To NReverse When Consing a List or By Pointer Manipulation, To Avoid It; That is the Question

Richard C. Waters

Mitsubishi Electric Research Laboratories
201 Broadway; Cambridge MA 02139
Dick@MERL.COM

A situation that arises all the time in Lisp is the need to create a list of elements where the order of the elements in the list is the same as the order that they are created in time—i.e., the first element computed is the first element in the list, the second element computed is the second element in the list, etc. There are two basic ways of doing this: the `nreverse` approach and the `rplacd` approach. In the `nreverse` approach, you push the elements onto the list as they are computed and then use `nreverse` to put the list into the correct order after all of the elements have been computed. In the `rplacd` approach, you maintain a pointer to the end of the list and use `rplacd` to put each element directly into its proper place in the list.

Which of the two approaches to creating a list is better?

Over the two decades that I have been writing Lisp programs, I have overheard (and participated in) quite a number of arguments about this question. Some people argue vehemently that the `rplacd` approach is **obviously** much faster and therefore better. Others argue just as vehemently that the `nreverse` approach is actually faster and given its greater simplicity is therefore **obviously** better. However, I have seen very little in the way of hard facts.

As discussed in detail below, the facts suggest that neither approach is **obviously** faster. It is just as easy to imagine Lisp implementations where one approach is faster as implementations where the other is faster. It is easiest of all to imagine implementations where the two approaches run at more or less the same speed.

Experimentation suggests that the `nreverse` approach is actually faster in many if not most Lisp implementations. However, more importantly, it supports the idea that the speed difference is not enough to be important. Therefore, given that the `nreverse` approach is easier to write and understand, I recommend using `nreverse` when creating lists.

A Specific Example

As a precise foundation for comparing the two approaches, it is best to look at a specific example. Consider implementing a simplified version of the standard Common Lisp function `maplist` that takes only one list argument—i.e., it enumerates each sublist in a list, calls a function on each sublist, and creates a list of the results. This example is convenient because it contains very little computation other than the creation of the output list.

The program `maplist-nreverse` shows how a one-list-argument `maplist` can be implemented using the `nreverse` approach to creating the output list.

```
(defun maplist-nreverse (f list)
  (do ((sub list (cdr sub))
       (r nil (cons (funcall f sub) r)))
      ((null sub) (nreverse r))))
```

The program `maplist-rplacd` shows how a one-list-argument `maplist` can be implemented using the `rplacd` approach to creating the output list. The code is less clear and less concise,

but avoids calling **nreverse**.

```
(defun maplist-rplacd (f list)
  (let ((r (cons nil nil)))
    (do ((sub list (cdr sub))
         (end r (let ((x (cons
                           (funcall f sub)
                           nil)))
                  (rplacd end x)
                  x)))
        ((null sub) (cdr r)))))
```

The code starts by creating a dummy cons cell **r** that is later discarded. This makes the main loop simpler and faster, because it avoids the need for code that handles the first output element specially. The savings is greater than the cost of the extra cons unless the input list is extremely short. (In the Lisp I use, the break even point is at an input length of five.)

In order to compare **maplist-nreverse** with **maplist-rplacd** one must look in detail at the function **nreverse**, since this should properly be considered as part of **maplist-nreverse**. As shown below, **nreverse** is a very simple function. It merely runs down a list applying **cdr** and **rplacd** once to each cons cell.

```
(defun nreverse (list)
  (prog ((prev nil) next)
        (when (null list) (return nil))
     lp (setq next (cdr list))
        (rplacd list prev)
        (when (not next) (return list))
        (setq prev list)
        (setq list next)
        (go lp)))
```

The key observation to make is that the code for **maplist-rplacd** is very much the same as the code for **maplist-nreverse** plus the code for **nreverse**. In particular, each approach calls **cons** to create the cells of the output list, and uses **rplacd** to place the cells in the correct order. The only difference is that taken together **maplist-nreverse** and **nreverse** traverse the output list twice as opposed to once for **maplist-rplacd**. This is a real difference, but not a large enough difference to be important.

Counting Instructions

To sharpen the comparison of the functions above, it is interesting to consider the best possible ways that the functions can be implemented using low level machine instructions. Since I do my work on an HP-9000 series machine and am more familiar with it than with other current machines, I will use HP's PA-RISC architecture [1] as the basis for the examples below.

Maplist_nreverse approximates the best PA-RISC implementation of **maplist-nreverse**. It is approximate because it makes many assumptions about the associated Lisp implementation.[1] In particular, it assumes that the implementation is using the standard PA-RISC calling conventions and that cons cells are implemented as a 4-byte car pointer followed by a 4-byte cdr, with **nil** implemented as **0**.

```
maplist_nreverse
.CALLINFO CALLER,SAVE_RP,ENTER_GR=5
r   .reg  %r3
sub .reg  %r4
f   .reg  %r5
    .ENTER
    LDI         0,r           ;(setq r nil)
    MOVB,=,n    %arg1,sub,DN
    MOVB        %arg0,f
 LP MOVB        sub,%arg0
    BLR         f,%rp
    MOVB        %ret0,%arg0 ;1st cons arg
    MOVB        r,%arg1     ;2nd cons arg
    BL          cons,%rp    ;cons
    MOVB        %ret0,r     ;(setq r ...)
    LDW         4(sub),sub
    COMIB,<>,n 0,sub,LP
 DN MOVB        r,%arg0     ;1st arg
    BL          nreverse,%rp;nreverse
    .LEAVE
```

As long as quantities are stored in registers, operations like **car**, **cdr**, **rplacd**, and **setq** can be implemented as single PA-RISC instructions. As a result, **maplist_nreverse** is very compact. The parts of the code that concern us are the seven instructions that create the output list. The correspondence between these instructions and parts of **maplist-nreverse** is indicated by comments.

The list **r** being constructed is stored in a

[1] The machine code shown is also approximate because it was not practical to test it. As a result, there might be minor errors; however, this should not effect the basic comparisons being made.

register. A load immediate instruction (LDI) is used to initialize r to nil. Two move instructions (MOVB) are used to set up the arguments of cons. A branch and link instruction (BL) is used to call the cons subroutine. A move is used to store the result returned by cons in r. The last two instructions call nreverse, whose result is returned as the result of maplist_nreverse.

Maplist_rplacd approximates the best PA-RISC implementation of maplist-rplacd. As above, the relationship between the instructions that create the output list and the code in maplist-rplacd is indicated by comments.

```
maplist_rplacd
.CALLINFO FRAME=4,CALLER,SAVE_RP,ENTER_GR=5
end .reg  %r3
sub .reg  %r4
f   .reg  %r5
    .ENTER
    STW         0,-52(%sp)  ;set cdr r nil
    ADDI        -56,%sp,end ;(setq end r)
    MOVB,=,n    %arg1,sub,DN
    MOVB        %arg0,f
 LP MOVB        sub,%arg0
    BLR         f,%rp
    MOVB        %ret0,%arg0 ;1st cons arg
    LDI         0,%arg1     ;2nd cons arg
    BL          cons,%rp    ;cons
    STW         %ret0,4(end);(rplacd end x)
    MOVB        %ret0,end   ;(setq end x)
    LDW         4(sub),sub
    COMIB,<>,n 0,sub,LP
 DN LDW         -52(%sp),%ret0;return cdr r
    .LEAVE
```

To save on overhead, the dummy header cons cell is simulated on the PA-RISC stack instead of calling cons. The first store instruction (STW) initializes the cdr of this cons cell to nil by storing 0 in the appropriate stack frame slot. The add immediate instruction (ADDI) initializes end to point to four bytes in front of this cdr. The car part of the cons cell never actually has to exist since it is never referred to. A store instruction is used to implement the required rplacd.

Comparing the loops in maplist_rplacd with maplist_nreverse shows that the cost of eliminating the call on nreverse is only one instruction execution per cons cell in the output list. This clearly opens the door to a savings in run time. However, it turns out that nreverse is so cheap to compute that the door is not opened very far.

To see how inexpensive nreverse is, it is useful to look at the modified implementation shown in nreverse-unrolled. By unrolling the loop so that three consecutive cons cells are handled on each cycle of the loop, one can eliminate the pointer shuffling that is required in the implementation shown above.[2] This reduces the number of basic operations per cons cell from five to three.

```
(defun nreverse-unrolled (list)
  (prog ((prev nil) next)
        (when (null list) (return nil))
     lp (setq next (cdr list))
        (rplacd list prev)
        (when (not next) (return list))
        (setq prev (cdr next))
        (rplacd next list)
        (when (not prev) (return next))
        (setq list (cdr prev))
        (rplacd prev next)
        (when (not list) (return prev))
        (go lp)))
```

Nreverse_unrolled approximates the best PA-RISC implementation of nreverse-unrolled and therefore nreverse. Cdr and rplacd are implemented with load and store instructions, as above. The tests for the end of the list are implemented using compare immediate and branch instructions (COMIB). In the loop, only three instructions per cons cell are require to reverse the input list.

Just as in maplist_rplacd, one instruction per cons cell is all that is needed to store the correct cdr pointers. The only overhead in comparison with maplist_rplacd is that nreverse_unrolled has to traverse the list a second time. This requires two instructions per cons cell, a null test and a cdr.

One way to summarize the results in this

[2]The nreverse-unrolled approach to implementating nreverse has been in use in various Lisp implementations since at least as long ago as the mid-1970s. At that time, JonL White used it in the PDP10 MacLisp implementation at MIT. He first discovered the trick by hand optimizing PDP10 Machine Language code, and only later noticing that Lisp variables could take the place of register names allowing for a higher level expression of the concept.

```
nreverse_unrolled
prev .reg %ret1
list .reg %ret0
next .reg %arg0
     .CALLINFO
     .ENTER
     LDI        0,prev
     MOVB,=,n   %arg0,list,DN2
 LP  LDW        4(list),next   ; cdr
     STW        prev,4(list)   ; rplacd
     COMIB,=,n  0,next,DN2     ; when
     LDW        4(next),prev   ; cdr
     STW        list,4(next)   ; rplacd
     COMIB,=,n  0,prev,DN1     ; when
     LDW        4(prev),list   ; cdr
     STW        next,4(prev)   ; rplacd
     COMIB,<>,n 0,list,LP      ; when
     MOVB,TR,n  prev,%ret0,DN2
 DN1 MOVB       next,%ret0
 DN2 .LEAVE
```

section is to say that the **rplacd** approach to creating a list has a clear theoretical advantage of two instructions per cons cell over the **nreverse** approach.

However, a better way to summarize the results is to consider the percentage improvement. Considering only the computation required to create the output list, the **nreverse** approach uses at least ten instructions to create each cons cell in the output (three in **nreverse_unrolled** and four in **maplist_nreverse** plus several instructions per cons cell for the call on **cons** even if it is coded in line.) The two instructions saved by the **rplacd** approach are at most only 20%.

It should be noted that the results presented above are not overly distorted by the fact that we have looked at only one specific hardware architecture. The PA-RISC architecture is at an intermediate level of complexity. There are RISC machines with much simpler instruction sets. There are non-RISC machines with much more complex instruction sets.

Switching to a simpler architecture would increase the number of instructions in the examples above. However, the programs are so similar that it is hard to imagine that the relative lengths of the critical loops would change much. The same can be said about switching to a more complex architecture.

Hand Tailored Code

It must be kept in mind that the speed advantage of the **rplacd** approach presented in the last section is only theoretical, because the hyper-efficient code shown is the result of careful hand coding, rather than being the output of a Lisp compiler. It is unlikely that any compiler will produce code that is anywhere near as efficient.

To start with, the typical compiler is likely to implement operations like **rplacd** as subroutine calls rather than inline instructions. In addition, it may store some intermediate values on the stack rather than in registers. Together, these and other factors are liable to lead to compiled code that is several times larger than the idealized code above.

The deficiencies of compilers are unfortunate in many ways, but in the main, they are not relevant to the current discussion. There is no reason to believe that the compiler will work better for any one function than for the others. Therefore, the quality of the compiler should not effect the comparisons being made here, except for one important thing.

Since **nreverse** is a built-in function, implementors may choose to write it using special implementation-specific subprimitives and/or to hand compile it. Either way, this could tilt the performance balance in favor of the **nreverse** approach, because the hand tailored code in **nreverse** could perform a good deal better than the equivalent user code required by the **rplacd** approach. Given that the typical compiler produces relatively voluminous code, this difference can be quite significant.

Memory Performance Effects

A potential difference between the **nreverse** and **rplacd** approaches is that memory is referenced in a different way—in two passes rather than in one. There are at least two negative effects that this might have.

First, there could be a negative interaction with garbage collection. If reference counters are used, then changing them at a later time can be more complex than setting them to a

correct value in the first place. Somewhat similarly, if an ephemeral garbage collection scheme is being used, then garbage collecting cons cells that have experienced an apparently arbitrary `rplacd` can be more complex than garbage collecting cons cells that have been created and used in a more controlled way.

Second, a two-pass approach might lead to poorer cache performance. Recently, processor speed has increased much faster than main memory speed. This has progressed to the point where the instruction cycle time is 1/10 of the memory cycle time or even less. This mismatch is overcome by using fast cache memory between the processor and the main memory. However, to work well, this requires good memory locality in order to minimize cache misses.

This could tilt the performance balance in favor of the `rplacd` approach, because that approach processes the cons cells created in a very local way. In contrast, the traversal of the output list initiated by `nreverse` does not begin until after the entire list has been created. If the output list is long enough, some of the cons cells in it may have fallen out of the cache before they are revisited by `nreverse`. If this happens, the main loop of `nreverse` could slow down by the equivalent of 10 additional instructions or more for each of these cons cells.

However, it is unlikely that this would be a significant difference for two reasons. First, since the `nreverse` visits the most recently created cons cells first, the initial cells it visits must be in the cache. Second, given that the typical compiler produces relatively voluminous code, 10 instructions is not liable to be a significant percentage difference.

Some Experiments

To assess the relative practical significance of the arguments above, experiments were performed in three very different environments: Lucid Common Lisp on an HP-730 machine, Allegro Common Lisp on an old (slow) Apple Macintosh, and MIT Scheme on an HP-715 machine.

Lucid. In Lucid Common Lisp on an HP-730 machine, the functions `maplist-nreverse` and `maplist-rplacd` compile into 54 and 65 instructions respectively—4 to 5 times the size of the idealized code. Using the compiled code, I determined the average time required per cons cell for various size input lists. I used `#'identity` as the map function to maximize the percentage of time spent actually creating the output list. The results of these experiments are shown in the table below.

	input list length				
	10	10^2	10^3	10^4	10^5
`nreverse`	3.1	2.6	2.6	2.9	6.2
`rplacd`	3.3	2.8	2.8	3.1	6.4

The numbers in the body of the table are the computation time in terms of microseconds per cons created. The data suggests that some difficulty arises when processing long lists, but does not reveal any relative penalty for the `nreverse` approach in comparison with `rlpacd`.

Interestingly, the `nreverse` approach is consistently faster than the `rplacd` approach—some 7% faster. It appears that this is due to hand-coding of `nreverse`.

Comparing the speeds of the system implementation of `nreverse` and the result of compiling `nreverse-unrolled` revealed that the user compiled version is 66% slower (1.0 microseconds per cons cell versus .6 microseconds per cons cell). This suggests that something was done to improve the machine code for `nreverse` in comparison with what a user can easily get the compiler to generate.[3] The hand-coding benefit obtained (.4 microseconds per cons cell) is easily large enough to account for the fact that the `nreverse` approach is faster than the `rplacd` approach, and to suggest that without the hand-coding benefit, the `nreverse` approach would be slower.

Allegro. In Allegro Common Lisp on a Macintosh, the functions `maplist-nreverse` and `maplist-rplacd` compile into 29 and 42 instructions respectively. This reflects the fact that

[3] According to JonL White, Lucid uses the `nreverse-unroll` technique to implement `nreverse`. It appears that it uses significant hand optimization in addition.

the Macintosh is not a RISC machine. Since I know very little about the machine instructions the Macintosh uses, I cannot comment on how close this is to the best that is possible.

Timing experiments identical to the ones above revealed the following.

	input list length				
	10	10^2	10^3	10^4	10^5
`nreverse`	95	75	75	75	73
`rplacd`	110	92	91	90	90

The data does not suggest any cache-miss penalty for either approach on long lists.

As above, the `nreverse` approach is consistently faster than the `rplacd` approach—some 17% faster. It appears that this is due to a major hand-coding effect for `nreverse`.

Comparing the speeds of the system implementation of `nreverse` and the result of compiling `nreverse-unrolled` reveals that the user compiled version is 208% slower (40 microseconds per cons cell versus 13 microseconds per cons cell). This suggests that `nreverse` has been very carefully hand coded. As above, the hand-coding benefit obtained (27 microseconds per cons cell) is easily large enough to account for the fact that the `nreverse` approach is faster than the `rplacd` approach, and to suggest that without the hand-coding benefit, the `rplacd` approach would be faster.

MIT-Scheme. Scheme differs significantly from Common Lisp. However, from the perspective of the comparisons being made in this paper, these differences are not important.[4] In MIT-Scheme on an HP-715 machine, timing experiments identical to the ones above revealed the following.[5]

	input list length				
	10	10^2	10^3	10^4	10^5
`nreverse`	2.0	2.0	2.1	2.2	2.2
`rplacd`	1.6	1.6	1.7	1.7	1.7

[4]But, note that in Scheme `nreverse` is called `reverse!`

[5]I am indebted to Franklyn Turbak for performing the Scheme experiments reported here.

The data does not suggest any cache-miss penalty for either approach on long lists.

In contrast to the results shown above, the `rplacd` approach is consistently faster—some 20% faster. It appears that this is due in part to the complete lack of any hand-coding effect for `nreverse`.

Comparing the speeds of the system implementation of `nreverse` and the result of compiling `nreverse-unrolled` reveals that the user compiled version is 24% faster (.47 microseconds per cons cell versus .62 microseconds per cons cell). This reflects the fact that in MIT-Scheme, `nreverse` is written as a simple user function and does not use any loop unrolling. The penalty caused by the lack of loop unrolling (.15 microseconds per cons cell) accounts for 1/3 of the difference between the `nreverse` and `rplacd` approaches. If a factor of 2 further gain in speed could be obtained by hand coding `nreverse`, the gap would disappear altogether.

Summary. The experiments suggest that of the three sources of speed difference between the two approaches (a theoretical advantage for `rplacd`, a hand-coding advantage for `nreverse`, and a cache performance advantage for `rplacd`) the hand-coding advantage usually wins out and therefore the `nreverse` approach is usually fastest.

However, more than this, it is clear that the speed difference between the two approaches is probably never very large. Therefore, if the computation being performed to compute the elements being consed together involves much more than just computing `identity`, the difference recedes into complete insignificance.

If you are interested in such things, you might run an experiment in your Lisp to see if any significant speed difference can be found. However, in the absence of clear evidence for such a difference, I recommend assuming that there is none.

Conclusion

The `rplacd` approach to creating an output list has a theoretical speed advantage, but as a practical matter this appears to be overwhelmed by the fact that `nreverse` is a system

function that can be hand coded by the system implementors. As a result, the **nreverse** approach is probably fastest in most Lisp implementations. Even if the **rplacd** approach is faster in a given Lisp, it is unlikely to be much faster. Therefore, since the **nreverse** approach is simpler and clearer, it is the best approach to use in almost every situation.

The only situation where I would consider using the **rplacd** approach is if I were a Lisp system implementor and had the opportunity to write a system function where I could hand tune machine code for creating a list. In this situation, the **rplacd** approach should be able to achieve its theoretical advantages and I would consider implementing a hand tuned version of the **rplacd** approach. However, it should be realized that there would be much more to be gained through the hand tuning than through the choice of which approach to tune.

In closing, I would like to note that the very best thing to do is to avoid writing code that conses lists. Whenever possible, you should use standard parts of Common Lisp that do the consing for you. In particular, you should use functions like **replace**, **map**, **reduce**, **remove**, **union**, etc. whenever they are appropriate. Beyond this, you should take advantage of looping macro packages such as **loop** and Series.

For example, using the extended features of **loop** that are available in the proposed standard for Common Lisp [2], a simple version of **maplist** could be written as follows.

```
(defun maplist-loop (f list)
  (loop for sub on list
        collect (funcall f sub)))
```

Alternatively, the Series macro package [3, 4] could be used as shown below.

```
(defun maplist-series (f list)
  (collect
    (map-fn t f (scan-sublists list))))
```

Either way, the resulting code is clearer, more compact, and no slower than anything else you can write.

References

[1] Hewlett-Packard, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, Hewlett-Packard, Cupertino CA, 1986.

[2] White, J.L., "Loop", in *Common Lisp: the Language*, 2nd Edition, 709–747, Steele G.L.Jr., Digital Press, Maynard MA, 1990.

[3] Waters R.C., "Automatic Transformation of Series Expressions into Loops", *ACM Transactions on Programming Languages and Systems*, 13(1):52–98, January 1991.

[4] Waters R.C., "Series", in *Common Lisp: the Language*, 2nd Edition, 923–955, Steele G.L.Jr., Digital Press, Maynard MA, 1990.