

# Memory Cache and Lisp

## Faster list processing via automatically rearranging memory

Richard Fateman  
Computer Science  
University of California  
Berkeley, CA 94720-1776, USA

July 22, 2002

### Abstract

It is well-known that modern computers' performance can be improved by organizing computations so that memory access patterns correspond more closely to the memory cache-loading patterns implemented in the hardware. Rearranging code and data are each possible, but if this is done by hand modification of source code, such improvements may also introduce program errors. In some technical circles it is believed that the now conventional copying garbage collectors will advantageously compress memory, as a matter of normal processing. Garbage collection (GC) routines tend to be very well-tested and quite robust, and so might reliably improve locality and therefore speed up programs. In fact, our tests show that they do. Using new hardware machine registers provided in new CPU chips to count cache misses, we see that GC reduces cache misses and speeds up computation.

## 1 Introduction

The Lisp programming language uses linked lists of `cons` cells (two pointers each) as a fundamental data constructor, and it would seem that with current architectures using these elements would be a poor choice in achieving highest computing performance. Cache memories thrive on sequential memory accesses which *appear* to be the antithesis of the “typical” model of Lisp: following pointers in linked lists from one random location to another in working memory. Major efforts in hardware implementations including

out-of-order or speculative execution, multiple execution units, etc. tend to be ineffective with a computation model where nearly each step requires another pointer-following memory access. In the worst case cache memory is defeated by the sequence of pointers.

In fact it is quite possible to allocate cells in a linked list in sequential locations in memory. Each cell then contains a pointer-sized datum that merely points to the next cell. If a cell contains only 2 pointers, as is the case for a Lisp cons cell, half of memory is consumed by “next” pointers<sup>1</sup>, and so such a program takes an immediate hit of 50% of the data cache.

While this seems to waste half of cache, remember that in the past when memory was the scarce resource, people persisted in using linked lists. There must be a payoff in terms of program convenience.

Oddly enough there may also be a memory locality payoff. In particular we have seen a pleasant result in lisp: that reorganization may *require no specific effort by the programmer or any change to the source code*. It is done by the storage reclamation activity of a copying garbage collector (GC), and as the GC runs normally without being explicitly called, this improvement probably requires no human effort. The effect of the garbage collector is to follow the links of live data structures and rearrange them in order in adjacent space. If the program traversing this data uses a similar path, the cache behavior can be quite improved compared to random allocation. In a linear list, there is not much question of the order of traversal.

There are other earlier papers on re-organization of memory; a good survey is contained in the paper by Mowry and Luk on memory forwarding [1]. (This paper advocates a hardware solution to the problem of locality: compress storage and as a safety measure plant a forwarding pointer (marked by an extra hardware bit) in the old location, in case there are any left-over references. This technique is necessary in undisciplined languages in which there is no guarantee that one can identify all pointers to a cell. Lisp does not have this problem.

## 2 A simple test

We constructed a simple test to determine if we were in fact benefiting from garbage collection improving memory locality.

Each of the tests below should be executing essentially the same number of data access instructions, namely enough to follow  $10^9$  pointers in succes-

---

<sup>1</sup>Special purpose Lisp architectures have been designed with various tricks to overcome this apparent space waste by using short pointers, but we will not pursue this here.

sion. In order to vary the locality we vary the number of different pointers. That is, this could be done by following the links in a list of length  $10^5$  repeatedly  $10^4$  times, or any other combination of repetitions and lengths.

Let us start with a Lisp list R consisting of 32,000 random single-word integers.

There are three computations of interest, and since the data seems to vary substantially from run to run, in some cases we repeated a peculiar result to see if it was somehow influenced by external activities on the system being timed.

Each of the three versions does the same computation: compute the length of R  $10^9/32000=31,240$  times.

In version a, 99.99% of pointers are to the next cell, as they are just allocated.

In version b, the cells are randomly ordered (99% of pointers are to addresses over  $2^{15}$  bytes away).

For version c, we run the garbage collector which returns the memory to an arrangement in which 99.99% of the pointers are to the next cell.

Most of our tests are on a Pentium III with a total L1 cache size of 32Kbytes. This is divided into 16Kbytes each for instructions and data, and a 256Kbyte L2 cache, 8-way set associative. Additional tests were performed on a Sun Microsystems SunBlade 100.

Occasionally when a recorded benchmark time seemed to be implausible, (perhaps caused by other activity on the computer system), or we re-tested the data point for other reasons and found a significant variation, we indicate the repeated time as well. The results for the 2,000 word test differ in nature from most of tests, but since the 2,000 cons cells just exactly fits the 16Kbyte data cache, we suspect we are seeing the effect of minor alignment differences, which in this case can make the “random” distribution of pointers better than the systematic and perhaps “counter-cache” access pattern. All cpu times are in seconds.

For 32,000 word-long lists traversed 31,250 times

	L1 misses	L2 misses	cpu time
a	2.54e8	1.22e8	18.7
b	9.57e8	1.15e8	22.7
c	2.25e8	0.72e8	13.0
c repeat	2.52e8	0.48e8	10.1

We show one larger, and then several smaller tests.  
 For 100,000 word-long lists traversed 10,000 times

a	2.56e8	2.52e8	34.0
b	10.23e8	7.55e8	112.0
c	2.56e8	2.52e8	34.0

for 2,000 word-long lists traversed 500,000 times

a	6.64e6	2.31e5	5.9
b	11.02e6	2.17e5	4.8 ?
b repeat	8.84e6	2.36e5	6.46
c	6.02e6	2.35e5	6.3 ?

for 1,000 word-long lists traversed 1,000,000 times

a	1.10e6	2.23e5	3.6
b	1.64e6	2.67e5	5.4
c	2.03e6	2.15e5	4.8

For 500 words or below, no consistent differences are apparent since our artificially generated data is likely to be all cache-resident, even if jumbled. An actual computation in which 500 “good” data points were slowly accumulated over many conuses would probably show better performance after GC.

Note that,  $10^6$  L1 cache misses is about 0.1% of the data accesses.

For the Sunblade-100, a 500Mhz system with a 256kb L2 cache, we have the following data which illustrates that there is a sharp break: once a list requires 32,000 or more 8-byte cells, the cache misbehaves if the list is not in order. We found one anomaly in the 2000 word list, where the sorted version takes twice as long; this is presumably due to an interaction of the cache association mechanism and memory locations. The problem is overcome with a slightly different size, as shown with the 2003 or 2011 word list.

for 100 word-long lists traversed 10,000,000 times

	time
a	6.48
b	6.47
c	6.50

for 1000 word-long lists traversed 500,000 times  
time  
a 6.06  
b 6.07  
c 6.07

for 2000 word-long lists traversed 500,000 times  
time  
a 6.03  
b 6.04  
c 15.90 ??

for 2003 word-long lists traversed 499,251 times or  
for 2011 word-long lists traversed 497,265 times  
time  
a 6.05  
b 6.05  
c 6.05

for 5,000 word-long lists traversed 200,000 times  
time  
a 16.1  
b 19.6  
c 16.0

for 10,000 word-long lists traversed 100,000 times  
time  
a 16.1  
b 18.8  
c 15.9

For 30,000 word-long lists traversed 33,333 times  
time  
a 16.6  
b 20.8  
c 17.7

For 32,000 word-long lists traversed 31,250 times

	time
a	18.4
b	31.3
c	18.4

for 50,000 word-long lists traversed 20,000 times

	time
a	28.6
b	101.2
c	28.7

Further notes on the tests:

The Lisp system we were using (Allegro Common Lisp 6.1) has a generation-scavenging copying/compacting garbage collector that has evolved over years. Its basic operation of copying data into new space in the first GC after being allocated is similar to most contemporary Common Lisp implementations. When a list  $R$  is allocated space from the Lisp free list it is probably composed of `cons` cells in sequential memory locations.  $R$  may not be *entirely* contiguous, but there should be only a few breaks in memory sequence. A typical measurement shows that for 99.9% or more of a newly constructed list, each pointer refers to an adjacent cell in memory.

For most tests we use a Pentium-III 933MHz workstation for these tests, although results should not differ qualitatively from other speed, cache sizes, or other popular cache organizations<sup>2</sup>. Thanks to the PAPI project<sup>3</sup>, these benchmarks were relatively pain-free.

The second timed (b) test of each sequence required randomizing the links. This was done by sorting the list “in place” by changing pointer values. (`setf R (mergesort R)`). Since the list was initially allocated as a list of pseudo-random numbers, this sort randomized the sequence of *memory locations* when the list cells were sorted by value. (We had to fuss to make sure that no garbage collection was triggered during sorting since, as will be seen in a moment, that would spoil the randomization. Consequently we wrote our own in-place list sorting routine rather than using the built-in one).

---

<sup>2</sup>We were also able to measure the L1 Instruction/ Data cache miss data, although not on the same exact run, since we can use only two hardware counters at the same time on our Pentium III.

<sup>3</sup><http://icl.cs.utk.edu/projects/papi/>

Before executing the third test (c), we forced a garbage collection. This had the effect of copying the list  $R$  so that it not only had sequential values but sequential memory locations. In this third test, the speed of scanning  $R$  is similar to the that scanning the length of the original list. As shown above, for size 100,000, it becomes over three times faster than scanning the scrambled list.

### 3 Why is the random list slower?

In a Pentium III each 32-byte cache line can hold 4 `cons` cells. If they are stored in memory that is *sequentially* accessed, starting at the beginning of a loaded cache line we will get 3 cache hits then a miss. If a sufficiently large non-repeating set of them is located in memory *randomly* accessed, essentially each reference will give us a cache miss. Under totally random pointer assumptions we should expect about 3 times more misses.

The Lisp copying/generation scavenging garbage collector has effectively reorganized the memory to regenerate locality in this test. (We hesitate to call this “automatic optimization” although it is clearly improved and certainly automatic.)

### 4 Can lists be better than arrays?

Some applications are more conveniently programmed with lists since one can insert or delete elements at a given location in  $O(1)$  time, rather than having to re-copy elements. On the downside, a conventional linked list does take storage for the links. Furthermore the typical management of such storage is more tedious: compare a whole array that could be freed in one step while linked list cells are individually managed.

Lists are not inevitably associated with garbage collection storage systems, but it has become more plausible for technical reasons, given the large amounts of memory available at low cost, as well as fashion: it is used in the popular language Java.

The obvious comparison data structure, with the nice property that it takes less space (no pointers), is the array. By contrast with the list, the advantage of the array is one can index into the middle of it in  $O(1)$  time. To see how this might compare on tests on the same computer, we examined comparable minimal processing of information in an array sequential storage structure. We allocated 16,000 single-word numbers in an array. This array could have the same contents as a 32,000 word list in memory, since the array

does not need space for pointers. But they both are within the 256Kbyte cache.

Traversing the list and the array the same number of times gives us this data:

Time (sec)	L1 misses	L2 misses	object traversed
4.54	2.51e8	6.07e5	list of 16,000 organized
5.48	1.26e8	1.90e5	array of 16,000
7.27	9.06e8	3.33e5	list of 16,000 randomized

Note that the organized list traversal time is shorter.

If we double the size of the array and list, we hit a cache limit with the list, but not with the array

5.23	1.26e8	2.26e5	array of 32,000
18.85	2.52e8	1227.14e5	list of 32,000 organized
27.37	2.54e8	1954.60e5	list of 32,000 randomized

The cache behavior of the array is not affected by sorting, but at least at the sub-L2 cache size, and comparing Lisp's compiled array accesses to Lisp's following links, the array storage – even given fewer cache misses – is still slightly slower. The advantage of the array storage is that it goes about twice as far before hitting the cache-miss wall.

We can push the array past that wall as well. A solution to this may be to proceed to buy the next CPU in the sequence, with twice the L2 cache size!

17.58	1.27e8	1.26e8	array of 128,000
34.05	2.54e8	2.53e8	list of 128,000 organized
120.65	10.06e8	8.18e8	list of 128,000 randomized

## 5 Conclusion

Here are two different ways to make better use of cache memories.

(a) The traditional one: Find a compact representation that works well with appropriate algorithms to provide rapid access and modification, using sequential access. This suggests a design that doesn't "waste space" for pointers etc. (like a packed array).

(b) The one suggested here: Leave data in lists and perform a copying garbage collection. In an ideal case the layout of the data in the copied list



as traversed by the garbage collector (following CDRs before CARs) will mirror the data access pattern of the associated program, and there may be substantial automatic improvement in locality.

The second of these possibilities, although probably obvious to persons writing garbage collectors, is not so well-known generally, although occasionally explored in the literature [1]. In this admittedly simple benchmark, it can speed the computation substantially.

In order to break away from memory band-width limits on the transfer between the caches and memory it seems that the programmer as well as the designer of the generational garbage collection system should consider the size of the caches, (for the Pentium III or the SPARC Sunblade-100, unless one is dealing with very small sets of data, the most relevant one would be the L2 256Kbyte cache), and in general look at ways of re-using already allocated memory. Detailed suggestions can be found in Wilson's 1992 paper [2].

The predicted continuing divergence between the speed of main memory and cache speed makes it even more important to try to make best use of cache. Fortunately we now have new tools such as PAPI and the hardware counters that make it possible to test programs in real-time rather than through painful simulation. Tests like these can be done to help compare data structures, and perhaps make critical choices, even as late as run-time, to improve throughput.

These tests do not directly bear on other data structures in Lisp, nor even on trees or graphs constructed of `cons` cells. Such linked structures, which may in any case be traversed in any number of ways, may not benefit by the GC restructuring. The trick here may be to try to deal with with a sufficiently small working set of not-necessarily contiguous list structure. In our case with lists of 500 or fewer elements, cache performance may not be the limiting factor in speed: we see essentially no difference once all the data is in L1 cache.

Finally, in any generalizations about program performance on today's complicated processors, and projecting to future processors, "your mileage may vary."

## 6 Acknowledgments

This research was supported in part by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley. Thanks for useful comments from Duane Rettig and John Foder-

aro.

Thanks also to the PAPI project.

## References

- [1] Todd C. Mowry, Chi-Keung Luk. Memory Forwarding: Enabling Aggressive Data Layout Optimizations by Guaranteeing the Safety of Data Relocation, CMU-CS-98-170 School of Computer Science, Carnegie Mellon Univ., October 1998.
- [2] Paul R. Wilson, Michael S. Lam, Thomas G. Moher, “Caching Considerations for Generational Garbage Collection,” *ACM Lisp and Funct. Prog.* 1992 32–42.

## Appendix: Selected program listings and notes.

```
(defun mrl(n) ;;make random list
  (let ((ans nil))
    (dotimes (j n ans)
      (push (random (random #.(truncate most-positive-fixnum 4))
                ans))))

(defun lengthm(l m) ;compute length of list l m times
  (dotimes (j m)
    (declare (fixnum j))
    (mylength2 l 0)))

(defun mylength2(a n) ;fast small no-checking version of length
  (cond ((null a) n)(t (mylength2 (cdr a)(1+ n))))

;;; an in-place mergesort to randomize a list

;;; split at least 2
(defun split (x) ;; break the list x into two parts, odd and even
  (cond ((null x)(values nil nil))
        ((null (cdr x))(values x nil))
        (t
         (let((a x)
```

```

        (b (cdr x))
        (c (cddr x)))
    (setf (cdr a) nil)
    (setf (cdr b) nil)
    (split1 c a b))))

(defun split1(x y z)
  (cond((null x)
        ((null (cdr x))(setf (cdr x)(cdr y)) (setf (cdr y) x))
        (t
 (let((a x)
      (b (cdr x))
      (c (cddr x)))
    (setf (cdr a) (cdr y))
    (setf (cdr y)a)
    (setf (cdr b) (cdr z))
    (setf (cdr z)b)
    (split1 c a b))))
    (values y z))

(defun mymerge(x y);; faster version of merging, destructively (cf. CL version)
  (cond ((null x) y)
        ((null y) x)
        ((>= (car x)(car y))
         (setf (cdr x)(mymerge (cdr x) y))
         x)
        (t (mymerge y x))))

;; the sort function itself

(defun mergesort(r)
  (cond ((null r) nil)
        ((null (cdr r)) r)
        (t(multiple-value-bind
            (x y)
            (split r)
          ;;(merge 'list (mergesort x)(mergesort y) #'<)
          (mymerge (mergesort x)(mergesort y))
          ))))

```

```
(defun lengtha(a m)
  ;;compute touch each element in array a m times
  (declare (type (simple-array t (*)) a)
    (fixnum m) )
  (let ((r 0))
    (dotimes (j m)
      (declare (fixnum j))
      (dotimes (k (length a))
        (declare (fixnum k))
        (setf r (aref a k))))))
```