# An Architecture for Efficient Lisp List Access

A. R. Pleszkun
M. J. Thazhuthaveetil

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706

## Abstract

In this paper, we present a Lisp machine architecture that supports efficient list manipulation. This Lisp architecture is organized as two processing units: a List Processor (LP), that performs all list related operations and manages the list memory, and an Evaluation Processor (EP), that maintains the addressing and control environment. The LP contains a translation table (LPT) that maps a small set of list identifiers into the physical memory addresses of objects. Essentially, the LP and LPT virtualize a list. The EP then operates on these virtualized lists. Such an organization permits the overlap of EP function evaluation with LP memory accesses and management, thus reducing the performance penalties typically associated with Lisp list manipulation activities. We used trace-driven simulations to evaluate this architecture. From our evaluation a relatively small LPT is seen to be sufficient, and to yield "hit rates" on data accesses higher than those of a data cache of comparable size.

## 1. Introduction

A number of approaches have been advocated for distributing the execution of Lisp programs over a few specialized processors. On the one hand, there are the approaches that try to exploit the implicit parallelism of a Lisp program by concurrently evaluating different parts of it [Hals84a, Yama83a]. Then there are the approaches that acknowledge the high cost of list manipulation operations and provide specialized processors for garbage collection or memory management, I/O, translation, evaluation, etc. [Will78a, Davi85a].

Based on our studies of Lisp list behaviour [Thaz86a], we believe that there is an advantage to be gained from greater emphasis on hardware for making list access and modification more efficient. We propose an organization in which a specialized list processor is in charge of performing all operations on lists. We also describe results from our evaluation of this organization.

The list data structure has not received much attention from computer architects in the past and support for list manipulation in Lisp machines has been minimal. Some machines provide micro-coded implementations of the Lisplist manipulation primitives. Examples of this are the MIT Lisp Machine and its descendents [Bawd77a, Moon 85a]. These and other Lisp architectures [Deut78a] aim for efficient list representation, and hence efficient list access, through compact representation schemes like cdr coding [Hans69a, Clar77a, Bobr79a]. We believe that having an efficient list representation for lists does not necessarily lead to efficient list access. Other than [Sohi85a] and [Pott83a] there have been few proposed Lisp systems that emphasize efficiency in list manipulation.

## 2. Proposed Lisp Machine Organization

The organization we propose for a Lisp machine is shown in Figure 1. The 2 main functional units are the EP (Evaluation Processor) and the LP (List Processor). The EP is in charge of program control, all non-list related data manipulation, and environment control,

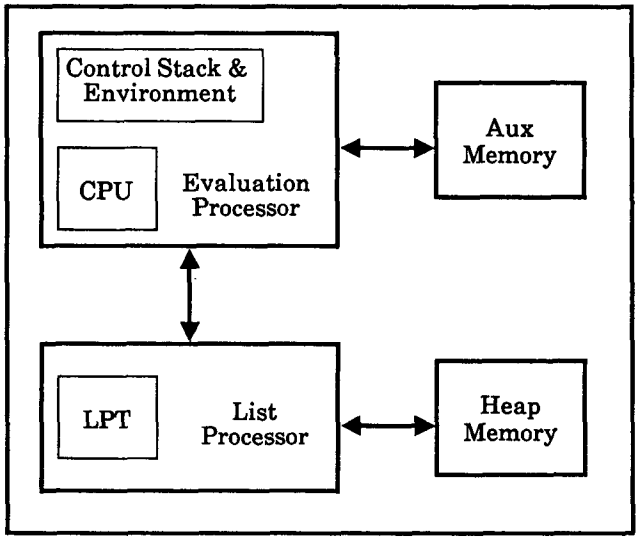


Figure 1. Proposed Organization



0884-7495/86/0000/0191$01.00 © 1986 IEEE

which includes control stack and association list manipulation for function calls. The EP passes all list related work to the LP. This general organization is suggested in [Bake78a] but to our knowledge has not been developed.

To make the communication between the EP and the LP efficient, the EP does not deal directly with real addresses. In managing list memory, the LP will have to move objects from place to place, and the EP must be shielded from using an address that has been rendered obsolete by LP memory management activity. We therefore map list cell addresses into small *identifiers* through a translation table in the LP.We call this translation table the *LPT*.

The EP specifies lists by directly addressing entries in the LPT, which is managed by the LP. In essence the LP and associated LPT *virtualize* a list. The EP then operates on these virtualized lists. This referencing scheme serves to reduce the semantic gap between the machine and the language. In most implementations the distinction between a list pointer and a Lisp object is lost. Though conceptually Lisp functions pass each other *object arguments*, this passing is implemented by passing *pointers* to those objects, and the concept of an object loses meaning. The LPT enables some of this distinction to be retained. Though this fact might not be of much importance in Lisp, it takes on added significance if these ideas are extended to machines for more object-oriented languages.

Figure 2 shows the components of the LPT; the fields of a table entry will be described in detail later. The address mapping is contained in the *identifier* and *address* fields. This mapping is like virtual memory mapping since both permit flexible use of physical memory without inconvenience to the application program.

| ID | CAR | CDR | COUNT | ADDR |
|---|---|---|---|---|
| L1 | - | - | 1 | a1 |
| L2 | - | - | 1 | a2 |

Figure 2. LPT Fields.

Of the various duties performed by the EP, the management of the environment is the most complicated. A Lisp environment changes on every function call and return. On a function call the variable-value bindings must be made, possibly with the saving of previous bindings. The environment is searched during the evaluation of the function to determine the current bindings for the names being referenced. If a referenced value happens to be a list object, then it is represented by an LPT address. On function return some of these bindings have to be undone, i.e. replaced with the values that they had before the function was called.

The EP communicates with the LP using operations similar to the list manipulation primitives of Lisp. The set of operations includes **car, cdr, cons, rplaca, rplacd, copy, readlist,** and **writelist.** For example, when the EP sends a **readlist** request to the LP, the EP expects to receive an *identifier* in return. It then binds this identifier to the program variable that is being read into. The LP, in turn, initiates I/O activity and updates its view of the lists currently being accessed to account for the newly created list object.

The details of LP operation revolve around the manipulation of LPT entries. Each entry in the LPT is basically an *(identifier, address)* tuple, where *identifier* is the short address used by the EP to identify list objects, and *address* is the actual physical memory address where that object is stored. To make list accesses fast we extend this tuple with 2 more fields, *car* and *cdr*. These are used to minimize recalculation of **car** or **cdr** operations on lists. The first time that a **car (cdr)** is computed, an LPT entry is created for the return value object and the *identifier* of that object is stored is stored as the *car (cdr)* of the argument list object. Future requests for the **car** or **cdr** of the list object can then be satisfied directly from the LPT. This feature of the LPT thus attempts to cache the more recently and frequently accessed parts of the list structure. We will see that it differs from a simple data cache in having a very sophisticated, Lisp specific replacement algorithm.

The LPT as described thus far will keep growing until it has an entry for each list object ever referenced. To reduce the size of the table, it becomes necessary to manage the table space just as one manages the heap memory in which the list cells reside. We consider a reference counting scheme [Coll60a, Weiz63a] to be the logical choice for this scenario. Reference counting does have its disadvantages, but we find that they are not critical in this setting. It is true that keeping track of a reference count for every list cell is wasteful of space and time. Since we have only one LPT entry for each list object currently being accessed, as we shall see in section 3.3, the number of counts being tracked is not excessive. Further, the reference count updating cost is a distributed heap management cost, and therefore not so much an overhead as an investment. So, we add a fifth field, *reference_count*, to the LPT tuple. An entry's reference count gets incremented when a new binding is made to that list object, and decremented when a binding gets unmade or when a reference count goes to zero.

Reference counting is generally considered unsuitable for real time applications because of the potentially unbounded amount of work that has to be done when a count goes to zero. We address this by the following optimizations: when an object's reference count goes to zero it gets reclaimed, but the reference counts of its children (the objects that are specified in its

*car* and *cdr*) will get their reference counts decremented only when the freed object gets reused. This makes the amount of work on object reclamation minimal at the expense of potentially keeping more LPT entries busy than necessary. To partially deal with this expense, free LPT entries are not remembered in a queue (first in first out) but on a stack (last in first out) implemented in the table. A Top of Stack register indicates the *identifier* of the next LPT entry to be allocated for use, and the stack is linked together through the *car* entries of the LPT. When an entry's reference count goes to zero, the only work that has to be done is to push it onto the free stack. If the children of the freed entry now actually also have reference counts of zero, they represent LPT space that is tied down but not actually referenced. Since we use a free stack rather than a free queue the most recently freed entry will be the first to be reused. Only when it is allocated for reuse are the reference counts of it's old children decremented. So, both LPT entry freeing and LPT entry allocation can be performed in a fixed amount of time.

Another common complaint with reference counts is their inability to reclaim circularly linked garbage. This is not strictly true; if we follow [Bobr80a] and [Frie79a] the LPT provides a means of distinguishing between internal and external pointers (a necessary condition for reclaiming circular lists using reference counts), and if we include an additional field in the LPT for the *circular list header* of each circular list, we can also reclaim circular lists. This would, however, impose an additional cost on all LPT accesses (to check and see if we are dealing with or creating a circular list) which is unacceptable. We opt, instead, to perform circular list reclamation at the time that the LPT is compressed to recover from table overflow.

*Compression* is an operation that is performed on the LPT when there are no more free table entries. If there are table entries that are only referenced from within the LPT we can compress them into their parents to make some table space available for immediate use. We call this condition *pseudo overflow*. Compression can be performed either when the LPT actually gets full or after it reaches a pre-determined level of occupancy. Further, compression can be carried out either until enough table space has been recovered to allow computation to continue (we will call this the Compress-One compression policy), or until there is no more compressible table space to reclaim (which we will call the Compress-All compression policy). We investigate the relative merits of these two policies later in this paper. Note that it is possible that there are no such compressible entries in the table; this. *true overflow* condition makes some form of LPT back-up storage necessary.

We illustrate some basic list manipulation operations in Figure 3. Figure 3a shows the LPT after 2 lists have been read in and designated as list objects L1 and L2 respectively. (Recall that there is an address field associated with each LPT entry.) The following operation is then performed on the two lists: {cons [cons (car L1) (cdr L2)] (car L2)}. First (car L1) is evaluated and as a result LPT entries L3 (for the car of L1) and L4 (for the cdr of L1) are created. Similarly, when (cdr L2) is evaluated LPT entries L5 and L6 are created. The LPT contents at that point are as shown in Figure 3b. The evaluation of (cons L3 L6) causes LPT entry L7 to be created. Figure 3c shows the LPT after the entire expression has been evaluated with return value L8. Note that to do 3 list accesses only 2 accesses of the actual list storage were necessary. The **cons** operations affect only the LPT and not the list heap memory. Due to this handling of **cons**, a value is returned to the EP with very little delay. The EP can thus continue its computation while the LP is in parallel updating LPT entries and allocating a new list cell in the heap. At the end of this evaluation the only external pointers are to

| ID | CAR | CDR | COUNT |
|---|---|---|---|
| L1 | - | - | 1 |
| L2 | - | - | 1 |

(a)

| ID | CAR | CDR | COUNT |
|---|---|---|---|
| L1 | L3 | L4 | 1 |
| L2 | L5 | L6 | 1 |
| L3 | - | - | 2 |
| L4 | - | - | 1 |
| L5 | - | - | 1 |
| L6 | - | - | 2 |

(b)

| ID | CAR | CDR | COUNT |
|---|---|---|---|
| L1 | L3 | L4 | 1 |
| L2 | L5 | L6 | 1 |
| L3 | - | - | 2 |
| L4 | - | - | 1 |
| L5 | - | - | 2 |
| L6 | - | - | 2 |
| L7 | L3 | L4 | 1 |
| L8 | L7 | L5 | 1 |

(c)

Figure 3. Example of List Manipulation in LPT.

the two initial objects, L1 and L2, and to the return value L8. If the LPT has size 8 entries, there is potential for a true overflow condition to occur. Since there are no compressible LPT entry pairs, any LP activity that requires an additional LPT entry at this point would cause a true overflow. Note also that Figure 3b is a potential pseudo overflow scenario for an LPT of size 6, since the LP could compress L3 and L4 into L1 to free 2 table entries for immediate use. In the Lisp machine organization outlined above, the LPT is intended to capture that subset of list structure nodes that is being actively accessed. Recalculation of Lisp access primitives is made unnecessary by storing some attributes of a list object in its LPT entry. Further, the organization deals effectively with temporary cons cells. It has been observed. [Fode81a] that in one large Lisp program (Macsyma) an average of 385 cons cells are allocated and quickly discarded before a more permanent one is allocated. This turns out to be a major part of traditional garbage collection activity. Our organization manages these temporary cons cells more efficiently since cons cells are created as LPT entries and cease to exist when their reference counts become zero. So, the transient cons cells soon disappear while permanent cons cells survive longer.

## 3. Evaluation

Before designing the proposed architecture in more detail and possibly implementing it, we have chosen to evaluate the effectiveness of its chief features: the translation table and LP operation. The possible problems that we forsee with these features are:

(1) frequent true LPT overflow,
(2) excessive reference count modification activity, and
(3) an insufficient number of access requests being met by the information contained in the table.

To investigate the above issues, we wrote a trace-driven simulator of our architecture.The traces to drive this simulator were derived from runs of a few large Lisp programs including: a circuit simulator (Slang), a PLA generator (PlaGen), an editor (Editor), and a VLSI design rules checker (Lyra)[1]. During typical interpreted runs of each program we caused trace information to be written on entry to and exit from (a) each Lisp primitive (name and arguments), and (b) each user defined function.(function name and number of arguments). The former information is needed to trace list object access and modification history, while the latter is needed to trace the amount of EP-LP activity relating to maintaining the program environment.

[1]Our benchmark selection procedure may seem to be heavily biased towards VLSI CAD tools. This was not intentional; it just so happens that most of the typical, large Franz programs we were able to get hold of are from that area. Small artificial Lisp kernels.[Gabr82a] are not suitable for our evaluation purposes since the nature of the interaction between the EP and the LP over function calls is one of our interests.

One difficulty with using this kind of trace is that two list arguments that look identical could actually be different objects, i.e. they could have been created independently and stored in different memory locations. Since we did not have access to the low-level details of the Franz Lisp system that we used, we could not access the information needed to overcome this difficulty. Thus, to deal with the traced list arguments, we can either consider all the list arguments to be independent or that identical list arguments always refer to the same list object. If we consider the list arguments to be independent, i.e. unique list objects, then our evaluation would become unduly pessimistic. We know however that these lists are not all independent since there is some degree of locality of reference in Lisp programs [Smit85a]. To simulate this locality we used the following strategy. If the list returned by one primitive function is identical to the argument of the next primitive function in the trace, we assumed that the two lists refer to the same object. In other cases, we assume that the list argument points to an existing non-local object, points to an existing local object, or points to a new object. One of these three possibilities is selected for the list argument based on a probability distribution specified as a parameter to the simulator.

The generated traces varied in length between 1437 and 160,933 primitive accesses performed among from 342 to 11907 user-defined function calls, with a maximum call depth of from 14 to 29. Table 1 characterizes this aspect of the traces.

| Trace | Functions | Primitives | Max Depth |
|---|---|---|---|
| Lyra | 11907 | 160933 | 27 |
| PlaGen | 8173 | 34628 | 15 |
| Slang | 620 | 2304 | 14 |
| Editor | 342 | 1437 | 29 |

Table 1. Content of the 4 Traces

The simulator monitors the contents of the LPT and the control-cum-binding stack over the function calls and list manipulating primitives of a trace. The mechanics of stack update on a function call are as follows: using the trace information (about the number of arguments to that particular function) a stack item is pushed for each argument, which is then randomly bound to something older to it on the stack. A randomly determined number of locals are then similarly bound on the stack. On function return these stack items are popped.

For a given simulation run, 5 simulator parameters can be specified: (1) *TableSize*, (2) *OverflowPolicy*, (3) *ArgProb*, (4) *LocProb*, and (5) *BindProb*. *TableSize* is the number of entries in the

LPT, and *OverflowPolicy* is the strategy to be employed on pseudo overflow (either Compress-One or Compress-All). The other three parameters specify probabilities to be employed in the random selection of arguments to the list manipulating Lisp primitives. They were used as follows: during the simulation, the argument of the primitive function whose execution is currently being simulated could be chosen from any of three classes of variables, viz (a) an argument of the currently active user-defined function, (b) a local variable of that function, or (c) a non-local variable. *ArgProb, LocProb, and 1 - ArgProb - LocProb* are the probabilities that we used in making this selection. A particular variable was then randomly selected from among the members of the selected class, and the primitive function evaluated, resulting in a return value. This return value was then either bound to a randomly selected variable on the stack (with probability *BindProb*) or just pushed onto the top of the stack (with probability *1 - BindProb*). For all the runs reported, the probability parameters. *ArgProb, LocProb,* and.*BindProb* were set at 0.6, 0.3 and 0.01 respectively. Note that this implies that 90% of all arguments to Lisp primitives are assumed to be arguments or local variables of the current user defined function, and only 10% are non-locals. We based this parameter setting on observations we made while selecting the Lisp benchmarks [Thaz86a].

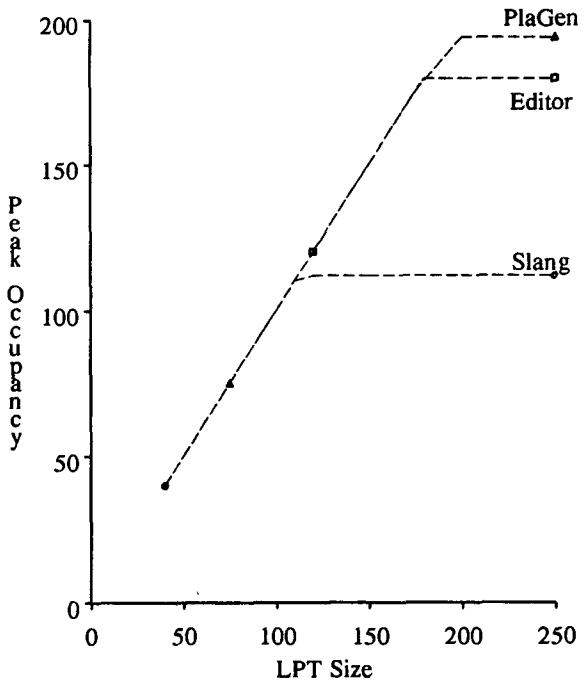


Figure 4. Peak LPT Usage Behaviour.

### 3.1. Optimal LPT Size

The first two issues described earlier in this section relate to LPT overflow conditions. When a pseudo overflow occurs the LP must spend some time on compressing entries to free space for the immediate need, this is an overhead that we would like to avoid. A true overflow has more severe penalties associated with it since some portion of the LPT must be written out to backup store. If we provide a sufficiently large table, overflow will occur rarely. Our first evaluation goal was to determine the optimal table size.

Figure 4 shows sample results from the studies on required table size. Each trace was run through the simulator for increasing values of the table size parameter using the same random number generator seed for each run. Three plots are shown on the graph, one for each of three of the sample traces. The first point for each plot is the smallest table size for which true table overflow did not occur. Even for the long Lyra trace of 160,933 references (the trace that was not plotted in Figure 4) true overflow occurred only when the table was less than a few hundred entries large. We expect that true table overflow will be extremely rare given a table of a few thousand entries. Each of the plots exhibit the same basic shape, a line with a 45 degree slope connected to a horizontal line. Such a curve will occur for any program trace. The knee of each plot occurs at that table size for which no form of overflow occurs. The plot for the Lyra trace is not shown in Figure 4, but is of the same shape as the curves shown, with a knee at 2000. So, in a translation table with 2K or 4K entries even pseudo overflows would occur rarely.

Figure 5 illustrates the largest number of LPT entries needed in.each of the traces. In this experiment we ran each trace through the simulator with between 60 and 90 different seeds and estimated the table size value where the knee (as seen in Figure 4) occurred. By re-seeding the random generator and re-running a trace we simulate a totally different access pattern. So, we are simulating the performance of different program behaviour. The number of runs was chosen to obtain acceptable confidence intervals for the observation for each trace.

The two points plotted for each trace in Figure 5 represent the extreme values of the knee (highest number of table entries used) observed over these runs. The interval for the Lyra trace stands out in this graph. This does not seem to be due to the fact that it is the longest of the four, since the difference in behaviour between the traces does not correlate with trace length. Even though there is more than an order of magnitude difference in trace length between the PlaGen and Editor traces, they show much the same trends in the graph of Figure 5. It appears that the behaviour shown by the Lyra trace is because of an intrinsic difference in program behaviour from that displayed in the other 3 traces. Because of the nature of Lyra's computation, it has a larger *working set* than the other traces; this is not entirely due to trace length.

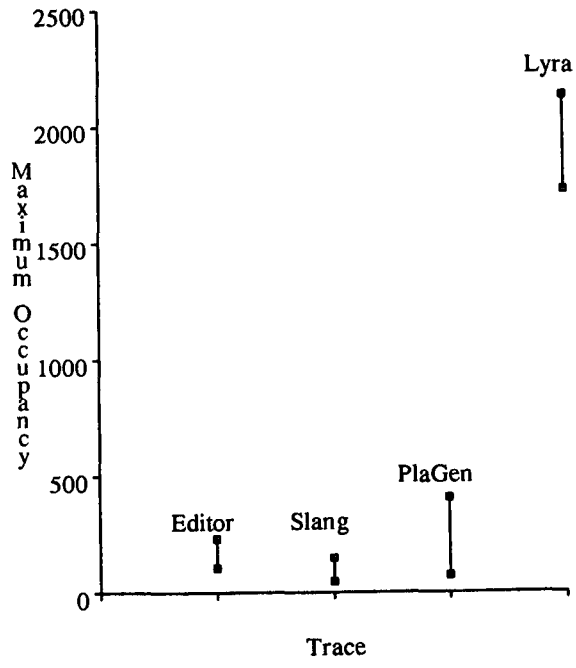


Figure 5. Maximum LPT Occupancy Levels.

### 3.2 Effect of Compression Policy

We next considered the two possible strategies for dealing with pseudo overflows; recall that we could either choose to compress LPT entries just enough to satisfy the immediate need (Compress-One), or we could perform compression over the whole table at the time of overflow (Compress-All). Figure 6 shows how this policy affects table performance. Using the same random generator seed with increasing limits on the table size we estimated the average LPT occupancy resulting from each of the policies. Note that these are plots of average and not maximal table occupancy, hence the jagged nature of the plots in Figure 6 compared to the straight line behaviour of maximal occupancy as plotted in Figure 4. The graph shows results from the Slang and Editor traces. The other 2 traces exhibited the same general behaviour.

As might be expected, the Compress-One policy causes the average LPT occupancy levels to be higher than the Compress-All policy. Given that the latter policy involves a compression phase of unbounded length, it is undesirable in a real time system. Fortunately, the graph indicates that the mean difference between the average LPT occupancy resulting from the 2 policies do not greatly differ. Recall that we left the reclamation of circularly linked garbage to be done at pseudo overflow compression time, implicitly assuming that the compress one policy would be used. A hybrid scheme is also conceivable. In such a scheme, Compress-One is used by default, but Compress-All is applied if pseudo overflows become frequent. We did not chose to evaluate this strategy.

### 3.3. LPT Activity

The second concern we expressed earlier in this section was that there might be excessive reference count arithmetic in the LPT. Our next evaluation goal was to investigate this issue. Table 2 summarizes measurements of the degree of LPT activity. The columns in Table 2 are: *Refops* (the number of times reference count arithmetic was performed), *Gets* (the number of LPT entry allocation requests), and *Frees* (the number of times reference counts went to 0 freeing an LPT entry). In computing. *Refops* we assumed that when a reference count goes to zero, the newly freed LPT entry, say L, gets pushed onto a stack of free entries, but that the reference counts of its children get

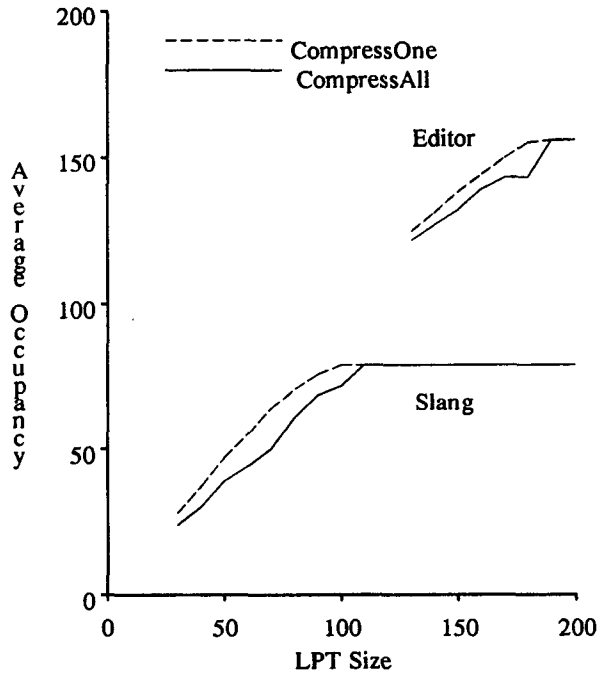


Figure 6. LPT Behaviour and Pseudo Overflow Policies.

| Trace | Refops | Gets | Frees | RecRefops |
|---|---|---|---|---|
| Lyra | 170232 | 29746 | 29746 | 27 |
| PlaGen | 92414 | 7248 | 7248 | 15 |
| Slang | 6852 | 1794 | 1794 | 14 |
| Editor | 4585 | 233 | 233 | 29 |

Table 2. LPT Activity

decremented only when LPT entry L is re-allocated for use *RecRefops*, on the other hand, is a count of the number of times reference count updating operations took place if a more simplistic policy is employed when a reference count goes to zero. Under this policy, when an LPT entry's reference count goes to zero the reference counts of its children are immediately decremented. Recall that we had discussed this in Chapter 4, but dismissed it as inferior since table freeing time becomes non-deterministic. From the differences between the *Refops* and *RecRefops* fields of Table 2 we see that this second policy leads to as much as a 47% increase (in the Editor trace) in the amount of reference count updating taking place. Note that the LPT maintenance costs are not excessive; using the information about the content of the traces from Table 1 we see that the statistics in Table 2 indicate between 1 and 3 reference count modifications per primitive list access, and between 1 and 4 table entry releases or allocations per user-defined function call.

Whether this amount of LPT activity seriously degrades performance depends on the hardware implementation. Results from the study just described suggest an optimization. We observed that a large percentage of the reference count activity was related to references from the stack. We could choose to only count references internal to the LPT in the *reference_count* field. The LPT would then have an additional bit field, *StackBit*, indicating whether or not there are any references to that entry from the stack. A separate reference count table could be maintained.*in the EP* for list references from within the stack. Only when one of those counts goes to zero need the LP be informed; it would then set the corresponding *StackBit* field to false. This scheme has the advantage of still maintaining strict control over all references to the heap while reducing the traffic over the EP-LP bus due to reference count activity.

### 3.4. LPT Buffering Capability

The use of the LPT requires some non-trivial amount of memory to buffer information about recently referenced lists. This is in fact a service that could be provided by a data cache. We decided to evaluate the buffering capability of the LPT by comparing it with a data cache of comparable size. Naturally, one must be cautious when interpreting these results since the LPT provides much more functionality than a cache; it detects the creation of garbage, and makes possible a decoupling of EP activity and heap activity. With a cache, more memory access are required for bookkeeping functions than with an LPT. None of these bookkeeping activities are accounted for in the results reported below. For this evaluation, we consider a fully associative, LRU replacement data cache of the same size as the number of entries in the LPT. Each cache line is one 2 pointer list cell large and no prefetching is attempted. Table 3 shows sample results from this comparison. The miss counts in the table represent the

| Trace | Size | LPTMisses | HitRate | CacheMisses | HitRate |
|---|---|---|---|---|---|
| Lyra | 1702 | 8221 | 97.27 | 12875 | 97.27 |
| | 2000 | 5682 | 98.12 | 9514 | 98.12 |
| | 2300 | 4314 | 98.57 | 7679 | 98.57 |
| PlaGen | 75 | 3510 | 94.31 | 6167 | 90.01 |
| | 150 | 825 | 98.66 | 1952 | 96.84 |
| | 225 | 230 | 99.63 | 934 | 98.49 |
| Slang | 40 | 625 | 78.63 | 868 | 70.31 |
| | 80 | 223 | 92.37 | 439 | 84.99 |
| | 120 | 46 | 98.43 | 204 | 93.02 |
| Editor | 120 | 240 | 91.14 | 325 | 88.00 |
| | 150 | 104 | 96.16 | 146 | 94.61 |
| | 180 | 53 | 98.04 | 89 | 96.71 |

Table 3. Comparison with Data Cache

number of times **car** and **cdr** requests were not satisfied by the *car* or *cdr* fields of LPT entries, or were misses in the data cache. We list both miss counts and hit rates to make clearer the difference in performance. Note that the LPT consistently produces more hits for the ranges of table size studied. The ratio of cache misses to LPT misses varied from 1.354 to 4.435. For large table sizes (over 2K entries) this study showed high hit rates for both cache and LPT, as seen in the Lyra results. However, the disparity between the actual number of cache misses and LPT misses displayed in Table 3 continued. Figure 7 shows the relative hit rates for

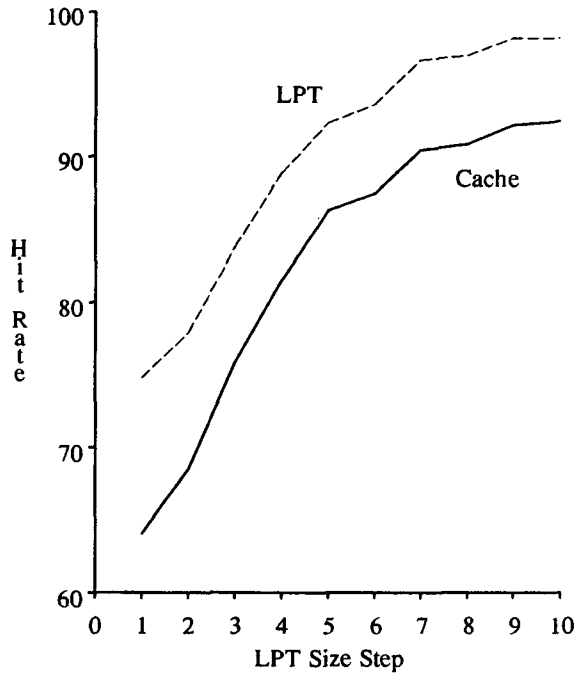


Figure 7. LPT/Cache Comparison with Slang Trace.

cache and table with increasing table size limits for one of the traces. We reiterate here that the LPT-cache comparison was conducted to evaluate the buffering capabilities of the LPT; the LPT is a more complicated device than a data cache and provides far more functionality.

## Conclusion

We have proposed an organization for a Lisp machine geared towards supporting efficient list manipulation. This Lisp architecture contains two processing units: a List Processor (LP), and an Evaluation Processor (EP). The key feature of the LP is the List Processor Translation Table (LPT). The LPT maps the short list object addresses used by the EP into actual physical memory addresses, while capturing relevant parts of the currently accessed list structure for fast access. This architecture was evaluated using a trace-driven simulator. The traces were derived from typical runs of 4 large Lisp programs. An LPT with 2K table entries is adequate to capture the list activity of our Lisp program traces. True LPT overflow will occur infrequently with a table of this size. Our evaluation also shows that the Compress-One pseudo LPT overflow recovery policy leads to faster LPT operation at the cost of higher average LPT occupancy, while the Compress-All policy leads to non-deterministic overflow recovery time. We suggested a hybrid LPT compression policy to minimize performance degradation due to the overflow handling overhead. The LPT is seen to capture the temporal locality of Lisplist access better than a data cache of comparable size. The results of these studies indicate that the use of such an organization can lead to high performance machines for the execution of Lisp programs.

## References


[Bake78a] Baker, H. G. Jr., "List Processing in Real Time on a Serial Computer," *Communications of the ACM* **21**(4) pp.280-294 (April 1978).

[Bawd77a] Bawden, A., et al., "Lisp Machine Progress Report," *MIT AI Laboratory Memo No. 444*, (August 1977).

[Bobr79a] Bobrow, D. G. and D. W. Clark, "Compact Encoding of List Structure," *ACM Transactions on Programming Languages and Systems*, (October 1979).

[Bobr80a] Bobrow, D. G., "Managing Reentrant Structures Using Reference Counts," *ACM Transactions on Programming Languages and Systems* **2**(3) pp. 269-273 (July 1980).

[Clar77a] Clark, D. W. and C. C. Green, "An Empirical Study of List Structure in Lisp," *Communications of the ACM* **20**(2) pp. 78-87 (February 1977).

[Coll60a] Collins, G. E., "A Method for Overlapping and Erasure of Lists," *Communications of the ACM* **3**(12) pp. 655-657 (December 1960).

[Davi85a] Davis, A. L. and S. V. Robison, "The FAIM-1 Symbolic Multiprocessing System," *Spring 1985 Compcon Digest of Papers*, pp. 370-375 (1985).

[Deut78a] Deutsch, L. P., "Experience with a Microprogrammed Interlisp System," *Proceedings of the 11th Annual Microprogramming Workshop*, pp. 128-129 (November 1978).

[Fode81a] Foderaro, J. K. and R. J. Fateman, "Characterization of VAX Macsyma," *Proceedings of the 1981 Symposium of Symbolic and Algebraic Computation*, p. 14 (1981).

[Frie79a] Friedman, D. P. and D. S. Wise, "Reference Counting Can Manage the Circular Environments of Mutual Recursion," *Information Processing Letters* **8**(2) pp. 41-44 (1979).

[Gabr82a] Gabriel, R. P. and L. M. Masinter, "Performance of Lisp Systems,"*Conference Record of the 1982 ACM Conference on Lisp and Functional Programming*, pp. 123-142 (1982).

[Hals84a] Halstead, R. H. Jr., "Implementation of MultiLisp: Lisp on a Multiprocessor," *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 9-17 (August 1984).

[Hans69a] Hansen, W. J., "Compact List Representation: Definition, Garbage Collection and System Implementation," *Communications of the ACM* **12**(9) (September 1969).

[Moon85a] Moon, D. A., "Architecture of the Symbolics 3600," *Proceedings of the 12th Annual Symposium on Computer Architecture*, pp. 76-83 (July 1985).

[Pott83a] Potter, J. L., "Alternative Data Structures for Lists in Associative Devices," *Proceedings of the 1981 International Conference on Parallel Processing*, pp. 486-491 (1983).

[Smit85a] Smith, A. J., "Cache Evaluation and the Impact of Workload Choice," *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pp. 64-73 (July 1985).

[Sohi85a] Sohi, G. S., E. S. Davidson, and J. H. Patel, "An Efficient Lisp-Execution Architecture with a New Representation for List Structures," *Proceedings of the 12th Annual Symposium on Computer Architecture*, (1985).

[Thaz86a] Thazhuthaveetil, M. J., "SMALL: An Architecture for Efficient Lisp List Access," Ph.D. Dissertation (in preparation), University of Wisconsin, Madison (June 1986).

[Will78a] Williams, R., "A Multiprocessing System for the Direct Execution of Lisp," *Proceedings of the 4th Workshop on Computer Architectures for Non-Numeric Processing*, pp. 35-41 (1978).

[Yama83a] Yamaguchi, Y., K. Toda, and T. Yuba, "A Performance Evaluation of a Lisp-based Data-driven Machine (EM-3)," *Proceedings of the 10th International Symposium on Computer Architecture*, pp. 363-369 (June 1983).