low-level optimization techniques for MCL 2.0

This document[1] lists techniques and suggestions for speeding up MCL 2.0. Most of the information has existed on the MCL bulletin board previously, or is available from Peter Norvig's excellent book *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp,* published by Morgan Kauffman.[2]

There are two parts to the document. Part 1 includes techniques for profiling and measuring code optimization. Part 2 provides Common Lisp techniques, as well as MCL specific suggestions, for optimizing time and/or space.

The document is by/for the general MCL community. Special acknowledgement is due to Bill St. Clair, and to Gary Byers on the MCL team for numerous suggestions and comments. If you have techniques of your own, suggestions, corrections, etc. please, please, please, pass them along for integration into the next posting. Send them to Ashok Khosla, on AppleLink at KHOSLA or on the Internet at KHOSLA@applelink.apple.com, or U.S. mail @ 1424 Harker Ave., Palo Alto, Ca., 94301, USA.

Although Bill St. Clair, and I have reviewed this document several times, errors may still exist. Indentation in MSWord format will differ from MCL indentation. Some forms may have missing parentheses, or other errors. Please report errors to Ashok Khosla, not to Bill St. Clair. Thank you!

MCL 2.0 is a great programming environment. Hopefully, these notes will help users to have fun with MCL, and to produce great applications. Happy programming!

---

1 This is version 1.0 , dated August 11, 1992

2 See especially *Chapter 10 — Low Level Issues.* Source code is available on the MCL 2.0 CD. See MCL 2.0 CD:User Contributed Code:Books:Norvig - "Paradigms of AI":

profiling and measuring tools for optimization

The following functions are useful in profiling and measuring functions

time.....time a form and return consing information
disassemble.....disassemble a function
get-internal-real-time.....returns the amount of time a Macintosh has been on - useful in building your own timing functions

time

time gives you one way of optimizing a function. If you are doing substantial optimization, you may first want to write profiling utilities built on time. See Norvig for sample utilities build on time and get-internal-real-time[3].

? **(time (dotimes (i 1000) (random 54)))**
(dotimes (i 1000) (random 54)) took 107 milliseconds (0.107 seconds) to run. Of that, 12 milliseconds (0.012 seconds) were spent in The Cooperative Multitasking Experience.
48 bytes of memory allocated.

You may wish to enclose the time form in a without-interrupts. Without the without-interrupts, you're timing MCL's event processing code as well as the code in which you're interested (MCL's event processing code is known as "The Cooperative Multitasking Experience"). Since the without-interrupts form disables interrupts, be careful when you use it; you won't able to interrupt or stop a long timing test!
? **(without-interrupts (time (dotimes (i 1000) (random 54))))**
(dotimes (i 1000) (random 54)) took 57 milliseconds (0.057 seconds) to run.
nil

disassemble

For advanced programmers, another useful technique is to look at the actual machine-code that is produced for the function. As an example, let's explore the impact a declare form has by examining the disassembled machine-code.

```
(defun f (a b)
        (+ a b))

(defun g (a b)
 (the fixnum (+ (the fixnum a) (the fixnum b))))
```

This disassembles into: (next page)

[3] The source code for Norvig's book is available on the MCL 2.0 CD.

This disassembles into:

```
? (disassemble 'f)
0 (jsr_subprim $SP-TWO-ARGS-VPUSH)
4 (move.l (vsp 4) d0)
8 (move.l @vsp d1)
10 (spop vsp)
12 (jmp_subprim $SP-ADD2ACC)

? (disassemble 'g)
0 (jsr_subprim $SP-TWO-ARGS-VPUSH)
4 (move.l (vsp 4) d0)
8 (add.l @vsp d0)
10 (spop vsp)
12 (rts)
```

Lisp has represented the diassembled output as a series of machine instructions For the function f the first machine instruction , jsr_subprim, is a jump to a subroutine which pushes the two arguments a and b onto the stack. In f, this is followed by two move instructions which move the arguments into the appropriate registers, a stack pop, and a jump to another subroutine to add the arguments together. Why another subroutine just to add two numbers? The types of the + function's arguments are unknown at compile-time. Consequently, a routine ($SP-ADD2ACC) must be called which does the appropriate type casting, and then calls the appropriate, type-based, machine instruction.

Since g had declare forms, the compiler knows at compile time, that a and b are fixnums. Consequently , the compiled code for g uses the CPU's own integer add instruction, instead of calling the more general, and slower $SP-ADD2ACC add subroutine. All of this is made possible by the declare forms. The disassembly confirms that the declare forms "worked".

Don't be intimidated by the Mc680x0 assembly-code. What's important is looking for subroutine calls vs. inline CPU instructions. For example, it's not that important to know what addressing mode move uses. More information can be obtained in Lap.Doc inside the MCL 2.0 Library folder, or from any good Mc680x0 assembler book (my favorite is *Programming the 68000* by Ed Rosenzweig, Hayden Press)

get-internal-real-time

Finally there is the get-internal-real-time function. You can use this to build your own profiling utilities. For information see Chapter 9.5 of Norvig.

MCL 2.0/Common Lisp optimization techniques

By using some of the more efficient techniques in common lisp you can increase your program up to 40x. The techniques described below include:

- use declarations
- use macros for type-dependant functions
- use floating point macros
- conserve cons'ing
- do explicit memory allocation and deallocation
- use "resources" if needed
- use the right data structure
- use accessor functions rather than caddrs or (second (first ....)))
- use do-list instead of mapc

use declarations

Declarations are a simple method of optimizing. By telling the compiler something about the intended run-time use and type of a symbol, it can optimize memory extent, and compiled code. Common declarations are memory declarations (declare (dynamic-extent...), type declarations (declare (fixnum x)), and optimization declarations (declare (speed 3)), (declaim (inline...))

Here are some examples of usage.

dynamic-extent

"(dynamic-extent *item1 item2 .... itemn*) declares that certain variables or function names refer to data objects whose extents may be regarded as dynamic" (CLtl2). The item's contents are allocated from the stack instead of from the Lisp's free memory pools, thereby saving consing and garbage collection time.

```
(defun average (a b &rest moreNumbers)
 (declare (dynamic-extent moreNumbers))
  (let ((sum 0)
              (count 2))
    (setf sum (+ a b))
    (dolist (aNumber moreNumbers)
       (incf sum aNumber)
             (incf count))
    (/ sum count)))
```

Don't forget to declare (&rest ....) parameters as dynamic-extent, when possible. See CLtL2 for more examples. Also note that declaring sum and count as dynamic-extent is unnecessary, and has no effect. This is because sum and count are bound to fixnums, and fixnums are immediate objects.

Improper use of dynamic-extent forms is unforgiving. MCL 2.0 will likely crash. What's an improper use? The dynamic-extent declaration guarantee says that the variables will not be needed outside the scope of the dynamic-extent declaration. Here's an example of what not to do:

```
;;;What not to do
(defun f (a b)
 (let ((c (list a b)))
   (declare (dynamic-extent c))
   c)) ;;c is returned even though it's dynamic-extent. oh-oh

(defun g (a b)
 (let ((c (f a b)))
   (format t "~%c is: ~S" c)
   c))
;;c is now assigned to something that has disappeared on the ;;stack

? (g 1 2)

c is: (657680 . 67108865)
#<BOGUS object @ #x5047A4>
```

Here is a list of contexts in which a dynamic-extent declaration has an effect:

- &rest args

- Closures created with flet or let

```
(defun my-mapcar (function list)
  ; MAPCAR is inlined. Want the functions below to go out of line.
  ; so that the closures will be (stack) consed.
  (mapcar function list))

(defun add-to-each-list-element (x list)
  (flet ((add-x (y) (+ x y)))
    (declare (dynamic-extent #'add-x))
    (my-mapcar #'add-x list)))

(defun add-to-each-list-element-2 (x list)
  (let ((add-x #'(lambda (y) (+ x y))))
    (declare (dynamic-extent add-x))
    (my-mapcar add-x list)))
```

- let bindings of a variable bound to the result of calling one of the following functions:

```
list
list*
multiple-value-list
make-list
cons
%int-to-ptr
%inc-ptr
%get-ptr
%new-ptr
%make-uvector
```

result of compiler macros for make-string and make-array
vector
ccl::%gvector (Many of the consers defined in "ccl:library;lispequ.lisp" for internal MCL objects expand into a call to ccl::%gvector.
make-string with no :initial-element arg
make-array with a first argument that is known (declared) to be a fixnum, an :element-type that is a compile-time constant, and no other keyword arguments

Currently the results of MAKE-INSTANCE and DEFSTRUCT generated constructor functions cannot be stack consed.

Example 1:
This is a fairly common idiom.

```
(defclass my-window () ())
(defmethod initialize-instance ((self my-window)
                                &rest rest &key window-show)
 (declare (dynamic-extent rest))
    (apply #'call-next-method self :window-show nil rest)
    (add-my-windows-subviews self)
    (if window-show
        (window-show self)))
```

Example 2 (contrived):

```
(defun find-substring (substring string
                       &optional (test 'string=))
  (let* ((string-len (length string))
                                 (substring-len (length substring))
                       (diff (- string-len substring-len))
                       (compare-string (make-string
                                        substring-len)))
          (declare (dynamic-extent compare-string)
                                  (fixnum string-len substring-len diff))
          (dotimes (i diff)
              (replace compare-string string
                                   :start1 0 :end1 substring-len
                                   :start2 i :end2 (the fixnum (+ i substring-len)))
              (when (funcall test substring compare-string)
                  (return i)))))
```

declare (type variable)

Since Lisp is a typeless language, type optimizations are based on a declare "guaranty". As an example, consider:

```
(defun quicker-add (a b)
 (declare (fixnum a b))
 (the fixnum (+ a b)))
```

The (declare (fixnum a b) says that a and b are guaranteed to be "fixnums" (ie. almost 29 bit integers). This type declaration allows the compiler to inline code the cpu add instruction, instead of calling the more general typeless addition procedure.. Note the use of "the fixnum" guarantee of the result also being a fixnum.

speed, safety, space

You can encourage the compiler to optimize by speed and safety macros. Here's one example[4]:

```
(defvar *optimize* nil)
(defmacro optimize (&rest the-code)
 (if (not *optimize*)
   `(locally (declare (optimize (speed 0)
                        (safety 3)
                        (compilation-speed 0)
                        (space 0)))
     ,@the-code)
   `(locally (declare (optimize (speed 3)
                        (safety 0)
                        (compilation-speed 0)
                        (space 0)))
     ,@the-code)))

(optimize
        (defun quicker-add1 (a b)
                (declare (fixnum a b))
                (the fixnum (+ a b)))

;;;This is also commonly written as
(defun quicker-add2 (a b)
        (declare (optimize (speed 3) (safety 0) (space 0)
                        (compilation-speed 0))
        (the fixnum (+ a b))
```

When you wrap a function with (optimize (speed 3) (safety 0)), as in the quicker-add1 function, the compiler does not emit any code to check for the correct number of arguments. If you call such a function with the wrong number of arguments, you will either get unexpected results or crash. The function WILL run faster, so this optimization is sometimes desireable when you know that the function will always be called correctly. The MCL compiler currently ignores the compilation-speed optimization quantity, though you may wish to include it if you are writing portable lisp code.

Finally, you can declare some functions inline.

```
(defmacro macro-func (a b)
 `(+ ,a ,b))

(declaim (inline inline-func))
(defun inline-func (a b)
 (+ a b))

(defun notinline-func (a b)
 (+ a b))
```

[4] Examine the compiler policy functions in the MCL manual as well

```
(defun time-macro-func (&optional (numtimes 1000))
 (dotimes (i numtimes)
   (declare (fixnum i))
   (macro-func 2 3)))

(defun time-inline-func (&optional (numtimes 1000))
 (dotimes (i numtimes)
   (declare (fixnum i))
   (inline-func 2 3)))

(defun time-notinline-func (&optional (numtimes 1000))
 (dotimes (i numtimes)
   (declare (fixnum i))
   (notinline-func 2 3)))
```

?(time (time-macro-func 10000))
(time-macro-func 10000) took 10 milliseconds (0.010 seconds) to run.
?(time (time-inline-func 10000))
(time-inline-func 10000) took 10 milliseconds (0.010 seconds) to run.
?(time (time-notinline-func 10000))
(time-notinline-func 10000) took 42 milliseconds (0.042 seconds) to run.

use optional args instead of keyword args

It may be faster to use optional args instead of keyword args.

```
(defun key-func (a &key (b 3))
 (+ a b))

(defun optional-func (a &optional (b 3))
 (+ a b))

(defun time-key-func (&optional (numtimes 1000))
 (dotimes (i numtimes)
   (declare (fixnum i))
   (key-func 2)))

(defun time-optional-func (&optional (numtimes 1000))
 (dotimes (i numtimes)
   (declare (fixnum i))
   (optional-func 2)))

(defun time-key-func-w-arg (&optional (numtimes 1000))
 (dotimes (i numtimes)
   (declare (fixnum i))
   (key-func 2 :b 3)))

(defun time-optional-func-w-arg (&optional (numtimes 1000))
 (dotimes (i numtimes)
   (declare (fixnum i))
   (optional-func 2 3)))
```

? **(without-interrupts (time (time-key-func 10000)))**
(time-key-func 10000) took 126 milliseconds (0.126 seconds) to run.

? **(without-interrupts (time (time-optional-func 10000)))**
(time-optional-func 10000) took 85 milliseconds (0.085 seconds) to run.

? **(without-interrupts (time (time-key-func-w-arg 10000)))**
(time-key-func-w-arg 10000) took 170 milliseconds (0.170 seconds) to run.

? **(without-interrupts (time (time-optional-func-w-arg 10000)))**
(time-optional-func-w-arg 10000) took 84 milliseconds (0.084 seconds) to run.

use macros for type-dependant functions

From: Michael Travers <mt@media.mit.edu>:

The MCL compiler does pay attention to declarations and will put many operations in-line if it can. I've enclosed some macros for doing fast integer arithmetic that I use (derived from Boxer).

Similar macros can speed up other operations. Car, cdr, and svref will compile in-line. [AREF will compile in-line when referencing a one-dimensional simple array
of a known type with a single index that is known to be a fixnum and the compiler policy says to inhibit safety checking (by default this is done by declaring (optimize (speed 3) (safety 0))) -ed.]

```
(defmacro def-arith-op (int-name reg-name)
    `(defmacro ,int-name (&rest args)
        `(the fixnum (,',reg-name ,@(mapcar #'(lambda (arg) `(the fixnum ,arg)) args)))))

(def-arith-op +& +)
; Only addition-type ops actually get any boost in MCL2.0
(def-arith-op -& -)
(def-arith-op incf& incf)
(def-arith-op decf& decf)
(def-arith-op 1+& 1+)
(def-arith-op 1-& 1-)
```

Using +& will compile an inline integer add instead of a subroutine call.

```
(defun inline-svref (vector index)
    (locally (declare (optimize (speed 3) (safety 0))
                                        (type simple-vector vector)
                                        (fixnum index))
        (aref vector index)))

(defun inline-float-aref (array index)
    ; Will inline the aref, but go out of line to cons a float. (locally (declare (optimize (speed 3) (safety 0))
                                        (type (simple-array double-float (*)) array) (fixnum index))
        (aref array index)))
```

use floating-point macros

On its own, MCL does no optimization for floating point arithmetic and MCL 2.0b1 supports implements only double floats. MCL 2.0 final adds short floats. If you have a procedure that does a great deal of floating point acceleration, you will want to look at the excellent inline floating point acceleration package written by Erann Gat. It is available on the MCL 2.0 CD[5]. Here is an example of a matrix multiply, along with accessor functions:

```
;;;Creates a 3D point as a lisp vector
(defmacro pt3 (x y z)
 `(make-array 3
            :element-type 'float
    :initial-contents (list (coerce ,x 'float)
                                                    (coerce ,y 'float)
                        (coerce ,z 'float))))


;;;Accessors for a point
(defmacro :x (aPoint)
  `(svref ,aPoint 0))

(defmacro :y (aPoint)
  `(svref ,aPoint 1))

(defmacro :y (aPoint)
  `(svref ,aPoint 1))

;;;Makes a 3x3 identity matrix
(defun matrix ()
  (let ((theNewMatrix (make-array 9 :element-type 'float
                                                    :initial-element (the float 0.0))))
    (setf (svref theNewMatrix 0) 1.0)
    (setf (svref theNewMatrix 4) 1.0)
    (setf (svref theNewMatrix 8) 1.0)
    theNewMatrix))

;;;Post multiplies a matrix by a 3D point
(defun pt3*matrix (aPt3d aMatrix &optional
                                        (resultPt (pt3 0.0 0.0 0.0)))
  (declare (type (simple-vector float *) aPt3d aMatrix resultPt))
  (setf (:x resultPt)
            (dot-product3-helper
                        (:x aPt3d) (:y aPt3d) (:z aPt3d)
                        (svref aMatrix 0) (svref aMatrix 3) (svref aMatrix 6)))
  (setf (:y resultPt)
            (dot-product3-helper
                        (:x aPt3d) (:y aPt3d) (:z aPt3d)
                        (svref aMatrix 1) (svref aMatrix 4) (svref aMatrix 7)))
  (setf (:z resultPt)
            (dot-product3-helper
                        (:x aPt3d) (:y aPt3d) (:z aPt3d)
          (svref aMatrix 2) (svref aMatrix 5) (svref aMatrix 8))))

(defun dot-product3-helper (x1 y1 z1 x2 y2 z2)
  (declare (float x1 y1 z1 x2 y2 z2))
  (fpc::fpc-inline (+ (* x1 x2) (* y1 y2) (* z1 z2))))
```

[5] See MCL 2.0 CD:User Contributed Code:Floating Point Compiler:fpc-v1.2a1.lisp.

The dot-product3-helper does all of the floating point work. Erran's fpc::fpc-inline macro inlines floating point coprocessor instructions and eliminates consing of intermediate results. Note that this code will not work with a machine lacking a Mc6888X coprocessor (use gestalt calls to verify that the machine has a floating point coprocessor if you want the code to be portable!).

conserve consing

Since garbage collection is such an "expensive" task, it pays to find ways to reduce consing. Here's the most obvious suggestion. More perceptive suggestions are available in Norvig's book — Chapter 10.4.

nconc, push, nreverse, delete

Good Lisp programmers who know what they're doing use destructive list functions such as nconc and delete, or push and nreverse idiom, instead of their cons-hungry equivalents such as append, reverse, and remove. An example:

```
(defun partition-if (pred list)
 "Return 2 values: elements of list that satisfy pred,
 and elements that don't."
 (let ((yes-list nil)
       (no-list nil))
   (dolist (item list)
     (if (funcall pred item)
         (push item yes-list)
         (push item no-list)))
   (values (nreverse yes-list) (nreverse no-list))))
```

Because these functions actually munge the input-list, don't use them if you are a novice. The truth is that you need to know when to use each. For example, macros very rarely want to do destructive operations during macro expansion (though they may want to generate code that does destructive operations).

do explicit memory allocation/deallocation

Objects of a fixed size and highly dynamic extent are prime candidates for "resources".
A pool of prebuilt resources is implemented using a vector/stack. Your explicitly allocating and freeing resources from the pool, augments the Lisp garbage collector with a more optimal memory allocation scheme.

These macros are from Norvig are useful.

```
;;;; Defresource:
(defmacro defresource (name &key constructor (initial-copies 0)
                            (size (max initial-copies 10)))
 (let ((resource (symbol '* (symbol name '-resource*)))
       (deallocate (symbol 'deallocate- name))
       (allocate (symbol 'allocate- name)))
   `(progn
      (defparameter ,resource (make-array ,size :fill-pointer 0))
      (defun ,allocate ()
        "Get an element from the resource pool, or make one."
        (if (= (fill-pointer ,resource) 0)
```

```
          ,constructor
          (vector-pop ,resource)))
     (defun ,deallocate (,name)
       "Place a no-longer-needed element back in the pool."
       (vector-push-extend ,name ,resource))
     ,(if (> initial-copies 0)
          `(mapc #',deallocate (loop repeat ,initial-copies
                                     collect (,allocate))))
     ',name)))

(defmacro with-resource ((var resource &optional protect) &rest body)
 "Execute body with VAR bound to an instance of RESOURCE."
 (let ((allocate (symbol 'allocate- resource))
       (deallocate (symbol 'deallocate- resource)))
   (if protect
       `(let ((,var nil))
          (unwind-protect (progn (setf ,var (,allocate)) ,@body)
            (unless (null ,var) (,deallocate ,var))))
       `(let ((,var (,allocate)))
          ,@body
          (,deallocate var)))))
```

use the right data structure

When needed, use vectors instead of lists . Vectors require half the space, and have a constant time element access. MCL will in-line array references only for vector's with element types of t, (signed-byte n) and (unsigned-byte n) where n is 8, 16, or 32 (some of these weren't supported in 2.0b1). As an example:

```
(defun rectangle ()
 (make-array 4
             :element-type '(unsigned-byte 16)
             :initial-element 0))

(defmacro rectangle-left (aRectangle)
 `(locally (declare (optimize (speed 3) (safety 0)))
    (the (unsigned-byte 16)
      (aref (the (simple-array (unsigned-byte 16) (*)) ,aRectangle)                    0))))
```

Note that in the array example above, you need to declare the type of the array and (declare (optimize (speed 3) (safety 0))) in order to get in-line array referencing. Declaring the type of the result is not enough. Note also that the size of the array's element matters. Declaring the array to be of type (unsigned-byte 32) also will not result in any needed optimization. rectangle-left will still go out of line because an (unsigned-byte 32) might be a bignum, and the out-of-line code checks for that. Also, declaring it to be an unsigned-byte 32 will also cons more (none for 16-bits, versus consing any bignum array elements for an (unsigned-byte 32)).

Norvig and Waters describe an optimized queue in *Implementing Queues in Lisp* in
Lisp Pointers, Vol IV, Number 4 (available from ACM Press). It is useful to study this article because of its clever avoidance of unnecessary consing. Norvig often does explicit surgery on cdrs of a list. For example – here is a memory efficient equivalent of "excise" written in the Norvig style:

```
(defun excise (alist at-position)
  "In place deletion of an item from a list at a given position (0 based)"
  (if (zerop at-position)
    (cdr alist)
    (let ((headlist alist))
      (declare (dynamic-extent headlist))
      (dotimes (i (1-& at-position))
        (setf headlist (cdr headlist)))
      (setf (cdr headlist)
            (cddr headlist))
      alist)))
```

use accessor functions

When you start optimizing your code you will appreciate the fact that you created "accessor" functions. For example, if you had a list which contained an address
use a function like last-name instead of cdar or second. That way you are free to use defstruct forms later. Defstructs use vectors for access, and the accessors are proclaimed inline. Consequently, defstruct accessors are very fast.

use do-loops instead of mapping functions

Use dolist instead of mapc functions:

```
; Set up a list of 1000 random numbers
(defvar foobar nil)
(dotimes (i 1000)
   (push (random 65535) foobar))

; Defun a test function map-test which maps a
; function over a list
(locally (declare (optimize (speed 3)
                            (safety 0)
                            (compilation-speed 0)
                            (space 0)))
 (defun map-test ()
   (let ((theSum 0))
     (declare (fixnum theSum))
     (mapc #'(lambda (x)
               (incf& theSum x))
           foobar)
     theSum)))

; Defun a test function map-test which
; uses a do loop over a list
(locally (declare (optimize (speed 3)
                            (safety 0)
                            (compilation-speed 0)
                            (space 0)))
 (defun dolist-test ()
   (let ((theSum 0))
     (declare (fixnum theSum))
     (dolist (anElement foobar)
       (incf& theSum anElement))
     theSum)))
```

```
#|
? (time (map-test))
(map-test) took 51 milliseconds (0.051 seconds) to run.
Of that, 15 milliseconds (0.015 seconds) were spent in The Cooperative Multitasking Experience.
32441687
? (time (dolist-test))
(dolist-test) took 3 milliseconds (0.003 seconds) to run.
32441687
?
|#
;;Do loops are much faster...
```