:     C9              leave
      91:     89F8            move  eax, edi
      93:     FD              std
      94:     C3              ret
L6:   95:     3B5DF8          cmp   ebx, [ebp-8]
      98:     7DF6            jge   L5
     100:     83C304          add   ebx, 4
     103:     89DA            move  edx, ebx
     105:     EBBC            jmp   L1
L7:  107:     89CA            move  edx, ecx
     109:     3B55FC          cmp   edx, [ebp-4]
     112:     7DED            jge   L6
     114:     83C104          add   ecx, 4
     117:     89CA            move  edx, ecx
     119:     EBBF            jmp   L2
```

The execution times on a X86 Linux machine with `n = 2000` are as follows. This results shows the importance of register allocator.

```
    SBCL           5.183 sec
    LispWorks      7.577 sec
    ACL           12.640 sec
```

6.2 Floating point arithmatics

The following function is a performance clinical part of a graph drawing function cited from a real application. Most operations are floating arithmetics here.

```
(defstruct bmk-kk-position
  (x 0 :type fixnum)
  (y 0 :type fixnum))

(defmacro expt2 (x)
  (let ((y (gensym)))
```

```
    `(let ((,y ,x)) (* ,y ,y))))

(defun bmk-kk-Exm (nvec dist L m xm ym)
  (declare (optimize (speed 3) (safety 0) (debug 0) #+lispworks (float 0))
           (type (simple-array t (*)) nvec)
           (type (simple-array fixnum (* *)) dist)
           (type double-float L)
           (type fixnum m xm ym))
  (let ((sum 0.0d0))
    (declare (type double-float sum))
    (dotimes (i (length nvec))
      (unless (= i m)
        (let* ((p (aref nvec i))
               (xi (bmk-kk-position-x p))
               (yi (bmk-kk-position-y p)))
          (declare (type fixnum xi yi))
          (let ((ym-yi (float (- ym yi) 0.0d0))
                (xm-xi (float (- xm xi) 0.0d0)))
            (incf sum (/ (- xm-xi (/
                                   (* L (aref dist m i)
                                      xm-xi)
                                   (sqrt (the (double-float 0.0d0 *)
                                           (+ (expt2 xm-xi)
                                              (expt2 ym-yi))))))
                         (expt2 (float (aref dist m i) 0.0d0)))))))))
    sum))
```

The execution times with `nvec` of 1000 elements and 10000 times calling of the function are as follows.

```
SBCL          0.400 sec
ACL           2.730 sec
LispWorks     9.440 sec
```

7 Conclusion

Each compiler can generate real instructions with real register allocation, which are basic requirements for real compilers. But their qualities are very different. Concerning this points, we think, the first is SBCL, and then ACL and LispWorks.

The most serious problem of ACL is its register usage. For X86, ACL uses only 3 registers as general registers. Thus spills easily arise.

This is not the problem of the register allocator but the problem of the policy of register usage. Anyway and unfortunately, X86 is the most widely used CPU. Thus, ACL needs to improve this to generate good code for X86.

Any compiler does not do useful code optimizations. But Lisp is a general purpose programming languate now and programmers expect Lisp to perform code optimizations. Common Lisp uses lexical scope and its specification is rigorously defined. And there are no problematical features such as common, equivalence in Fortran, or unlimited pointers in C. Thus, we think, implementing the optimizations mentioned in this report in Lisp seems not hard.

In other words, any Lisp compiler can be improved more. Lisp has high productivity with mature develop environment. Lisp should be wildly used. Therefore, we expect a Lisp compiler which perform optimizations we mentioned in this report.

A Comments to section 3

The generated codes for test program of common-subexpressions elimination and loop optimization by SBCL and LispWorks, ommited in section 3, are as follows.

A.1 SBCL

This is the generated code for `test-common-subexpression-elimination`. Common-subexpression elimination is not performed as it has three subtractions.

```
 0AE9670A:       8BC2             MOV EAX, EDX
       0C:       29F8             SUB EAX, EDI <<
       0E:       8BC8             MOV ECX, EAX
       10:       8BC2             MOV EAX, EDX
       12:       29F8             SUB EAX, EDI <<
       14:       01C1             ADD ECX, EAX
       16:       29FA             SUB EDX, EDI <<
       18:       01D1             ADD ECX, EDX
       1A:       8BD1             MOV EDX, ECX
```

This is the generated code for `test-loop-invariant-hoisting`. The loop invariant `(+ n 10)` is not moved outside of the loop.

```
 0AE75C82:       31C0             XOR EAX, EAX
       84:       31C9             XOR ECX, ECX
       86:       EB13             JMP L1
       88: L0:   8BDA             MOV EBX, EDX
       8A:       C1FB02           SAR EBX, 2
       8D:       83C30A           ADD EBX, 10  <<
       90:       C1F802           SAR EAX, 2
       93:       01D8             ADD EAX, EBX
       95:       C1E002           SHL EAX, 2
       98:       83C104           ADD ECX, 4
       9B: L1:   39D1             CMP ECX, EDX
       9D:       7CE9             JL L0
       9F:       8BD0             MOV EDX, EAX
```

A.2 LispWorks

This is the generated code for `test-common-subexpression-elimination`. Common-subexpression elimination is not performed as it has three subtractions.

```
       0:      8B7C2404         move  edi, [esp+4]
       4:      89FB             move  ebx, edi
       6:      2BD8             sub   ebx, eax      <<
       8:      89FA             move  edx, edi
      10:      2BD0             sub   edx, eax      <<
      12:      03DA             add   ebx, edx
      14:      2BF8             sub   edi, eax      <<
      16:      8D043B           lea   eax, [ebx+edi]
```

This is the generated code for `test-loop-invariant-hoisting`. The loop invariant `(+ n 10)` is not moved outside of the loop.

```
       0:      89C1             move  ecx, eax
       2:      33FF             xor   edi, edi
       4:      83F900           cmp   ecx, 0
       7:      7E21             jle   L2
       9:      55               push  ebp
      10:      89E5             move  ebp, esp
      12:      50               push  eax
      13:      33DB             xor   ebx, ebx
      15:      89CE             move  esi, ecx
      17:      83EE04           sub   esi, 4
      20:      8975FC           move  [ebp-4], esi
L1:   23:      8D4128           lea   eax, [ecx+28]  << 28H = 4 * 10
      26:      03C7             add   eax, edi
      28:      89C7             move  edi, eax
      30:      3B5DFC           cmp   ebx, [ebp-4]
      33:      7D0B             jge   L3
      35:      83C304           add   ebx, 4
      38:      89DA             move  edx, ebx
      40:      EBED             jmp   L1
L2:   42:      89F8             move  eax, edi
      44:      FD               std
      45:      C3               ret
L3:   46:      C9               leave
      47:      EBF9             jmp   L2
```

B 64 bits CPU

Recently, 64 bits CPUs, such as X86-64, are about to be used widely. Our experience shows that some ACL applications, firstly developed on X86, run significantly faster on X86-64 by just re-compiling them. We describe one of the essential reasons by comparing generated codes by ACL for X86 and X86-64.

The enhancement for X86-64 includes bitwidth extensions of the registers to 64, and also increasing the number of registers to 16. The latter is, we think, the reason for the significant performance improvement on X86-64 because ACL generates for `test-regalloc-nreg-4` the following spill-less concise code for X86-64; its generated code included spills for X86.

```
 0: 4c 8b ef       movq       r13,rdi
 3: 49 83 c5 08    add        r13,$8
 7: 4c 8b e7       movq       r12,rdi
10: 49 83 c4 10    add        r12,$16
14: 4c 8b df       movq       r11,rdi
17: 49 83 c3 18    add        r11,$24
21: 48 83 c7 20    add        rdi,$32
25: 4d 03 ec       addq       r13,r12
28: 4d 03 eb       addq       r13,r11
31: 49 03 fd       addq       rdi,r13
```

The result we investigate the spill threshold by the similar way shows that ACL use 9 registers of X86-64 as its general registers. The following concise code is for X86-64 of `test-intop-spill` whose generated code included many spills for X86.

```
 0: 45 33 ed       xorl       r13d,r13
 3: 45 33 e4       xorl       r12d,r12
 6: 4c 8b df       movq       r11,rdi
 9: 4d 3b e3       cmpq       r12,r11
12: 7c 0a          jl 24
14: 49 8b fd       movq       rdi,r13
17: f8             clc
18: 4c 8b 74 24 10 movq       r14,[rsp+16]
23: c3             ret
24: 45 33 c9       xorl       r9d,r9
27: 4c 8b c7       movq       r8,rdi
30: 4d 3b c8       cmpq       r9,r8
33: 7c 06          jl 41
35: 49 83 c4 08    add        r12,$8
39: eb e0          jmp        9
41: 33 c9          xorl       ecx,ecx
43: 48 8b d7       movq       rdx,rdi
46: eb 08          jmp        56
48: 49 83 c5 08    add        r13,$8
52: 48 83 c1 08    add        rcx,$8
56: 48 3b ca       cmpq       rcx,rdx
59: 7c f3          jl 48
61: 49 83 c1 08    add        r9,$8
65: eb db          jmp        30
```

And the execution times with `n = 2000` are as follows.

```
X32         21.344 sec
X32-64      4.992 sec
```

The spill-less code for X32-64 runs about 4 times faster.

C SMP LispWorks

Recently, multi-core CPUs are about to be used widely. Lisp systems supporting such CPU can divide a given task into some Lisp threads and feeds them to the cores; this can improve the throughput aggressively.

Concurrent LispWorks 6.0 is such a Lisp system. We show, for reference, the evaluation result of the Lisp on the multi-core CPU Xeon(R) 2GHz with 8 CPUs.

```
(defun parallel-naive-pi (&optional (nproc 1) (ntimes 1))
  ;; Calculates pi using NPROC processes with initial function naive-pi.
  (let ((pis (make-list (* nproc ntimes) :initial-element 0)))
    (format t "Nproc=~d RealTime:" nproc)
    (dotimes (i ntimes)
      (let ((lock (#+allegro   mp:make-process-lock
                   #+lispworks mp:make-lock))
            (nrests (list nproc))
            (time (get-internal-real-time)))
        ;; Create NPROC processes.
        (loop
            for i from 1 to nproc
            for c on pis
            do (mp:process-run-function (format nil "~d" i)
                 ;; Each process records its (actually the same) result
                 ;; to the list, then increments the semaphore and exit.
                 #+lispworks nil
                 #'(lambda (c)
                     (setf (car c) (naive-pi))
                     (#+allegro   mp:with-process-lock
                      #+lispworks mp:with-lock
                       (lock)
                       (decf (car nrests))))
                 c))
        ;; Wait for the processes to complete.
        (mp:process-wait "wait" #'(lambda (x) (zerop (car x))) nrests)
        (format t " ~s" (/ (float (- (get-internal-real-time) time))
                           internal-time-units-per-second))))
    (format t "~%")
    ;; Returns the mean of results from the processes.
    (/ (apply #'+ pis) (* nproc ntimes))))

(defun naive-pi (&optional (n  #xfffffff))
  ;; Calculates pi using the Gregory series of N (<= #xfffffff) terms.
  (declare (optimize (speed 3) (safety 0) (debug 0) #+lispworks (float 0))
           (type fixnum n))
  (* 4.0d0 (loop
               for i fixnum from 1 to n
               for k fixnum = 1 then (+ k 2)
               for s of-type double-float = 1.0d0 then (- s)
               sum (/ s (float k 0.0d0)) of-type double-float)))
```

The function `parallel-naive-pi` runs `nproc` threads in parallel that each of them executes a relatively heavy numerical computation `naive-pi` with synchronization that waits until all of the threads halt and prints the execution time; this trial is repeated `ntimes` times. The following list is the execution result of the function with `nproc = 1 ..  24` and `ntimes = 4`.

```
CL-USER 3 > (loop for n from 1 to 24 do (parallel-naive-pi n 4))
Nproc=1 RealTime: 10.282 10.234 10.281 10.235
Nproc=2 RealTime: 10.282 10.281 10.281 10.297
Nproc=3 RealTime: 10.281 10.281 10.282 10.297
Nproc=4 RealTime: 10.281 10.281 10.281 10.297
Nproc=5 RealTime: 10.281 20.563 20.562 20.532
Nproc=6 RealTime: 10.281 10.281 20.578 20.563
Nproc=7 RealTime: 10.281 10.281 20.485 20.515
Nproc=8 RealTime: 10.453 20.532 20.515 30.813
Nproc=9 RealTime: 20.5 20.5 20.078 20.516
Nproc=10 RealTime: 20.109 20.578 20.313 20.343
Nproc=11 RealTime: 20.563 20.531 30.578 20.516
Nproc=12 RealTime: 30.5 30.937 30.453 30.954
Nproc=13 RealTime: 30.593 30.844 20.313 30.796
Nproc=14 RealTime: 30.594 30.844 30.562 30.75
Nproc=15 RealTime: 30.61 30.844 40.687 30.828
Nproc=16 RealTime: 40.813 30.984 21.672 41.187
Nproc=17 RealTime: 41.079 40.859 30.969 31.218
Nproc=18 RealTime: 31.235 41.015 41.094 41.328
Nproc=19 RealTime: 41.125 41.141 41.281 41.406
Nproc=20 RealTime: 37.485 41.39 35.922 41.141
```

```
Nproc=21 RealTime: 38.594 41.25 50.89 41.125
Nproc=22 RealTime: 36.391 51.75 38.953 51.516
Nproc=23 RealTime: 51.375 51.531 51.391 39.5
Nproc=24 RealTime: 51.421 33.016 51.375 33.031
```

Although the measured times for each trial somewhat unstable, each load of a trial is evenly shared until `nproc = 8` which is the number of cores, and even for further `nproc`, loads increase smoothly.

ACL also implements multi-threading, but unfortunately does not support multi-core CPUs as the following result shows, which is measured in the same condition.

```
CG-USER(4): (loop for n from 1 to 10 do (parallel-naive-pi n 4))
Nproc=1  RealTime:  9.078  9.062  9.063  9.062
Nproc=2  RealTime: 18.047 18.063 18.046 18.094
Nproc=3  RealTime: 27.078 27.11  27.156 27.187
Nproc=4  RealTime: 36.157 36.14  36.094 36.188
Nproc=5  RealTime: 45.14  45.219 45.25  45.188
Nproc=6  RealTime: 54.203 54.234 54.141 54.156
Nproc=7  RealTime: 63.406 63.172 63.297 63.219
Nproc=8  RealTime: 72.39  72.266 72.313 72.312
Nproc=9  RealTime: 81.313 81.25  81.406 81.266
Nproc=10 RealTime: 90.297 90.344 90.343 90.391
```

Apparently, each load of tasks is not shared by 8 cores. We would like to expect ACL to support multi-core CPUs.