Memory Cache and Lisp: Faster list processing via automatically rearranging memory

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

November 17, 2003

Abstract

The speed of modern computers can be increased 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. Here we concentrate on automatic rearrangement of data, and examine the belief, common in some technical circles, that modern generational copying garbage collectors (GC) will improve data caching by relocating and compressing data, as a matter of normal processing. Since GC routines tend to be very well-tested and quite robust, if this belief can be confirmed by benchmarks, a GC could be a "free" way of reliably speeding up programs *in practice*.

In fact, our tests show this speedup phenomenon can be measured in some but not all sample Lisp programs. A novelty in our tests (at least when this paper was written) was using a full Lisp system linked to free software (PAPI) to access hardware machine registers. PAPI allows us to count cache misses during full-speed computation. We conclude that after a GC cache misses may be significantly reduced in some examples. Reorganization by GC speeds up computation by as much by a factor of 4, but in some cases the effect is negligible (or may even slow computation slightly). In any case, no extra effort is required of the programmer or user to take advantage of the potential speedup.

1 Introduction

The Lisp programming language uses linked lists of `cons` cells (two pointers each) as a fundamental data constructor. Current computer architectures (and memory systems) are slow at following links, so it would seem that 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 list pointers from one "random" location to another in working memory. Major efforts in improving hardware implementations such as 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 could be totally 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[1], and so such a program appears to require twice as much data cache to store a sequence compared to a sequential non-linked-list allocation.

Why would a programmer waste half the data cache? Consider that in past when memory was far more expensive, people invented, and persisted in using, linked lists. There must be a perceptible payoff in some form, for example, programmer convenience.

Oddly enough there can also be a memory locality payoff for using linked lists. In particular we have seen a pleasant result in a typical modern implementation of Lisp; an improvement in speed by memory reorganization may *require no specific effort by the programmer or any change to the source code.* It is done by the storage reclaimation activity of a copying garbage collector (GC), and as the GC runs normally without being explicitly called, this improvement usually 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 memory space. If the program traversing this data uses a similar path, the cache behavior can be quite improved compared to some random allocation. In a linear list, there is not much question of the order of traversal, and so the GC and the usual traversal agree.

There are other earlier papers on re-organization of memory; a good

[1]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.

survey is contained in the paper by Mowry and Luk on memory forwarding [3]. (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.) A more recent paper by Chilimbi and Larus [2] specifically addresses some of the same issues as we do, extending the range of applicability by suggesting how profiling and rearranging object storage in memory can attain better performance. They conclude that such techniques may help narrow, or even reverse, the performance gap between high-level languages such as Lisp, and low-level languages. Unfortunately this paper [2] has benchmarks on only a single architecture and a relatively unknown language, Cecil. Their references provide useful pointers into the literature.

2 A simple test

We constructed a suite of simple tests 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 succession. 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 10^9 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. This is our "good cache behavior" layout, and traversing data in the form should be as fast as possible.

In version b, the cells are randomly ordered (99% of pointers are to addresses over 2^{15} bytes away). This is our effort to provoke bad cache behavior in our data.

For version c, we start with the cells arranged as in version b, but trigger the GC which in effect returns the memory to an arrangement in which 99.99% of the pointers are to the next cell, resembling version a. (Does this include the time for the GC? It doesn't matter since this time is negligible[2]. In a real-world application the GC time tends to be 5 percent or so of processing time overall; clearly for our test we are revisiting the same locations repeatedly, so the cost for moving them (once) is relatively small compared to the rest of the computation.)

Most of our tests are on a Pentium III with a total L1 cache size of 32 Kbytes. This is divided into 16 Kbytes each for instructions and data, and a 256 Kbyte L2 cache, 8-way set associative. Additional tests were performed on a Sun Microsystems SunBlade 100. Tests can be performed with the same code on any Allegro Common Lisp system on any architecture. The cache counts show below also require PAPI [3] as well. PAPI [1] runs on many computers.

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. Recall that each of the tests is essentially traversing the number of pointers (10^9), and in a cache-less machine would take the same time. The table below shows a variation between 3.6 and 112 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

[2] In these tests the GC times are 16-32 ms. Even for a 100,000 word list, the GC need only scan the data structure once and copy it to "new space".

[3] http://icl.cs.utk.edu/projects/papi/

```
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 `cons`es 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 256Kbyte 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 each of the following benchmarks on the Sun, we give three times,
in seconds for tests {a,b,c}. In a non-cache-memory world,
the {a,b,c} times  would be the same.

List Length    Number of    Times in Seconds
```

in words	iterations	
100	10,000,000	{6.48, 6.47, 6.50}
1,000	500,000	{6.06, 6.07, 6.07}
2,000	500,000	{6.03, 6.04, 15.90 ??}
2,003	499,251	{6.05, 6.05, 6.05}
2,011	497,265	{6.05, 6.05, 6.05}
5,000	200,000	{16.1, 19.6, 16.0}
10,000	100,000	{16.1, 18.8, 15.9}
30,000	33,333	{16.6, 20,8, 17.7}
32,000	31,250	{18.4, 31.3, 18.4}
50,000	20,000	{28.6, 101.2, 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 used a Pentium-III 933MHz workstation, although results should not differ qualitatively from other speed, cache sizes, or other popular cache organizations[4]. Thanks to the PAPI project these benchmarks can be performed simply by calling programs to read the registers before and after each run. The benchmarks can be run on any other (unaltered) Lisp program using a half-page of "foreign function" declarations. Since there are PAPI bindings for other languages, similar tests could be performed on other systems.

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

[4]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.

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).

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 was stored in sequential memory locations. In this third test, the speed of scanning R is similar to the time required to scan the 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 use 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 to linked list cells which are individually managed.

Lists are not inevitably associated with GC 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

5.48               1.26e8         1.90e5   array of 16,000
4.54               2.51e8         6.07e5   list of  16,000 organized
7.27               9.06e8         3.33e5   list of  16,000 randomized
```

Note that *organized list traversal* is faster.

If we double the size of the array and the list, we reach a cache limit with the list, but not with the array. Here *array indexing* is much faster.

```
 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 data cache.

(a) The (by now) 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). Developing tools for optimization of data layouts with respect to cache performance for instructions and data are popular topics in the architecture and programming language research communities. [1, 2].

(b) The one suggested here: Leave data in lists and allow the system (or provoke the system) to perform a copying GC. In an ideal case the layout of the data in the copied list as traversed by the GC (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, appears to be not so well-known generally, Comments on this possibility appeared in the literature at least as early as 1980 [4], and are discussed in the literature [3].

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 GC 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). In general the programmer can look at ways of updating already allocated memory, but this may not be advantageous. A newly allocated list can be compressed more easily than one that has been "tenured" in a generational GC system. Detailed suggestions can be found in Wilson's 1992 paper [5].

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 via profiling [2] or even later at 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.

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

vary," but in this admittedly simple benchmark, use of a GC can speed the computation substantially.

6 Acknowlegments

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 Foderaro, and the anonymous SIGSAM Bulletin referee.

Thanks also to the PAPI project for making their measurement tools available.

A draft of this paper has been available on-line since July 22, 2002, at `http://www.cs.berkeley.edu/~fateman/papers/cachelisp.pdf`. Other material in that directory may be of interest.

References

[1] London, K., Dongarra, J., Moore, S., Mucci, P., Seymour, K., Spencer, T. "End-user Tools for Application Performance Analysis, Using Hardware Counters." *Proc. Int'l Conf. on Parallel and Distributed Comp. Systems*, August, 2001.

[2] T. M. Chilimbi and J. R. Larus. "Using Generational Garbage Collection to Implement Cache-Conscious Data Placement." *Proc ISSM '98*, ACM Press, 37–47.

[3] 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.

[4] J. L. White. "Address/memory management for a gigantic LISP environment, or, GC considered harmful," *Conf. Record of the 1980 LISP Conference*, ACM Press. 119-127.

[5] 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))))))
```