A Structured Memory Access Architecture for LISP

by

Matthew J. Thazhuthaveetil

Computer Sciences Technical Report #658

August 1986

A STRUCTURED MEMORY ACCESS ARCHITECTURE FOR LISP

by

MATTHEW JACOB THAZHUTHAVEETIL

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

August 1986

A STRUCTURED MEMORY ACCESS ARCHITECTURE FOR LISP

Matthew Jacob Thazhuthaveetil

Under the supervision of Assistant Professor Andrew R. Pleszkun

Lisp has been a popular programming language for well over 20 years. The power and popularity of Lisp are derived from its extensibility and flexibility. These two features also contribute to the large semantic gap that separates Lisp from the conventional von Neumann machine, typically leading to the inefficient execution of Lisp programs. This dissertation investigates how the semantic gap can be bridged.

We identify function calling, environment maintenance, list access, and heap maintenance as the four key run-time demands of Lisp programs, and survey the techniques that have been developed to meet them in current Lisp machines. Previous studies have revealed that Lisp list access streams show spatial locality as well as temporal locality of access. While the presence of temporal locality suggests the use of fast buffer memories, the spatial locality displayed by a Lisp program is implementation dependent and hence difficult for a computer architect to exploit. We introduce the concept of *structural locality* as a generalization of spatial locality, and describe techniques that were used to analyse the structural locality shown by the list access streams generated from a suite of benchmark Lisp programs. This analysis suggests architectural features for improved Lisp execution.

The SMALL Lisp machine architecture incorporates these features. It partitions

functionality across two specialised processing elements whose overlapped execution leads to efficient Lisp program evaluation. Trace-driven simulations of the SMALL architecture reveal the advantages of this partition. In addition, SMALL appears to be a suitable basis for the development of a multi-processing Lisp system.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last two decades, a gap has been emerging between computer architects and high level programming language designers. This gap is described in terms of what has come to be called the *semantic gap*. To quote Myers [Myer82a],

> Most current systems have an undesirably large semantic gap in that the objects and operations reflected in their architectures are rarely closely related to the objects and operations provided in programming languages.... There is a large gap in semantics between the programming environment and the representation of program concepts at the architecture level, ... (which) contributes to a large number of significant problems, among them ... (program) execution inefficiency.

Several approaches for overcoming the semantic gap problem come to mind, with the general idea of either raising the machine to the level of the language, or lowering the language to the level of the machine. Under one approach, a machine would be incrementally designed by adding to it the common architectural features desirable for the various languages to be used on it. Under another, the machine would be of a simple design, but time and effort would be invested in complicated software language processors to translate source programs into an efficiently computable form for the machine. Yet another approach would design special purpose machines for specific languages, accepting as a fact that the semantic gap cannot be effectively bridged otherwise.

Today, with the von Neumann model of computer architecture still largely dominant, few languages suffer from a wider semantic gap than Lisp. Despite this handicap, Lisp has been a popular programming language for well over 20 years. The power of Lisp is derived from the ability to build a powerful, friendly system from a few basic primitives and data types; it is extremely extensible. It has been likened to a ball of mud [Mose69a]: you start with a small one, add features to it, and it is still a ball of mud. Further, Lisp is dynamically typed and ideally suited for incremental program development, making it a good choice for the fast prototyping of software systems. As a result, several current symbolic manipulation and algebraic systems [Mart71a, Gris78a], design and graphic description systems, expert systems, and other heavily used non-numeric systems are based on an underlying Lisp system. Recent interest in Fifth Generation Computer Systems has sparked renewed interest in systems for the efficient execution of Lisp and Lisp-like languages. Typically these systems do not run efficiently due to the large semantic gap between list manipulating languages like Lisp and the conventional von Neumann machine.

This thesis investigates the semantic gap separating the programming language Lisp from conventional machines, and how that gap can be bridged or made smaller. Our research goals were two-fold: first, to characterize the nature of the locality of reference displayed by Lisp list access streams, and second, to design a Lisp machine architecture that takes advantage of this locality of reference to execute Lisp programs efficiently. This dissertation begins with a survey:

in Chapter 2 we identify *function calling*, *environment maintenance*, *list access*, and *heap maintenance* as the four key run-time demands of Lisp programs, and survey the techniques that have been developed to meet them in current Lisp machines. Based on this survey we examine issues relating to list access and representation in more detail in Chapter 3; we describe empirical studies of Lisp list behaviour that we conducted along with studies performed by other researchers. Previous studies have revealed that Lisp list access streams show spatial locality as well as temporal locality of access. The drawback of these studies has been that they were highly dependent on implementation details. We overcome this problem by extending the concept of spatial locality into what we refer to as *structural locality* of reference. In Chapter 3 we describe studies that we conducted into the structural locality of list referencing; we develop techniques to partition a Lisp list access stream into structurally related locales of high temporal locality of reference which we call *list sets*. This partition suggests architectural features that can improve Lisp execution efficiency. Chapter 4 contains a description of the organization and operation of SMALL, the Structured Memory Access of Lisp Lists architecture, that incorporates these features. The SMALL architecture partitions functionality across two processing elements whose overlapped execution leads to efficient Lisp program evaluation. We evaluate the effectiveness of the SMALL organization using a trace driven simulator in Chapter 5. In Chapter 6 we examine how the architectural ideas of SMALL can be extended to multiprocessor systems. We conclude, in Chapter 7, with a summary of the results reported in the thesis along with suggestions for future work.

Chapter 2

Supporting Lisp Execution

2.1. Introduction

Lisp ranks, along with Fortran, as one of the oldest programming languages in wide-spread use today [McCa78a]. It originated in the late 1950's as a list processing language [McCa60a], This version of Lisp is today known as Lisp 1 or "pure Lisp" and was mathematically elegant but awkward to program. Since the 1950's, Lisp has undergone a steady evolution, with Lisp 1.5 [Weis67a], Lisp 2 [Abra66a], Maclisp [Moon74a], Interlisp [Teit75a], Scheme [Suss75a], Franz Lisp [Fode79a], and T [Rees82a] being among the more prominent stages on the way. This proliferation of Lisps has prompted efforts to arrive at a Lisp standard, first in Standard Lisp [Mart78a] and, more recently, in Common Lisp [Stee84a].

The studies described in this dissertation are not directed at any specific Lisp implementation; we aimed to study issues that are important to all Lisps. We start this chapter, therefore, by discussing what takes place during typical Lisp execution. Based on this, we enumerate the run-time requirements of a Lisp system and identify potential obstacles to good machine performance. In section 2.3, we provide a classification of the Lisp machines that we encountered during our survey and examine the techniques employed in these machines to cater to Lisp's run time requirements.

2.2. Run Time Requirements of a Lisp System

A Lisp program is organised as a collection of functions that call each other. Lisp execution can therefore be thought of as a series of nested function evaluations. Extensive function calling is typical of Lisp. Another characteristic of Lisp is the data structures on which these functions operate - they are, by and large, lists. Understanding these Lisp hallmarks lays the foundation for discussing and contrasting different Lisp machine architectures.

2.2.1. Function Calling

In general, function calls and returns are expensive operations. Lisp function calls, however, are more complicated than those in a lexically scoped language like Pascal, since Lisp function evaluation takes place in a dynamically bound context; the *latest active value* bound to a variable name is used when that variable is referenced. At any instant in Lisp execution, there are a number of dynamically nested, uncompleted function calls. Only one of these, the most recent, is active. Each of these function calls has a *referencing context* associated with it. The referencing context is a set of name-value pairs that specifies the current bindings of the variable names used in the function. We will use the term *environment* to refer to the collection of referencing contexts corresponding to all the function calls that are uncompleted at a given time. When a function call completes and returns control to its caller, the referencing context of that

caller must be restored to allow it to become active and continue execution. It therefore becomes necessary to update the environment upon every function call and function return. When a variable name is encountered during the evaluation of the body of a function, the environment is *interrogated* for the current binding of the name. We will also refer to environment interrogation as *name lookup*. Maintaining the environment is not conceptually difficult; a simple scheme using a name-value binding stack would simply have to add and delete items from the top of the stack on function calls and returns in order to update the environment. In Lisp, where function calling is very frequent, it is essential that environment interrogation be fast; the simple name-value binding stack could result in slow lookup.

In a simple form of Lisp function evaluation, the definition of a function specifies exactly how many arguments the function expects. Calling a function with a different number of arguments would be an error. When the function is called, the arguments in the call are evaluated and then bound to the formal arguments of the function. The environment must be modified so that these new bindings are present in the currently active referencing context. The body of the function is next evaluated, and a return value made ready for the calling function. Finally, the environment is again modified to mirror the referencing context of the calling function, to which control is now being returned. Several variations to this mode of function evaluation have arisen. In one variation functions are allowed to accept a variable number of arguments. This is useful in program development, and has made Lisp a popular proto-typing language. Other variations do not evaluate arguments before binding them to formal arguments. The syntactic conventions used to specify these function calls vary from Lisp to Lisp [Jone82a]. For example, in Franz Lisp the user can define
(1) **exprs**, i.e., functions with a ***fixed*** number of arguments that are ***all evaluated***, using
(DEF <function name> (LAMBDA (<lambda variables>) <function body>)).
(2) **lexprs**, i.e., functions with a ***variable*** number of arguments that are ***all evaluated***, using
(DEF <function name> (LEXPR (<lambda variables>) <function body>)).
(3) **fexprs**, i.e., functions with a ***variable*** number of arguments that are ***not evaluated***, using
(DEF <function name> (NLAMBDA (<lambda variables>) <function body>)).

Finally, there are variations in which a function is allowed to return more than one value to the function that called it. All of these variations make it difficult to provide generally applicable architectural support for function calling in Lisp.

Unfortunately, the woes of Lisp function calling do not end there. In Lisp functions can be passed as arguments. This adds to the complexity of function calling. Such a functional argument gets bound to a formal argument as with all arguments, but when it is executed, the evaluation must be conducted in the referencing context that was present when the functional argument was initially passed as an argument. The problem of maintaining the environment consistently under such conditions is called the ***funarg problem***; a funarg is a function-environment pair [Alle78a]. This implies that function calling and returning is not strictly LIFO (last in first out) in Lisp, and complicates the implementation of environments. If function calling and returning was strictly

LIFO, the environment could be implemented using a LIFO stack. With the advent of functional arguments, information about the referencing context of a function call might have to be retained even after the call has returned. A technique to deal with this is described in [Bobr73a]. This technique has become fairly well accepted, and we will therefore not address this issue further.

Thus, there are two problems associated with supporting function calling in Lisp - the existence of a number of calling conventions, and the need to maintain the environment across function calls. Other than the funarg problem, Lisp function calls are not more complicated than function calls in other languages. It is the frequency with which these calls taks place and the updating of the environment that makes efficient function calling critical to Lisp system performance.

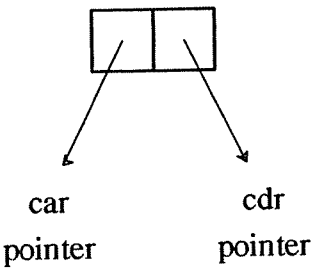
2.2.2. Dealing With Lists

The fundamental data structure in Lisp is the list. More specifically, Lisp data objects are called *s-expressions,* short for symbolic expressions. Two special cases of s-expressions are *atoms* and *lists.* Examples of atoms are numbers and names (character strings). A list is a collection of atoms and other lists. This definition makes recursive data structures possible. In Lisp notation, a list consists of a left parenthesis followed by zero or more atoms or lists separated by spaces and ending with a right parenthesis. Lists are typically represented as linked lists of *list cells;* a list cell is composed of a pair of pointers - the *car* pointer points to the contents of that list cell (which could be an atom or another list), and the *cdr* pointer is a link to the next list cell in that list. The special atom *nil* terminates lists. We will use the term *heap memory* to refer to the memory containing all the list cells. Figure 2.1 illustrates a list cell, with its *car* and *cdr* pointers, and how these list cells are used to represent lists.

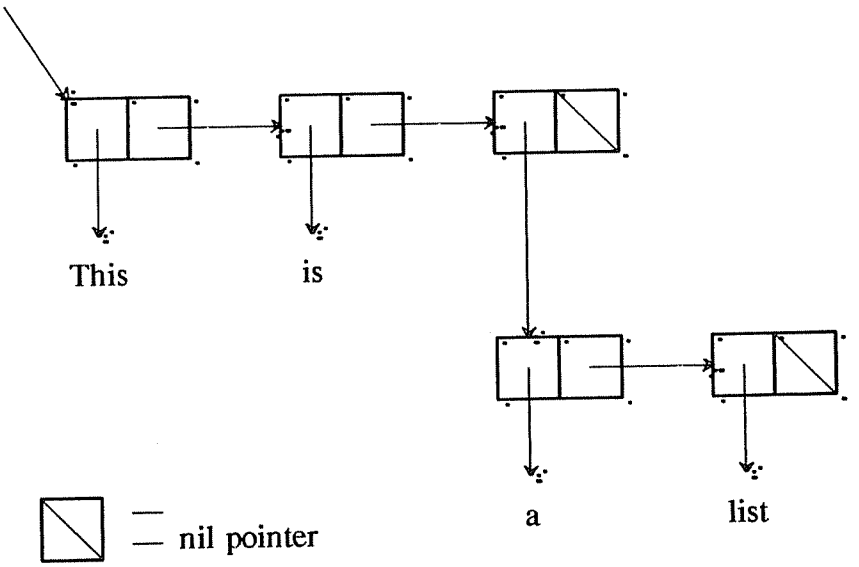
Lisp provides a set of pre-defined, primitive functions to manipulate lists. This includes **cons** (used to create a new list cell), **car** (returns the first element of a list argument), **cdr** (returns everything following the first element of a list argument), **rplaca** (used to replace the *car* pointer of the list argument), and **rplacd** (used to replace the *cdr* pointer of the list argument). The data manipulated by Lisp programs is stored in lists, and accessing it often involves traveling down several pointers at run-time using this set of primitive functions. It has been conjectured that a large fraction of Lisp execution time is spent in following these pointers [Fate78a].

There are no type declarations in Lisp. This facilitates quick program development, but complicates run time operation. Due to Lisp's dynamic nature, all data type checking is done at run-time. This could be less efficient than conducting these checks at compile time. As a simple example of the need for run-time type checking, consider an arithmetic function call *(subtract X Y).* The values of X and Y could be either integers or real numbers; a different action must be initiated in each case. This can be decided only at run-time. To facilitate this run-time type checking, Lisp machines use tagged memories and usually contain type checking hardware.

Another important issue in dealing with lists is the management of heap space. In a language like Fortran a compiler can predict the exact run-time



(a) A list cell, with car and cdr pointers



(b) Representation of (This is (a list)) using list cells

Figure 2.1. A List Cell and a Sample List.

memory requirements of any program. This is not possible in Lisp, where the number of list cells required during the evaluation of a program cannot be predicted by examining the program. Further, while the programmer explicitly

causes new data objects to be created, the programmer does not explicitly cause them to be reclaimed when they are no longer referenced. Left to itself, then, a Lisp program would soon run out of heap space. To prevent this from happening list cells have to be recovered, and made available for reuse, when there are no more extant pointers pointing at them. This task is performed by the Lisp system. Heap management has become an issue of prime concern to Lisp system designers. Lisp did not attain wide-spread popularity until there were efficient solutions to this problem.

2.2.3. Summary

Based on the discussion of the last two sections we consider the four major problem areas in efficient Lisp execution to be :

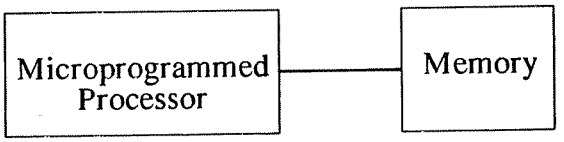
(1) efficient function calling,
(2) environment maintenance,
(3) list access and representation, and
(4) heap maintenance.

This assessment is confirmed by a survey of current Lisp machine designs; most of the special architectural features included in these machines address one or the other of the issues listed above.

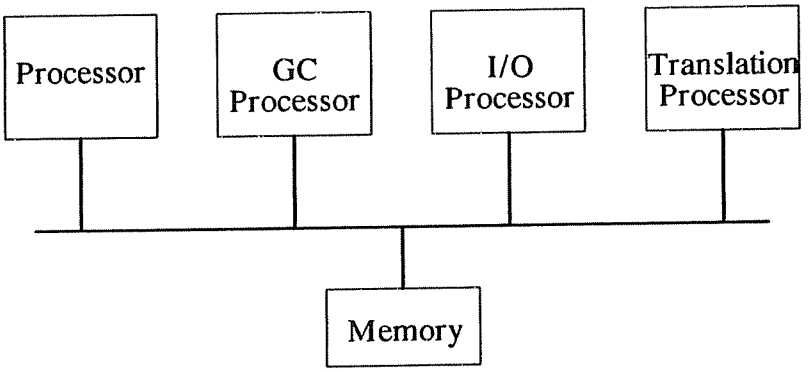
2.3. Approaches to Lisp Machine Design

The first implementation of Lisp was done between 1958 and 1960 on an IBM 704. In fact, it is from the architecture of the 704 that the Lisp access primitives ***car*** and ***cdr*** get their names. The 36 bit data word of the 704 had two 15 bit fields, called the address and the decrement, that could be independently fetched to index registers using special instructions. The names ***car*** (contents of the address part of the operand) and ***cdr*** (contents of the decrement part of the operand) evolved from the representation of two-pointer list cells in these data words. Later on, it was with Lisp in mind that the designers of the DEC PDP-6 and PDP-10 computers included half-word instructions and stack instructions in these architectures. The Interlisp and Maclisp Lisp implementations developed at MIT on these machines caused Lisp to gain in popularity in the artificial intelligence community. Interactive Lisps on time-sharing systems soon followed, and both user programs and Lisp implementations steadily increased in complexity in the years that followed.

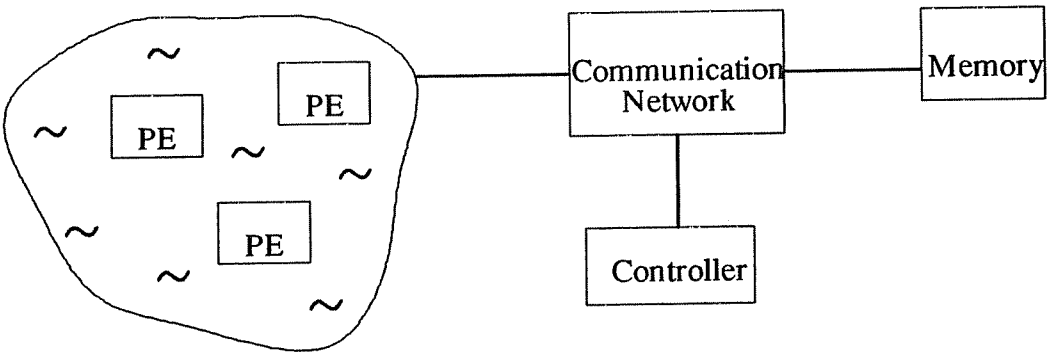
The next landmark in Lisp implementation came in 1977, with the MIT Lisp Machine project [Bawd77a]. Inspired by work done on personal Lisp machines at Xerox PARC [Deut78a, Deut73a, Deut80a, Burt80a], the MIT Lisp machine was a single-user computer, thereby assuring the user of a higher degree of service than that obtained on a time-sharing system. The MIT Lisp machine was implemented with a microprogrammed architecture, using "a very unspecialized processor" for reasons of speed, cost, and ease of microprogram debugging. Key Lisp primitive functions were implemented directly in microcode. Many of the commercial Lisp machines that have appeared on the market since then are based on the MIT Lisp Machine design experience. Examples include the Symbolics 3600 [Road83a, Moon85a], the LMI Lambda, and the TI



Class M Machine: Unspecialized Microprogrammed Processor



Class S Machine: Special-purpose Processors



Class P Machine: Pool of Identical Processors

Figure 2.2. Alternative Lisp Machine Organizations.

Explorer.† Another commercially available Lisp machine is the FACOM Alpha [Haya83a]. At the same time, there has been on-going research on alternative Lisp machine architectures, mainly at the university level.

The Lisp machines that we studied can be divided roughly into three classes as illustrated in Figure 2.2. First, there are the ***Class M*** machines, which are unspecialized microcoded Lisp processors, like the MIT Lisp machine or the Xerox PARC Lisp machine projects mentioned above. Second, there are the ***Class S*** machines, which are multiprocessor organizations where each processor serves a specialized function. Examples of this are MLS (an airborne multiprocessing Lisp system at the University of Illinois) [Will78a], the Fairchild FAIM-1 multiprocessor AI machine [Deer85a, Davi85a], and a Lisp machine project undertaken at Keio University in Japan [Yama81a]. Finally, there are the ***Class P*** machines, which are multiprocessor systems composed of pools of identical processing elements, aiming for high performance through concurrent evaluation of different parts of a Lisp program on separate processors. Examples of this class of Lisp machines are Guzman's AHR Lisp machine [Guzm81a], the EM-3 machine [Yama83a], and the Evlis machine project at Osaka University [Yama81a]. Within the framework of this classification we can discuss how different machines address our four architectural issues.

2.3.1. Function Calling

In section 2.2.1 we saw several Lisp function calling conventions. The simplest convention expected a fixed number of arguments that were all evaluated before the body of the function itself was evaluated, and the function returned a single value. More complicated conventions allowed for a variable number of arguments, arguments not being evaluated, or the return of multiple values. In practise, it is not necessary for a Lisp machine to support all of these function calling conventions. For example, suppose a function calling convention expects a fixed number of arguments that are all to be evaluated. To implement a calling convention that does not evaluate the function arguments, the evaluation must be suppressed. In most Lisps this can be done using the **quote** function, which suppresses the evaluation of its arguments. To allow a variable number of arguments, the actual arguments can be passed as a single argument in the form of a list, which is then split up into its parts (using **car** and **cdr**) in the function. Similarly, to implement a calling convention that evaluates its arguments using a calling convention that does not, we could cause the arguments to be explicitly evaluated in the function before the function body itself is evaluated.

Lisp functions are represented as lists. Low level primitives, like **car** and **cdr**, are typically implemented as a few machine instructions. During the evaluation of user defined functions, however, the Lisp interpreter must access the internal list representation of the function. The interpretation process thus involves frequent accessing of the list representation of the function, interspersed with the execution of machine instructions corresponding to primitive operations. To speed up the evaluation process Lisp compilers have been developed. A

---

† Explorer is a trademark of Texas Instruments Inc.

compiler reduces a function to a set of machine instructions, and results in faster execution. On the negative side, compilation sometimes requires the programmer to declare the properties of functions and variables. This weakens the flexibility of Lisp programming.

Class M and Class S machines typically provide a range of function calling support. In the MIT Lisp Machine, for example, a function can exist in any of three forms - as slow interpreted code, as slightly faster macro-compiled code, or compiled directly into microcode. The programmer chooses from among the three based on how frequently a function is used. In the case of interpreted or macro-compiled code there are two function call instructions, **CALL0** for calls with no arguments, and the slower **CALL** instruction for other functions. **CALL** initiates the evaluation of the function's arguments and their binding to formal parameters. The Symbolics 3600 divides this functionality over several instructions - there is a set of function calling instructions (including a special instruction for function calls that have no arguments), and a set of argument binding instructions. Both the MIT Lisp Machine and the Symbolics 3600 have function return instructions that allow multiple values to be returned. An interesting variation on the **CALL** instruction was used in a MicroLisp project at Xerox [Deut73a]. Based on an observation that most functions have between 0 and 6 arguments, 8 kinds of function calls are supported - for functions with 0, 1, 2, 3, 4, 5, 6, and more than 6 arguments. In the last case, the actual number of arguments is specified as a separate argument in the call instruction.

Several Class P machines have attempted to add another dimension to function calling by evaluating the arguments of a function call in parallel. In Guzman's multi-microprocessor Lisp machine, AHR [Guzm81a], and the Evlis multiprocessor Lisp machine at Osaka University [Yama81a], the evaluation of each argument is forked off onto a separate processor. As soon as all of a function's arguments have been evaluated that function gets scheduled on a free processor. The EM-3 machine [Yama83a] attempts even more. To trigger increased amounts of parallel evaluation, the EM-3 allows incomplete results to be sent forward. Thus, when all of the arguments of a function have become available, a pseudo-result is generated for that function and returned to its caller. Since some of those arguments could themselves be pseudo-results, the evaluation of a function keeps going in parallel with that of its arguments. Evaluation gets blocked when actual results, as against pseudo-results, are needed. Such an evaluation scheme clearly involves a complicated control mechanism. The complexity of this control mechanism can be reduced if the programmer provides hints by specifying the parts of the program where parallel argument evaluation can be safely performed. Several researchers hace suggested language extensions which enable the Lisp programmer to provide these hints [Hals84a]. We will be discussing these and other issues relating to parallel argument evaluation in Chapter 6.

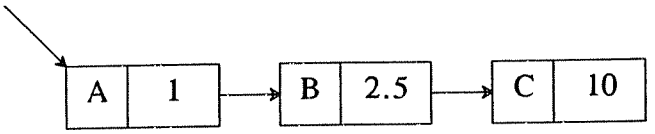
2.3.2. Maintaining the Environment

Recall that we refer to the collection of referencing contexts (made up of name-value binding pairs) in affect at a given time as the environment of Lisp execution. This environment must be modified on function call and on function return.

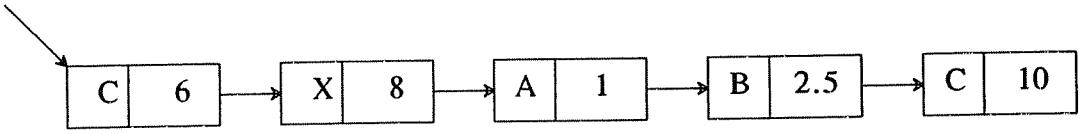
The most straight-forward way to implement the environment would be as a linear linked list of name-value pairs. New items get appended to the head of this *association list* on function calls, and deleted from the head of the list on function returns. Whenever a variable is referenced during the evaluation of a function the association list is searched (from the head) for the first, and hence most recently active, instance of that variable name. In the worst case, this variable lookup might involve scanning the entire association list. This implementation of the environment is called *deep binding*. Figure 2.3 illustrates how the association list changes over a function call. An alternative, called *shallow binding*, would be to maintain a table with one entry for each variable name, containing the current value binding of that name. The table is often called the *oblist* or *global symbol table*. Each variable name thus has a value cell (in the table) associated with it. Figure 2.4 illustrates how a shallow bound system maintains the environment over the same function call used in Figure 2.3. Shallow binding changes the name interrogation problem from one of list search to a simple table lookup. To maintain consistency, the table of value cells has to be modified on function call and return. Bindings that are in danger of being over-written by a newer value have to be saved (typically on a stack), to be used in restoring the table to its original state on function return. This modification procedure is more complicated than environment modifications in a simple deep bound association list, where all deletions and additions to the list take place at the head. In choosing one implementation over the other, there clearly is a trade-off to be made between fast function calling and fast variable lookup. Further, it is possible to conceive of a continuum of implementation schemes between deep and shallow binding. These schemes selectively cache portions of the environment in dealing with this trade-off.

The MIT Lisp Machine supports a shallow bound environment. Variable referencing is done through an *invisible pointer* to the value cell of the variable name. Invisible pointers form a special data type, distinct from ordinary pointers. A reference to a list cell containing an invisible pointer is *automatically* dereferenced by the hardware to the list cell pointed at by that invisible pointer. On a function call, some of these value cells get modified to reflect the new name-value bindings associated with the call. These values are stored on the control stack, from where, on function return, the modified value cells are restored to their original values. A similar scheme is used in the Symbolics 3600. As an optimization, there is a bit in each frame on the control stack that indicates whether or not there are value cell modifications associated with that call. This saves simple function calls from having to pay the shallow binding overhead.

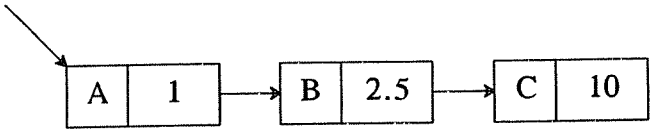
Most of the other machines that we surveyed support a deep bound implementation of environments. The penalties associated with variable lookup are reduced with the help of architectural support. For example, in [Deut78a] and [Deut73a] a caching scheme is used so that repeated references to the same variable in the same function cause only one expensive lookup to be made. Deutsch



(a) The Association List is a linked list of name-value binding pairs. On variable lookup the list is searched starting from its head. The environment shown has 'A' bound to the value 1, 'B' bound to 2.5, and 'C' bound to 10.

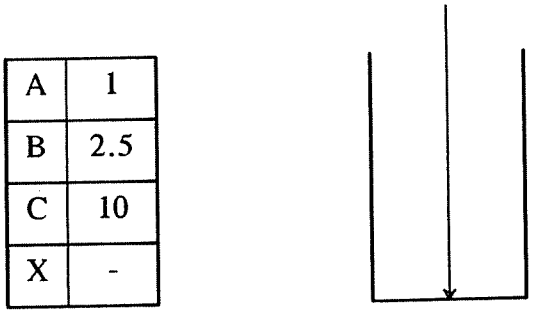


(b) When a function call takes place new bindings get added to the head of the association list. In this example, two new bindings are added, one for 'C' and one for 'X'. The current environment still has 'A' bound to 1 and 'B' bound to 2.5. 'C' is now bound to 6 and 'X' to 8.

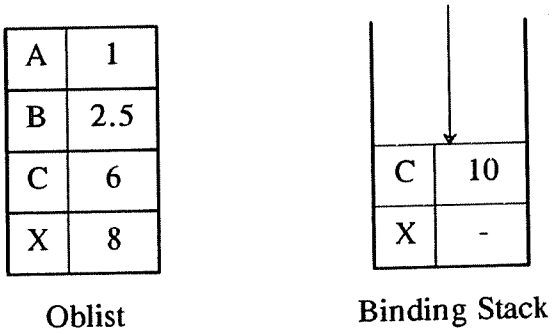


(c) On function return the bindings that were added on that function's call are removed from the head of the association list, leaving the environment in the same state that it was in before the call.

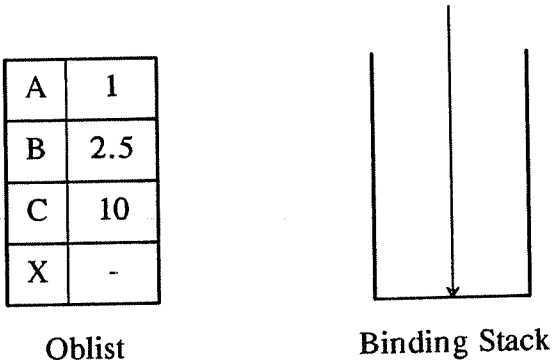
Figure 2.3. Deep Binding: Association List Modification over a Function Call.



(a) A shallow bound environment is maintained with a table of bindings (oblist) and a stack of old bindings for names that have been reused. This stack is initially empty.



(b) On a function call some oblist entries get updated to reflect new bindings. The old values of these bindings are pushed onto the stack. Variable lookup is simplified to oblist table lookup.



(c) On function return, the items are popped from the binding stack and used to update the oblist. The environment is thus modified to the state that it was in prior to the function call.

Figure 2.4. Shallow Binding: Oblist Modification over a Function Call.

estimated that a cost savings of as much as 80% results from this. He also describes a compiler optimization for references to variables that are bound at the top level and never re-bound. In a normal deep bound implementation, each such reference would involve a search of the entire association list, since the bindings made at the top level would be at the beginning of the association list. The compiler deals with such references by resorting to a shallow binding technique - using the value cell of the name directly. Unfortunately, this optimization cannot be used for interpreted code.

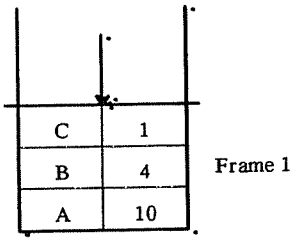
The FACOM Alpha [Haya83a, Akim85a] supports deep bound environments with a *value cache*. The value cache is an associative memory device that is searched before the association list during the lookup process. The association list is maintained in the frames of the control stack; the name-value bindings added to the environment on a function call are stored in the control stack frame associated with that function call. Each value cache entry is made up of a valid bit, a stack frame number (to identify which function call it belongs to), and fields for the variable name and value binding. When a function is called, the value cache is searched for the names of its formal arguments and local variable names referred to in it. Such cache entries are invalidated. Then, for each variable lookup that is a miss in the value cache, the usual lookup (in the association list) is done, after which the corresponding value cache entry is updated, and the value cache entry validated. On function return, the value cache is again searched, and all entries whose frame numbers are the same as that of the current function are invalidated. Figure 2.5 illustrates how using a value cache improves lookup time in system with deep bound environments.

Class P machines have added problems in maintaining the environment. The environment of each on-going evaluation must be available; instead of an association list, there is an association tree rooted at the global environment of the top level. A new branch gets added on each function evaluation. Since arguments get evaluated in parallel, these branches grow in parallel. To perform a variable lookup, a processor specifies the variable name and the head of the association tree branch corresponding to its environment. Class P machines typically use deep bound implementations of environments. An efficient method of specifying which branch of the association tree a reference belongs to is described in [Padg83a].

Though we have described Lisp as being dynamically scoped, several modern Lisps are lexically scoped. Examples include Scheme, T, Common Lisp and NIL. Under lexical scoping, a variable name is visible only in the lexical context of its binding, as in a language like Pascal. This modification simplifies the task of compiling Lisp functions, and removes the name look-up problem. Lexical scoping also raises language issues; the designers of Scheme argue against dynamic scoping since it violates *referential transparency*, i.e., the requirement that "the meanings of the parts of a program be apparent and not change, so that such meaning can be reliably depended upon" [Stee78a] through its global effect on the meaning of names.

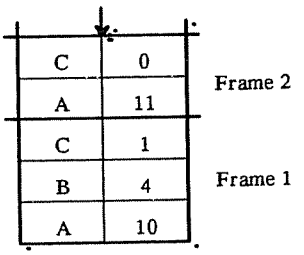
In summary, there are two main schemes for maintaining run-time referencing environments - deep binding and shallow binding. Deep bound environments allow for fast function calls and returns at the expense of potentially slow name interrogation, while shallow bound environments make name interrogation fast at the expense of slower function calls.

Name	Value	Valid?	Frame No.
A	10	Yes	1
B	4	Yes	1
C	1	Yes	1



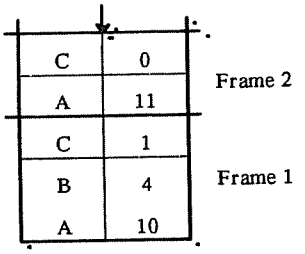
(a) The FACOM Alpha uses a Value Cache and an association list (maintained in the control stack) in it's optimized deep bound environment.

Name	Value	Valid?	Frame No.
A	10	No	1
B	4	Yes	1
C	1	No	1



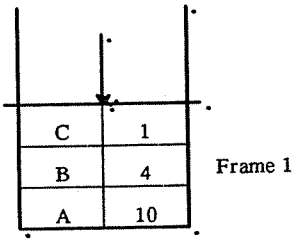
(b) If a function call with 'A' as formal argument and using 'C' as a local variable name is made, the value cache and stack are updated as shown.

Name	Value	Valid?	Frame No.
A	11	Yes	2
B	4	Yes	1
C	0	Yes	2



(c) When 'A' and 'C' are referenced in the function body, the a-list is searched for the latest binding. The Value Cache is then updated.

Name	Value	Valid?	Frame No.
A	10	No	2
B	4	Yes	1
C	1	No	2



(d) On function return a stack frame is popped and all Value Cache entries with that frame number are invalidated.

Figure 2.5. Facom Alpha Value Cache Operation.

2.3.3. Efficient List Representation

In early Lisps, lists were represented as linked lists of two-pointer list cells. Each of the two pointers was large enough to address all of memory. Figure 2.6 illustrates the two pointer list cell representation. This representation proves to be efficient as far as list accessing is concerned. With these two-pointer list cells, both the **car** and **cdr** primitives can be implemented as simple memory read operations, **rplaca** and **rplacd** as simple memory write operations, and **cons** as a list cell allocation followed by two memory write operations. However, one of the problems with this representation of lists can be seen when we try to ***traverse*** a list, i.e., access all the elements of a list. During a traversal, the address of the list cell to be accessed next is contained in one of the pointers of the cell that has just been accessed. The address can be forwarded to the memory system only after the previous access has been completed. This addressing bottleneck slows down list traversal; the two pointer list cell representation is considered time inefficient. Further, the representation is highly space inefficient; studies have shown that it is not necessary for both the ***car*** and ***cdr*** pointers to span the entire address space [Clar77a]. Lisp machines generally provide more compact representations of lists. We classify these compact list representation schemes as being either ***vector-coded*** or ***structure-coded.***

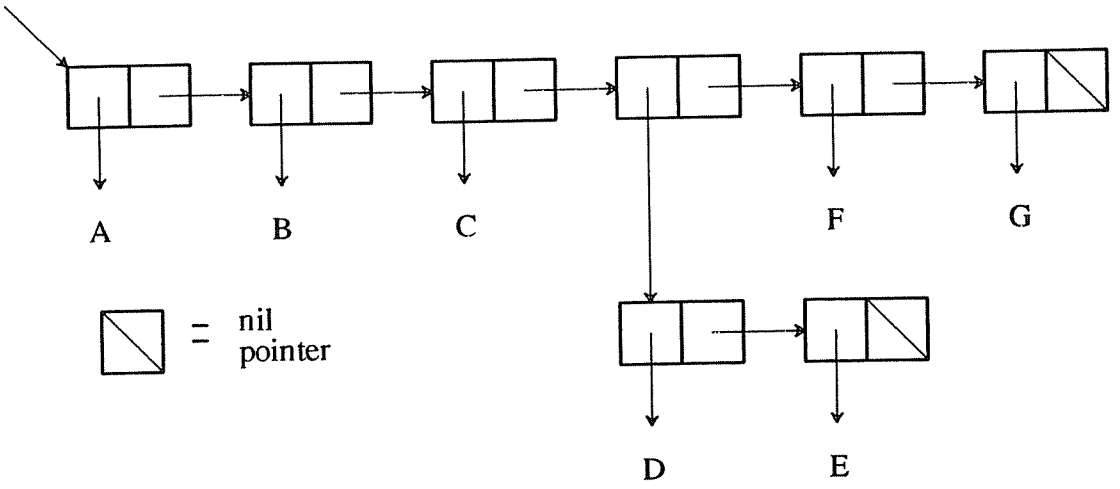


Figure 2.6. Two Pointer Cell representation of (A B C (D E) F G).

2.3.3.1. Vector-Coded Representations of Lists

The basic idea behind vector-coded list representation schemes is to represent *linear lists* as linear vectors of symbols. We call a list linear if none of its elements is itself a list. Under such a scheme, a list cell is represented by a vector element, with its *car* pointer assumed by default to be a pointer to a symbol, and its *cdr* pointer assumed by default to be a pointer to the next element in the vector. Pointers that differ from these default types are dealt with by providing for exception conditions. In traversing a list the address of the next list cell to be accessed is, by default, the next location in the vector. The address generation bottleneck appears only when the exception condition occurs. This kind of representation was first proposed in [Hans69a]. Two examples of this basic vector-coded representation are the *conc* representation [Kell80a] and the *linked vector* representation [Li85a].

The conc representation calls its vectors tuples. A tuple is a list of elements stored in contiguous memory locations. It is accessed through a descriptor which specifies the number of elements in the tuple, and a pointer to the beginning of the tuple. There are special tuples called *conc cells* whose elements are pointers to other conc cells or to tuples. Conc cells are used to implement list concatenation without having to modify the list structure; in concatenation of lists L1 and L2 in the two-pointer list cell representation, list concatenation involves modifying a pointer at the end of L1 to point to L2. In the conc representation the operation involves allocating a conc cell and setting its fields to L1 and L2.

The linked vector representation is another basic vector-coded representation. To take care of exception conditions, each vector element is tagged as either being a default cell or an indirection cell. Default cells contain list elements, while indirection cells contain pointers to elements in other vectors (or to nil). The last cell in a vector is assumed to be an indirection cell. Further, as an optimization to make frequent vector compactions unnecessary, each vector element can be tagged as *unused*. All of the information about a vector element can be encoded into a two bit tag, with the four tag field sequences being use to differentiate among the *cdr* being nil, the *cdr* starting from the next cell, the current cell being an indirection cell, and the current cell being unused. Figure 2.7 illustrates the linked vector representation.

Notice that vector-coded representation schemes are both space efficient and time efficient. The main problem with the basic vector-coded representation scheme follows from the fact that the unit of memory allocation is the vector. If fixed sized vectors are used, either internal fragmentation (if the vectors are too large) or an excessive number of indirection cells (if the vectors are too small) results. If, on the other hand, variable sized vectors are used, memory management becomes difficult. A separate free list must be maintained for each vector size, and on every vector allocation, a decision must be made as to which vector size should be allocated. The cdr-coding schemes that we describe next overcome this problem by allocating memory in units of vector elements.

The cdr-coding representation scheme is used in the MIT Lisp Machine, the Symbolics 3600, and many other Lisp machines. In it, list are represented using cdr-coded list cells, which are made up of a large *car* pointer and a small cdr-code. For example, in the MIT Lisp Machine, the two fields are 29 bits and 2 bits wide respectively. The four possible cdr-code bit sequences are called

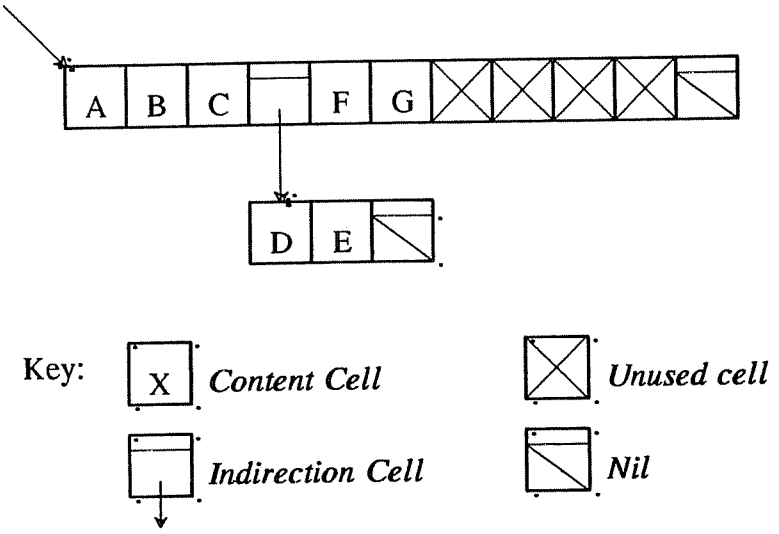


Figure 2.7. Linked Vector representation of (A B C (D E) F G).

cdr-normal, cdr-error, cdr-next, and cdr-nil. Cdr-next and cdr-nil provide an approximation to a vector-coded representation. A vector is made up of a set of contiguous cells whose cdr-codes are cdr-next, except for the last cell in the vector, which has a cdr-code of cdr-nil. The ***cdr*** of each list cell is simply the cell next to it. The cdr-normal code is provided for cases where such a vector representation is not possible. A cell with a cdr-code of cdr-normal is assumed to have its ***cdr*** pointer in the next cell. The ***car*** of such a cell is in its 29 bit ***car*** field. Its neighbouring cell will have a cdr-code of cdr-error. Thus, if a cell has a cdr-code of cdr-normal, then that cell and its neighbour resemble a normal two-pointer list cell. When lists get destructively modified (by **rplaca** and **rplacd** operators) during Lisp execution, the compact vector-coded parts of a list might have to be modified into less space-efficient structures, possibly using indirect pointers. Dereferencing such indirect pointers during list access involves extra memory activity. Invisible pointers can be used to reduce this indirection cost. Recall that an invisible pointer is automatically dereferenced by the hardware, thereby creating little or no overhead in the data structure reference. Figure 2.8 illustrates the MIT Lisp Machine cdr-coded representation.

The compact list representation scheme employed in [Deut78a] is also called cdr-coding. This scheme uses a 24 bit ***car*** field and an 8 bit cdr-code. A cdr-code of 0 means that the ***cdr*** is nil, while cdr-codes between 1 and 127 are interpreted as offsets to be added to the current list cell address to obtain the address of the ***cdr*** of that list cell. A cdr-code of 128 means that the ***cdr*** is located at the address specified in the ***car*** field of the list cell, and cdr-code values 129 to 255 represent the offset from the current list cell where the address of the ***cdr*** is

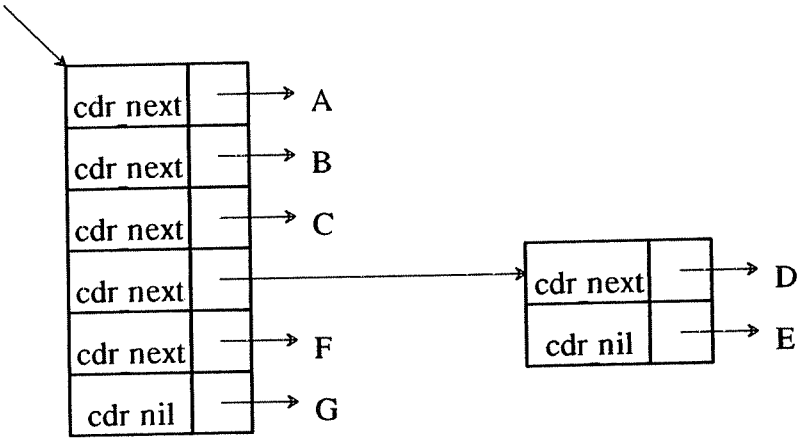


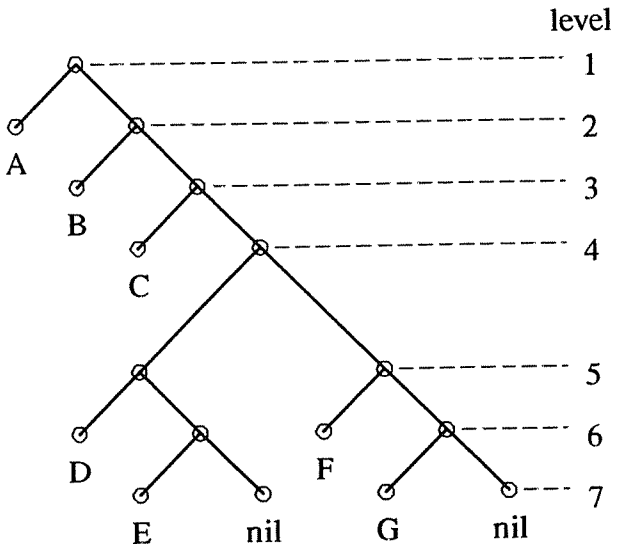
Figure 2.8. MIT Lisp Machine cdr-coded representation of (A B C (D E) F G).

located. This interpretation of cdr-codes was chosen largely from working set considerations, since the Lisp system operated in a paged virtual memory system with a page size of 256 words.

2.3.3.2. Structure-Coded Representations of Lists

The basic idea behind structure-coded list representation schemes is to attach to each list cell a tag that specifies the position of that cell in the list structure. In addition, list structures are stored on associative memory devices. The combination of detailed structural information at each node and associative search capabilities leads to the potential for fast list access and traversal. In list traversal, the address of a list element can be generated without having to look at the list cell that was accessed prior to it.

A scheme for representing binary trees on associative devices is described in [Mins73a]. Each node of the tree is tagged with a pair (l,k), where l is the depth of the node in the tree, and k is the maximal number of nodes at level l that could precede (l,k) in left to right order. Figure 2.9 illustrates this list representation. An extension of this node tagging scheme is suggested for use in the BLAST Lisp machine architecture[Sohi85a], where the pair (l,k) is compressed into a single node number, $N = 2^{l-1} + k$. A list can be mapped into such a tree with all the symbols in the list mapping into leaves in the tree. The symbolic information contained in the list is present in the leaf nodes of this tree, and the structural information of the list is described by the shape of the tree. It then becomes possible to represent a list compactly by remembering only its symbols, and tagging each with its node number from the corresponding tree; a list is thus



Node value	Minsky pair	BLAST number
A	(2,0)	2
B	(3,2)	6
C	(4,6)	14
D	(6,28)	60
E	(7,58)	122
F	(6,30)	62
G	(7,60)	126

Figure 2.9. Tree Coded representation of (A B C (D E) F G).

encoded as a set of *(node_number,symbol)* tuples. In BLAST, these tuples are stored in tables called *exception tables.* An associative searching capability on such tables is useful in implementing list manipulation operations efficiently.

Two schemes that achieve a similar encoding are described in [Pott83a]. One of these, called *CDAR coding,* tags each symbol in a list with a string of 0's and 1's. This string specifies the series of **car** (represented by 0's in the string)

and **cdr** (represented by 1's in the string) operations that when applied to the list yield that symbol as the result. This is equivalent to the node number used in BLAST; it specifies the position of the symbol in the list, and can be used to make list access operations efficient. The other encoding scheme, called the *explicit parenthesis storage (EPS) representation*, tags each symbol with 3 pieces of information - the number of left parentheses in the list preceding the symbol, the number of right parentheses in the list preceding and immediately following the symbol, and the position of the symbol in the list. Figure 2.10 illustrates the

(A B C (D E) F G)

		← EPS Representation →		
Node Value	CDAR Code	Left	Right	Position
A	000000	1	0	1
B	000001	1	0	2
C	000011	1	0	3
D	000111	2	0	4
E	010111	2	1	5
F	001111	2	1	6
G	011111	2	2	7

Key: *Left* : *Number of left parentheses in list to left of atom*

Right : *Number of right parentheses in list to left of and immediately following atom*

Position *Position of atom in list*

Figure 2.10. CDAR coded and EPS representations of (A B C (D E) F G).

CDAR coded and EPS list representations.

2.3.3.3. Summary

The traditional way to represent lists is using two-pointer list cells. The major drawback with this scheme is that the information needed to address a list cell is entirely contained in another list cell. Because of this, in performing a list access, the processor must wait for one memory access to complete before the next one can be initiated. We have seen two classes of more complicated list representation schemes - vector-coded and structure-coded. Simple linear lists can be represented compactly and accessed efficiently using vector-coded representation schemes. Unfortunately, all lists are not linear, and vector-coded representation schemes include exception conditions to take care of more complex list structures. In such schemes, the information needed to address a list cell is still partially contained in another list cell. The Lisp machines that we surveyed used either a naive two-pointer list representation or a vector-coded list representation (typically cdr-coding). Structure-coded representation schemes make it possible to address the elements of a list independently by attaching to each list cell a tag that describes its position in the list. There is no clear consensus on which type of list representation scheme is preferable. A major factor in making this determination would be how typical Lisp programs access lists. For instance, if lists are accessed in a very poorly structured, or random, manner then structure-coded representation schemes would be preferred; under such a list representation scheme, the various elements of a list can be accessed in the same amount of time, regardless of their position in the list structure.

2.3.4. Heap Maintenance

Finally, we turn our attention to heap maintenance strategies. By and large, Lisp machines are tagged architectures. Tags are used to specify data types; type checking in a dynamic language like Lisp in which there are no type declarations is greatly facilitated by this. Tags also make it possible to distinguish between list pointers and list data, which turns out to be useful in managing the heap space. Recall that in Lisp, it is the system's responsibility to manage the allocation and reclamation of list cells in the heap, since the user does not explicitly deallocate list objects. Since the reclamation of these ***garbage cells*** is its most complicated part, heap maintenance has come to be largely identified with ***garbage collection.*** Several basic garbage collection schemes are described in [Cohe81a]. Clearly, the goal of garbage collectors is to reclaim garbage quickly and with a low overhead.

Garbage collection is a two stage process. We call the two stages ***garbage detection*** and ***garbage reclamation.*** The garbage detection stage involves identifying which heap memory cells are no longer referred to, after which the garbage is treated and made ready for reuse in the garbage reclamation stage. Garbage reclamation often only involves adding the newly recognised garbage cell to a list of free list cells available for re-allocation. In systems where lists are represented using compact representations, reclamation might also involve compacting the parts of the heap that are in use. Garbage detection can be done in two ways, either by using ***reference counts*** or by ***marking.***

In marking [Scho67a], all accessible list cells are marked starting with a set of root cells that are known to be non-garbage cells, following the pointers contained within them to other list cells, and so on. All list cells that are not marked at the end of this operation are not accessible and are hence reclaimable as garbage. Marking involves an overhead of at least one bit per cell to be used for the mark. In marking schemes, garbage collecting is initiated only when there is no more heap space available and a request for more space is made. So, the overhead of heap management need be felt only when there is no more heap space.

In reference counting [Coll60a], a count is maintained for each heap cell of the number of extant pointers to it. A cell is known to be garbage when its reference count goes to zero. Reference counting has several drawbacks. One problem is the actual size of the reference count field. It would seem that each reference count field would have to be large enough to hold the total number of list cells in the heap. Another problem is that heap users pay the price of garbage detection continuously in the form of a space overhead (a reference count field for every list cell), and a time overhead (the updating of these counts on heap accesses). Further, reference counting has the disadvantage of not, in general, being able to reclaim circular lists. Also, reference counting can lead to poor real time performance. Consider what happens when the reference count of a list cell goes to zero. Before the list cell can be added to a free list of cells, the reference counts of the two list cells that it points at are decremented by one. This could cause the reference counts of these two cells to go to zero, in which case they would have to be reclaimed, with the reference counts of their descendents being decremented by one. Thus, the seemingly simple operation of reclaiming a list cell could initiate an arbitrarily large amount of reference count updating and list cell reclamation.

Several variations to these two basic schemes have been developed. Some variations use a combination of the two schemes [Deut76a]. Others use clever marking schemes to enable marking to be done in parallel with the useful Lisp evaluation [Dijk78a, Stee75a, Ram85a]. The two parallel processes are commonly referred to as the ***collector*** and the ***mutator***. These schemes work by using more than one bit to mark each list cell. These bits are used to implement a form of mutual exclusion of access to the list cells by the collector and the mutator. Another approach [Bake78a, Feni69a] divides the heap space into two semispaces, the "oldspace" and the "newspace", either one of which will be undergoing garbage collection. The two semispaces are simultaneously active but the garbage collector tries to relocate, or copy, accessible cells from the "oldspace" to the "newspace". When that job has been completed the two semispaces get flipped. These schemes are called ***copying garbage collectors***. They can be made to work in real-time by performing a fixed number of relocations on every heap allocation request, yielding copying, incremental garbage collectors.

The MIT Lisp Machine supports Baker's real-time, copying, incremental garbage collection scheme [Bake78a]. Hardware features help in making this feasible. For instance, since a tagged memory is used, it is possible to detect whether or not a list cell contains a pointer by examining its tag. Further, memory is organised as ***areas***, where each area contains related list cells. The user can declare an area to be static and save the garbage collector the task of processing that area.

The Symbolics 3600 uses a similar garbage collection algorithm with additional hardware support. Each memory page has an associated *page-tag* which indicates whether or not that page contains pointers to memory areas that are of interest to the garbage collector. These tags help in reducing the work of the garbage collector. Unlike the MIT Lisp Machine, where static areas were declared by the user, in the Symbolics 3600, areas are identified as static (unlikely to become garbage), ephemeral (assumed to contain cells that are likely to become garbage soon), or dynamic (assumed to contain cells of intermediate lifetimes) based on how much garbage collection activity they have required in the past. The virtual memory software maintains this information, as well as tables with other information useful to the garbage collector. For example, it maintains a table identifying the swapped out pages that contain pointers to ephemeral areas. Since the garbage collector concentrates its attention on these ephemeral areas, this table will help in reducing the amount of paging activity due to garbage collection.

Despite its drawbacks, reference counting has also been incorporated into Lisp machine garbage collectors. The Machine for Lisp Like Languages, M3L, Project [Sans82a] uses a 3 bit reference count field. This count does not include pointers that are on the run-time stack or in registers. The paper reports studies which suggest that this reference count suffices to reclaim about 98% of all inaccessible list cells. If stack and register pointers are included, reference counts would grow in proportion to the number of function calls, due to the fact that arguments are passed on the stack. A separate 1 bit reference flag is therefore maintained for each cell to indicate whether or not that cell is referred to by pointers on the stack and in registers. This has the unfortunate consequence that the reference flag might have to be updated on each stack operation. On a stack pop that involves a stack item containing a pointer to a list cell, the entire stack must be checked to see if there are any other references to that list cell in order to determine whether or not to modify the list's reference flag.

The Facom Alpha also uses a reference counting garbage collection scheme [Haya83a]. The Alpha memory is organised as a number of sub-spaces. Reference counts are used to reclaim sub-spaces, not list cells. So, there is one reference count for each sub-space. To keep down paging activity due to garbage collection, a table of reference count updates to sub-spaces that are currently swapped out is maintained. Those reference count modifications are performed when the page gets swapped back in. The reference count of a sub-space counts only the pointers to cells in that sub-space that originate from other sub-spaces. By not counting pointers that are in the same sub-space, it becomes possible to reclaim those circular lists that are entirely contained in a single sub-space. In each sub-space, a free list of list cells is maintained. A marking scheme is used to detect garbage cells in sub-spaces. The marking process starts from the pointers on the stack; these pointers were not included in the reference counts of sub-spaces, and can therefore be used as root pointers in the marking process. Marking is initiated either when the number of free cells in a sub-space goes below a threshold value, or when the processor is otherwise idle.

In summary, there are two classes of techniques for dealing with garbage - maintaining reference counts and using mark-and-sweep algorithms. Most machines provide for some form of mark and sweep garbage collection. We include copying garbage collectors in this class. Different machines support

variants of this basic technique so that the time spent in garbage collection is amortized over the entire run of the program. A few machines have also incorporated reference counting techniques into a basic mark and sweep strategy.

2.4. Summary

A survey of Lisp machines reveals that the designers had, by and large, four issues in mind in arriving at architectural support for Lisp: fast function calls, environment maintenance, efficient list representation, and heap maintenance.

We do not see much variety in the architectural support for function calling and environment maintenance. Most machines provide a small set of call instructions to support the most frequent forms of function calls. They use either a shallow bound implementation of environments or a deep bound implementation. Some machines include a scheme for caching name-value bindings in order to make name lookups faster. The interesting research in these two areas seems to be in extensions to concurrent execution and arriving at new evaluation paradigms.

All of the machines that we surveyed provide some degree of support for heap maintenance. Commercial machines generally provide support for real-time copying incremental garbage collectors. Hybrid reference counting and mark-and-sweep schemes have also been used.

How should lists be represented internally? The simple list representation scheme, with the two-pointer list cell, is both space inefficient and time inefficient in terms of the time required to traverse a list. We classified list representation schemes as either *vector-coded* or *structure-coded*; vector-coded representation schemes aim for space efficiency, while structure-coded schemes aim for access efficiency. Which representation scheme is best? The answer to this question will depend on what lists look like typically, and how they are accessed. For instance, if most lists are simple linear lists that do not get modified much then vector-coded representation schemes would be preferable. If they are not, but happen to be accessed in well structured ways, then structure-coded list representation schemes can make efficient list access possible; recall that list traversal is more efficient in structure-coded lists than in vector-coded lists. We investigate these issues further in the next chapter.

Chapter 3

Lisp Lists and Their Manipulation

3.1. Introduction

In this chapter we will seek to answer the questions relating to Lisp lists that were raised by the survey of Chapter 2. These questions fall into two broad categories: those pertaining to how lists are best represented, and those about the nature of detectable patterns in Lisp list access streams.

List representation is interesting since a given s-expression could be represented in several different ways, each representation scheme with its advantages and disadvantages. We define a representation scheme to be **uniform** if it provides a unique representation for every s-expression. A non-uniform representation scheme typically encodes structural information about a list in tags to minimize the amount of space occupied by the list, at the expense of slower accessing speeds. One representation scheme is more uniform than another if it provides for fewer such exception cases. For example, the two-pointer list cell provides a uniform representation (with no exception conditions), but is space-inefficient. The cdr-coded list cell, on the other hand, is space-efficient but less uniform. Any representation scheme can be evaluated in terms of (a) how efficiently it makes use of limited list memory space, and (b) how well it supports list manipulation primitives. We will see that most Lisp list studies performed to date provide data useful for an evaluation of list representation.

More important, however, from an architectural viewpoint, is an analysis of Lisp list access streams. We need to determine whether these access streams contain access patterns, and if they do, we need to study their exact nature in more detail. For example, if there is temporal locality of reference to the list structure we might expect access patterns to take the form of repeated or periodic accesses to portions of the list structure. Such locality of reference can be exhibited by appropriate architectural support, analogous to the way vector processors take advantage of patterns in array accessing. In order to design such support we need a better understanding of the nature of the Lisp list access stream.

3.2. Past Studies: At the List Cell Level

The main body of work on studying Lisp lists was performed by Clark [Clar77a, Clar79a]. In the first paper [Clar77a], Clark describes a series of "static" Lisp list studies. The studies are called "static" since they examine the list structure that survives program execution, and not the "dynamic" changes to the list structure that take place *during* program execution. These "dynamic" properties of lists are the subject of the second paper [Clar79a]. Other studies in the area [Deut78a, Fode81a, Olss83a, Pond83a, Smit85a] have been aimed at specific Lisp systems, system features, or programs and do not yield sufficiently general insight into the nature of Lisp list manipulation to justify describing them here at length. They will, however, be cited wherever relevant.

3.2.1. Clark's Static Studies

Five large Lisp programs with substantial list activity were used in this study. A typical run of each program was done and the lists extant at the end of the run (either bound to an atom at the top level or in an atom's property list) were analysed. This technique was used in 2 different kinds of investigations.

In the first, statistics about the values of the list cell pointers were accumulated to suggest guidelines for efficient alternative list representation schemes. For instance, Clark found that *cdr* pointers rarely point at atoms; they point mostly at list and *nil*, in the ratio 3:1. *Car* pointers on the other hand rarely point at *nil*, pointing mainly at atoms and lists, in the ratio 3:1. Also, few list cells are pointed at more than once, indicating that there is not much substructure sharing in Lisp lists. Further, pointers tend to point at list cells that are a small distance away. This suggests that there is considerable scope for space saving through compact pointer encoding schemes like *cdr*-coding *hash linking*[Bobr75a] and *offset addressing*[Bobr79a]. In [Bobr79a] these results are used in an analysis of alternative compact list encoding schemes. The difference in content of car and cdr pointers also led Clark to investigate the advantages of *linearizing* lists. Linearization is a procedure that relocates list cells so that list cell pointers typically point at neighbouring list cells. This could be done either in the car or in the cdr direction; in a list that has been linearized in the cdr direction most *cdr* pointers point at their neighbours.

In the second set of static experiments Clark evaluated list linearization and the effectiveness of various algorithms implementing the **cons** primitive. He ran his benchmarks on interpreters that used different **cons** algorithms and used the state of the list memory after program termination as a measure of the effectiveness of these algorithms in linearizing lists. He observed that a naive **cons** algorithm performed almost as well as a more clever one in keeping pointer distances small, indicating that this is an inherent feature of Lisp list behaviour and not of the efficacy of a particular **cons** algorithm.

3.2.2. Clark's Dynamic Studies

In [Clar79a], Clark reports results from his "dynamic" studies, which were conducted during short runs of 3 of the 5 programs used in the "static" study. On every list cell access, the instruction doing the access, the list cell address, the values (addresses) of the *car* and *cdr* pointers of that list cell, and (if it was a modify operation) the replacement value (an address), were written into a trace file which was later analysed.

Some of these "dynamic" observations roughly confirm the "static" findings on pointer data type frequencies and list pointer distances. Among Clark's other dynamic observations: **car** and **cdr** account for the vast majority of all primitives executed, with **cdr** being slightly more common. Recall the static observation that pointer distances are, on the average, small. These two observations suggest that there is considerable spatial locality of reference in Lisp list access streams, something confirmed in the data cache simulations of [Smit85a] where the hit ratios for Lisp workloads were as high as those for any other workload. Clark found that among the other Lisp primitives, **cons** is the most frequently occurring. He also studied the locality of list cell reference using the LRU stack

model. 20-30% of all references were found to be to the most recently accessed cell, with about 50% to one of the 10 most recently accessed, and about 80% to one of the 100 most recently accessed cells. This suggests considerable temporal locality of reference in list access streams. Finally, in studies on the persistence of linearization Clark found that once a list was linearized it tended to stay fairly well linearized. As a consequence, long-lived lists were typically well linearized; their list pointer distances were either one or small.

3.2.3. Implications of Clark's Studies

Clark's studies into Lisp list behaviour provide at least three interesting observations.

(1) The *car* and *cdr* pointers of list cells get treated quite differently.

(2) Lisp programs show both spatial and temporal locality in access reference.

(3) Once they have been linearized, lists tend to stay that way. Old lists do not change much.

The first of these observations has been widely recognized; in Chapter 2 we saw that compact list representation techniques, like *cdr*-coding, are commonly employed in Lisp systems. The other two observations have not received as much attention.

We believe that Clark's "dynamic" studies were *not dynamic enough.* Dynamic studies are preferable to static studies; they capture a better picture of the system being studied by frequently sampling the behaviour of a system during program execution. Static studies, on the other hand, capture only summary information, which is often not truly representative of system behaviour. Clark's "dynamic" studies were performed on very small simulated runs of the programs. This was necessary for two reasons. First, the experimental setup was based on a simulator whose speed made longer runs difficult. Second, garbage collection could not be allowed during experiments since lists were studied in terms of their physical addresses, which could change across garbage collections. Further, even though the statistics were gathered dynamically, they were effectively averaged out over time. No temporally local patterns in access were studied; neighbouring list access events were not studied relative to one another other than in the limited sense of pointer distances, which is related to spatial locality. Such access patterns might be significant and should therefore be investigated more carefully.

Our own studies of Lisp lists, described next, attempt to answer some of the unanswered questions about patterns in Lisp list access streams.

3.3. Our Studies: At the Data Structure Level

We chose to conduct our studies at what we call the *list data structure level.* At this level we are more concerned with the high level consequences of what sequences of list operations do to the list structure than with what happens to individual list cells. Rather than dealing with lists in terms of physical list cell addresses as Clark did, we attempted a higher level study, tracing the history of

list access and modification at the s-expression level.

The investigation was carried out in two stages. In stage 1 we repeated some of Clark's experiments in order to calibrate our benchmark programs with his as well as to verify his results with observations made from a different perspective. In stage 2 we went on to investigate patterns in the access streams.

3.3.1. Stage 1: Benchmark Characterization

We know of no set of programs that can be described as "standard Lisp benchmarks" particularly for a study as far removed from the hardware level as ours. Lisp system benchmarking is a fine art. It could be conducted at several levels, ranging from the machine level to the Lisp instruction level. The reader is referred to [Gabr82a, Gabr85a] for guidelines on benchmark selection. Our benchmark program selection problem is simplified by our not being concerned with overall system performance, but only with list manipulation operations.

We used a suite of Lisp programs including a PLA generator (PLAGEN), a circuit simulator (SLANG), a VLSI design rule checker (LYRA), an editor (EDITOR), and PEARL (Package for Efficient Access to Representations in Lisp). The programs were run on a Franz Lisp interpreter modified such that on the call of a list access or modify function, the function name and its arguments (in s-expression form) were written to a trace file which was later analysed. This trace was not intended to capture all list accesses; if it did, we, like Clark, would have had to confine our studies to very short runs. To ensure that the traces are representative of the true list accessing behaviour of the five programs, they were generated using realistic inputs to those programs. PLAGEN was used to generate a PLA for a traffic light controller [Mead80a]. The circuit simulator, SLANG, was used to simulate the behaviour of a BCD to decimal convertor as well as another simple Boolean function. The VLSI design rules checker, LYRA, was used to conduct CMOS design rules checks on a portion of an 8 bit multiplier. The program that we call EDITOR is actually the Interlisp TTY function editor. We used it to perform an editing script on one of the editing functions; this script included performing global substitutions, searches, modifications, as well as the tutorial session described in the Interlisp Reference Manual [Teit75a]. Finally, PEARL was used to construct a small database management system and perform lookup and update operations on it. The length of the 5 traces (i.e. the number of list primitive calls made in the run) were as follows: PLAGEN (59,967), SLANG (19,846), LYRA (252,951), EDITOR (33,790), and PEARL (1,572).

We used a simple parameter to place our programs in perspective with regard to Clark's Lisp programs: the distribution of Lisp list primitives in each execution trace. The primitive function frequencies are presented in Figure 3.1 for our five benchmarks along with those from three of Clark's programs: NOAH, CONGEN and PARSER [Clar79a]. The figure is a histogram of the percentage of all traced functions that were **car** (lowest portion of each bar), **cdr** (middle portion), or **cons** (uppermost portion); the other primitives together covered less than 10% of all functions traced and were not plotted.

While the PLAGEN, LYRA and EDITOR runs follow Clark's programs in having a predominance of access primitives, the PEARL and SLANG runs show

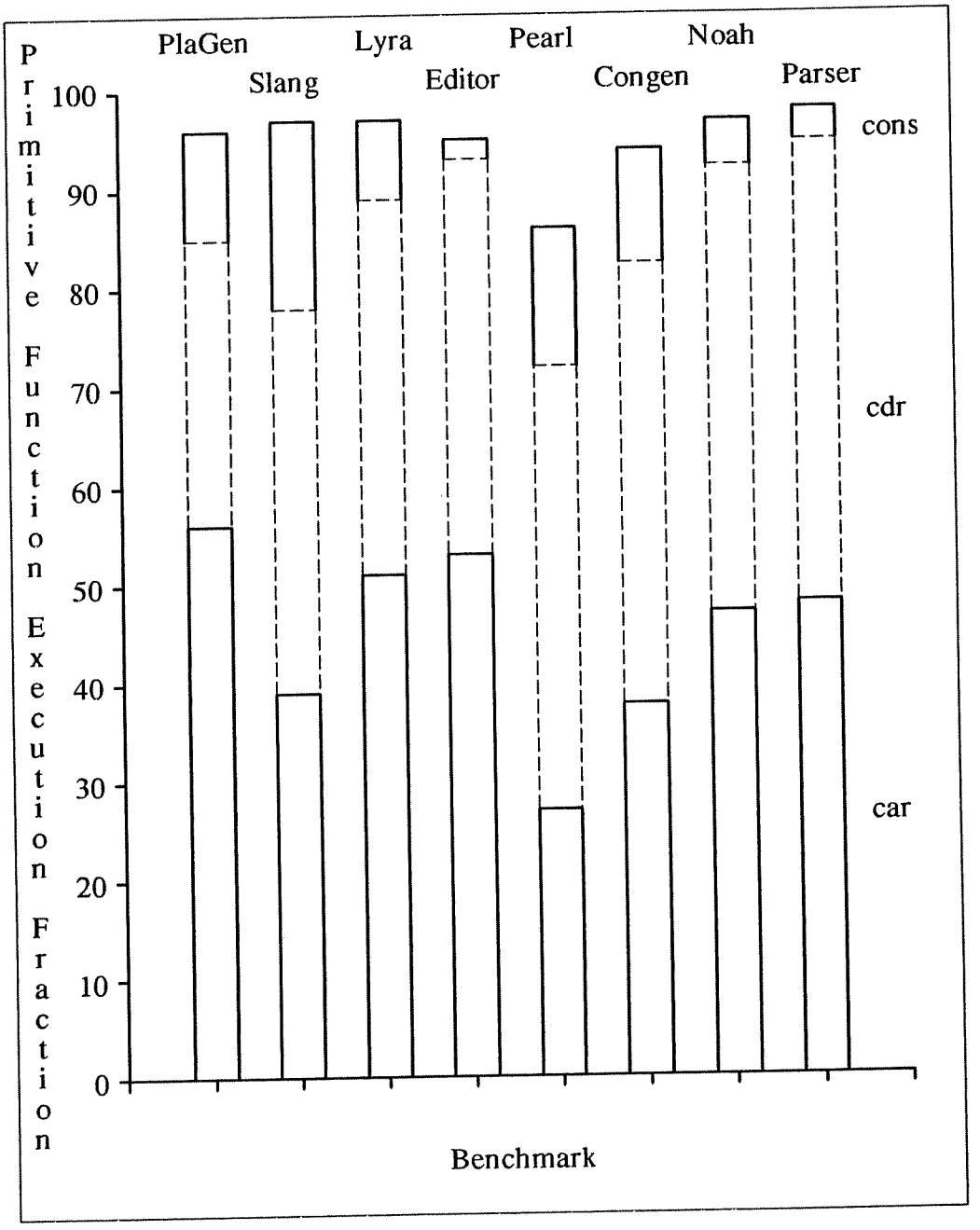


Figure 3.1. Execution Frequencies of Primitive Lisp Functions.

different characteristics, with SLANG having a higher **cons**, and PEARL a higher **rplaca/rplacd** percentage than any of the other programs. So, some of the members of our program suite behave like Clark's programs did while others show different characteristics. It would appear, then, that our programs cover a wider range of Lisp program behaviour than Clark's did.

As a further checkpoint we compared the complexity of the list structure operated on by our program suite with that in Clark's programs. Clark characterized lists in terms of the percentages of all car and cdr pointers that pointed at lists, or symbols, or *nil*, etc. We used simpler measures: for each list encountered we noted **n**, the number of symbols in a list, and **p**, the number of internal parenthesis pairs in the list. These measures were chosen since they compactly summarize information describing the complexity of a list. The metric **p** is a measure of the ***structure*** of a given list; we loosely define a simple linear list as being unstructured, while a list the depth of whose tree representation is far less than the list's length is considered as highly structured. Further, the sum **n+p** is proportional to the amount of space required to represent the list using 2-pointer list cells or cdr-coded list cells. Using a compact structure-coded list representation only **n** cells are necessary to represent a list. So, these two statistics can be used in evaluating alternative list representation schemes.

The two examples in Figure 3.2 illustrate this. It takes **n+p** 2-pointer list cells (or cdr-coded list cells) to represent a list with **n** symbols and **p** internal parenthesis pairs. For a more compact representation scheme like cdar-coding the amount of space required is proportional to **n**.

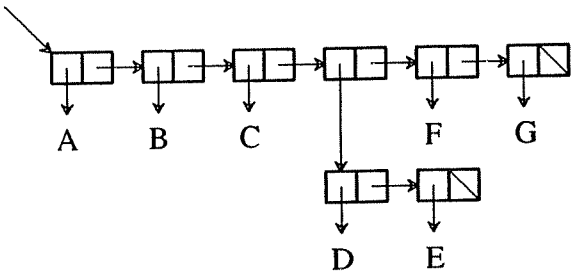
Table 3.1 shows the average **n** and **p** values for each of the 5 runs. Notice that in 4 of the 5 runs the average value of **p** is less than 3. This corresponds to lists of, on the average, simple structure. The EDITOR run deals with more complex lists. The **n** values average out to about 10, indicating that lists were not too long other than in the EDITOR run. These observations are along the same lines as Clark's, which led him to speculate that lists on the average are not long. Figures 3.3a and 3.3b give a better idea of the distributions of the observed **n** and **p** values. We will be using these distributions in Chapter 5 to construct a typical list access stream for our simulation.

3.3.2. Stage 2: Locality of Reference in Lisp List Access Streams

From the studies surveyed in Section 3.2 it appears that there is both spatial and temporal locality in Lisp list access. In this section we will investigate this locality of reference in an ***implementation and representation independent*** manner. This is a major departure from the work of Clark; he studied list pointer distances (which are highly representation dependent), on an Interlisp system (making the study implementation dependent). We begin this section, therefore, by describing our implementation and representation independent vantage point.

(A B C (D E) F G)

$n = 7, p = 1$



Node	Code
A	000000
B	000001
C	000011
D	000111
E	010111
F	001111
G	011111

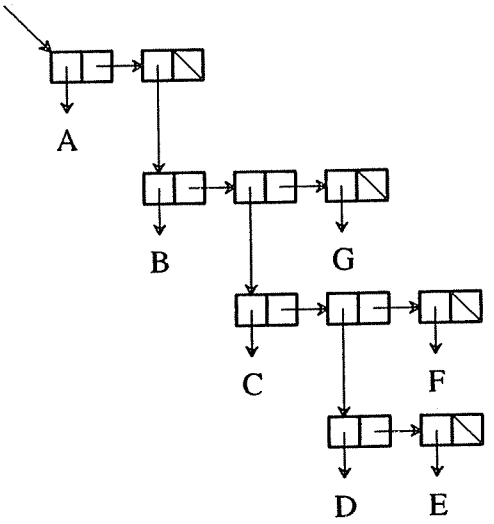
8 two-pointer list cells

7 cdar codes

(a)

(A (B (C (D E) F) G))

$n = 7, p = 3$



Node	Code
A	000000
B	000001
C	000011
D	000111
E	101111
F	011111
G	000111

10 two-pointer list cells

7 cdar codes

(b)

Figure 3.2. Significance of **n** and **p**.

Table 3.1. Average Values of **n** and **p**.

Benchmark	n	p
SLANG	10.04	1.99
PLAGEN	12.40	2.90
LYRA	9.70	1.55
EDITOR	74.74	20.98
PEARL	13.98	2.79

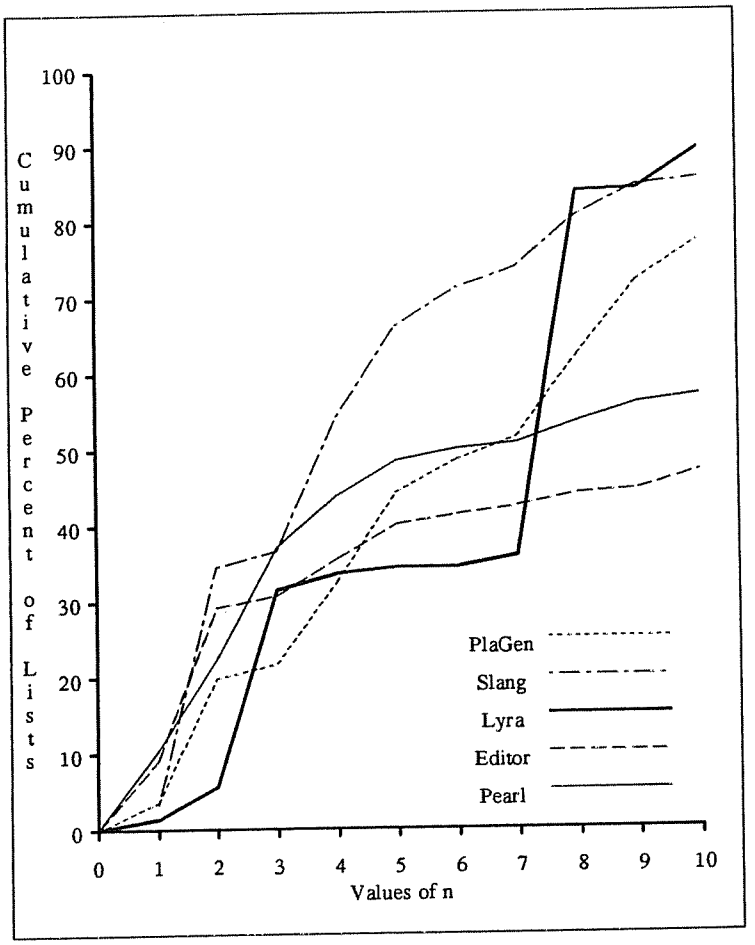


Figure 3.3a. Distribution of **n** over Lists.

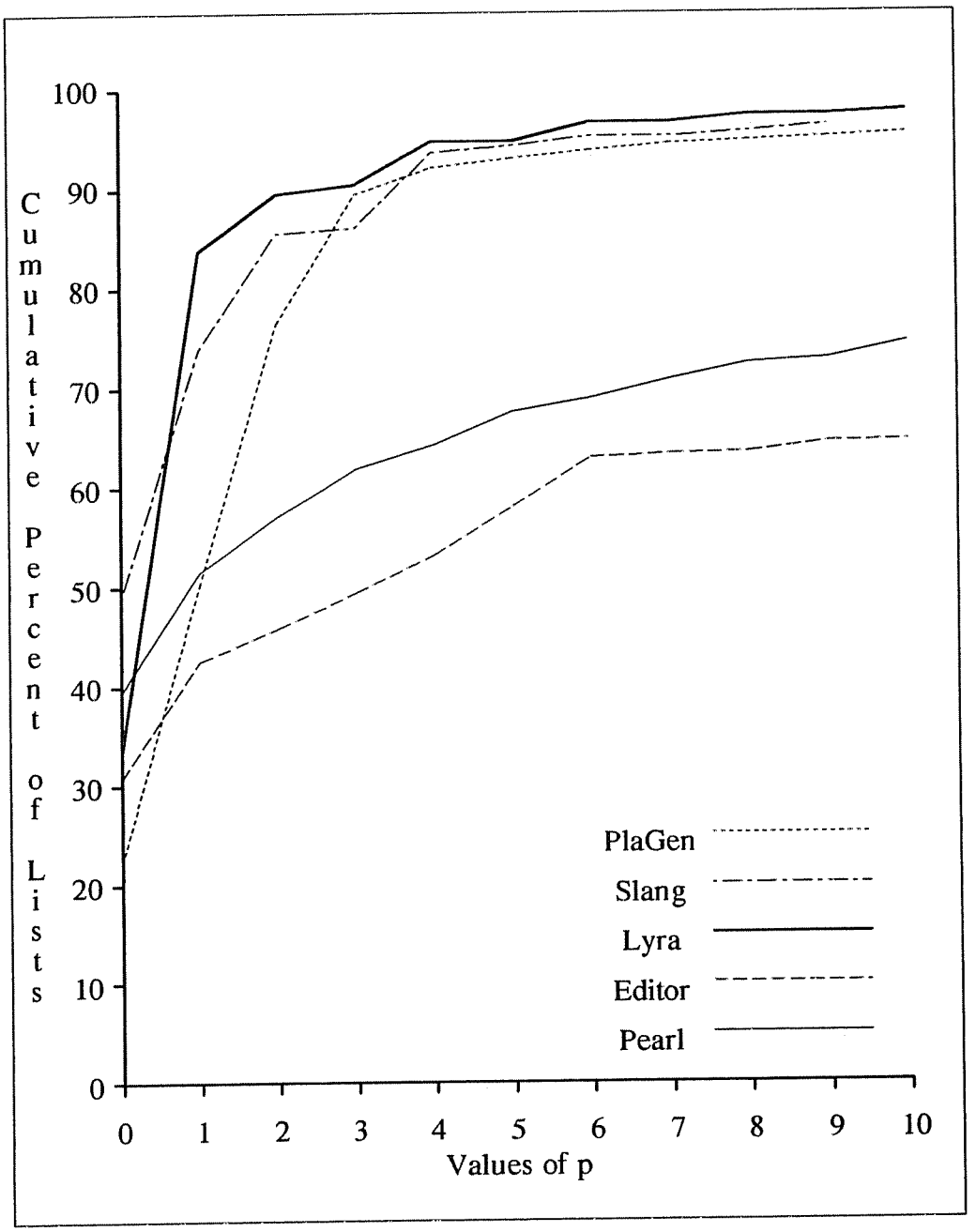


Figure 3.3b. Distribution of **p** over Lists.

3.3.2.1. Structural Locality and List Sets

A Lisp trace shows spatial locality because it accesses different parts of a list whose elements are close together in memory. This locality could probably be better described as structural locality of reference. So, for example, stepping through a cdr-coded linear list would display the same spatial locality as a DO loop stepping through a one dimensional array by sequentially accessing its elements. In dealing with arrays, by storing the elements of an array close together in memory, we can take advantage of the structural locality shown by array access streams. Structural locality is manifested as spatial locality of reference which can be exploited by simple architectural support. If the elements of a list are not stored close together in memory, then even if a Lisp program is very local in its referencing behaviour (in that it references only a small set of lists most of the time), a study like Clark's would detect little locality of reference. Clark conjectures that the locality he observed was caused not by clever space allocation in the memory management algorithms, but by an inherent property of how list structures get constructed in Lisp computation. Our study is far enough removed from the representation level to make it possible to evaluate this from a data structure viewpoint.

The concept of structural locality needs a little more elaboration. To this end we extend the concept of spatial locality to the list s-expression level. We say that two list references are ***related*** if one is the car or cdr of the other. A list access reference stream can then be partitioned into ***list sets***, where each list set is a closure of related list references with the added constraint that no two temporally adjacent members of the list set are separated in the access trace by more than 10% of the total length of the trace. We further define the ***lifetime*** of a list set as the distance between the temporally first and last members of that list set.

The list set concept is intended as a representation-independent means of detecting the scope for spatial locality of reference at the s-expression level. In very general terms, it seeks to divide the list structure manipulated by a Lisp program into array-like domains of high spatial locality of reference. A list set partition of an access stream is, in some sense, a working set partition of the entire list structure. The 10% separation constraint is included in the list set definition to preclude list sets from having long lifetimes but few members; such a trivial list set would not represent spatial locality that could be exploited by architectural support. If there are several trivial lists sets in the partition of an access stream, then that partition describes spatial locality of reference that is not temporally important, and hence not of architectural significance. The separation constraint thus serves to ensure that the lifetime of a list set is the period during which that list set is being actively accessed. We chose 10% as a compromise between a larger fraction, which would result in more trivial list sets, and a smaller fraction, which would result in large list sets being split up into several smaller list sets.

An analysis of the list set partition of an access trace can yield insight into the nature of referential locality contained in the trace. Note, however, that a list set partition of a reference stream cannot make a strong statement about the scope for sub-structure sharing among the lists accessed. Each member of a list set is a list reference, not a list. Two list references could be mistaken for each other if they were made to identical lists. This does not overly concern us since, as we have seen in Section 3.2.1, Clark's studies suggest that typically there is little

sub-structure sharing in Lisp programs.

3.3.2.2. Locales of Reference in the List Structure

Figure 3.4 shows how the list access stream for each of the five program runs was partitioned into list sets. The graph is a plot of the cumulative number of list sets against the cumulative percentage of all list references that they contain. It indicates the number of list sets in the five access streams. In interpreting Figure 3.4 we define the size of a list set as the number of list references that it encompasses. The inverse-exponential nature of the plots indicates that there are few large list sets and several small list sets in the list set partition of each access stream. Further, notice that it took only as few as 10 list sets to cover as many as 80% of all list references encountered. Our partitioning procedure has thus determined that a small number (about 10) of significant structural locales of reference represent a large percentage (about 80%) of all the list references in each trace.

Our next concern was the lifetime of these list sets. Figure 3.5 shows how the lifetimes of the list sets varied for the 5 traces and is a cumulative plot of the number of list sets against list set lifetime, where list set lifetime is expressed as a percentage of the total length of the reference stream. There is a wide variation of behaviour over the programs. In the PLAGEN and LYRA traces the rapid rise from the origin indicates that a large percentage (70-90%) of all list sets are short lived, surviving less than 10% of the total program run. Further, from Figure 3.5 it appears that in general there are few long-lived list sets. In all five traces, less than 20% of the list sets survived for more than 60% of the program trace length. Of the five traces, the LYRA trace has the highest percentage of long-lived list sets; about 15% of all list sets had lifetimes of more than 95% of the total trace length. In the other traces less than 10% of all list sets had lifetimes of over 90% of total trace length.

How are list references distributed over these list sets? Do most lists belong to short lived list sets or to long lived list sets? In Figure 3.6 we combine Figures 3.4 and 3.5 to gain insight into this question. Figure 3.6 is a plot of the cumulative number of list references that belonged to list sets of various lifetimes. The plots show 2 kinds of behaviour: either an almost equal distribution of list sets of all lifetimes, or most list sets being of short lifetime. The PEARL and EDITOR plots can be approximated to straight lines through the origin with slope 1, indicating an even distribution of lists over list sets of all lifetimes. The PLAGEN, SLANG and LYRA traces, on the other hand show near inverse-exponential behaviour; this corresponds to most lists being in list sets of long lifetimes.

Combining these observations on list set size and lifetime for our five reference traces, we have a list set partition where there are

(1) few large list sets and several small list sets,
(2) few list sets are long lived, and
(3) and most lists belong to long lived list sets.

We conclude that the few large list sets that survive for long periods of time contain a large percentage of all list references, while other list sets are more transient in their access behaviour. List sets would therefore appear to be equivalent to working sets of referencing locality. Note that the LYRA trace contains a far

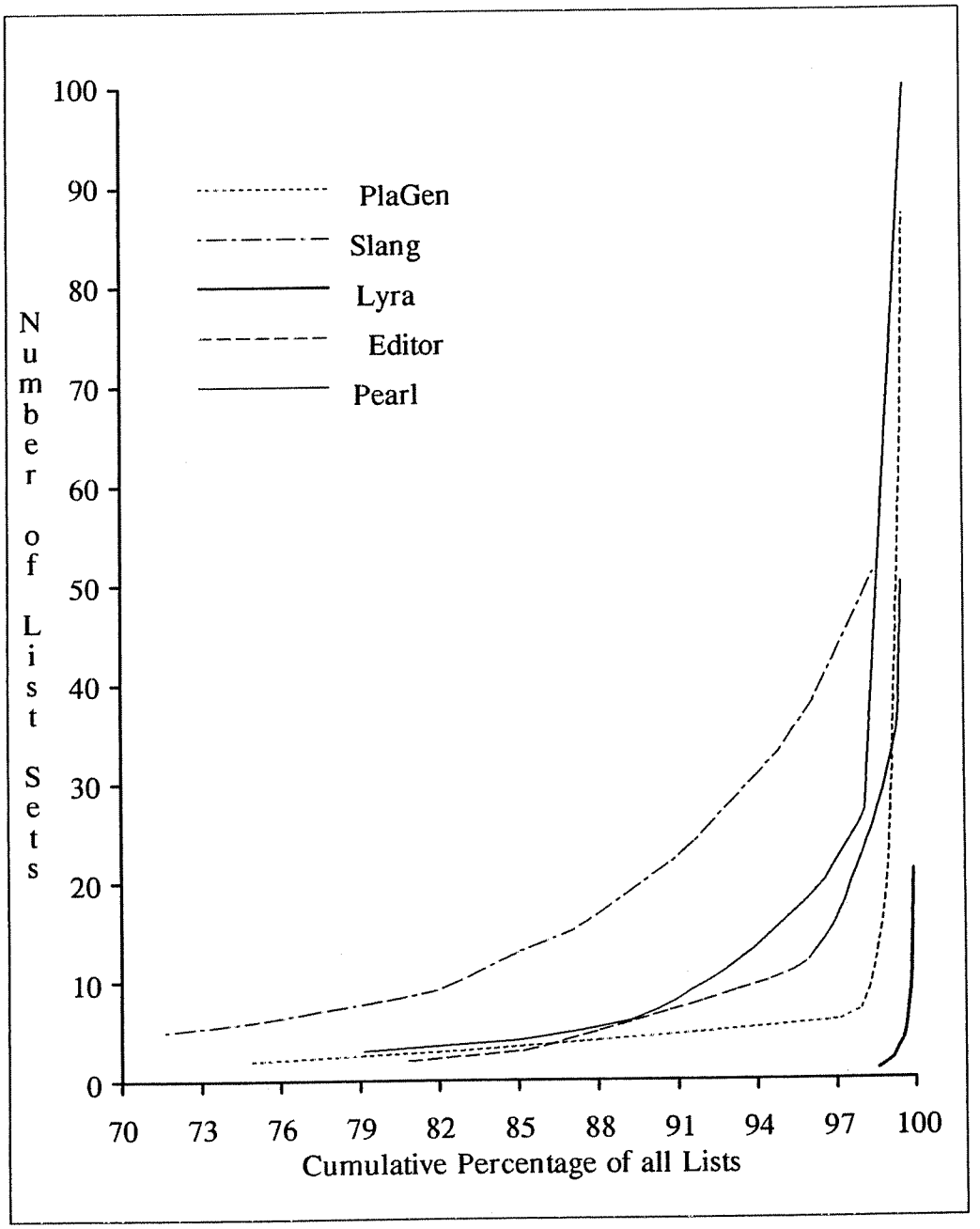


Figure 3.4. Distribution of Lists over List Sets.

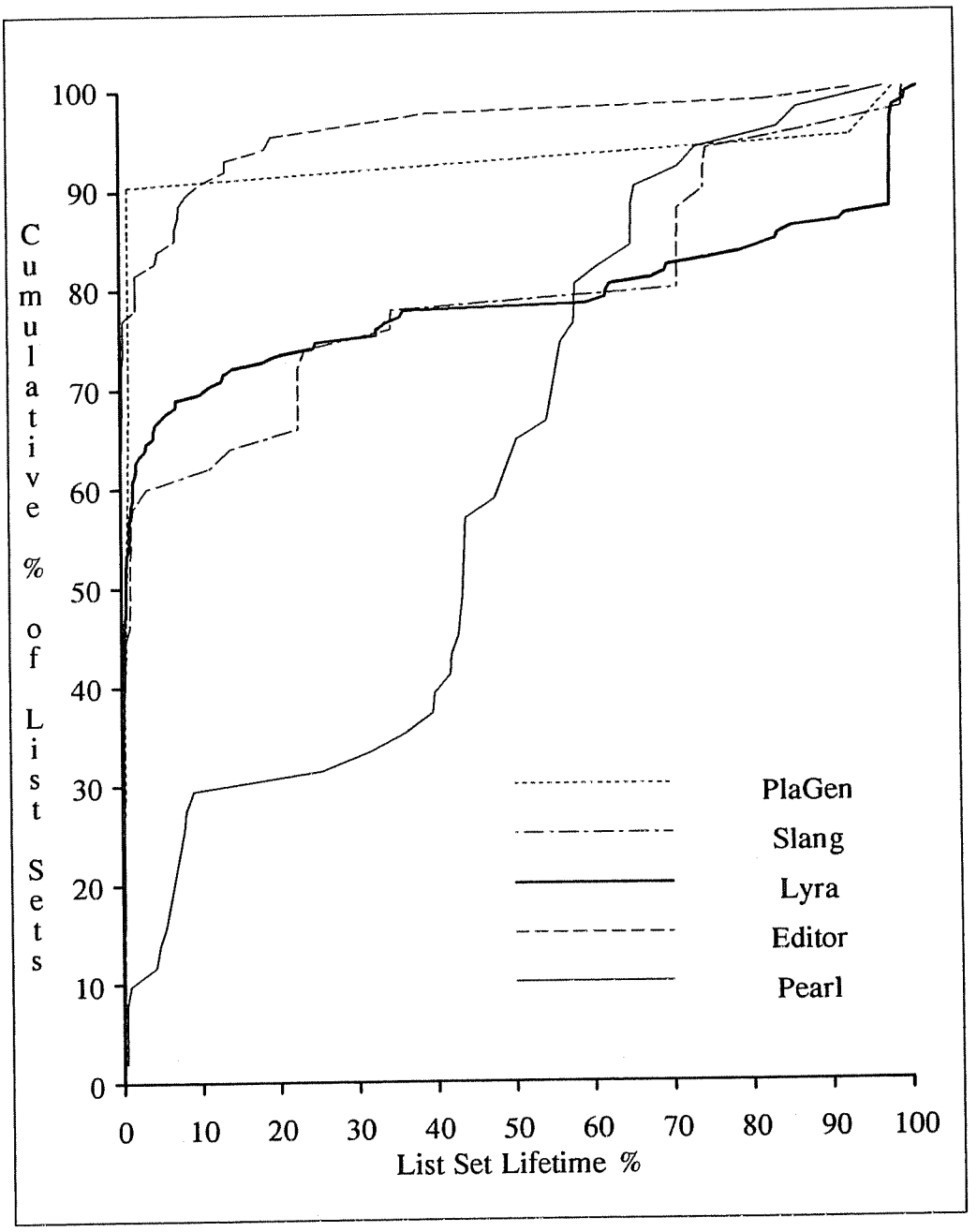


Figure 3.5. Distribution of List Set Lifetimes over List Sets.

larger percentage of large, long-lived list sets than the other traces, and we would therefore expect it to have a larger working set than the other traces as a consequence.

3.3.2.3. List Access Characterization

Our next goal was to investigate the temporal locality of access to these list sets. This amounts to taking a closer look at exactly how frequent and far apart references to members of a list set are during the lifetime of that list set. The Least Recently Used (LRU) stack model was used in this study. We used the algorithm of [Matt70a], where a single pass over the program trace is enough to calculate the success rates for a whole range of LRU stack sizes. Clark used the same method. Figure 3.7 shows the percentage of all list references that fall into different depths of the LRU stack. Even though Clark's LRU stack study was at the list cell level his graphs are of the same general shape as ours. There is wide variation in behaviour over the programs in both studies. What they show in common is that a stack depth of 4 list sets captures from 70-90% of all accesses. List sets are, therefore, objects of high temporal reference locality.

A list set is, in essence, a collection of list cells that are linked together by car and pointers. Our studies show that at any given instant the Lisp working set consists of a small collection of list sets. Architectural support for list manipulation should provide fast access to the current working set. A hitherto unanswered question is whether or not any specific paths into the list structure should get preferential treatment - are there sequences of accesses into lists that occur with sufficient frequency to warrant special hardware? To answer this question, we looked for patterns in our list access streams. We say that **primitive function chaining** has occurred if the value returned by one primitive function is immediately passed to another primitive function. Since higher level function calls are transparent to our studies these two primitive calls might actually be separated by several function calls, but we are guaranteed that no list pointer creation or modification has taken place in between. Table 3.2 below shows the percentage of all **car** and **cdr** calls that were inside such function chains. The percentages are significant in 4 of the 5 programs, with only the Pearl benchmark showing a low level of function chaining, and it would therefore appear that function chaining is generally common. In Pearl the major program data structures are maintained as Franz Lisp *hunks* for performance reasons. These are direct access data structures; each element of a hunk can be accessed without having to go indirect through another element, as is the case for lists represented by simple two-pointer list cells. A single hunk access would have been a sequence of chained access function calls on a Lisp implementation that did not support the hunk data structure. Though the most frequent primitive function call sequences varied from program to program, certain short chains (e.g. **car-cdr**) consistently occurred more frequently than others. The exact nature of the chaining varied from program to program. In short, we can draw no generally applicable observations regarding primitive function chaining patterns. The study does indicate, however, that in a Lisp program that uses lists as the main data structure, 25-80% of all list manipulating primitive calls are made from within function chains.

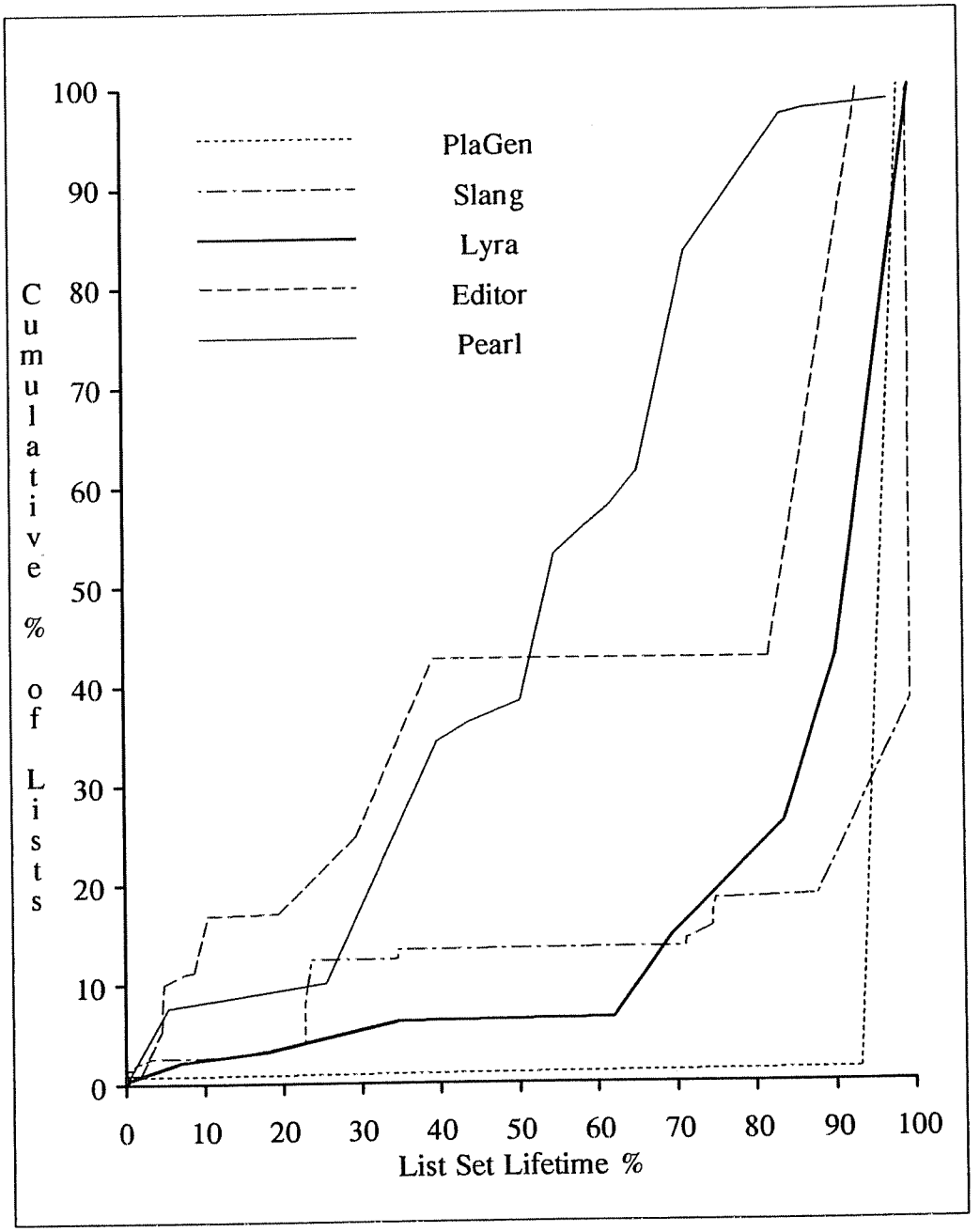


Figure 3.6. Distribution of List Set Lifetimes over Lists.

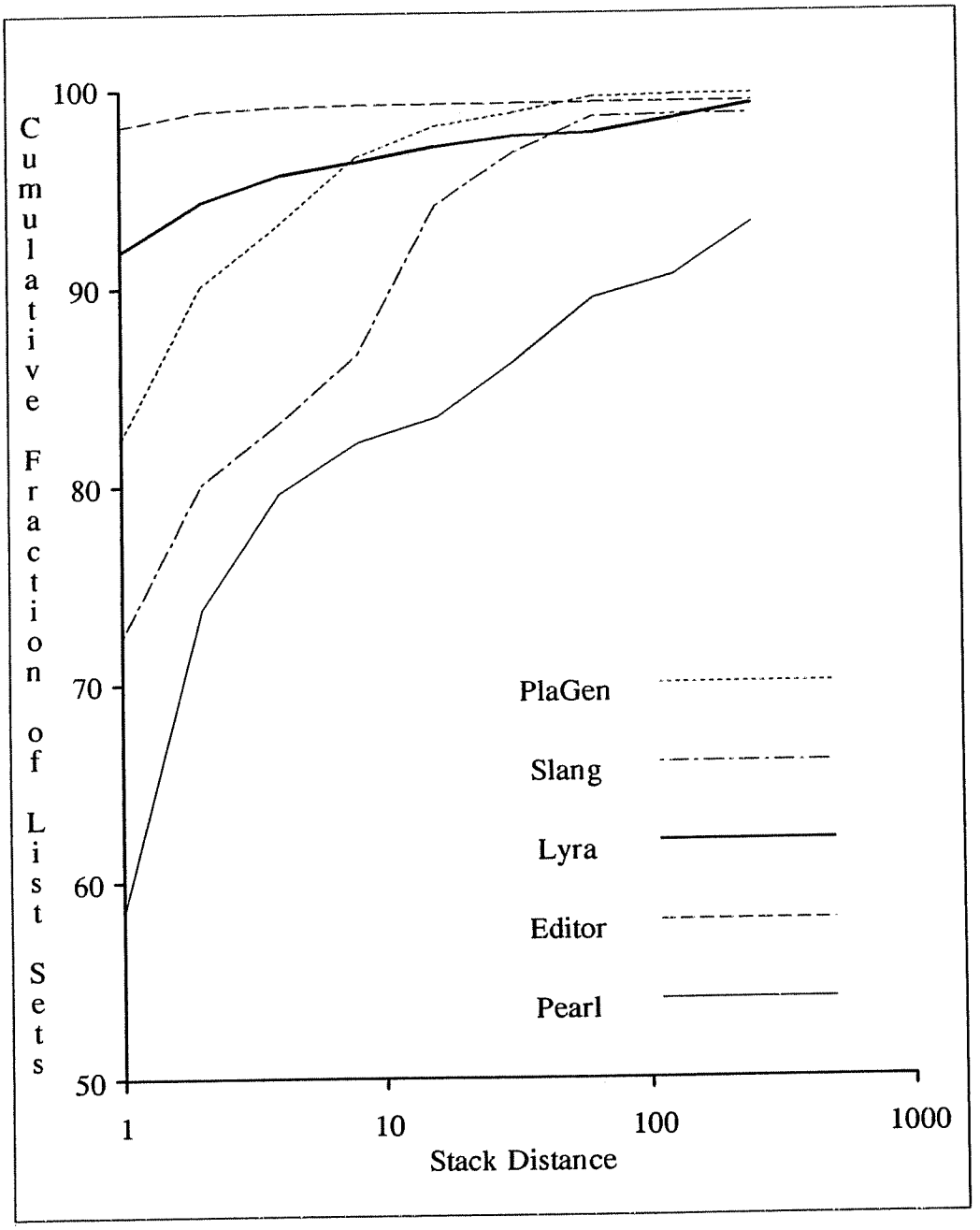


Figure 3.7. Distribution of List Set LRU Stack Distances.

Table 3.2. Percentage of CxR calls that Occurred inside a Function Chain

Benchmark Name	Primitive Function	
	CAR	CDR
SLANG	55.68	26.71
PLAGEN	26.68	40.89
LYRA	82.75	68.99
EDITOR	47.21	38.72
PEARL	0.88	1.00

3.3.2.4. Sensitivity Analysis

The studies described in this chapter suggest that the list set partition of a Lisp list access stream defines structural locales in the list structure that are accessed with high temporal locality of reference. Our list set partitioning procedure used a *separation constraint* to define a window in the access stream within which list reference relationships were searched for; we used a separation constraint of 10% in gathering the observations reported thus far. How sensitive are our observations to perturbations in this parameter? To answer this question we conducted two sets of sensitivity experiments.

In the first set of experiments we varied the separation constraint from 5% upto 100% and examined the resulting list set partition for the SLANG trace. Figures 3.8-3.10 show the results of this study. These figures show how the SLANG lines of Figures 3.4-3.6 get modified as the separation constraint is varied. It is clear that while the graphs differ with different separation constraints, they consistently display the same general behaviour. Also, as we had anticipated, a smaller separation constraint results in a smaller number of large list sets. The lines for the two largest separation constraints, 50% and 100%, are identical.

In the second set of sensitivity experiments, we ran the PLAGEN, SLANG, LYRA and EDITOR traces through the simulator using a separation constraint that was constant over all four traces rather than a fraction of the length of each trace. The constant that we chose was 10% of the length of the shortest trace (SLANG); this was 0.79%, 3.34% and 5.91% of the lengths of the LYRA, PLAGEN and EDITOR traces respectively. Figures 3.11-3.13 are the results of this study. These graphs are the constant separation constraint counterparts of the graphs of Figures 3.4-3.6, which were plotted using a 10% separation constraint. A comparison of Figures 3.4 and Figure 3.11 reveals little change in behaviour other than for the LYRA trace. This trace shows a shift towards a list set partition in which there are several small list sets and few large ones; Figure 3.11 shows that the 100 largest list sets in the LYRA trace represented only 88%

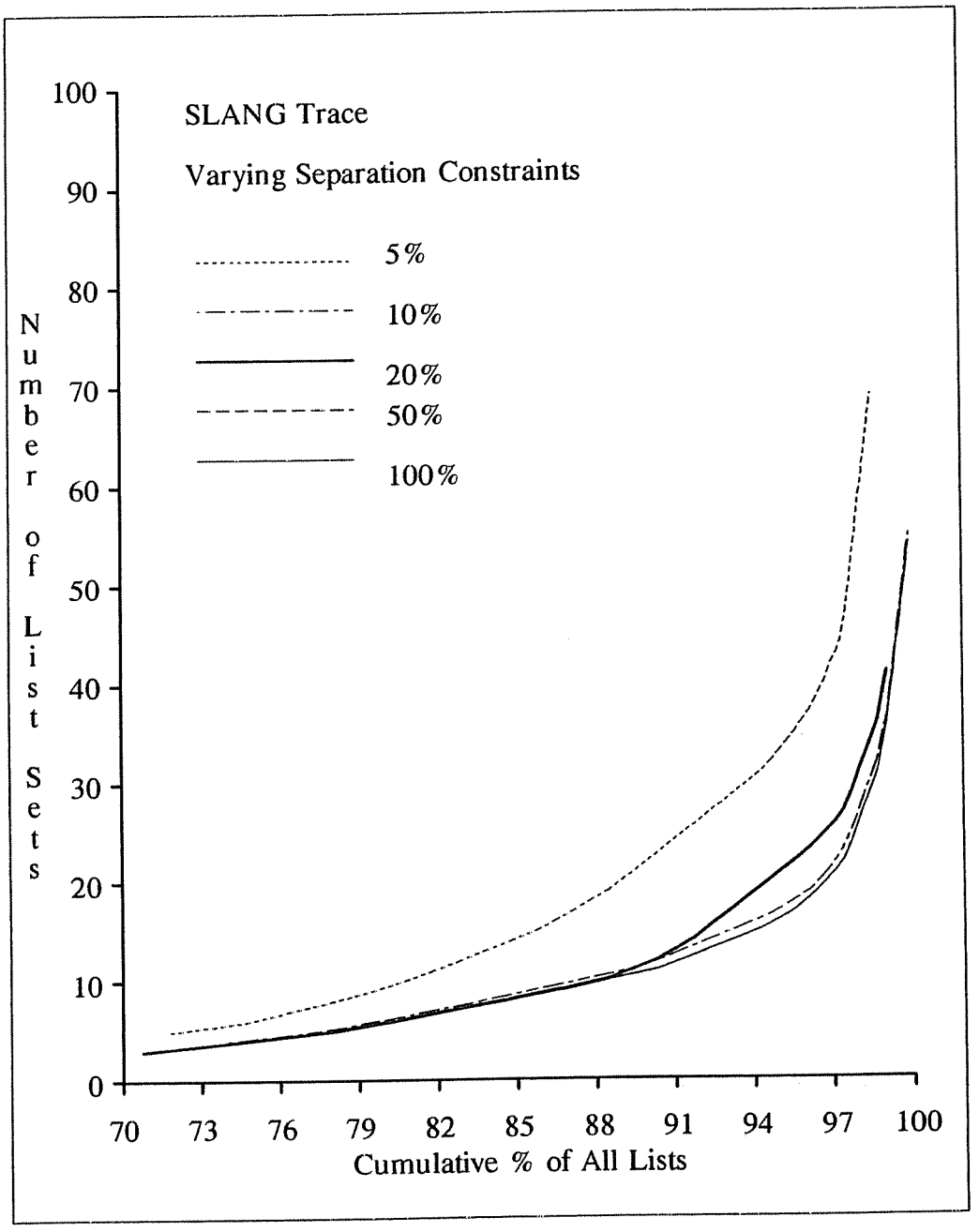


Figure 3.8. Varying Separation Constraint: List Distribution.

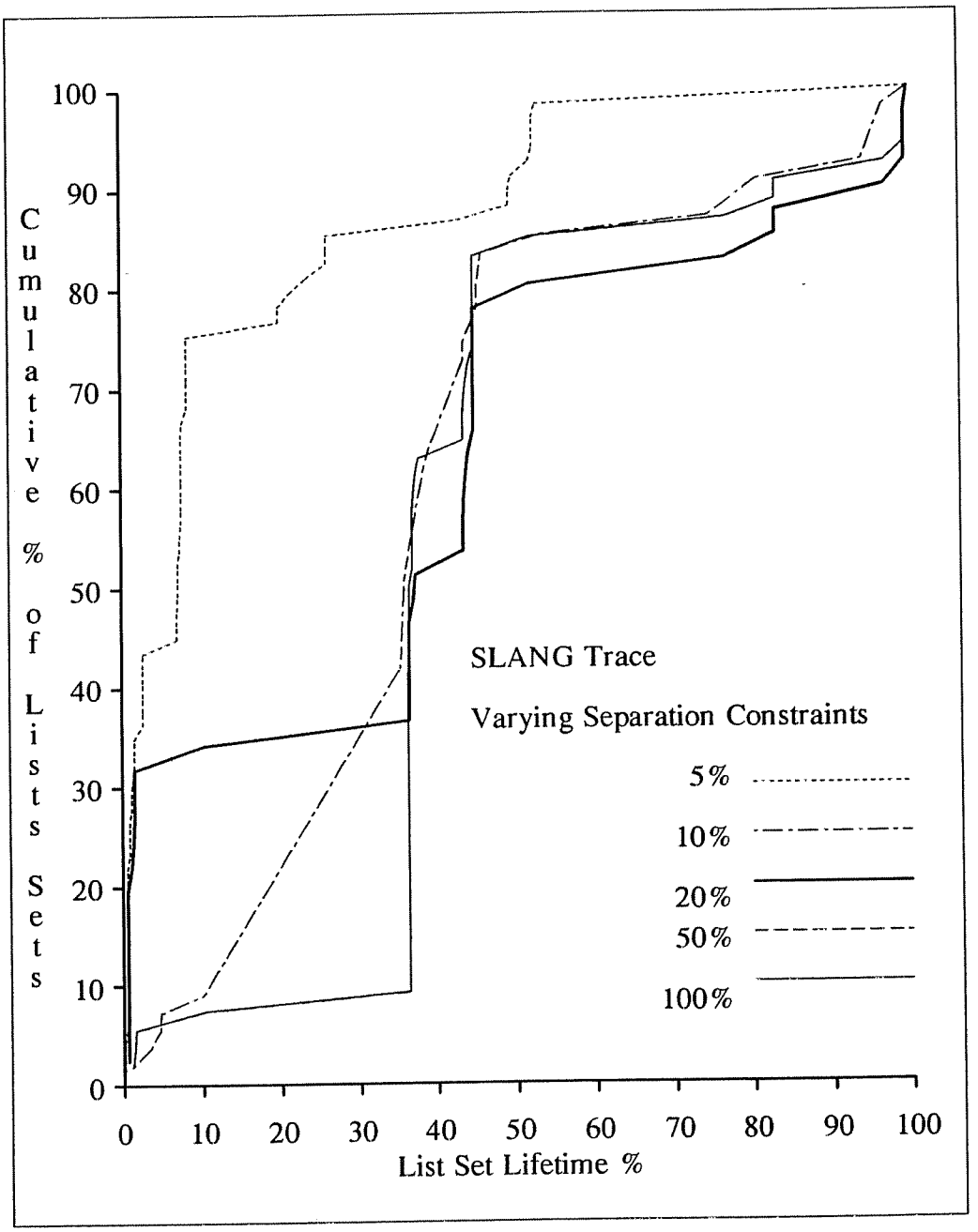


Figure 3.9. Varying Separation Constraint: List Set Lifetime.

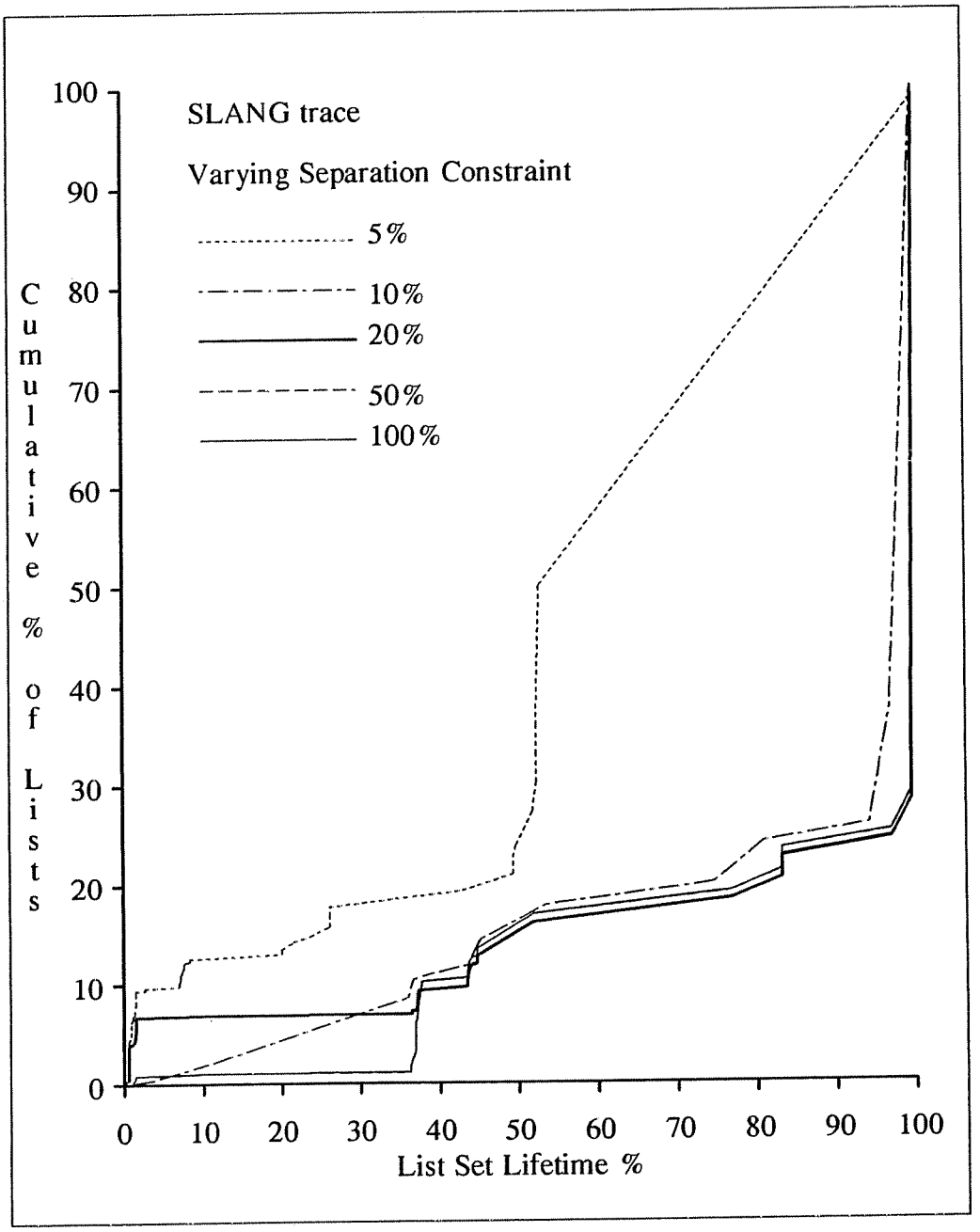


Figure 3.10. Varying Separation Constraint: List Lifetime.

of all the list references in the trace, as against almost 100% in Figure 3.4 (the 10% separation constraint case). The other traces do not show such a pronounced change. This is probably because of the extent by which the separation constraint was changed for each trace - from 10% to 0.79% for the LYRA trace, but only from 10% to 5.91% for the EDITOR trace.

The distribution of list set lifetimes over list sets, as plotted in Figure 3.12, does not undergo much change either. Notice that the LYRA and EDITOR lines do not contain any list sets that lived for more than 50% of the total trace length. This indicates that these traces contain less structural locality than the other two traces; a reduction in separation constraint window size results in a drastic reduction of the number of large, long-lived list sets. They are replaced by smaller, shorter-lived list sets. This is re-iterated in Figure 3.13, which is a cumulative plot of the number of lists belonging to lists of various list set lifetimes. Notice that the SLANG and PLAGEN lines in Figure 3.13 do not differ much from their counterparts in Figure 3.6.

We conclude that the observations we made are not very sensitive to minor changes in the separation constraint of our list access stream partitioning procedure. We were able to anticipate the changes in the nature of the list set partition with varying separation constraint sizes.

3.4. Summary

The studies that we described in this chapter serve two purposes. Firstly, they provide statistics describing the structure of Lisp lists. Clark's results on list cell pointer distances are one source of such information. Our studies used a new set of measures (in **n** and **p**) to study this issue. These measures appear to be both concise and informative.

Secondly, and more importantly, this chapter provided new insight into the locality of reference present in Lisp reference streams. Previous studies of Lisp list behaviour have revealed that Lisp programs show good spatial locality of reference. Spatial locality in list access is dependent on the representation scheme used; in a system where memory is largely dynamically heap allocated, the spatial position of data items in memory is highly dependent on the heap management policies employed. Our definition of list sets provides a means to study locality of reference in a representation independent way which we call structural locality. The studies described in this chapter illustrate that there are high levels of structural locality in Lisp program traces. The degree of primitive chaining that we detected confirms this observation. When a Lisp list access stream is partitioned into list sets using the procedure we described in this chapter, most list references are members of large, long-lived list sets. We were able to identify a small number of such large list sets in each trace that we analysed. A Lisp machine should be able to take advantage of this accessing regularity to improve list accessing efficiency.

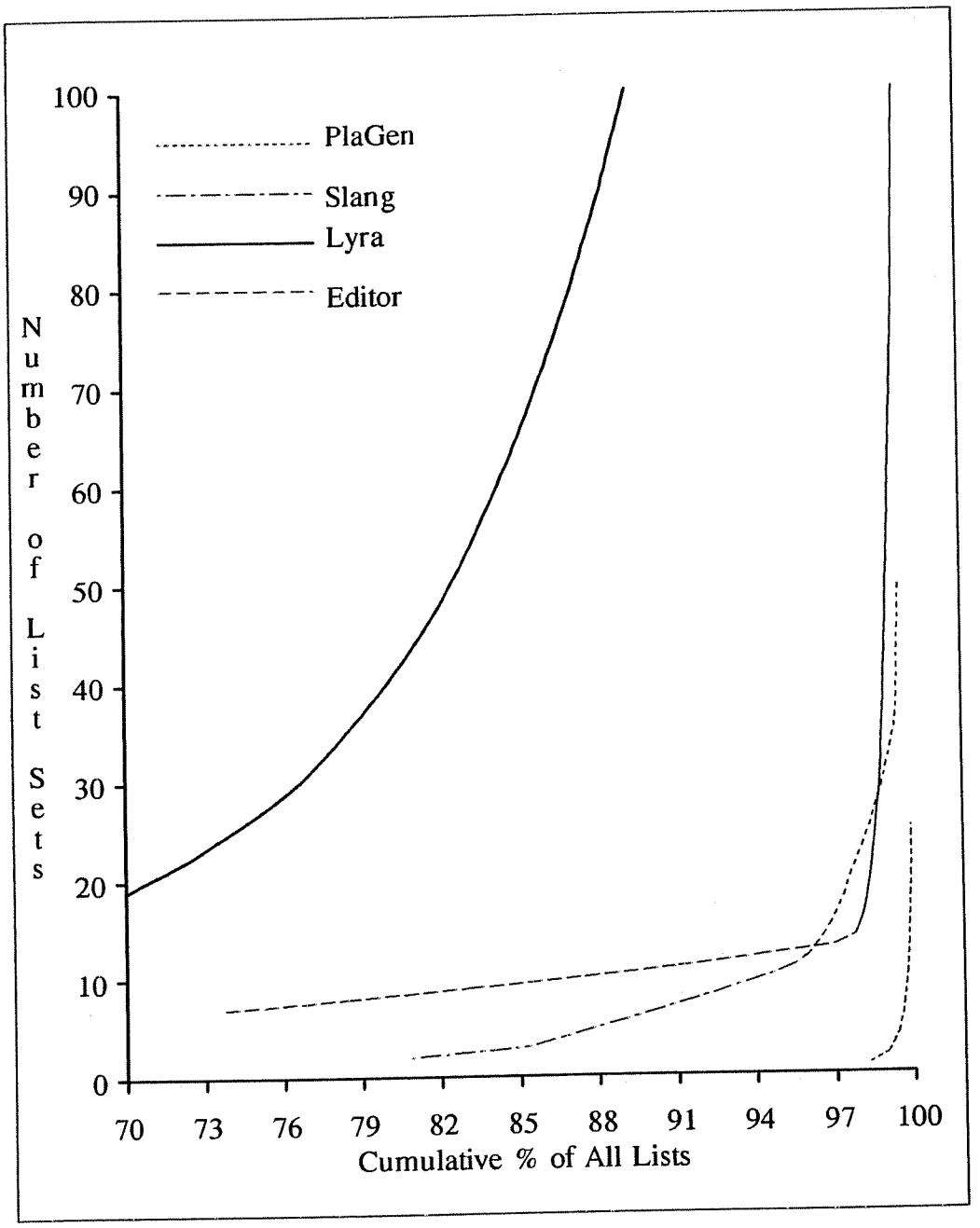


Figure 3.11. Fixed Separation Constraint: List Distribution.

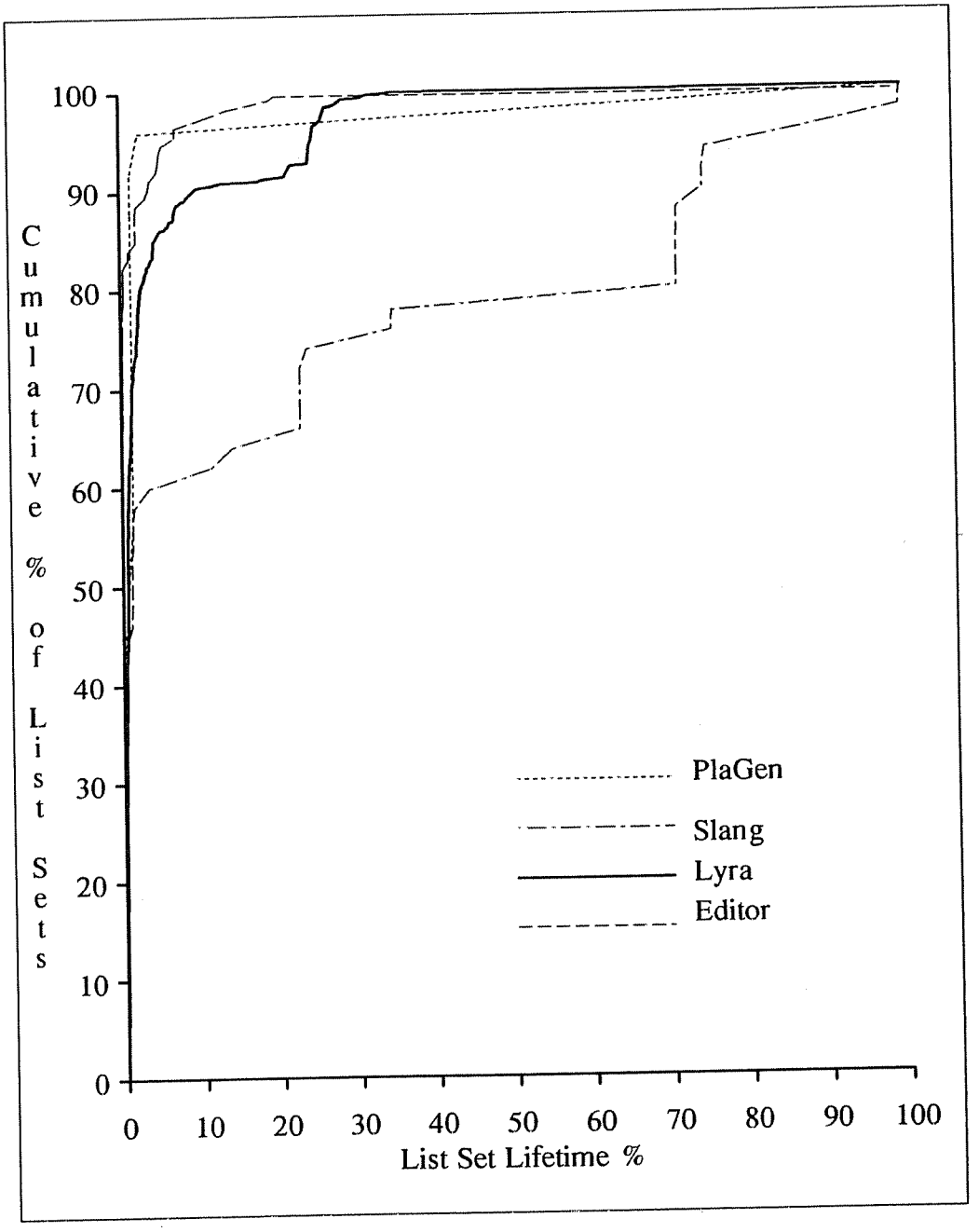


Figure 3.12. Fixed Separation Constraint: List Set Lifetime.

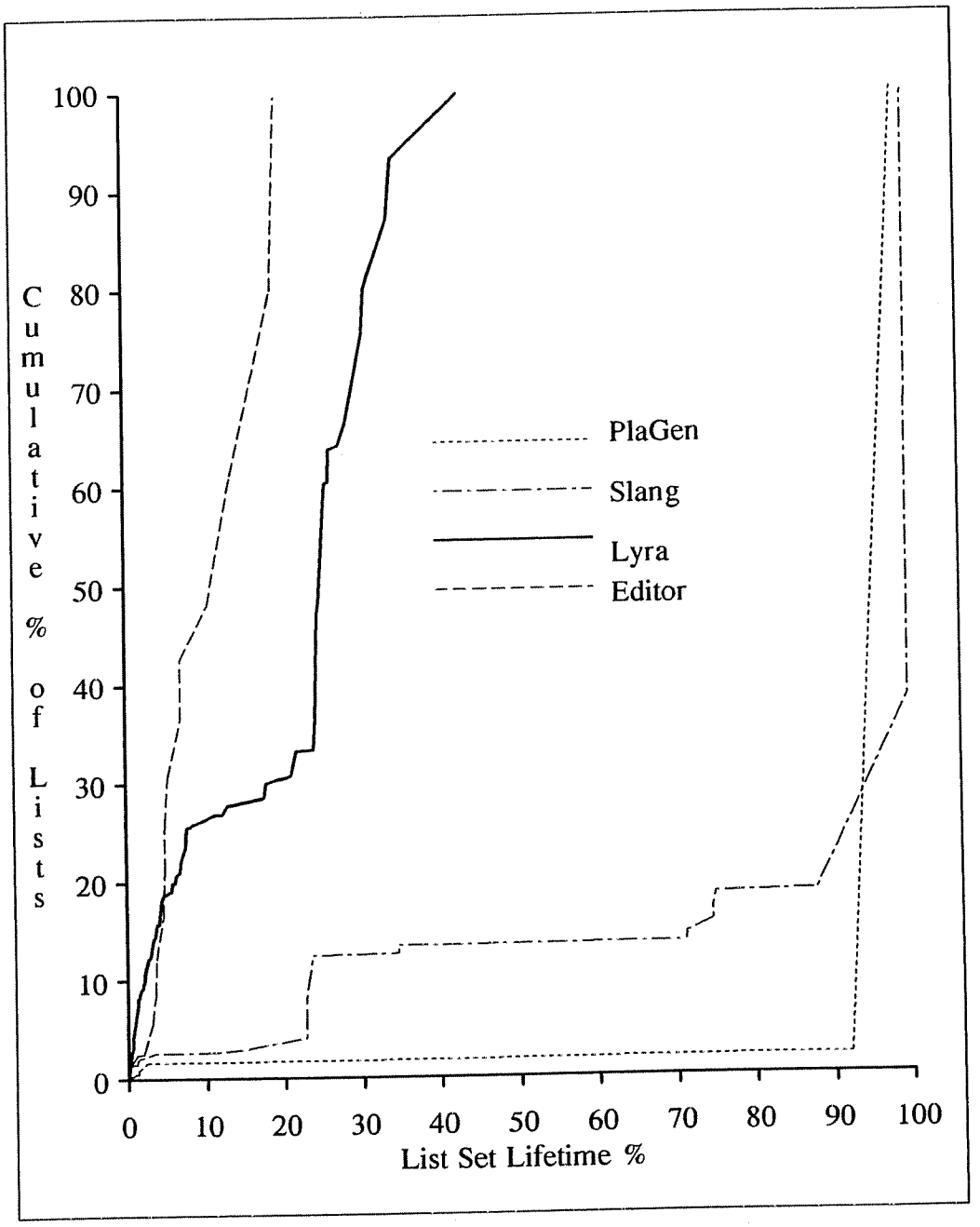


Figure 3.13. Fixed Separation Constraint: List Lifetime.

Chapter 4

A SMALL Lisp Machine Architecture

4.1. Introduction

Our studies reveal regularities in the list data access in Lisp execution that suggest solutions to the data space management problem for Lisp systems. The SMALL architecture we are about to describe is geared towards achieving efficiency in

(1) accessing list data,
(2) use of limited processor-memory bandwidth, and
(3) the management of dynamically allocated (heap) memory space,

in an environment where function calls occur frequently.

The traditional approach to making data access efficient has been to take advantage of locality in data reference; the use of register files and data caches are examples of this approach. Processor-memory bandwidth is conserved both by this and by the encoding of addressing information. Finally, support is traditionally provided for heap management in two main ways. The first involves explicit support for garbage collection, say in the form of marking bits and hardware to update them. The second involves attenuating the debilitating affects of the heap management problem through parallel activity.

4.2. Rationale

Traditional solutions to the data space management problem will not suffice in a high performance Lisp system. Consider the use of fast general purpose registers. Since there are typically only a small number of such registers, they can be addressed efficiently using a few bits. At the same time, register access speed is far faster than main memory access speed. There is, however, an overhead in using registers - work has to be done on each function call and return to ensure consistency of the data space view presented by the register file and memory. Multiple window register files have been used to reduce the register saving overhead [Patt81a]. While this approach might work for a C programming environment, it is not clear that it could cope with the magnitude of register saving and restoring that would result in a Lisp environment. Elaborate register allocation effort at compile time can make the use of registers worthwhile in certain function call intensive programming environments, like C. Lisp's dynamic nature makes this difficult. Register speed data access, then, seems out of the question.

Another way to minimize the amount of time the processor has to wait on a slow memory system is to introduce a fast buffer memory between the processor and memory. Studies have shown that data caches can get reasonably high hit ratios during Lisp evaluation [Smit85a]. Data cache designs typically use simple replacement algorithms like LRU (Least Recently Used), approximations to LRU, or random replacement. LRU replacement might displace data that is going to be accessed in the near future even if the cache contains garbage, which would obviously be the best data to displace. From the studies of Chapter 3 we

have seen that there is considerable locality of reference in Lisp list access streams related to the structure of Lisp lists. What we would like to see, then, between the processor and slow memory system, is a fast memory structure that selectively captures list data in a Lisp-specific way. This is what we aim for in SMALL, the Structured Memory Access of Lisp Lists Architecture.

4.3. The SMALL Architecture

The organization we propose for a Lisp machine is shown in Figure 4.1. 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 (manipulation of the control stack and association list over function calls). The EP passes all list related work to the LP. This general organization is suggested in [Bake78a] but to our knowledge it has not been developed further.

To make the communication between the EP and the LP efficient, the EP does not directly deal with main memory addresses. In managing list memory,

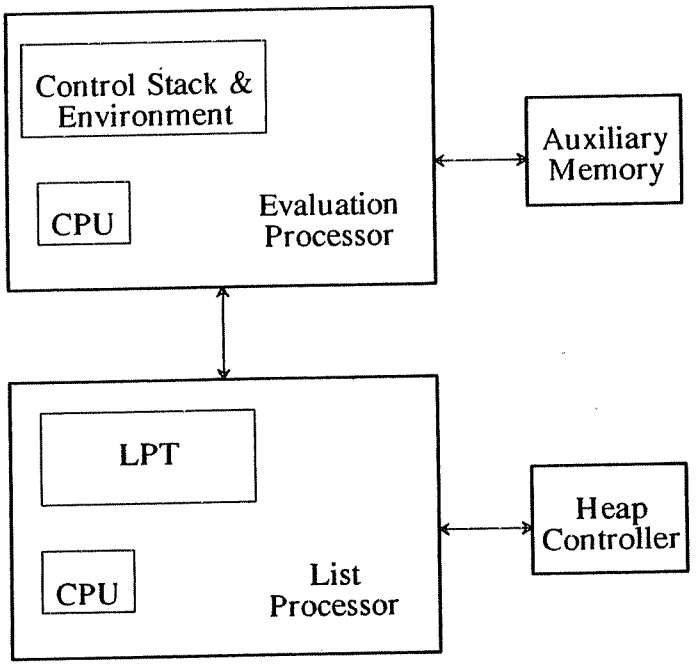


Figure 4.1. Proposed Organization.

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 in meaning. The LPT enables some of this distinction to be retained. More importantly, the LPT serves as a buffer for the heavily accessed parts of the list structure; the LP implements algorithms necessary to capture the currently accessed portions of the list structure in the LPT.

Figure 4.2 shows the components of the LPT; the various 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 similar to virtual memory mapping since both make flexible use of physical memory possible without inconveniencing the user.

4.3.1. The Evaluation Processor

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.

ID	CAR	CDR	REF	ADDR	MARK
L1	-	-	1	a1	-
L2	-	-	1	a2	-

Figure 4.2. Fields of an LPT Entry.

Consider that the environment is maintained as a stack of name-value binding frames. This corresponds to a deep bound implementation of addressing environments. Recall that a deep binding implementation leads to fast function calls and returns at the expense of look-up time, while shallow binding leads to fast look-up at the expense of slowing down function calling and returning. Also, shallow binding is not the preferred implementation method for environments in multi-processor Lisps. For our purposes, a stack implementation of environments is convenient for illustrating the principles involved.

We assume that Lisp code to be run on SMALL undergoes some simple pre-processing. Part of this serves to speed up the name look-up process. The pre-processing enables function arguments and locals to be looked-up as known offsets in the environment stack, thus saving on look-up time. In the case of non-locals the pre-processing cannot predict the run-time location of the name-value pair, and so non-locals result in run-time searches of the environment.

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 EP also maintains the control stack, which contains a stack frame for each active function call. There is one control stack frame for each function call; there is also a set of bindings added to the environment for each call. We will assume that the environment (name-value bindings) is incorporated in the frames of the control stack. Each stack frame thus contains a set of name-value bindings, and the information necessary to return from function call. This information includes a return address, and a stack frame pointer to indicate how big the frame is; the latter is necessary to determine how much of the stack must be removed on return from that function. Stack items that contain name-value bindings are composed of a name field, a tag field (specifying the type of the value), and a value field (which contains an *identifier* if the value is a list). In the case of list values, the tag field also indicates whether the value is a copy; this field is set when a value is passed as a call-by-value parameter. Before such a value can be modified the EP must create a copy of it by sending a **copy** request to the LP and binding the returned *identifier* to the appropriate name. At function return time, before control is transferred back to the function that initiated the call, a reference count decrementing request is sent to the LP for each stack item that represents a name-value binding added during that call, and that item is then popped from the stack.

4.3.2. The List Processor

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. For the LP to keep track of the relationship between the various list pointers represented by these LPT entries, we would like to be able to identify which entries belong to the

same list set. So, we extend the LPT tuple with 2 more fields, *car* and *cdr* that will relate LPT entries to each other. The *car* field of LPT entry **L1** will be set to **L2**, where **L2** is an index into the LPT, and the list reference **L2** is the **car** of list reference **L1**. 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 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. The *car* and *cdr* fields therefore serve to minimize recalculation of **car** or **cdr** operations on lists and make list access faster. This feature enables the LPT to cache the more recently and frequently accessed parts of the list structure in a Lisp-specific way.

4.3.2.1. LPT Management

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 LPT, 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 Chapter 5, 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 to the object from within the table ceases to exist. An entry ceases to exist when its reference count goes to zero.

In Chapter 2 we have seen that 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 list objects are specified in its *car* and *cdr* fields) will get their reference counts decremented ***only when the freed object gets re-used.*** 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 *addr* LPT entry field. Figure 4.3 illustrates this operation. 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 occupied but not actually referenced. This is the price paid for trying to minimize the delay in the freeing of LPT entries. Only when an LPT entry is allocated for reuse are the reference counts of it's old children decremented. Since we use a free stack rather than a free queue the most recently freed entry will be the first to be re-used. This minimizes the

ID	CAR	CDR	REF	ADDR
L1	L2	L3	1	-
L2	-	-	1	-
L3	-	-	1	-
L4	-	-	0	-
L5	-	-	0	L4
L6	-	-	0	L5

TOS

L6

L1 gets freed

ID	CAR	CDR	REF	ADDR
L1	L2	L3	0	L6
L2	-	-	1	-
L3	-	-	1	-
L4	-	-	0	-
L5	-	-	0	L4
L6	-	-	0	L5

TOS

L1

Request for new LPT entry

ID	CAR	CDR	REF	ADDR
L1	-	-	1	-
L2	-	-	0	L6
L3	-	-	0	L2
L4	-	-	0	-
L5	-	-	0	L4
L6	-	-	0	L5

TOS

L3

Figure 4.3. LPT Entry Freeing and Allocating.

period during which more LPT space than is necessary is occupied. So, both LPT entry freeing and LPT entry allocation can be performed efficiently in fixed amounts 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 (the check to 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. This is described later in this chapter.

4.3.2.2. LP List Manipulating Primitives

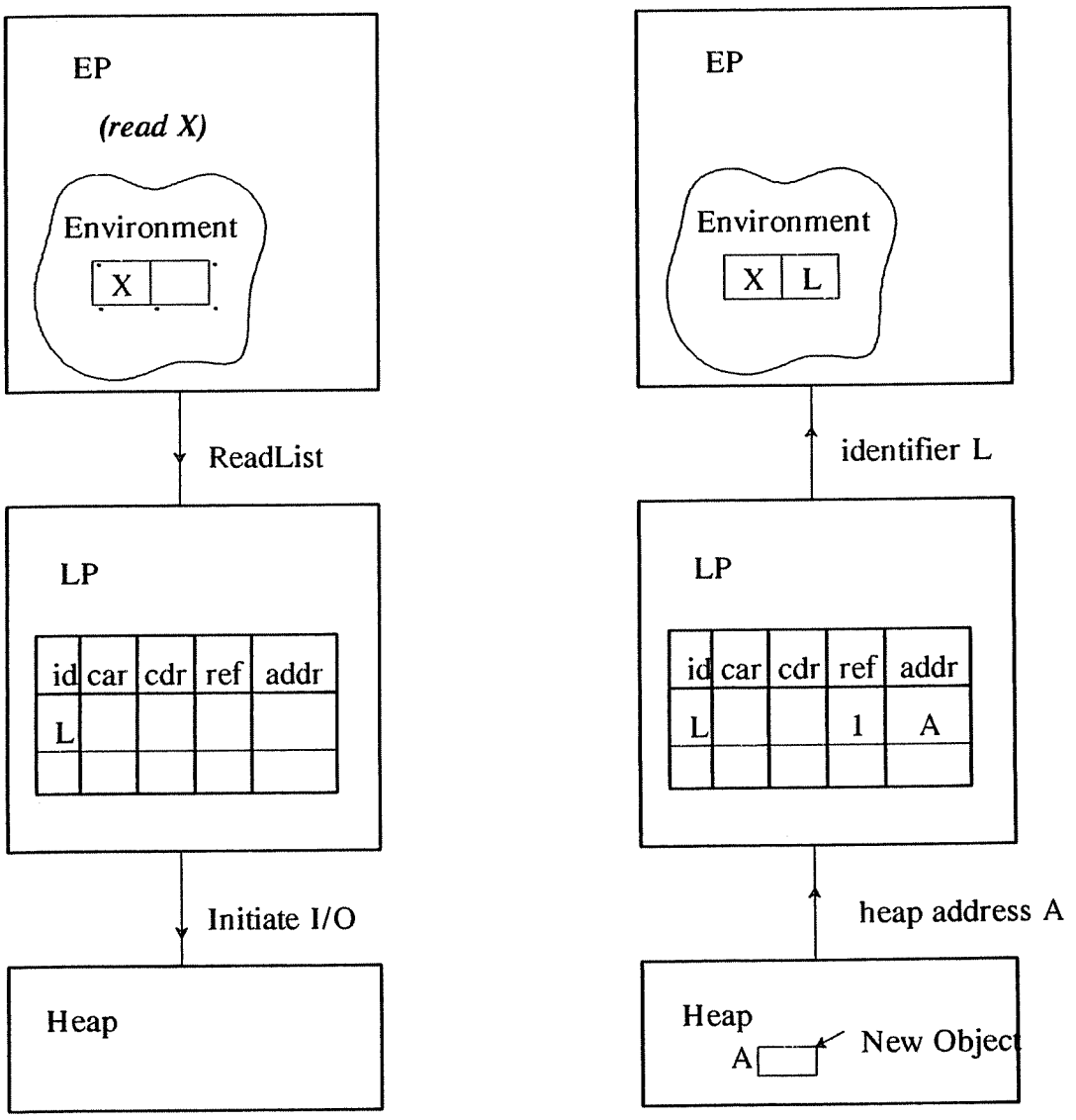
With this picture of the LPT in mind we now look in more detail at how Lisp list manipulating primitives are carried out in the SMALL architecture. Recall that in EP-LP communication lists are specified by *identifier*s which are indices into the LPT. The next 4 sub-sections describe list I/O, simple list access, simple list modification, and list **cons**ing. The illustrations show only the relevant LPT fields.

4.3.2.2.1. Reading in List Data

Consider a Lisp list input function call of the form *(read X)*. In evaluating this call the EP determines what the name *X* represents by examining the environment. If *X* has an *identifier* L1 associated with it the EP then sends a **readlist L1** request to the LP. Otherwise the EP sends a **readlist 0** request to the LP. If the EP's request had specified a list object L1, the LP decrements the reference count of LPT entry L1. The LP then initiates I/O activity which results in the list data being read into the heap memory. A new LPT entry is allocated and updated with the address information returned by the heap memory controller. The *identifier* of this LPT entry is returned to the EP as the return value of the *readlist* request. The EP updates the environment by binding *X* to this *identifier*. The case where the name *X* had no *identifier* associated with it is illustrated in Figure 4.4.

4.3.2.2.2. Simple List Access

In evaluating a list access function call of the form *(car X)* the EP looks up the environment for the *identifier*, say L1, corresponding to the name *X* and sends a **car L1** request to the LP. If the *car* field of the LPT entry for L1 is set the LP returns the value of that field to the EP and updates its reference count by one. Otherwise, the LP requests the heap memory controller to **split** the list object L1. A **split** results in a heap list object being split into two list objects, the car and the cdr of the original list object. The heap memory controller returns the addresses of these two pieces; the LP uses the addresses in setting up two new



(a)

To evaluate the read call the EP sends a ReadList request to the LP. The LP causes I/O to be initiated. Notice that the name X has no value bound to it, and entry L is unused.

(b)

A new heap object is read in at heap address 'A', which is returned to the LP. The LP allocates a new LPT entry, L, updates its fields and returns the value L to the EP which binds this value to X.

Figure 4.4. Reading in a List.

LPT entries corresponding to the *car* and *cdr* of object L1. The LP then returns

from the **car** L1 request with the *identifier* corresponding to the *car* of the newly split object L1. Figure 4.5 illustrates the steps involved in performing list access if a split takes place (the other case is straightforward).

4.3.2.2.3. Simple List Modification

In evaluating a list modification call of the form *(rplaca X Y)* the EP looks up the environment for the *identifier*s corresponding to the names *X* and *Y*, say Lx and Ly, and requests the LP to **rplaca Lx Ly**. The EP can continue with its evaluation after initiating the LP. In the LP, if the *car* field of LPT entry Lx is set, say to La, the LP decrements the reference count of LPT entry La, increments that of LPT entry Ly, and sets the *car* field of entry Lx to Ly. Otherwise it requests the heap memory controller to **split** the list object Lx first. Figure 4.6 illustrates the case where no splitting is necessary.

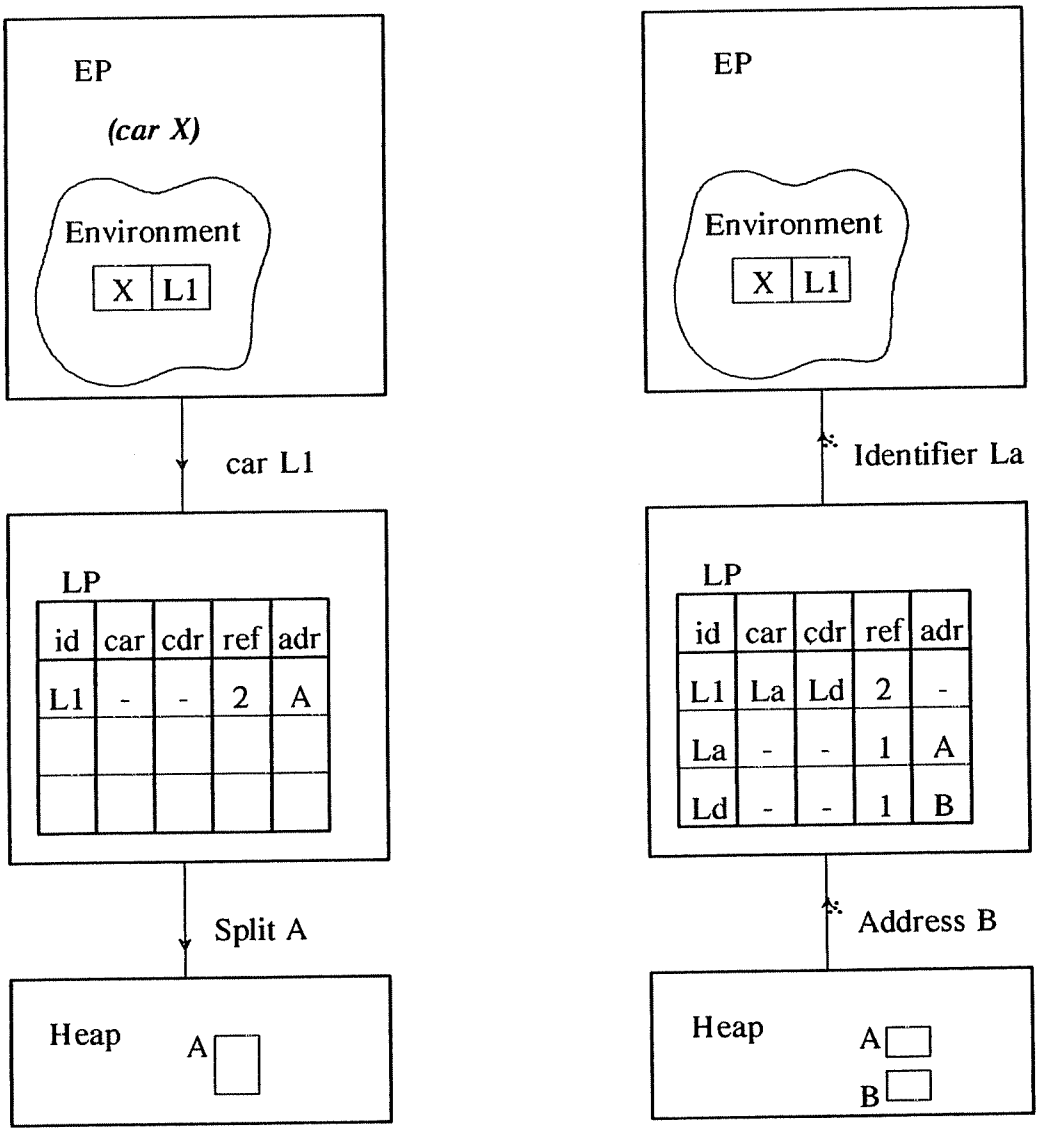
4.3.2.2.4. List CONSing

In evaluating a primitive call of the form **(cons X Y)** the EP looks up the environment for the *identifier*s corresponding to the names *X* and *Y*, say Lx and Ly, and requests the LP to **cons Lx Ly**. The LP allocates a new LPT entry, say Lz, increments the reference counts of entries Lx and Ly, sets the *car* and *cdr* fields of LPT entry Lz to Lx and Ly respectively, and sets the reference count of entry Lz to 1. The LP sends *identifier* Lz as return value to the EP immediately after the LPT entry has been allocated and before the LPT entry fields have actually been set, allowing the EP to continue with its evaluation work. Notice that **cons**ing involves no heap activity. This causes dynamically created parts of the list structure to be built up as an endo-structure present only in the LPT (rather than in the heap), and therefore readily available for fast access. Figure 4.7 illustrates the **cons** operation.

The decoupling of activity between the EP and the LP along with the virtualization of heap addresses achieved in the LPT combine to create the scope for parallel activity in the EP, LP and heap controller. As we have seen, in performing a **cons** the LP returns a value to the EP as soon as a new LPT entry has been allocated. While the LP is setting the fields of the new LPT entry and updating the reference counts of the entries being **cons**ed, the EP can continue with its evaluation work, possibly using even using the value returned by the **cons** operation.

4.3.2.3. LPT Overflow

Since the LPT is a fixed sized table, it is possible that the LP runs out of LPT space during Lisp execution. We call this condition *LPT overflow*. However, if there are LPT entries that are only referenced from within the LPT we might be able to compress them into their parents to make some table space available for immediate use. We call this condition *pseudo overflow*. *Compression* is the operation that is performed on the LPT when there are no more free table entries under a pseudo overflow condition. Figure 4.8 illustrates what happens when a compression takes place. Since L2 and L3 have reference counts of 1 and are pointed at only from within the LPT, they can be compressed into their



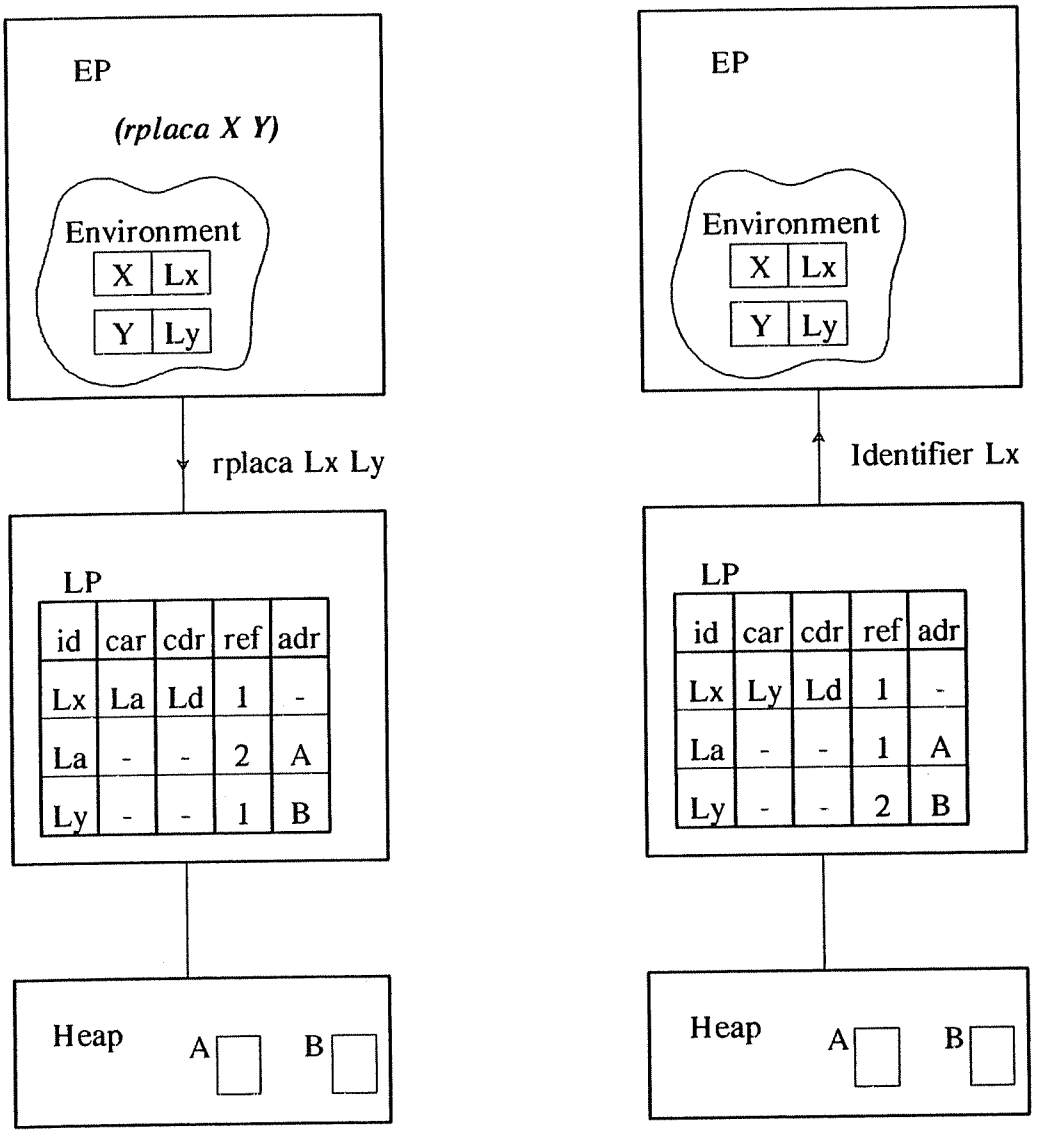
(a)

In executing a 'car', control first flows from the EP to the LP to the heap manager.

(b)

Results are returned from the heap manager to the LP, and from the LP to the EP. The identifier La is the value of (car X).

Figure 4.5. Simple List Access.



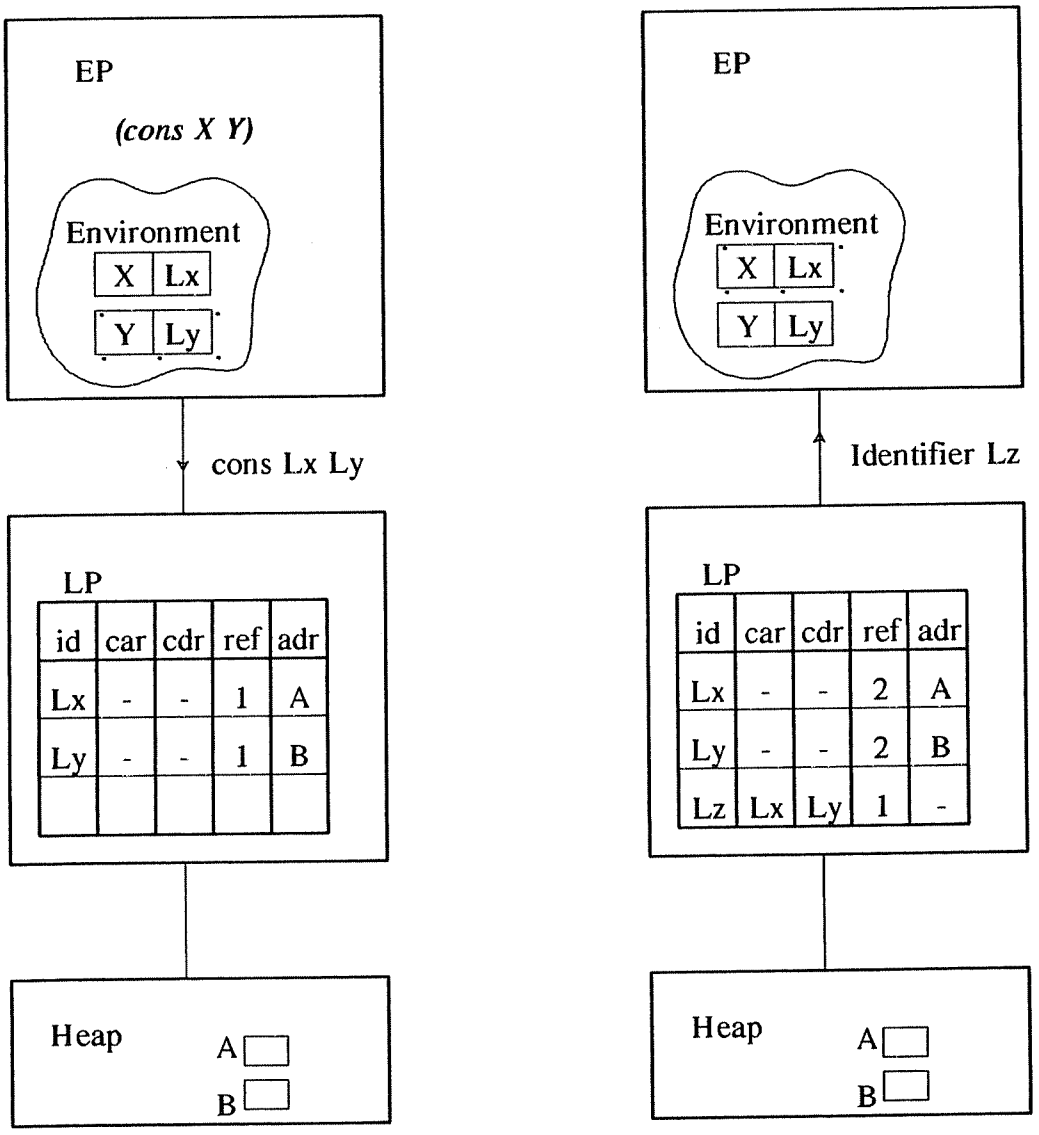
(a)

In executing (rplaca X Y) the EP looks up the current bindings of X and Y and requests the LP to perform the list modification operation.

(b)

The LP modifies LPT entries to reflect the requested modification. Control can be passed back to the EP while these LPT changes are being made.

Figure 4.6. Simple List Modification.



(a)

In executing (cons X Y) the EP finds the current bindings of X and Y and passes the request on to the LP.

(b)

The LP allocates a new LPT entry, Lz, sets its fields, updates the reference counts of Lx and Ly, and returns Lz to the EP. No heap activity is involved.

Figure 4.7. **cons**ing two Lists.

parent, L1. The heap controller is instructed to **merge** the objects at addresses *A*

and *B*, returning the address of the merged objects as *C*. Figure 4.8a shows the pseudo overflow condition before compression and figure 4.8b shows the LPT after merging has successfully completed. Two LPT entries are thus freed. Notice that **merge** is the inverse of **split**.

There are several possibilities regarding when LPT compression should be performed and how much LPT space should be freed on each compression. It can be initiated either when the LPT actually gets full or after it reaches a pre-determined threshold 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). Note that it is possible that there are no such compressible entries in the table; we call this a ***true overflow*** condition. When true overflow occurs there are no LPT entries available for the LP to allocate in carrying out operations requested by the EP. It is essential that the LP be able to recover from such this condition.

ID	CAR	CDR	REF	ADDR
L1	L2	L3	2	-
L2	-	-	1	A
L3	-	-	1	B

(a)

ID	CAR	CDR	REF	ADDR
L1	-	-	2	C
L2	-	-	0	-
L3	-	-	0	-

(b)

Figure 4.8. Compressing LPT Entries.

Recovery from true overflow turns out to be fairly expensive in terms of the heavy performance degradation between the normal mode of operation (which we call **fast mode**) and the operation under true overflow conditions (which we call **overflow mode**). Therefore, before switching to the overflow mode of operation the LP attempts to recover from true overflow by searching for hidden free LPT space. This involves looking for cycles of unused LPT entries. This space was not recovered under normal operation since we use reference counting to manage the LPT. A simple cycle detection algorithm would be used, making use of the *mark* bits to mark LPT entries and then performing a conventional sweep of the LPT to reclaim the entries forming the cycle. If any LPT space is made available by the breaking of cycles SMALL operation would continue in the fast mode. Otherwise it degrades to overflow mode.

In overflow mode the LPT essentially gets bypassed; the EP communicates in terms of heap addresses (through the LP) with the heap memory controller. To accommodate large heap addresses in the value parts of name-value bindings on the stack, there is a special tag value for *overflow mode identifier*, and the actual (large) heap address is contained in the next stack item. The LP keeps count of how many such large identifiers are referred to in the EP, and when this count goes to zero, initiates a mode change back to the fast mode of operation. Note that in overflow mode the EP references lists in two ways: using the LPT *identifier*s that are still valid, and using large heap addresses in other cases. Overflow mode is slow for several reasons. Firstly, an extra tag condition has to be checked for in the control stack for each lookup operation on the environment. Secondly, the EP-LP bus is no longer wide enough for EP-LP communication and more time is spent in operand specification. Further, the LP has to check the addresses being returned by the heap in response to list manipulation requests initiated by the LP in order to maintain the consistency of the LPT.

Clearly, operating in the overflow mode will result in performance degradation. One way to prevent this from happening is to provide an LPT large enough to make true overflows rare occurrences. Our simulation studies, described in Chapter 5, show that for an LPT with a few thousand entries true overflow does not occur during runs of our set of benchmark programs. So, operating in the overflow mode will not be a performance problem in a well tuned SMALL machine.

4.3.2.4. Example

We illustrate some basic list manipulation operations in Figure 4.9. Figure 4.9a shows the LPT after 2 lists have been read in and designated as list objects L1 and L2 respectively. (Recall that there are other fields 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 4.9b. The evaluation of (cons L3 L6) causes LPT entry L7 to be created. Figure 4.9c 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

ID	CAR	CDR	REF
L1	-	-	1
L2	-	-	1

(a)

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

(b)

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

(c)

Figure 4.9. Examples of List Manipulation in the LPT.

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 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 4.9b 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.

4.3.2.5. Concurrency in EP/LP Activity

The examples of LP operation suggest that EP and LP activity can be overlapped during Lisp evaluation on a SMALL machine. The extent of this concurrency will depend on implementation dependent parameters such as LPT access time, LPT entry modification time, reference count update time, name lookup time, etc. While the exact timing of EP-LP interaction will depend on these factors, we can get an idea of the scope for concurrency in SMALL list manipulation by assigning approximate values to these timing parameters and constructing timing diagrams for typical operations. Figures 4.10 through 4.13 are such timing diagrams for the four primitive LPT list manipulating examples that we used above. Each timing diagram contains two horizontal time lines, one for the EP and one for the LP. The relative sizes of intervals on these time lines are dependent on the specifics of the SMALL implementation. A gap in a time line indicates that the processing element is idle for that interval of time. We use dashed vertical lines to specify the action being performed in the indicated time interval.

Each diagram starts with the EP interrogating the environment regarding the current bindings of the names used in the list manipulating instruction being executed. The EP then sends a list manipulating request to the LP. Depending on the nature of this request the LP then initiates I/O, allocates a new LPT entry, or updates the fields of a particular LPT entry. The LP returns a value to the EP as early as possible, which is, in most cases, a short period after the request was made. There are a few exceptions to this quick response. One example is in reading in list data, as in Figure 4.10. Notice that the EP has to remain idle for a period of time waiting for the return value from the LP. This is necessary since the LP cannot predict the type tag of the value being read in until the I/O is complete. Similarly, when **car** or **cdr** accesses are being made and **splitting** is needed, the LP must wait for the return value from the heap controller specifying the type of the newly split object, since the **split** could result in an atomic value. In other cases, the time during which the EP is forced to wait idly for a response from the LP is small.

One possible concern that these diagrams raise is EP-LP interaction during primitive function chaining, or when the EP's instruction stream contains several consecutive list manipulating instructions. Even though the LP responds quickly to each EP request, it is not ready to accept the next request from the EP for a short period after that while modifying LPT entries. For example, in the case of the **cons** operation, after the LP sends the return value to the EP, it still has to update the LPT free list, set the fields of the newly allocated LPT entry, and update the reference counts of the two LPT entries being **consed** before it can accept the next EP request. If several such requests occur one after the other, the EP might have to wait idly for the LP to become ready. This will depend on the relative sizes of the various time parameters.

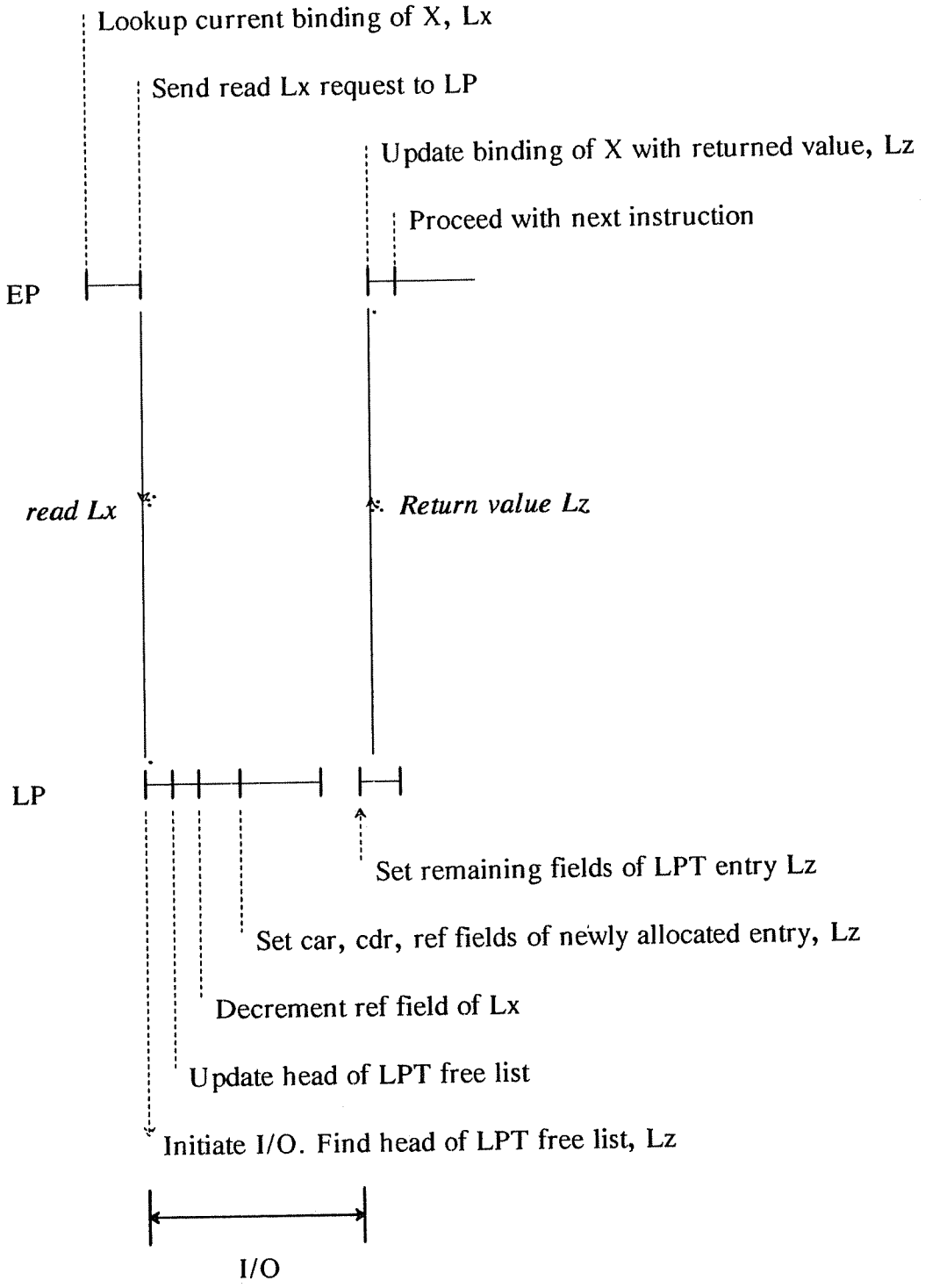


Figure 4.10. Timing in Reading in List Data.

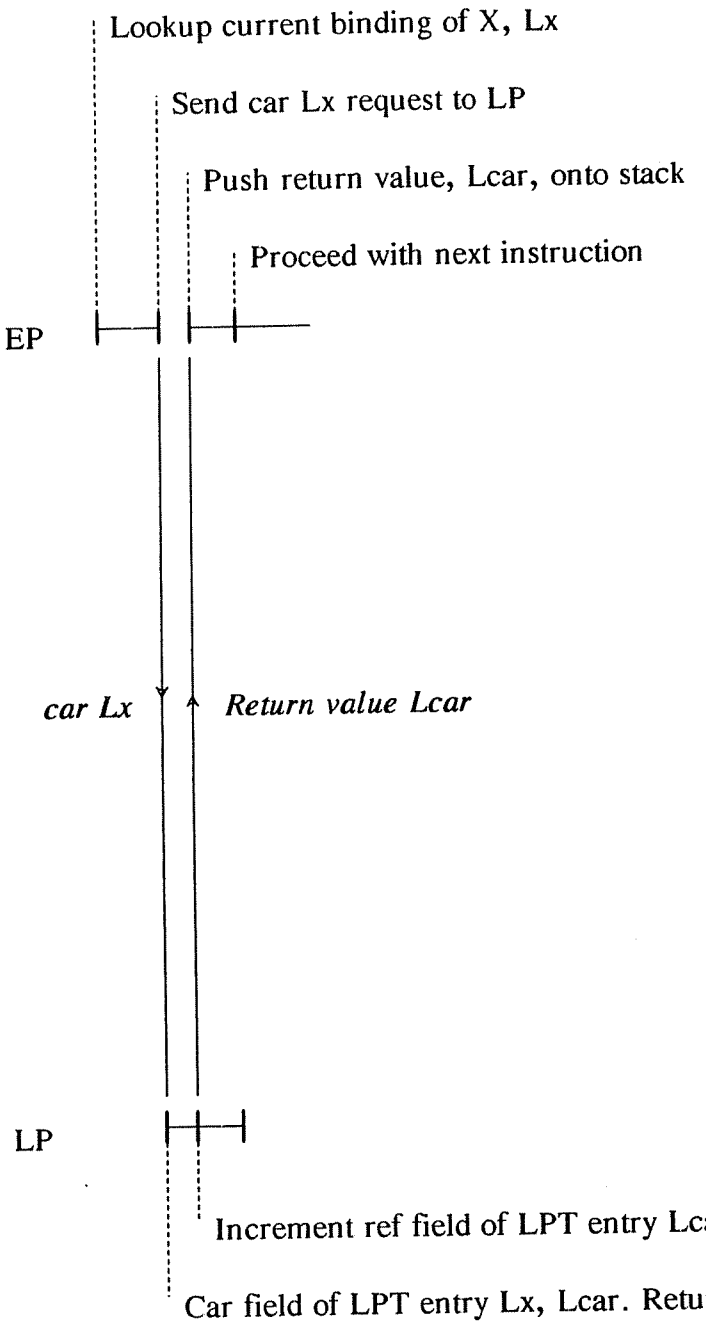


Figure 4.11. Timing in Simple List Access.

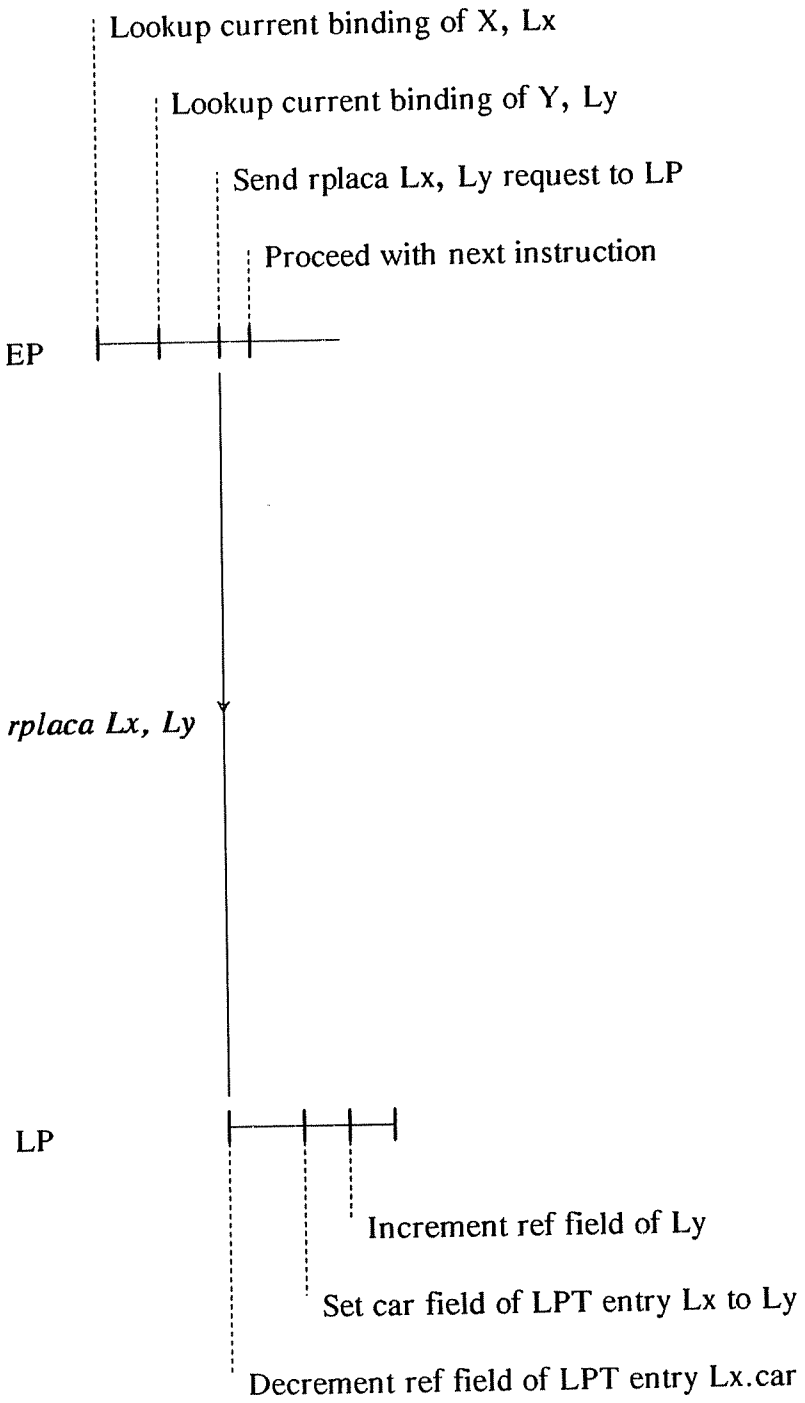


Figure 4.12. Timing in Simple List Modification.

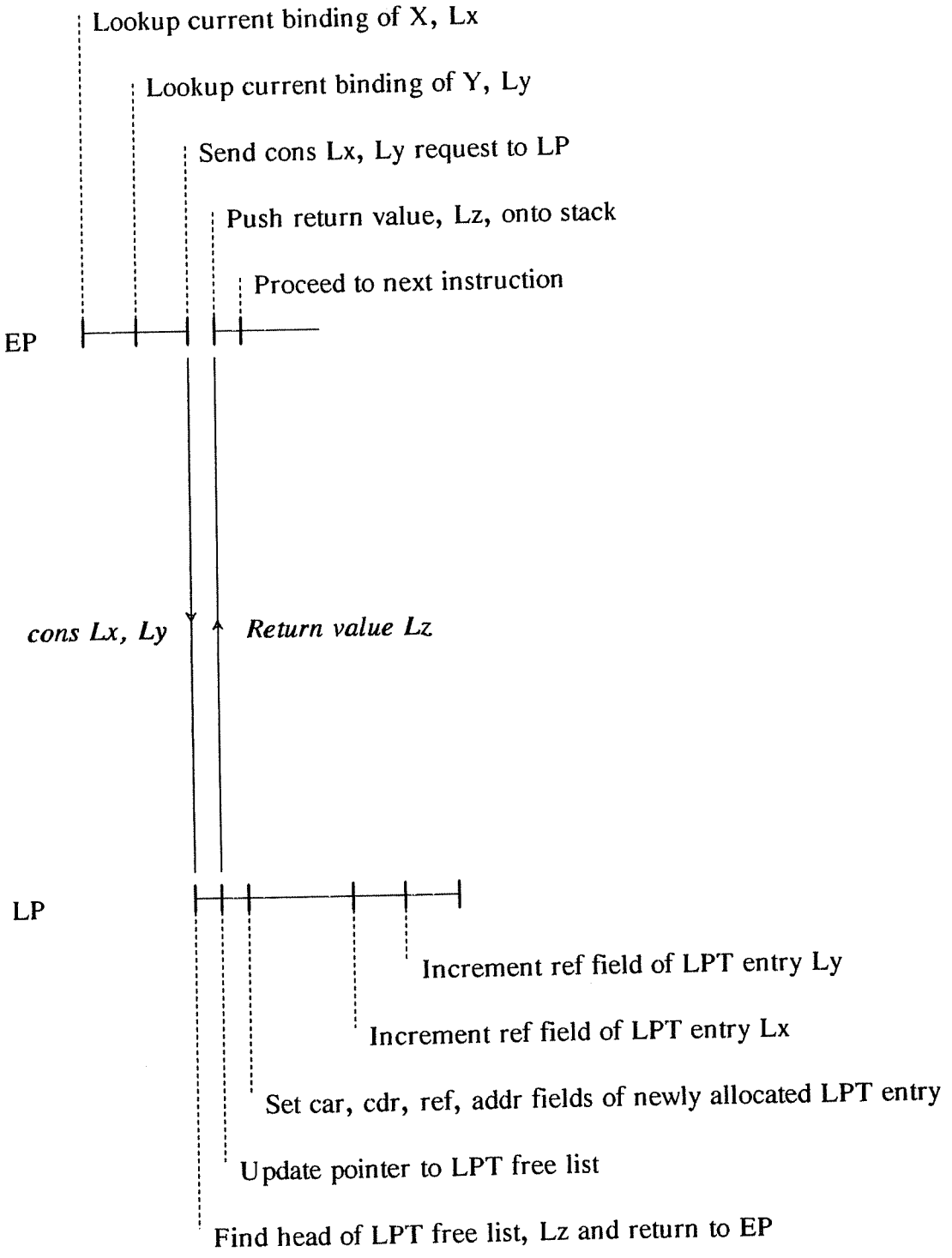


Figure 4.13. Timing in List consing

4.3.3. Heap Memory

From what we have seen of LP operation it is clear that considerable functionality is required of the heap memory controller. The controller has to manage the free heap space, perform the **splitting** and **merging** of list objects, and manage the input and output of list data. These are all strongly dependent on the actual list representation scheme used in the heap memory; any list representation scheme should be evaluated in terms of the ease with which these operations can be performed. While we have not attempted a design of the heap memory controller, the discussion that follows examines a few of the issues involved in that design.

4.3.3.1. Managing Free Heap Space

The management of heap space is greatly simplified by the reference counts maintained in the LPT. When a reference count goes to 0 and that LPT entry gets put onto the LPT free list, the LPT signals the heap memory controller to reclaim the space occupied by that object (using the *address* information from the LPT entry). Freeing this space could be an arbitrarily complicated operation depending on how list objects are represented. For example, if 2-pointer list cells are used in the heap, reclaiming the space occupied by a list involves traversing the list while adding the list cells traversed to the heap's free list. This requires a stack to keep track of the intermediate nodes that are ancestors of untraversed parts of the list. The object freeing operation takes time proportional to the number of list cells in the list representation. But, the stack space is only used temporarily, and the LP need not wait for the heap controller to finish freeing the list object. There could, in fact, be a queue of free requests serviced by the heap controller whenever convenient. The queue size could be limited as a means of flow control. This prevents the LP from running far ahead and ensures that there are not large portions of heap memory waiting to be freed, and therefore unused.

The traversal would be simpler if list objects were represented using fixed sized blocks of contiguous memory cells, as suggested, for example, for the implementation of exception tables in the BLAST architecture [Sohi85a]. The disadvantage with this scheme is the wastage of memory through internal block fragmentation, i.e. large portions of these fixed sized blocks being empty. Since SMALL list accesses involve splitting the object at the lowest level, there could be considerable internal fragmentation depending on the choice of block size. Using blocks of a few fixed sizes solves this problem to some extent but complicates other aspects of heap management; it now becomes more difficult to allocate blocks and to manage the free heap space.

4.3.3.2. Merging and Splitting List Objects

Recall that the LP requests the heap controller to **split** the object resident at a specified address on the first access to every list. Subsequent accesses to that object get satisfied from the information in the LPT. The amount of splitting that takes place will depend on the extent to which list accesses are satisfied by the information already contained in the LPT. However, even if splitting is a low frequency operation it must be performed as quickly as possible. When the LP

sends a **split** request to the heap controller, it must wait for the return information from the heap controller before in turn returning a value to the EP. This is necessary since the accessed value might be an atom, in which case a type tag must be included in the value returned by the LP to the EP; this information must come from the heap controller during the splitting process.

Splitting objects represented using two pointer list cells is simple. To split the object at address *X* the heap controller simply returns the values of the 2 pointers and frees the list cell at address *X*. Splitting is more complicated in an exception table represented list heap. A split involves scanning each entry of the exception table corresponding to the object, copying each into one of two new exception tables depending on which sub-tree of the parent list object it belongs to. This is direct fall-out from the fact that the exception table representation aims for compactness by encoding structure information in each table entry. The more compact a representation scheme is the more difficult it becomes to split list objects.

Merging two pointer list cell objects, say at addresses *X* and *Y* is, like splitting, a simple operation. A simple merging algorithm would allocate a new heap cell, say at address *Z*, set its *car* and *cdr* fields to *X* and *Y* respectively and return *Z* to the LP. Note that this value would be returned to the LP as soon as cell *Z* is allocated, so that the LP could continue while the fields of the newly allocated list cell are being set in the heap memory. In an exception table represented heap, to merge exception tables *X* and *Y* a simple merging algorithm would allocate a new exception table, *Z*, with just two entries in it - forwarding pointers to *X* and *Y*. While these entries of exception table *Z* are being set, the heap controller would return the address *Z* to the LP enabling LP computation to continue. This makes merging quick at the expense of increasing the number of forwarding pointers and the amount of internal fragmentation in the exception table memory.

4.3.4. Instruction Set and Function Compilation

To get a better feel for how Lisp functions get evaluated we developed software tools to emulate their execution on SMALL. We assumed that Lisp code is preprocessed into an executable form by a program transformer (which we will call a compiler). Specifying a complete instruction set for SMALL was not our goal in this experiment - we leave this for future work. This was intended as an exercise to suggest what the SMALL instruction set might look like and how Lisp functions might be compiled into it. We therefore started by defining a basic set of primitive functions that the compiler would accept. The set included the basic list manipulating primitives (**car, cdr, cons, rplaca, rplacd**), **cond** and **prog** constructs (along with a **goto** function and **label** specification), predicates (**atomp, nullp, equalp, greaterp**), arithmetic functions (**add, subtract, multiply, divide**), logical functions (**and, or, not**), assignment (**setq**), input and output functions (**read, write**), function definition (**def**) and return. This subset is comparable to Lisp 1.0. Only the simple function call described in Chapter 2 was supported. In this simple Lisp the only valid numeric type is integer.

The compiler accepts a file containing a function call and any number of function definitions. It scans the input file using *lex* and generates code for each function by traversing the function definition tree, producing code for a node

when code has been produced for all of its children, and backpatching forward calls when the function definition is encountered. Code was generated for a stack machine with the list manipulating functionality of SMALL. The instruction set included instructions for function call and return, adding a new binding to the environment, looking up the current value bound to a name and pushing it on top of the stack, pushing immediate values onto the stack, input and output, list manipulating operations, arithmetic and logical operations, unconditional branching, and conditional branching based on predicate testing of the current value on top of the stack. We emulated the code produced by this compiler to test its correctness. The emulator operated by tracing the state of three key SMALL structures: the stack (control and environment), the LPT and the heap.

We present two examples of how a function gets compiled into this stack machine instruction set. Figure 4.14 shows a function that computes the factorial of a number and the stack machine code that it compiles into. Figure 4.15 shows an example of how list manipulation and function calling are compiled.

```
(DEF fact(x)
   (COND
      ((= x 0) 1)
      (T (* x (fact(- x 1)]
```

```
fact:   BINDN      x      / bind argument to name 'x'
        PUSHSTK    1      / push value of argument 1 (x) onto stack
                          / argument 'x' at a known stack location
        PUSHSYM    0      / push constant 0 onto stack
        NEQUALP    labl   / goto 'labl' if top 2 stack values unequal
        PUSHSYM    1      / push constant 1 onto stack
        FRETN             / return
labl:   PUSHSTK    1      / push value of argument 1 (x) onto stack
        PUSHSTK    1      / push value of 'x' onto stack
        PUSHSYM    1      / push constant 1 onto stack
        SUBOP             / subtract TOS from TOS-1 and push result
        FCALL      fact   / recursive call to fact
        MULOP             / multiply TOS and TOS-1 and push result
        FRETN             / return
```

Figure 4.14. Factorial function.

```
(DEF print(junk)
   (WRITE (CDR junk)]

(DEF doit()
   (PROG(list)
      (READ list)
      (print list)
      (SETQ list (CDR (CDR list)]
```

```
print:  BINDN     junk    / bind argument to name 'junk'
        PUSHSTK   1       / push value of 'junk' onto stack
        CDR               / cdr of list identfier at TOS
        WRLIST            / write value returned from CDR
        FRETN             / return

doit:   BINDN     list    / bind stack item for PROG variable 'list'
        RDLIST    1       / read list into 'list'
        PUSHSTK   1       / push value of 'list' onto stack
        FCALL     print   / call 'print'. Argument on the stack
        PUSHSTK   1       / push value of 'list' onto stack
        CDROP             / cdr of list identfier at TOS
        CDROP             / cdr of list identfier at TOS
        SETQ      1       / assign TOS value to 'list'
        FRETN             / return
```

Figure 4.15. List manipulation and function calling.

Our compiler, instruction set and emulator provide guidelines for the development of a more complete SMALL Lisp implementation.

4.4. Summary

The SMALL Lisp machine architecture is made up of two main processing elements, one dedicated to heap activity (List Processor) and the other dedicated to function evaluation (Evaluation Processor). The special-purpose hardware in the LP should lead to efficient heap management. The EP-LP partition makes concurrent evaluation and list access possible.

In the LP, the LPT captures that subset of list structure nodes that is being actively accessed. Recalculation of Lisp access primitives is thus made unnecessary by storing some attributes of a list object in its LPT entry. Further, the

organization deals effectively with temporary cons cells. This is a major part of traditional garbage collection activity. The SMALL organization manages these temporary cons cells efficiently since cons cells are created as LPT entries and cease to exist when their reference counts go down to zero. So, transient cons cells soon disappear while permanent cons cells survive.

While we have explained the rationale behind various SMALL features, it is not clear how efficiently they will perform in practise. We address this issue through a more detailed evaluation in the next chapter.

Chapter 5

A SMALL Evaluation

5.1. Introduction

The art of evaluating a computer architecture typically involves a wide variety of techniques that are used at different stages of its development. Early paper design ideas can sometimes be evaluated using analytical models. When this is not possible more qualitative techniques must be used. Once the details of major parts of the architecture have been finalized, simulation is a powerful evaluation tool. Further on in the design process, an actual implementation of the architecture can be evaluated by timing the execution of benchmarks programs; the choice of benchmarks is usually guided by the nature of the features that are being evaluated. In this chapter we describe our evaluation of the SMALL architecture.

5.2. Quantitative Evaluation

We first describe our quantitative evaluation of SMALL's chief features: the LPT (translation table) and basic LP list manipulation. The possible problems that we foresee with these features are:

(1) frequent true LPT overflow,
(2) excessive reference count modification activity, and
(3) low LPT hit rates.

5.2.1. Simulation Set-up

To investigate the issues listed above we used 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). During typical interpreted runs of each program we caused trace information to be written on entry to and exit from (a) each Lisp primitive (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.

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, but would be mistaken for each other. 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 from Chapter 3 that these lists are not all independent

since there is locality of reference in Lisp programs. 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 is either the value of some existing non-local variable, or that of some existing local variable. One of these possibilities is selected for the list argument based on a probability distribution specified as a parameter to the simulator.

We implemented this by first pre-processing the trace files. Each list argument was replaced by 2 integers: a unique identifier, and a chaining flag. Lists that look identical are allotted the same unique identifier. The chaining flag was set to 1 if the list argument happens to be the value returned by the previous call in the trace. Otherwise it was set to zero. In the simulator, in deciding the argument of a particular primitive, the chaining flag is first examined. If it is 1, the argument is assumed to be available on top of the simulated run-time stack. Otherwise, a local or non-local variable is selected based on a set of probability distributions described later in this section.

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 5.1 characterizes this aspect of the 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, 6 simulator parameters can be specified: (1) ***TableSize***, (2) ***OverflowPolicy***, (3) ***ArgProb***, (4) ***LocProb***, (5) ***BindProb***, and (6) ***ReadProb***. ***TableSize*** is the number of entries in the LPT, and ***OverflowPolicy*** s the strategy to be employed on pseudo overflow (either Compress-One or

Table 5.1. Content of the 4 Traces.

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

Compress-All). The other four parameters specify probabilities to be employed in the random selection of arguments to the list manipulating Lisp primitives. They are 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 chosen from among the members of the selected class. In order to enhance the realism of the simulation we did not always use the value currently bound to that variable. The current value was used with probability *1 - ReadProb*; with probability *ReadProb* we assumed that a new list object had been read in and bound to that variable since it was last accessed. The primitive function was then evaluated using the argument thus selected, 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*). Note that the four probability parameters are thus used to decide three orthogonal issues:

(a) whether the primitive argument is a function argument, local or non-local variable (decided by *ArgProb* and *LocProb*) ,

(b) whether the variable thus selected has been read into since last accessed (decided by *ReadProb*) , and

(c) whether the return value should be bound to a variable or just pushed onto the top of the stack (decided by *BindProb*).

For all the runs reported, the probability parameters were set at 0.6, 0.3, 0.01, 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 (leaving only 10% as non-locals), that 1% of all arguments are to lists that got read into variables during the execution of the program (i.e. not during the initialization of the program), and that 1% of all primitive function return values get bound to a variables on the stack (as against just being pushed onto the stack). We consider these parameter settings as reasonable for Lisp based on our benchmark set. The sensitivity of the simulations to parameter variations is discussed in Section 5.2.6.

The simulator used results from our own studies and from Clark's studies to determine the structure of the lists being manipulated. This information was required to perform **splits** and **merges** of heap objects. In doing a **split** the simulator assigned addresses to the car and cdr parts of the list object based on pointer distance distributions from Clark's studies. The actual size of each part was assigned using the **n** and **p** distributions from our studies, as plotted in Chapter 3.

5.2.2. LPT Size Requirements

The first two concerns expressed earlier in this section relate to LPT over-flow 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. If we provide a sufficiently large table, overflow will rarely occur. Our

first evaluation goal was to determine the expected LPT size requirements. Figure 5.1 shows sample results from the studies on required table size.

Each trace was run through the simulator with increasing values of the LPT 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 trace of 160,933 references (the trace that was not plotted in Figure 5.1) 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 slope of 1 passing through the origin, connected to a horizontal line. Such a curve will occur for any program trace. The points on the sloping part of the curve represent simulation runs in which pseudo overflows occurred; the peak LPT usage observed was equal to the LPT size. The Compress-One policy was used in LPT compression in these runs. The knee of each plot occurs at that LPT size for which no form of overflow (neither true nor pseudo overflow) occurs. It represents the minimum LPT size required for the trace to run without an overflow occurring. Increasing the LPT size beyond this value does not affect the observed peak LPT usage; this is seen in the horizontal part of the plot after the knee. The curve for the Lyra trace is identical in shape to the three curves of Figure 5.1 but was not plotted for scaling reasons. For the Lyra trace the maximal number of table entries needed was about 2000.

Figure 5.2 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 5.1) 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 5.2 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 5.2. 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 (recall that we concluded this from the curves of Figures 3.5 and 3.6 in Chapter 3); this is not entirely due to trace length. From this graph we conclude that in a translation table with 2K or 4K entries even pseudo overflows would rarely occur.

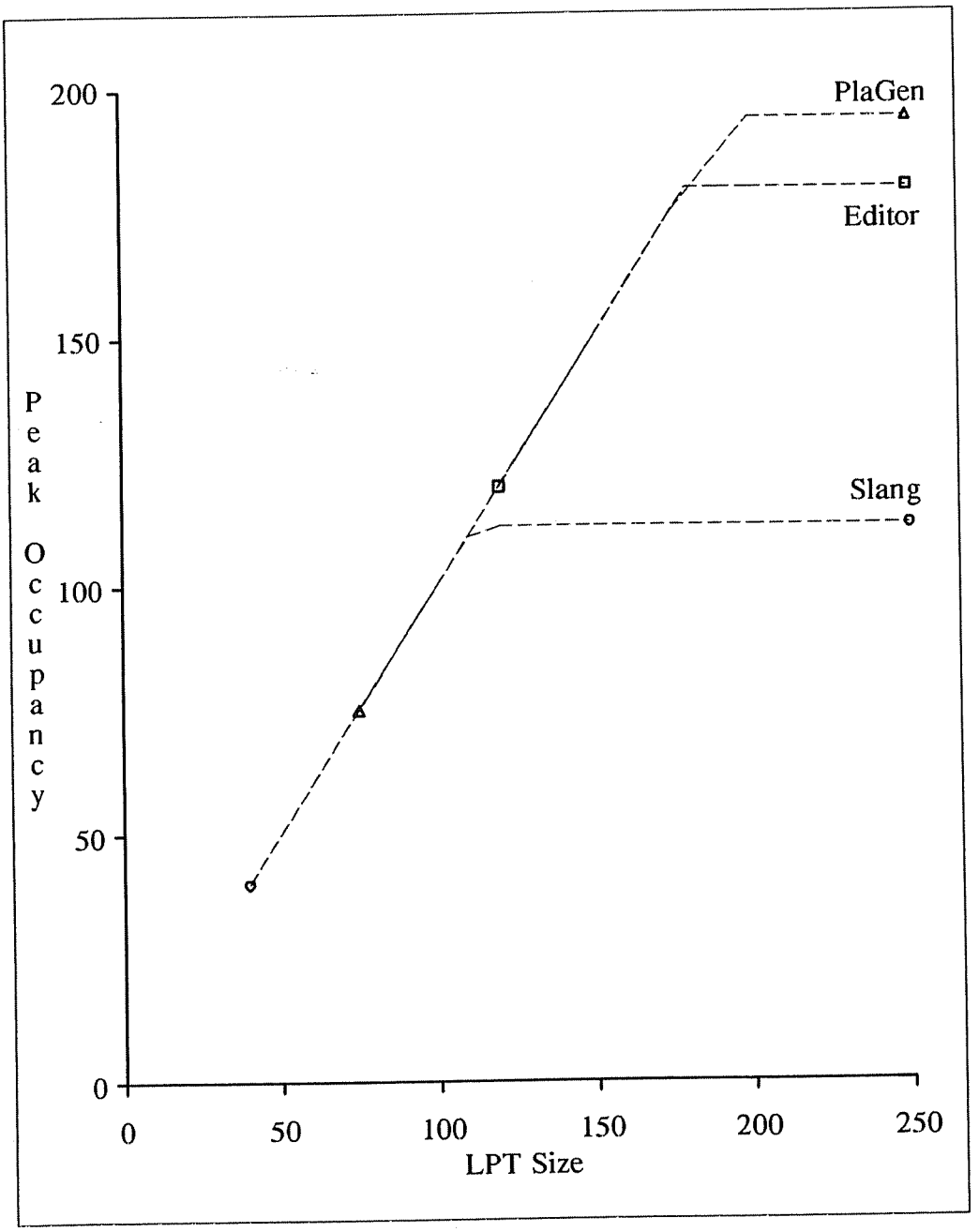


Figure 5.1. Peak LPT Usage Behaviour.

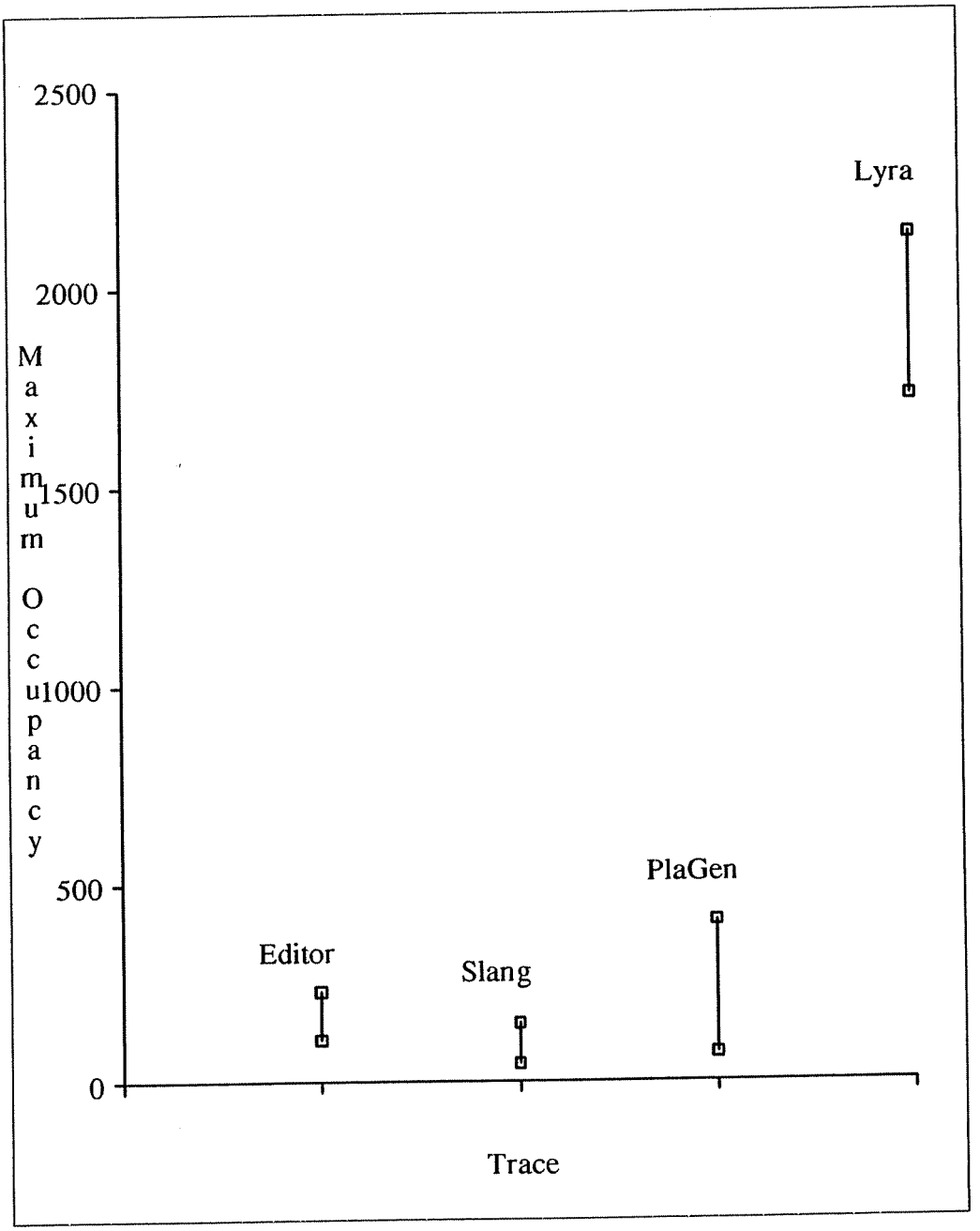


Figure 5.2. Maximum LPT Occupancy Levels.

5.2.3. 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). Under the Compress-One compression policy, when a pseudo overflow occurs the LPT is scanned for compressible entries. The first compressible pair of LPT entries found is compressed releasing two LPT entries and normal LPT activity is resumed. Under the Compress-All policy, on the other hand, the entire LPT is scanned for compressible entries and compressed when a pseudo overflow occurs. Figure 5.3 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 5.3 compared to the straight line behaviour of maximal occupancy as plotted in Figure 5.1. 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 true 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.

5.2.4. 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 5.2 summarizes measurements of the degree of LPT activity. The columns in Table 5.2 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 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 5.2 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.

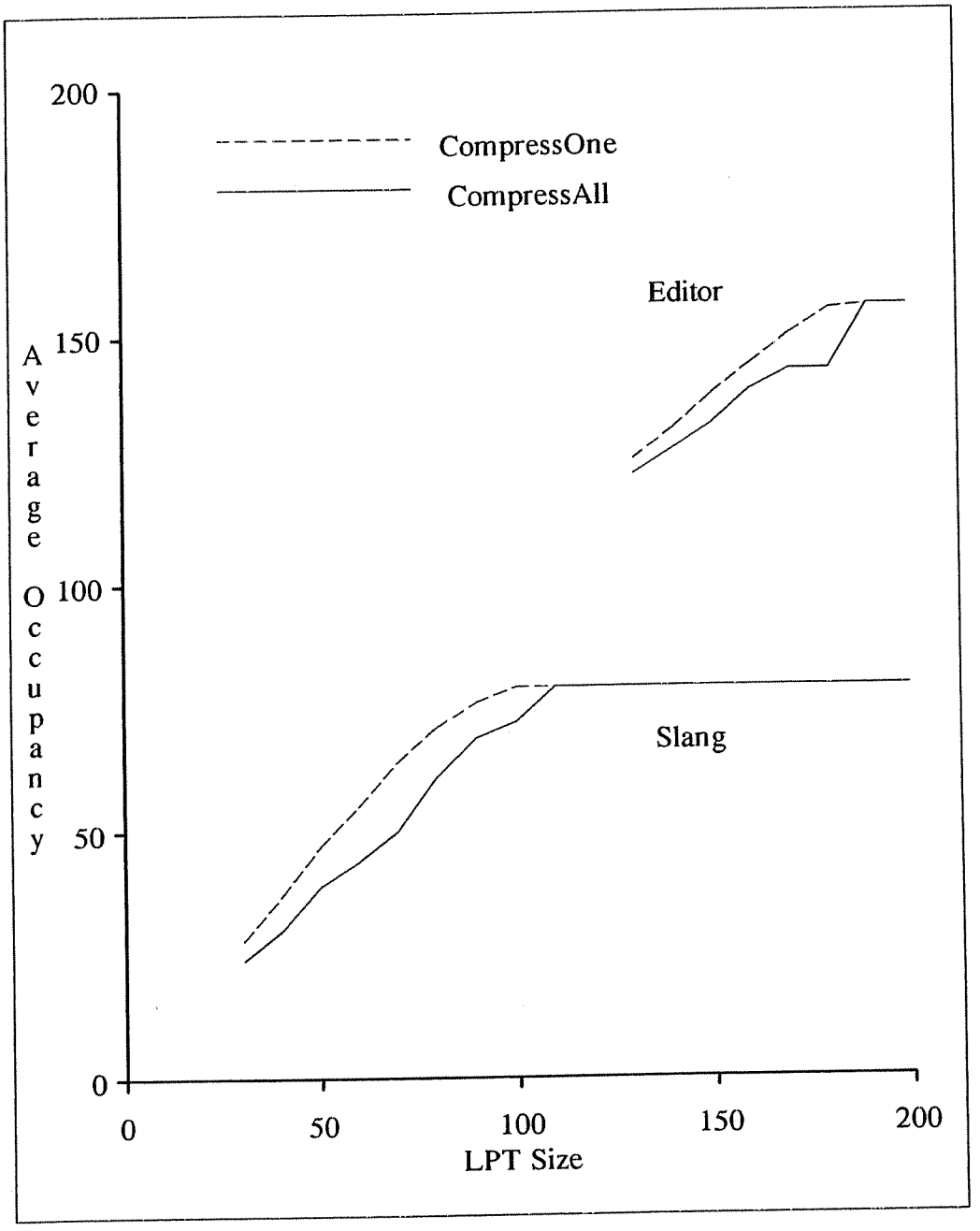


Figure 5.3. LPT Behaviour and Pseudo Overflow Policies.

Table 5.2. LPT Activity.

Trace	Refops	Gets	Frees	RecRefops
Lyra	170232	29746	23006	213532
PlaGen	92414	7248	6971	106216
Slang	6852	1794	573	9580
Editor	4585	233	30	6749

Note that the LPT maintenance costs are not excessive; using the information about the content of the traces from Table 5.1 we see that the statistics in Table 5.2 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 due to reference count activity over the EP-LP bus. To evaluate this scheme we modified the simulator to keep track of reference counts using a reference count table in the EP in addition to the counts maintained in the LPT. The extent of the reduction in EP-LP traffic is illustrated in Table 5.3. The table shows how the number of reference count updating operations changes from the old scenario, where all reference counts are maintained in the LPT (labeled **Then**), to the scenario, where reference counts of pointers in the stack are maintained in a table in the EP (labeled **Now**). It also shows how the maximum values that these reference counts reached. From the near order-of-magnitude difference between the **Then** and **Now** columns, it is clear that by splitting the reference count up as described above the traffic over the EP-LP bus can be greatly reduced. Further, the maximum sizes of the reference counts can be used in deciding how large the count fields should be made.

Table 5.3. Evaluation of Split Reference Counts.

Trace	Refops		MaxCounts	
	Then	Now	In LPT	In EP
LYRA	170232	17905	8	47
PLAGEN	92414	6258	10	39
SLANG	6852	1363	7	39
EDITOR	4585	471	64	82

5.2.5. LPT vs. Cache

The use of the LPT requires some non-trivial amount of memory to store information about recently referenced lists. This is in fact a service that could be provided by a data cache. We decided to evaluate whether the LPT or a data cache was a more efficient use of memory by comparing the hit rate for each type of structure. Naturally, one must be cautious when interpreting these results since the LPT provides much more functionality than a cache. With a cache, more memory accesses are required for bookkeeping functions than with an LPT. None of these bookkeeping activities are accounted for in the results reported below.

In this evaluation, we considered a fully associative, LRU replacement data cache with the same number of entries as the LPT. Since our traces do not include any actual address information we had to generate an address for each list reference with which to drive the cache simulation. This was done as follows. We maintained a counter that represented the next address to be used; it was initialized to zero. Whenever a new list reference was encountered in the simulation, a size was assigned to it based on our **n** and **p** distributions of Chapter 3. The value of the counter was assigned as the address of that list reference, and remembered by associating it with the LPT entry corresponding to that list reference (so that the next time that list reference was encountered the address would be available for the cache simulation). The counter was then incremented by the size. When an object was accessed (split), addresses were assigned to its ***car*** and ***cdr*** based on the ***car*** or ***cdr*** pointer distances listed in Clark's thesis [Clar76a], and calculated as an offset from the address of the object itself. Using this procedure we were able to generate an address for each list reference in the trace. The simulator kept track of the contents of the cache based on these addresses.

A 2 pointer list cell was assumed to be the cachable unit. Table 5.4 shows sample results from a comparison where each cache line is the size of one such 2 pointer list cell and no prefetching is attempted. The miss counts in the table represent the number of times **car (cdr)** requests were not satisfied by the ***car***

Table 5.4. Comparison with Data Cache.

Trace	Size	LPTMisses	HitRate	CacheMisses	HitRate
Lyra	1700	8221	97.27	12875	95.73
	2000	5682	98.12	9514	96.84
	2300	4314	98.57	7679	97.45
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

(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. In almost all of the runs quoted, the cache misses outnumber table misses by at least a factor of 2. 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 5.4 continued. Figure 5.4 shows the relative hit rates for cache and table for one of the traces (SLANG). The runs represented by Figure 5.4 were generated using a different random number sequence than was used in producing the results of Table 5.4. As a result, the two sets of results do not correspond exactly though they show the same general behaviour. The cache and LPT hit rates are shown for different values of total cache/LPT size.

Since there is spatial locality of reference in Lisp access streams, a cache designer would opt for a more complicated cache structure than the unit cache line size structure that we used in our simulation model. Cache hit rates would be higher with line sizes greater than 1 due to the pre-fetching that would take

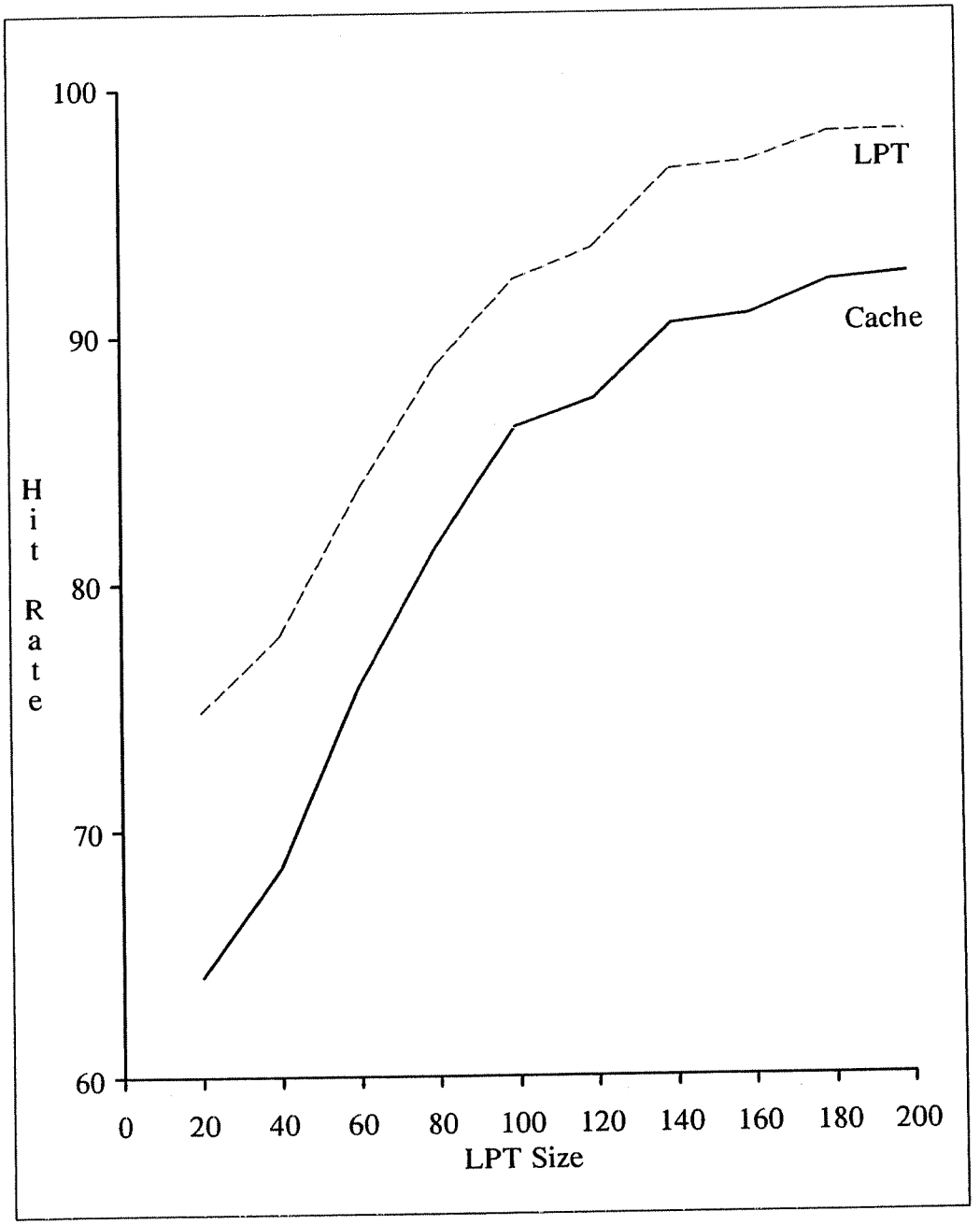


Figure 5.4. Hit Rates for LPT and Data Cache with Slang Trace.

place into a cache line. We therefore studied the relative hit rates for data cache and LPT using a modified cache model. In this new model the cache was fully associative, with an LRU replacement policy and of the same size as the LPT as before. We assumed in addition that each cache entry is half the size of each LPT entry, which is more realistic than the previous model of equal entry sizes. We varied cache line sizes from 1 to 16 entries.

Sample results for the Lyra, Slang and Editor traces are shown in Figure 5.5. The graph shows the ratio between the data cache miss rate and the LPT miss rate, plotted against data cache line size. For each setting of the cache/LPT size parameter, these miss ratios were estimated for increasing cache line sizes. Increasing the cache line size with a fixed cache size results in a smaller number of cache entries; while the cache size remains the same, the cache is configured differently. Each trace was run with increasing values of cache/LPT size. Thus, the 4 lines for the SLANG trace in Figure 5.5 represent 4 different values of the cache/LPT size parameter. Several points must be noted in interpreting these graphs. Based on our earlier cache comparison study, one might expect the LPT to outperform the cache; the ratio of the cache miss rate to the LPT miss rate should then be greater than one. Instead, in Figure 5.5 the ratio varies from 0.7 to 2.8, with several points below 1. This is due to the modified cache model - since each LPT entry is twice the size of each cache entry there are half as many LPT entries as there are cache entries, which results in more cache hits relative to the previous study. Once again, we would point out that the results of this cache comparison should be assimilated with caution since the LP provides far more functionality than a simple data cache. As before, cache performance improves relative to LPT performance as the size of the LPT (and cache) increases; this is seen by the fact that the miss ratio line for each of the three traces is lower with a higher LPT (and cache) size. In some of the curves, the improvement in cache performance improves upto a point and then starts falling off. This is governed by the degree of structural locality in the trace. A trace that shows high structural locality will show improving cache performance with increasing cache line size due to the prefetching achieved by fetching a whole cache line at a time from the heap. The LP, on the other hand, does only limited prefetching into the LPT (in that it fetches both ***car*** and ***cdr*** in splitting an LPT entry), and so the ratio of cache misses to LPT misses decreases. This trend continues for increasing cache line size only upto the point where useful data is being prefetched, which is dependent on the extent of structural locality shown by the trace. Based on this explanation, the Editor trace shows the highest degree of structural locality.

5.2.6. Sensitivity to Parameter Changes

Recall that our simulator accepted a set of probability parameters that were used to select arguments to the primitive function calls in the traces. In all of the experiments described, the same setting of these parameters was used, viz, **ArgProb** = 0.60, **LocProb** = 0.30, **BindProb** = 0.10, and **ReadProb** = 0.10. These values were chosen based on observations that we made during the selection of the 4 Lisp benchmarks.

To evaluate how critical the choice was, we conducted a limited investigation of how sensitive our observations are to variations in the settings of the

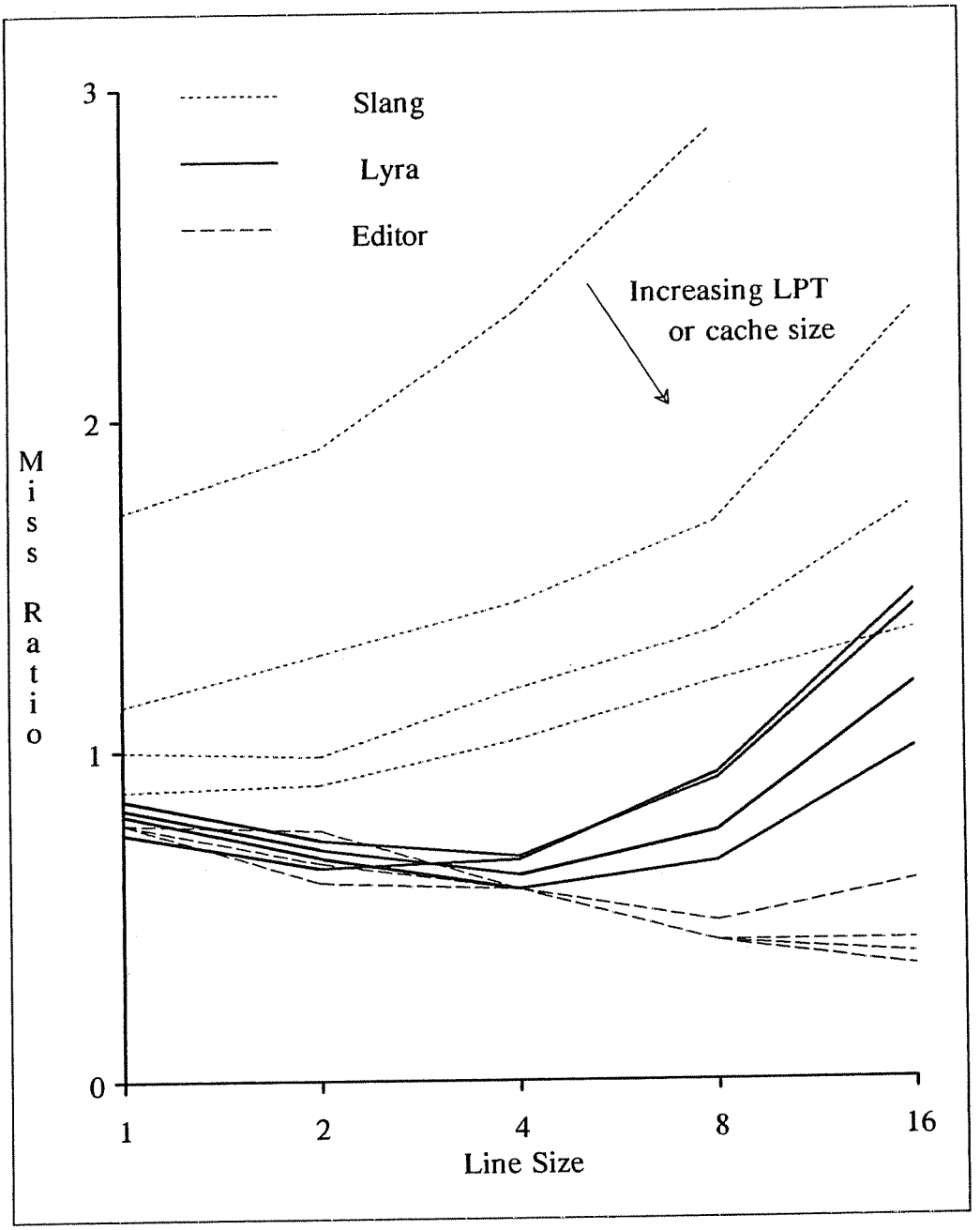


Figure 5.5. Ratio of Cache Misses to LPT Misses versus Line Size.

probability parameters. We ran the SLANG trace through the simulator with 5 different settings of the parameters. The five runs were

(1) a control run that used the parameter values listed above.

(2) a high **ArgProb** run with all parameters as in the control run except for **ArgProb**, which was raised to 0.85, and **LocProb**, which was lowered to 0.125 to compensate.

(3) a high **LocProb** run with all parameters as in the control run except for **LocProb**, which was raised to 0.60, and **ArgProb**, which was lowered to 0.30 to compensate.

(4) a high **BindProb** run with all parameters as in the control run except for **BindProb**, which was raised to 0.03.

(5) a high **ReadProb** run with all parameters as in the control run except for **ReadProb**, which was raised to 0.03.

Sample results are shown in Table 5.5. Clearly, the measures fluctuate by small amounts. What the table does not show is that the general trends observed during the course of this chapter do not change; for each setting of the parameters we re-ran the experiments described in this chapter (on the SLANG trace), and observed that the graphs did not change in general shape, but were shifted by small amounts of the order of the fluctuations illustrated in Table 5.5. This degree of sensitivity does not overly concern us, though it might if we were doing a more detailed simulation of a more completely specified design.

Table 5.5. Sensitivity of Simulation to Probability Parameters.

Statistic	Control	HiArg	HiLoc	HiRead	HiBind
Ave LPT Count	49	50	51	52	51
Max LPT Count	64	64	64	64	64
LPT Hits	2755	2725	2783	2622	2768
Cache Hits	2765	2730	2786	2630	2770
Max Refcount	29	34	24	30	39
Refops	12062	12127	12060	12088	12229

5.3. Discussion

In this section we discuss the impact of the SMALL architecture on three key aspects of Lisp execution: accessing list data, managing the list heap, and support for function calling.

5.3.1. Accessing List Data

The average speed of list access in SMALL depends on the percentage of accesses that are satisfied by the information in the LPT. A successful LPT access returns a result in the time that it takes to index into the LPT and reference to the relevant field of the LPT entry. We studied this "LPT hit ratio" through simulation earlier in this chapter. Other accesses result in splitting activity in the heap, followed by the setting up of LPT entries for the two children of the split object, before a result can be returned to the EP command that initiated the access activity. The actual difference in speed between these two types of access is, of course, highly dependent on how the LP algorithms are implemented.

One accessing question that we can address without simulation is how efficiently list traversals can be performed on the SMALL architecture. We define a *list traversal* as a sequence of accesses to several elements of the same list, and an *ordered traversal* as a traversal in which each element of a list is accessed once in some canonical order based on a structural relationship. Examples of ordered traversals are in-order, pre-order, and post-order traversals. We further define a *random traversal* as one in which extensive (not necessarily exhaustive) accesses to a particular list are made, but in a random or repetitive order.

Not much can be said about random traversals on SMALL. When an access is made to a list node that is not represented in the LPT a split takes place. Future accesses to that node can then be performed by simply referring to the LPT entry. The larger the number of such repetitive accesses in a traversal the lower the total traversal time. Ordered traversals, on the other hand, correspond to highly predictable access streams and therefore lend themselves to more detailed analysis.

In discussing ordered traversals we will assume that an s-expression maps into a binary tree in which symbols are represented by leaf nodes as shown in Figure 5.6. The figure shows how the list **(((A B) C D) E F G)** is represented in a binary tree. Leaf nodes correspond to symbols in the list. Internal nodes describe the structure of the list. This is just one of several possible tree representations of lists. For a tree rooted at node T the three ordered traversal mentioned above are defined as follows:

```
Pre-order(T):
        Visit(T)
        Pre-order(Left Subtree of T)
        Pre-order(Right Subtree of T)

In-order(T):
        In-order(Left Subtree of T)
        Visit(T)
```

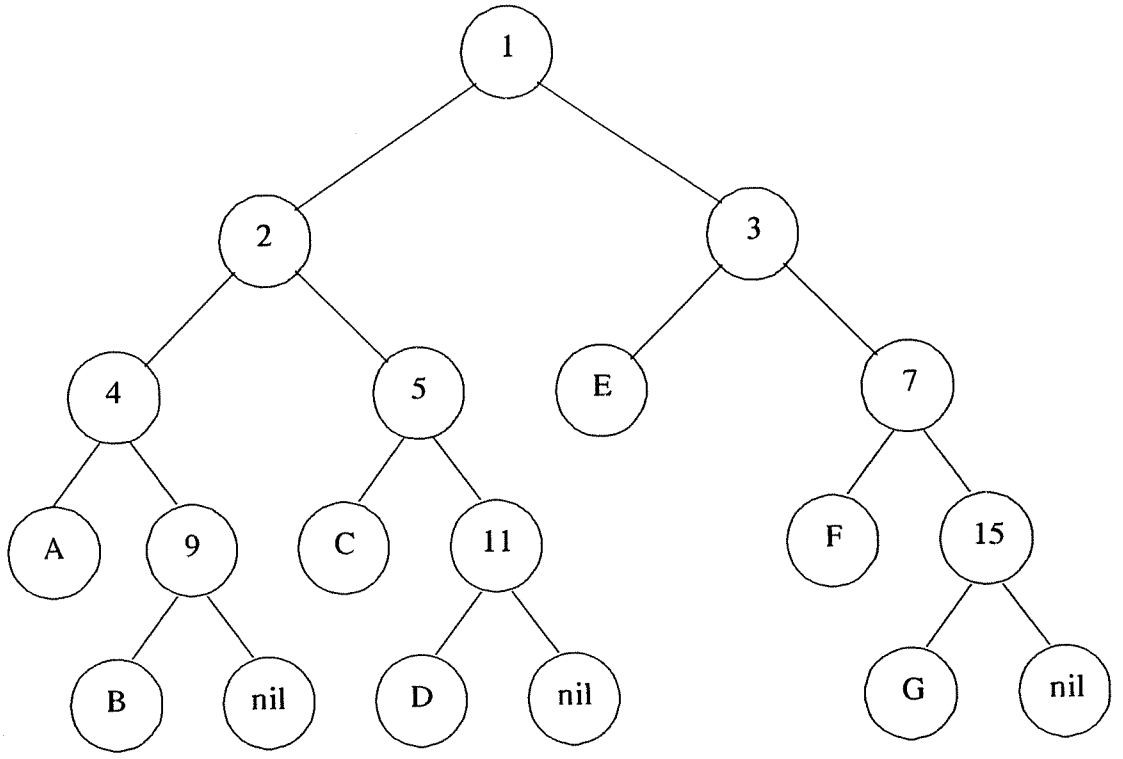


Figure 5.6. Tree Representation of the list (((A B) C D) E F G).

In-order(Right Subtree of T)

Post-order(T):
Post-order(Left Subtree of T)
Post-order(Right Subtree of T)
Visit(T)

The order in which the nodes of the tree are visited are

Preorder: **1 2 4 A 9 B 5 C 11 D 3 E 7 F 15 G**

Inorder: **A 4 B 9 2 C 5 D 11 1 E 3 F 7 G 15**

Postorder: **A B 9 4 C D 11 5 2 E F G 15 7 3 1**

These are sub-sequences of the actual super-sequence in which the nodes are touched in performing the traversal. In all three cases the traversal super-

sequence is

1 2 4 A 4 9 B 9 4 2 5 C 5 11 D 11 5 2 1 3 E 3 7 F 7 15 G 15 7 3 1

Note that each internal (i.e. non-leaf) node is touched 3 times. The pre-order traversal sub-sequence visits each internal node when it is first touched, while the in-order traversal sequence visits internal nodes on the second contact, and the post-order traversal visits an internal node on the third (and final) contact. For all three traversals, if we assume that the LPT does not overflow during the traversal, the number of **splits** that will occur is equal to the number of internal nodes in the tree. The LPT entry corresponding to an internal node is **split** after it is first contacted during the traversal. The earliest that it can be **merged** with its sibling is after their descendents have been traversed and **merged**. So, exactly the same sequence of LPT entry **splits** and **merges** will occur in each of the ordered traversals.

A list with **n** atoms and **p** internal left parentheses has **n+p+1** leaf nodes (**n** atomic leaves and **p+1** nil leaf nodes) and **n+p** internal nodes, with a total of **2n+2p+1** nodes in the tree. So, a complete traversal of the list without LPT overflow requires **n+p splits**. Recall that each internal node is accessed three times and each leaf node once. There would then be **n+p** LPT misses and **3n+3p+1** LPT hits, or a hit rate of 75%. Notice that this is a guaranteed hit rate since even if pseudo overflow occurs during the traversal, the LPT entries corresponding to the leaf nodes cannot be **merged** as noted above.

5.3.2. Recycling Garbage

As we have seen in Chapter 2, garbage collection has been a primary concern in Lisp systems since heap space is not explicitly deallocated by the user. Garbage is created when all of the references to a list object cease to exist. Garbage collection involves two stages: detecting the presence of garbage *(garbage detection)*, and reclaiming the heap space occupied by it *(garbage reclamation)*.

The SMALL philosophy on garbage collection is that while garbage must be detected as quickly as possible in the LPT, the speed of the reclamation of the heap space is not that critical. Quick garbage detection is greatly aided by the mapping between list identifiers and heap addresses maintained in the LPT. Heap addresses need not be known to detect the existence of garbage; the unique identifier assigned to each list object suffices for this purpose. Having the LP communicate with the EP in terms of object *identifier*s instead of heap addresses also enables changes in the environment to be signaled to the LP efficiently. Garbage is then detected almost immediately after its creation - when a reference count goes to zero in the LPT. Garbage reclamation is initiated by the LP, and it takes place in two stages. First the LPT space corresponding to the new garbage is reclaimed by the LP. Then the actual space occupied by the object in the heap is reclaimed by the heap controller. Transient cons cells get reclaimed quickly since they get allocated in the LPT, not in the heap. These list cells form a large part of the garbage collection load of traditional Lisp systems; a study of the symbolic algebra system Macsyma [Fode81a] revealed that only 1 out of 385 cons cells allocated did *not* become garbage soon after they were allocated.

This garbage collection policy is concurrent; there are no periods during which computation ceases while waiting for the garbage collection process to clean up the heap. It is also incremental, in that the garbage detection process

could be considered to be constantly running and not just triggered when heap space runs out. The cost of garbage collection gets amortized over the cost of every LPT activity in the form of reference count activity. The reference count updating and LPT space reclamation strategies described in Chapter 4 ensure that the garbage collection related overhead added to each LPT operation is not too large.

5.3.3. The Ecology of Function Calling

There are two major book-keeping overheads in Lisp execution: managing the heap space and managing the dynamic name binding environment. SMALL has a special purpose processor (the LP) to cope with the heap space management overhead; the Evaluation Processor must see to the maintenance of the environment. The speed of function calling in SMALL is dependent on how fast the incremental changes to the environment corresponding to the call can be made. Recall that in our discussion of the EP and the environment in Chapter 4 we did not make a strong case for either deep bound or shallow bound implementations of environments. In either implementation there is a burst of EP-LP activity on every function call, as new bindings get made and added to the environment (in the case of a call), or bindings are discarded from the environment (in the case of a return), which translates into reference count activity in the LP: incrementing reference counts on a function call and decrementing reference counts on a function return. The EP need not wait for these operations to complete before continuing with its function evaluation work. Even if there are several such consecutive reference count bursts, as would happen if several small functions were called one after the other, the LP will not fall far behind the EP, since updating a reference count is a simple operation that can be implemented efficiently.

5.4. Summary

The SMALL 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 was seen to be adequate to capture the list activity of our Lisp program traces. True LPT overflow will occur infrequently with a table of this size. In the handling of pseudo LPT overflow 2 policies were studied, Compress-One and Compress-All. The Compress-One 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. Our LPT/cache comparison revealed that the LPT captures the temporal locality of Lisp list access comparable to an LRU data cache. Further, from our studies of the amount of reference count updating that takes place, it appears that reference counting is a viable heap management scheme in the SMALL environment. Reference counting provides a mechanism for garbage to be detected concurrently with LPT activity, and is guaranteed to detect garbage soon after it is created.

Chapter 6

A SMALL Multilisp

6.1. Introduction

Over the past few decades, advances in technology and design have produced computers with steadily increasing performance potential. At the same time, the complexity of the problems being solved on these machines has been increasing. As part of this trend, when it was detected that the processing power of sequential, uniprocessor architectures was no longer adequate, achieving increased computing power through larger numbers of computing elements and concurrent computation has become an important issue. Efforts to enhance Lisp performance in this way have been underway for some time now [Guzm81a, Hals81a, Mart83a, Sugi83a]. More recently, proposals have been made for expressing concurrency in Lisp programs [Gabr84a, Hals84a]. We use the term *Multilisp* to describe the language supported by such multiprocessing Lisp systems.

Before discussing Multilisp systems in particular, we recall the communication techniques employed in general multiprocessing systems. Any multiprocessing system must make provision for processors to communicate with each other to achieve their common goal. This communication can either be done through shared memory, or by inter-processor message traffic. In multiprocessors like C.mmp [Harb81a] and Cm* [Swan77a] at CMU, the NYU Ultracomputer [Gott83a], or the BBN Butterfly, inter-process communication is conducted through shared memory. In more decentralized systems, like the Apiary machine [Hewi80a] and Concert multiprocessor [Hals84a] at MIT, the AMPS project [Kell79a], the Intel iPSC[1] (and other hypercube architectures), the Connection machine [Hill85a], or the Bath Concurrent Lisp Machine project [Mart83a] the multiprocessor system is made up of several processors, each with its own memory, which communicate with each other via messages.

In this chapter we discuss some of the issues involved in designing a Multilisp system. We also indicate how the SMALL architecture can be extended to provide a suitable environment for such computation.

6.2. Parallelizing Lisp

Whatever the organization of the Multilisp system may be, there are two interesting sets of problems involved in programming it. The first set of problems is associated with determining what the parallel activity should be, and how it should be distributed over the available processors. The second set of problems is concerned with system run-time support, including both heap and environment maintenance.

---

[1] iPSC is a trademark of Intel Corp.

6.2.1. Detecting the Scope for Parallelism in Lisp Code

In Chapter 2 we saw that some experimental Lisp systems attempt to evaluate the arguments of a Lisp function in parallel [Guzm81a, Yama83a]. These efforts are aimed towards taking advantage of the *implicit parallelism* present in Lisp code. Often there is scope for further parallelism that is not immediately obvious from examining the program code, but which is known to the programmer. Efforts have been made to make it possible for programmers to express such instances explicitly. This is done by extending the language with features for describing *explicit parallelism.* We next discuss these two kinds of parallelism.

6.2.1.1. Implicit Parallelism

Recall that in Lisp execution, a function's argument list must be evaluated before the body of the function can be evaluated. In sequential Lisp this argument list is evaluated from left to right. Frequently, these arguments can be evaluated in parallel; this is one source of implicit parallelism in Lisp. There are other potential sources of parallelism in Lisp code. One example is the conditional construct, **cond**, which is made up of a series of *(condition, body)* tuples called *cond-legs.* **Cond** is evaluated by evaluating the conditions one by one from left to right until one returns a non-nil value, and returning the value of the body of that cond-leg. All of the conditions could potentially be evaluated simultaneously in parallel, just like function arguments. Note, however, that it is essential that parallel argument (or condition) evaluation be consistent with conventional left to right sequential evaluation; parallel evaluation, if performed, must not violate the semantics of sequential Lisp.

How can this consistency be ensured? In parallel argument evaluation on the Evlis multiprocessor Lisp machine at Osaka University [Yama81a] consistency is maintained by only evaluating function arguments in parallel when it is obvious from the function definitions that the arguments cannot affect each other by altering lists. In implementing this test it is necessary to be conservative, which reduces the number of cases where implicit parallelism can be taken advantage of. Another technique that has been employed is to detect the scope for parallel evaluation by compile time dataflow analysis [Mart80a]. Typically, Lisp compilation is a simple process of storing functions as trees, and then attempting to flatten these trees into a form that more closely resembles ordinary machine code, but which provides support for Lisp like function calling and variable accessing. This includes producing code for accessing local variables at known offset locations in the run-time stack frame, tail recursion elimination, and some peephole optimization techniques. Note that these are all local optimizations, since Lisp functions are typically compiled independently. Marti's compiler uses a less local dataflow analysis technique to detect a large subset of all possible parallel argument evaluation possibilities. The compiler uses a set of heuristics to determine which functions are worth parallelizing, and then computes for each the set of functions that could be run in parallel with it. This would amount to a fair amount of computing for a Lisp program made up of a large number of functions; while it might be worth compiling frequently used Lisp programs in this manner, the scheme is not generally applicable.

In Chapter 2 we saw that in the EM-3 data driven multiprocessor Lisp machine [Yama83a] functions returned pseudo-results in order to spark

additional parallel evaluation. This is an example of *eager evaluation*, where the evaluator seeks to evaluate as much as possible as soon as possible (as opposed to a *lazy evaluator*, which is demand driven, and evaluates a sub-expression only when the result is needed). Even in eager evaluation, care must be taken that the sub-expressions being concurrently evaluated are independent, and do not modify lists used by each other. If they do, the semantics of sequential Lisp would be violated.

6.2.1.2. Explicit Parallelism

The schemes that we have seen are not guaranteed to detect all the implicit parallelism contained in a Lisp program; they might detect an insignificant part of the parallelism that could actually be exploited in certain programs. More extensive, and yet safe, implicit parallelism is difficult to detect [Gabr84a]. The alternative is to extend the language with constructs for expressing concurrency. Several such Multilisp proposals have been made [Hals84a, Gabr84a, Prin80a]. Since the three schemes do not greatly differ semantically, we discuss only one of them in more detail below.

In Halstead's Multilisp [Hals84a], the default form of evaluation is sequential. So, in the call to function *F* with arguments *A, B* and *C*, represented as

(F A B C)

A, *B*, and *C* are evaluated one after the other in the usual left to right order. To force the arguments to be concurrently evaluated, this call is changed to

(PCALL F A B C)

When this construct is evaluated, an implicit 3-way fork takes place (corresponding to the evaluation of the three arguments), followed by a join, after which the body of function *F* is evaluated.

Halstead's Multilisp includes one other extension, called *futures*. The future construct enables the programmer to specify exactly where a pseudo-result (borrowing the terminology of the EM-3) should be generated and returned while an expression is being evaluated. The pseudo-result is called a *future*. Thus, when the construct

(future X)

is evaluated, a future is immediately returned to the caller, which proceeds with its evaluation. Concurrently, the expression *X* is being evaluated, and when that evaluation is completed, the return value takes the place of the future. In the mean time, any process that tries to access the future gets suspended. Thus, the future construct enables controlled concurrency between the computation of a value and its deployment. Note that the *PCALL* construct is more conservative than the future construct. During the evaluation of a *PCALL* the concurrent evaluation of its arguments is guaranteed to complete before the evaluation of the *PCALL* is complete. On the other hand, while the expression in a future is being evaluated an arbitrary amount of evaluation occurs in the rest of the program before the future is required. The future construct thus provides a more flexible source of parallelism.

Constructs for expressing explicit parallelism are gaining in popularity. They leave the task of deciding the degree of parallel activity to the programmer, who best knows how the program is likely to be applied. The programmers expert knowledge can be used to exploit as much parallelism as possible from a

program. On the other hand, the chore of debugging a program is made more difficult by this added complexity; a future in the wrong place could lead to unpredictable run-time behaviour.

6.2.2. Heap Maintenance

We next discuss the second problem that arises in a Multilisp system - maintaining the heap. Heap maintenance can be a major problem in a Multilisp system. A list cell becomes garbage when it is no longer referenced by *any processor*. Thus, garbage collection will involve communication among the processors. The extent of this communication overhead depends on the architecture of the system and the particulars of the heap manager. We will examine two proposed schemes in this section.

Baker's two semispace scheme [Hals84a], which we saw in Chapter 2, is extended in the heap manager described by Halstead for Multilisp on the Concert multiprocessor at MIT. This system is made up of 32 MC68000s sharing 20 MB of memory, but with individual processors having slightly faster "local" access to nearby memory modules. Processors interleave periods of garbage collection with ordinary computation. Each processor has its own oldspace and newspace in its "local" memory. It creates new objects there; with the expectation that this will reduce memory contention, increase the fraction of accesses to "local" memory, and dynamically increase the locality of reference observed. Object relocation can go on in parallel in each processor's garbage collection period but for consistency reasons semispace swapping must be coordinated on all the processors. A "non-local" object reference causes special difficulties if the object is currently being relocated by its processor (in a garbage collection period). These collisions are dismissed as being infrequent and unlikely. In general, to prevent collisions such pointers would have to be locked prior to object updating and objects would have to be locked prior to being relocated. One drawback with the Concert style of heap management is that the strict partition of the heap space over the processors might cause a particular processor to run out of space while there is more memory available elsewhere. A suggested solution is to organize the two semispaces as collections of dynamically allocated parcels rather than as contiguous regions of memory. - when a processor's newspace gets full, it asks for and gets an extension parcel for that space.

Another modification of Baker's scheme has been suggested by Lieberman and Hewitt for the Apiary multiprocessor at MIT [Lieb83a]. This heap manager tries to optimize garbage collection activity by spending more time on short lived objects based on the empirical observation that they yield most of the garbage to be reclaimed. It divides the address space into several regions rather than into two semispaces. Regions are organized into generations. The object relocation strategy sees that long lived objects end up in older generation regions which get garbage collected less frequently than younger generation ones. Here too, each processor is given a strict partition of memory. To take care of inter-processor object references each processor maintains two tables: an *Exit Table* listing all references that this processor makes to other processors, and an *Interest Table* listing the references from other processors to this processor's objects. These tables are "scavenged" along with the local heap when a region in a processor's heap space has to be garbage collected. A related approach to heap management

is described in [Moha84a].

Both of these heap maintenance schemes partition the heap address space, allocating to each processor a portion of the shared heap, which we will call a *heaplet*, to manage. When garbage collection is performed, either interprocessor message traffic is generated (the more the locality of reference in program execution, the more the processors access their own heaplets, and the less the message traffic), or objects/pointers/tables-entries have to be locked. This is because of references made by processors to portions of the heap that are not under their direct control (these references have been variously described as "external" and "non-local" references) and the undesirability of forcing all the processors to do garbage collection at the same time. Given a favourable allocation of tasks over the processors and a high locality of reference, that would not pose much of a problem since the number of such references should be low.

6.3. SMALL Multilisp

In this section we demonstrate how the SMALL organization can be extended into a system that accommodates Multilisp execution. We will not concern ourselves with the problems of how the concurrent Multilisp evaluation is initiated; we have seen that this can be done by either taking advantage of the implicit parallelism in Lisp code, or by allowing the programmer to specify explicitly where concurrent evaluation is to be performed. Thus, we start by assuming the existence of concurrent computation (however it may be initiated) where each process is a piece of code and an environment in which to execute. Processes access a common shared heap space.

Several SMALL features make it a reasonable choice for extension to this multiprocessing environment. The virtualization of list addressing achieved by the address mapping performed in the LPT provides a means for efficient global naming (addressing) and routing (address decoding) in a multiprocessor system. List reference virtualization also serves to shield the EPs of the system from the actual heap activity taking place at the LP level. Further, the SMALL philosophy of continuous heap space management should lead to reduced communication overheads related to garbage collection in a multiprocessor system. In the SMALL organization of Chapter 4 this was achieved using reference counts; we will see that such a scheme will not suffice for the SMALL Multilisp system.

6.3.1. SMALL Multilisp System Organization

The SMALL organization could be extended into a multiprocessing system in several ways. In this section we suggest three alternative organizations. In all three organizations the global heap is partitioned into a set of heaplets, each managed by an LP.

In the *homogeneous* organization, the Multilisp system is composed of processing elements, each made up of an EP and an LP. The EP-LP pairs are connected to a communication network through which they communicate with each other using messages. Each processing element has a unique node identifier associated with it; node identifiers are used in specifying the sender and receiver of a message, and also in constructing unique system wide identifiers for list

references. A list reference is composed of its *identifier*, as maintained in the LPT of the LP that manages it, concatenated with the node identifier of the processing element containing that LP. Note that this implies that all LPT fields that are *identifier*s must be extended to hold the node identifier extension. An EP can access lists managed by its LP faster than it can access non-local list objects, which are managed by other LPs. This organization is uniform in that all processing elements are identical, with fast local list accessing within a processing element. Non-local references are made through messages containing the access request sent to remote processing elements. The accessed value is sent back in another message.

Computation starts on a single EP-LP processing element; other nodes get activated when parallel evaluation is triggered. For example, when a future is encountered, a new node is activated to execute the expression of the future. Any reference to the value of the future is forwarded to that node, where it is blocked until the expression has been evaluated. When new lists are created during evaluation at a processing node, they are allocated in the heaplet managed by the local LP. This is done with the expectation of increasing the locality of reference in list access, along the same lines as the schemes suggested for the Concert and Apiary systems mentioned above.

Because of the distinction between local and non-local list accessing in the homogeneous organization, the manner in which evaluation tasks are allocated to processing elements determines the performance of the system. The *decoupled* Multilisp organization removes this problem by making all list accesses non-local. In a decoupled organization, EPs and LPs are not associated with each other on a one-to-one basis; the system is made up of a pool of pure EP nodes and pure LP nodes connected through a communication network. Each node has a unique node identifier associated with it, and just as in the homogeneous case a list reference is made up of an LPT *identifier* and a specification of which LP node it is managed by. Since every list access is non-local, the problem of allocating processes on processors becomes trivial; it doesn't matter which node a process runs on. On the other hand, the speed of list access is guaranteed to be slow. The decoupled organization thus provides an upper bound for the amount of message traffic generated in evaluating a Multilisp program.

Recall that in the homogeneous organization each node managed a portion of the heap, and was able to access the lists in that heaplet quickly. In reality some of the concurrent streams of execution might require more heap space than others. The decoupled organization went to the extreme of making all EPs equal in having to do all list accessing non-locally. The *hybrid* organization, where the Multilisp system is made up of EP-LP processing elements, EP nodes, and LP nodes all connected to a communication network, provides a compromise between the homogeneous and decoupled organizations. The EP-LP processing elements are intended for the streams of concurrent evaluation that do excessive local list accessing. Clearly, this organization will require a more complicated task allocation algorithm than the other two.

Which of these organizations is preferable? We believe that there is no simple answer to this question; it is an issue worthy of further research. The decoupled organization suffers from an excess of non-local heaplet accesses; it would result in more message traffic due to list referencing than in any of the other organizations. The extent of this message traffic can be reduced in the

homogeneous and hybrid organizations if processes are allocated to processors appropriately. If a process is allowed to run on an EP-LP node whose LP manages the heaplet that is most frequently accessed by that process, then much of the heap memory activity can be handled locally, resulting in faster execution.

Under any of these organizations, the EP and the LP that we described in Chapter 4 will have to be modified. Every node, be it an EP-LP processing element, a plain EP node, or a plain LP node, must have a queue in which to buffer incoming messages. It must also have a unique identifier with which to be addressed by other nodes. These node identifiers will also be used to extend LPT entry *identifier*s. Further, each processing node (a EP-LP processing element or a simple EP node) will require a link to its parent node, i.e. the processing node from which its activity was initiated. This link might be necessary in doing non-local lookups. Figure 6.1 illustrates schematics of the three kinds of nodes.

All three of these Multilisp organizations assume the existence of a globally addressable heap that is distributed over several heaplets, with each heaplet being managed by an LP. The LPT of each LP will contain all the list cells of the heaplet that are currently being accessed along with all the list cells of the heaplet that contain pointers to other heaplets. This follows directly from the way **mergeing** and **splitting** are performed in the LPT. We next discuss how this distributed heap can be managed.

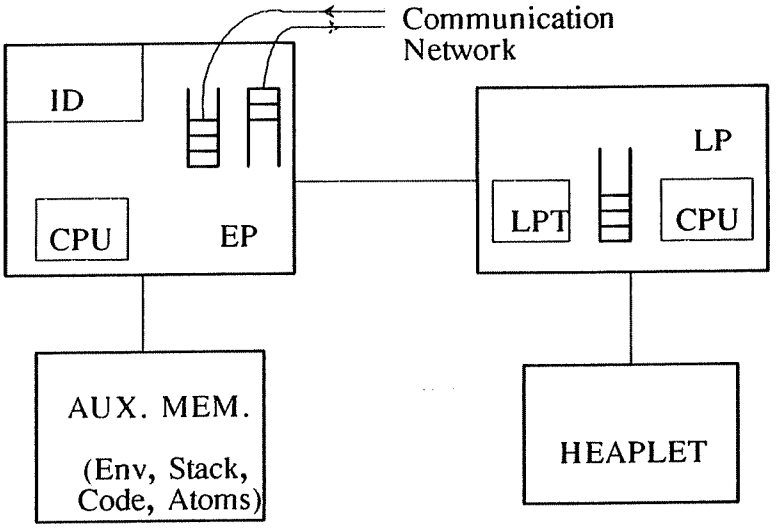
6.3.2. Heap Maintenance

The simple SMALL organization was able to keep tight control over the heap. This was possible largely due to the immediate detection of garbage made possible by the use of reference counts. Unfortunately, reference counting does not extend well to a multiprocessing system.

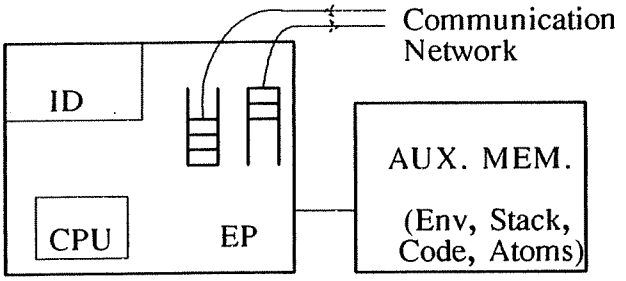
6.3.2.1. Reference Counting and Multiprocessing

There are two main reasons for the incompatibility of reference counting and multiprocessing. One is the volume of message traffic generated by reference count incrementing and decrementing operations. In the simple SMALL organization these operations were not much of an overhead since they were performed in the LP concurrently with useful evaluation in the EP. In a Multilisp system non-local reference count updating is done using messages. This uses up communication network bandwidth, and might be preventing useful work from using the network. So, the more the volume of this message traffic the more the impact of heap management on useful computation.

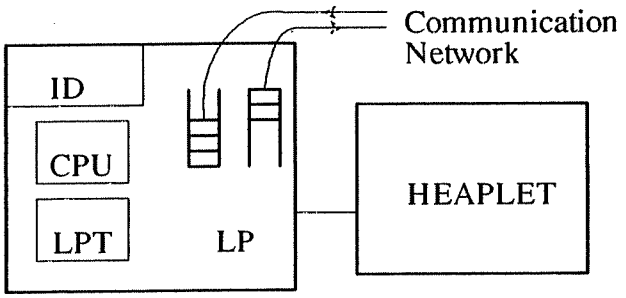
A more serious problem falls out of the distributed nature of the system. At any given instant, several reference count updating messages could be in transit in the communication network. For example, in the simple situation illustrated in Figure 6.2 there is a reference count increment message, M1, in transit from node B to node A, and a reference count decrement message, M2, in transit from node C to node A. Suppose that message M1 logically precedes message M2. Consider what happens if, due to delays in the communication network, message M2 arrives at node A and is serviced before message M1. This



(a) EP-LP Processing Element

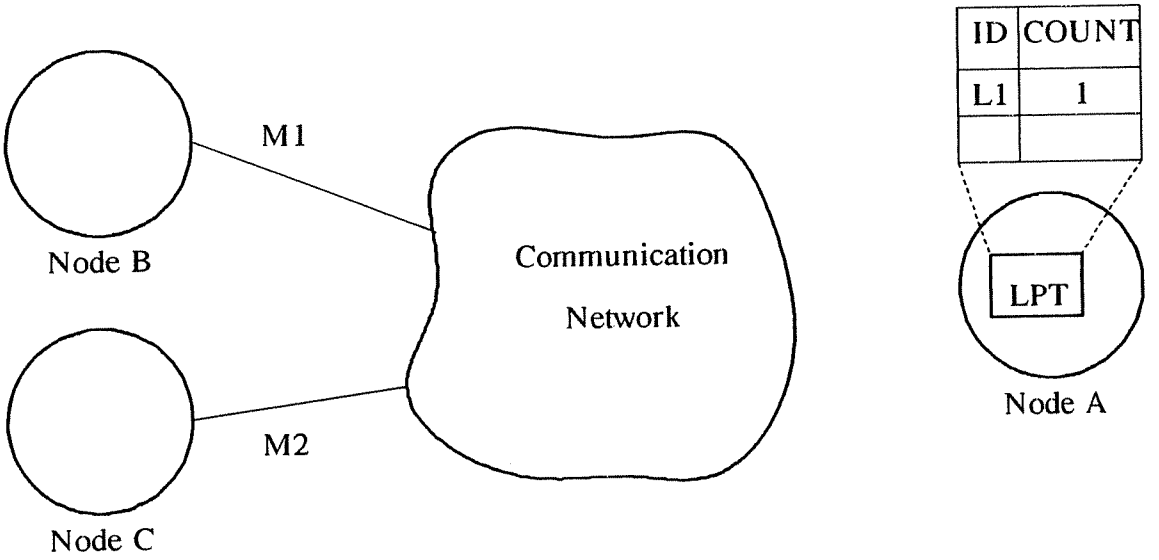


(b) EP Node

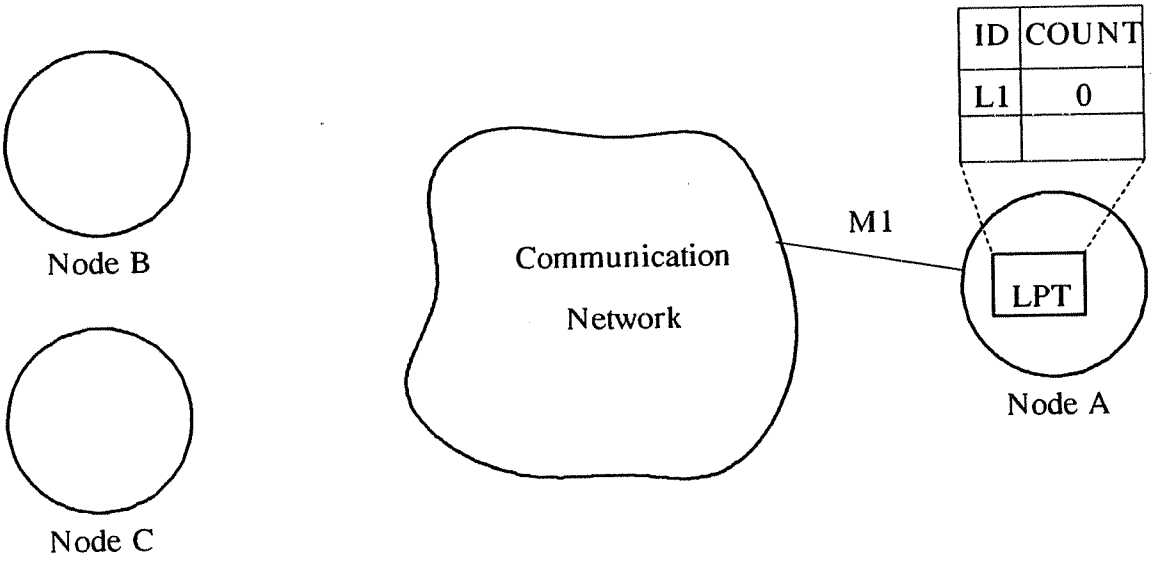


(c) LP Node

Figure 6.1. SMALL Multilisp System Nodes.



(a) Messages M1 (Incr A.L1) and M2 (Decr A.L2) are sent to node A from nodes B and C respectively



(b) If M2 reaches node A before M1 an error condition arises since the reference count of L1 is now 0

Figure 6.2. Reference Counting in a Multiprocessing System.

decrement operation leaves the reference count at zero; the heap space associated with the object will be reclaimed for reuse. So, when message M1 arrives at node A and is serviced it will result in the error condition of attempting to increment the reference count of a non-existent object. Clearly then, in such a scenario reference count messages must arrive *in order* to ensure error free evaluation.

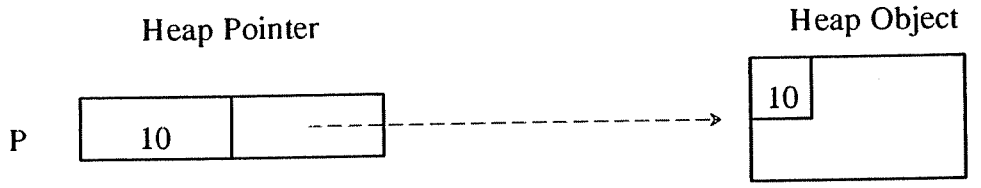
6.3.2.2. Reference Weights

One solution to these problems is to use *reference weights* instead of reference counts [Weng79a]. In such a scheme a reference weight is maintained for each list object. Each list pointer also has a reference weight associated with it. The sum of the reference weights of all the pointers to an object is equal to the reference weight of that object. When a pointer *P* is destroyed, the reference weight of the object *P* points at is decremented by *P's* reference weight. This is similar to how reference counting operates; in reference counting, the reference weight of every pointer is 1. The two schemes differ in their treatment of pointer copying. In reference counting, when a pointer is copied the reference count of the corresponding object is incremented. In reference weighting, on the other hand, when a new pointer, *C*, is created and initialized by copying the value of another pointer, *B*, into it, the reference weights of *B* and *C* are adjusted so that their sum is equal to the old reference weight of *B*. So, the reference weight of the object itself does not have to be modified.

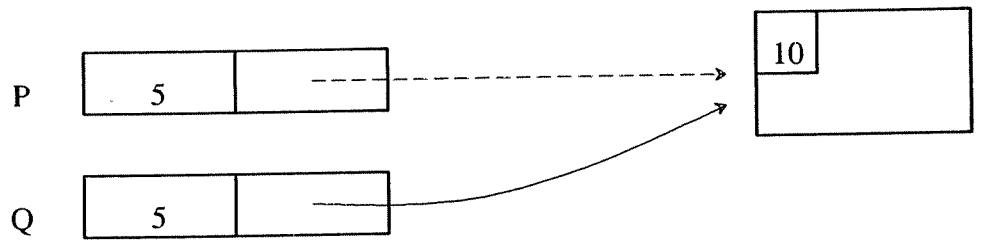
There are two conditions under which messages will be generated in maintaining the heap. When a pointer *P* is no longer used, a reference weight decrementing message must be sent to the node where the object *P* points at. Unlike in reference counting, no message need be sent to this node on pointer copying; the new pointer gets a share of the old pointer's reference weight. Occasionally, however, reference weight incrementing messages have to be sent. For example, if a pointer *P* is copied into pointer *Q*, but *P* has a reference weight of 1, then this reference weight cannot be split between *P* and *Q*. *P*'s reference weight must first be incremented; this requires the reference weight of the object that *P* points at to be incremented by the same amount. However, since these are the only count/weight increment operations that are required, there will be less message traffic than in the equivalent reference counting system. Notice also that these messages need not arrive in order, given that increment messages are rare and that the arrival order of decrement messages is immaterial. Reference weighting is illustrated in Figure 6.3.

6.3.2.3. Reference Weights on SMALL

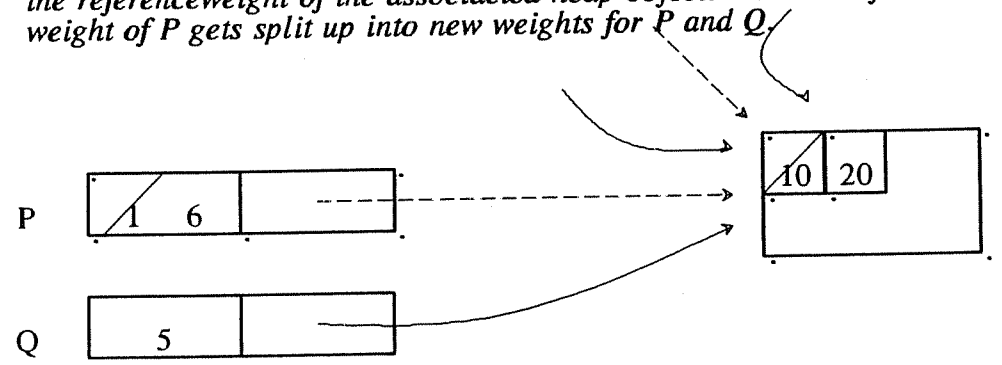
Note that reference weighting would not have been a viable alternative to reference counting for the SMALL architecture of Chapter 4. Reference weighting has a number of hidden costs associated with it. Whereas the updating arithmetic involved in reference counting is restricted to incrementing and decrementing by one, in reference weighting the increments and decrements could be arbitrarily large. So, while a reference counting LP could be optimized to manage without an ALU to do this arithmetic, a reference weighting LP cannot be so optimized. Another hidden cost is the additional space required for reference



(a) Every heap object has a reference weight associated with it. A heap pointer is madeup of a reference weight and a pointer to the heap object. The sum of the reference weights of all the pointers to an object is equal to the reference weight of that object.



(b) When pointer P is copied into pointer Q, no change need be made to the referenceweight of the associacted heap object. The old reference weight of P gets split up into new weights for P and Q.



(c) If P is to be copied into Q but has a reference weight of only 1, then the reference weight of the object is incremented (in this case by 10), and P's weight is incremented by the same amount before being split.

Figure 6.3. Reference Weighting.

weights. Each LPT entry must have three reference weight fields : one for the object it is pointing at, one for its ***car*** field, and one for its ***cdr*** field, as against a single reference count field. The two new fields are necessary since the ***car*** and ***cdr*** fields are actually pointers to other LPT entries. Further, every EP increment/decrement request to the LP must be accompanied by a reference weight. This will either necessitate a wider EP-LP communication path or more clock periods to send a single request across. Finally, in the EP's environment each name-value binding that includes a list pointer must be extended with a reference weight field. So, the environment will occupy more space. On the other side of the balance, all that a reference weighting scheme can offer is a reduction in the number of increment operations.

We have seen that reference weights are preferable to reference counts in a Multilisp system. To implement this change in heap management policy minor modifications have to be made to the LPT described in Chapter 4; the new structure of an LPT entry is as shown in Figure 6.4. The disadvantages of using reference weighting as outlined in the previous paragraph still persist. They can, however, be minimized.

If we assume that the allocation of tasks to processors has been done intelligently, then the majority of all heap accesses will be local due to the locality of reference shown by Lisp code. Suppose that all local reference weights are kept at a value of 1. In addition, the following strategy is employed for pointer copying. We differentiate between cases where a pointer to an object is copied and stored locally, and cases where the pointer is copied and stored non-locally. For local copying the reference weight of the copy is set at 1, while for non-local copying the reference weight of the copy is set to a value greater than 1. In each case the corresponding local increment reference weight operation is performed in the LP. As a net result, as far as local operation is concerned this strategy reduces to reference counting. The LP hardware can be tuned towards this, giving local list accesses a quick response and making non-local references pay a

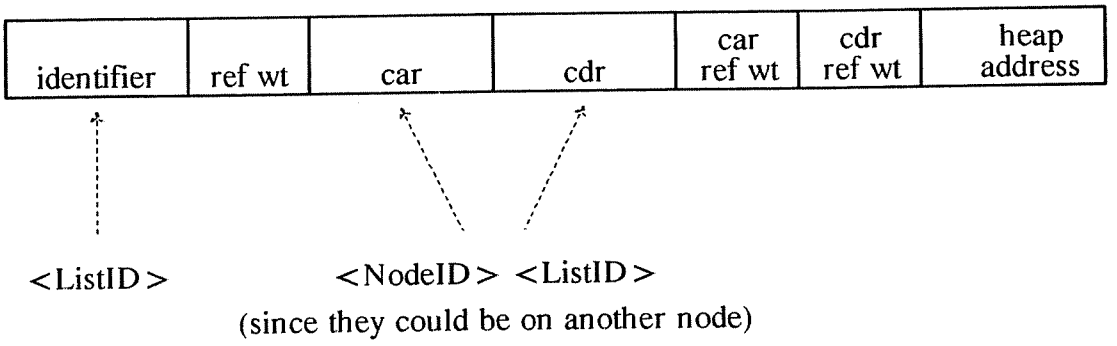


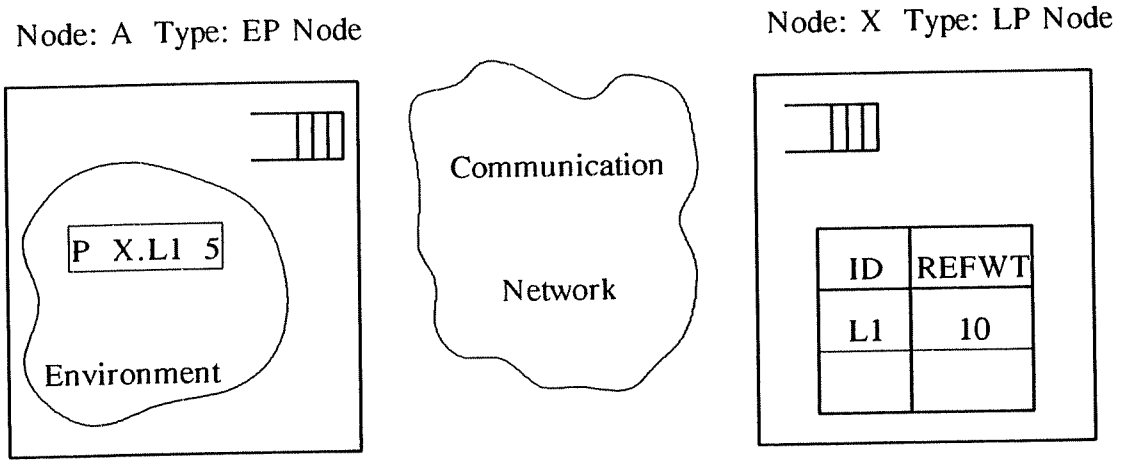
Figure 6.4. New LPT Organization.

higher performance penalty. This strategy still retains the better features of reference weighting; since non-local pointers have reference weights larger than 1 they can be non-locally copied without having to update the reference weight in the LPT entry corresponding to the list object being pointed at as illustrated in Figure 6.5.

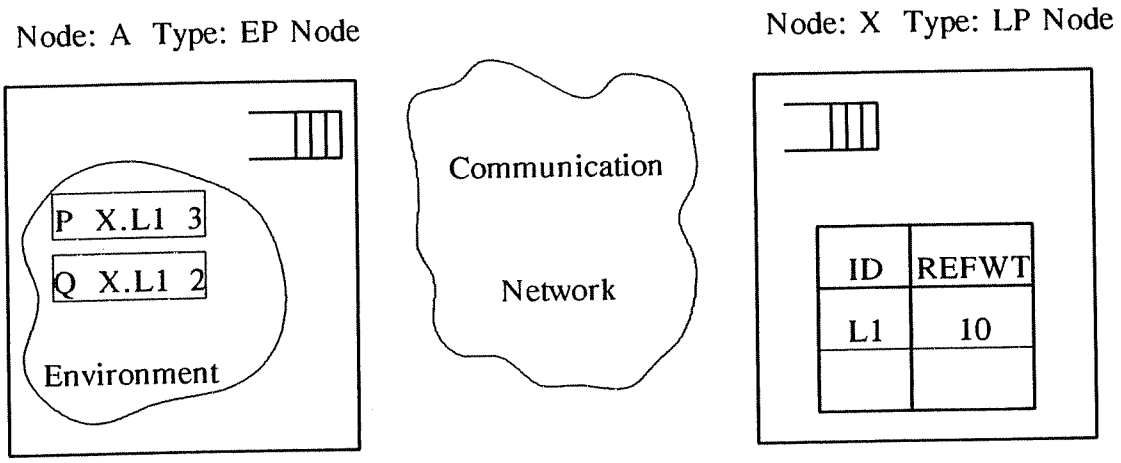
At each LP, the updating of reference weights is a low priority operation; other LP operations are more critical since the evaluation task of an EP could be blocked awaiting the return value from such an operation. So, depending on the level of activity in the system, reference weight update requests could get queued up waiting to be serviced. A good measure of the success of our approach to distributed heap management would be the length of these reference weight update messages queued up at the LPs of the SMALL Multilisp system. One approach to cope with excessive queued up update requests would be to attempt to ***combine*** them in the queue as illustrated in Figure 6.6.

6.4. Summary

Our goal in this chapter was to illustrate how SMALL can be extended into a Multilisp system. Such a system could be organized in many different ways, with the ease of task allocation being a degree of freedom in the selection of which organization to use. The use of reference weights to overcome the disadvantages of reference counting in a multiprocessing environment was described. We did not try to answer a key question: Is the SMALL philosophy of tight heap control preferable to heap management schemes base on mark-sweep algorithms in a multiprocessing environment? This and several of the issues that were raised in this chapter are left for future research.

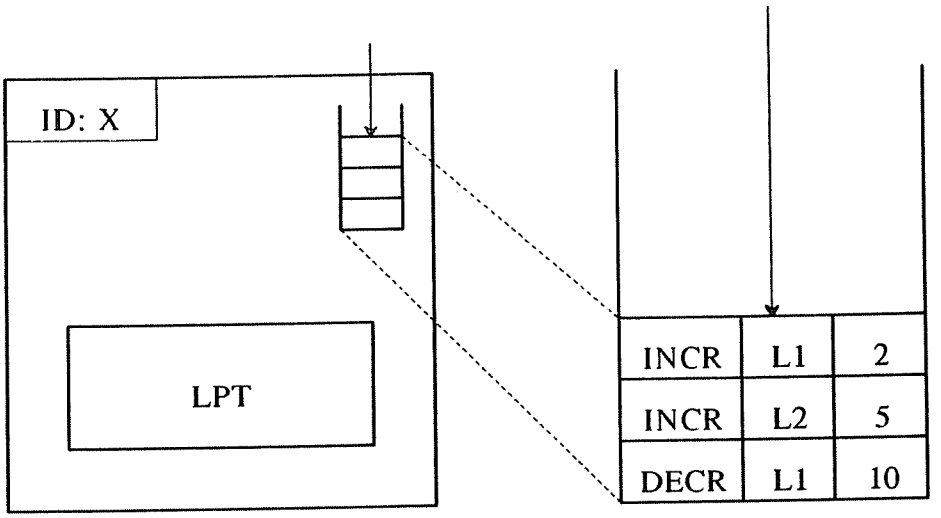


(a)
In the environment of the process running at Node A, the name 'P' is bound to the (non-local) object L1 magaged by LP Node X. The reference weight of this pointer is 5, while the sum of the weights of all pointers to L1 is 10.

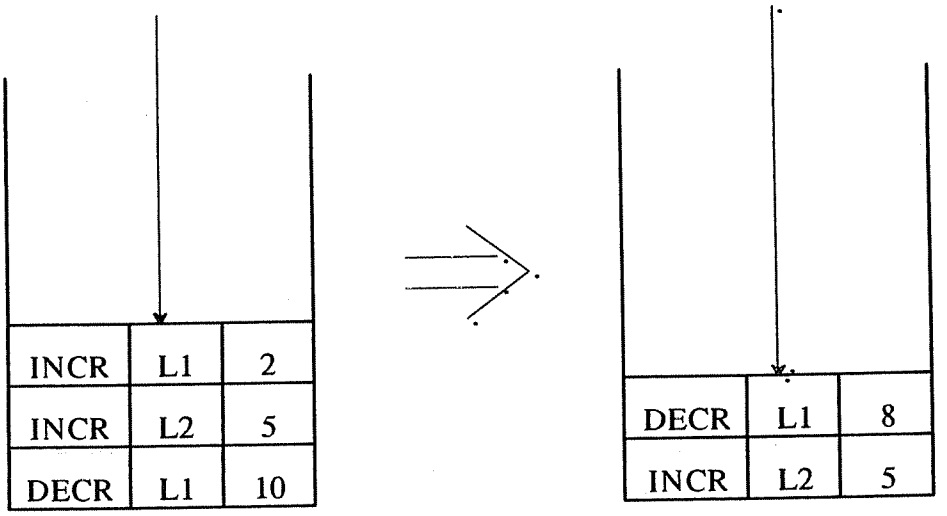


(b)
Node X is not involved when 'P' is copied into 'Q' local to node A.

Figure 6.5. Non-local Copying Using Reference Weights.



(a) Depending on the heap addressing activity in the Multilisp system and the speed of reference weight updating in LP nodes, there could be long queues of reference weight update requests.



(b) These requests can be combined in LP queues.

Figure 6.6. Combining Reference Weight Updates in Queues.

Chapter 7

Conclusion

7.1. Summary of Results

Lisp has been a popular programming language for well over 20 years. Recent interest in Fifth Generation Computer Systems has sparked renewed interest in systems for the efficient execution of Lisp and Lisp-like languages. Typically these systems do not run efficiently due to the large semantic gap between list manipulating languages like Lisp and the conventional von Neumann machine. This thesis presented an effective organization for a Lisp machine.

Our analysis of the run-time demands of Lisp computation revealed four potential areas for increased architectural attention: function calling, environment maintenance, list accessing, and heap maintenance. Based on a survey of techniques used in state of the art Lisp machines we chose to concentrate our efforts on studying the behaviour of lists in Lisp. Previous studies have shown that Lisp list access streams display temporal as well as spatial locality of reference. In a Lisp system, where memory is largely dynamically heap allocated, spatial locality is highly implementation dependent; the spatial position of objects in memory will depend on how the dynamic memory space is managed at run-time. The studies conducted in the past examined only these spatial properties of Lisp list accessing.

To evaluate the regularity of access shown by Lisp programs in the dynamic Lisp environment we extended the traditional concepts of spatial and temporal locality of access with the concept of *structural locality* of access. Structural locality of access provides an implementation independent means of describing the scope for spatial locality. We used a partitioning procedure to partition a Lisp list access stream into structurally related locales of high temporal locality of reference, called list sets. Our studies of this list set partition reveal that it captures regularities in Lisp list access streams that can be taken advantage of by suitable architectural features.

The SMALL Lisp machine architecture was the outcome of this investigation. In SMALL, the Lisp evaluation work is partitioned across two processing elements. The Evaluation Processor (EP) manages this evaluation and maintains the name-value binding environment. The List Processor (LP) manages the storage and accessing of lists as well as the management of the heap using a hardware table called the LPT. It is optimized towards efficient heap maintenance (using reference counting), efficient list addressing (lists being addressed with short LPT indices), and efficient list accessing (through the buffering of list properties in the LPT).

In SMALL, both LPT space and the heap space of list cells are managed by the LP using a reference count in each LPT entry. Garbage is detected almost immediately after it is created; the LP identifies an LPT entry as unused when its reference count goes to zero. SMALL provides a list processing environment in which reference counts can be maintained with low overhead, making the elaborate mark-and-sweep garbage collection schemes found in other Lisp machines unnecessary.

Trace driven simulations indicate that the SMALL organization deals effectively with the problems of list accessing and heap maintenance. They also yield indications as to the size requirements of the LPT. True LPT overflow will be extremely rare given an LPT of a few thousand entries; even for our longest trace, true overflow occurred only when the LPT was less than a few hundred entries large. In a translation table with 2K or 4K entries even pseudo overflows would rarely occur. In our evaluation of pseudo overflow recovery policies we found that while the Compress-One policy results in less efficient use of LPT space than the Compress-All policy, the mean difference between the average LPT occupancy resulting from the 2 policies do not greatly differ. We suggest a hybrid scheme, where Compress-One is used by default, and Compress-All is applied if pseudo overflows become frequent.

Our evaluation indicated that the amount of LPT activity that occurred was not excessive. But, in case the degree of LPT activity is found to be excessive in a particular SMALL implementation we suggested an optimization. We observed that a large percentage of the reference count activity was related to references from the stack, which suggested the use of a split reference count. Our evaluation showed that split reference counts can reduce the count updating activity in the LP by almost an order of magnitude, at the expense of increased hardware costs.

We also studied the list access buffering capability of the LPT by comparing it with a data cache, and counting the number of misses on **car** and **cdr** accesses in each case. The LPT performed almost twice as well as a data cache with line size of 1. Even with larger cache line sizes the LPT compared favourably with a data cache.

Finally, the decoupling of evaluation and list accessing achieved by the EP-LP partition makes SMALL a prime candidate for extension to Multilisp systems, i.e. multiprocessor systems in which the evaluation of a single Lisp program is allocated across several processors. We investigated this possibility briefly in Chapter 6.

7.2. Suggestions for Future Work

Several interesting questions about Lisp, SMALL and Multilisp implementations present themselves at this point. We describe them briefly in this section, but leave their deeper study as future work.

In our analysis of Lisp program traces, we concentrated on the Lisp list manipulating primitives, **car, cdr, cons, rplaca,** and **rplacd,** which are the lowest level list manipulation operations. At a higher level, most Lisps include a set of pre-defined functions that access lists in highly structured ways. For example, in Franz Lisp, there are **map, mapcar** etc. In a way, these functions are to Lisp what the DO loop is to Fortran. Their calling behaviour could be studied to get a better feel for typical access patterns to lists, possibly leading to more involved architectural support for list access.

As far as the SMALL organization is concerned, we see several interesting research issues. One is the operation of the Evaluation Processor. We described the EP's operation in Chapter 4, but made no attempt to evaluate its effectiveness. The EP's effectiveness would depend on how the environment was

maintained (deep binding, shallow binding, or some new binding scheme), on how the control stack was maintained, etc. Another SMALL issue of interest is the choice of how to represent lists in the heap memory. While we addressed the relative merits and demerits of a few schemes in Chapter 4, this is an issue worthy of more careful analysis. The representation should be capable of quick splitting and merging while not being too space inefficient. Also, we raised several questions in describing SMALL Multilisp. We suggested three alternative organizations for the multiprocessor system, homogeneous, decoupled, and hybrid. The advantages of using each scheme need to be studied more, as does the effectiveness of using reference weights in multiprocessor heap management. Finally, the development of more detailed emulation and compiling tools for Lisp on SMALL would be useful in fine tuning the architecture.

Finally, we would point out that while SMALL has an LP to make list processing efficient, other specialized memory managing processors could be added to make access to other kinds of data structures more efficient. For example, if arrays were heavily used, a memory access processor that kept track of array references could be used, as in the Structured Memory Access architecture [Ples82a]. The SMALL organization could be extended with other such special purpose memory accessing functional units to improve performance.

References

[Abra66a] Abrahams, P. W., "The LISP 2 Programming Language and System," *Proceedings of the Fall Joint Computer Conference*, pp. 661-676 (1966).

[Akim85a] Akimoto, H., S. Shimizu, A. Shinagawa, A. Hattori, and H. Hayashi, "Evaluation of the Dedicated Hardware in Facom Alpha," *Spring 1985 Compcon Digest of Papers*, pp. 366-369 (1985).

[Alle78a] Allen, J. R., *Anatomy of Lisp*, McGraw-Hill Artificial Intelligence Series (1978).

[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., R. Greenblatt, J. Holloway, T. Knight, D. A. Moon, and D. Weinreb, "Lisp Machine Progress Report," MIT AI Laboratory Memo No.444 (August 1977).

[Bobr73a] Bobrow, D. G. and B. Wegbreit, "A Model and Stack Implementation of Multiple Environments," *Communications of the ACM* **16**(10) pp. 591-603 (October 1973).

[Bobr75a] Bobrow, D. G., "A Note on Hash Linking," *Communications of the ACM* **18**(7) pp. 413-415 (July 1975).

[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).

[Burt80a] Burton, R. R., L. M. Masinter, D. G. Bobrow, W. S. Haugeland, R. M. Kaplan, and B. A. Sheil, "Overview and Status of DoradoLisp," *Conference Record of the 1980 ACM Lisp Conference*, pp. 243-247 (1980).

[Clar76a] Clark, D. W., "List Structure: Measurements, Algorithms, and Encodings," Ph. D. Dissertation, Carnegie-Mellon University, Pittsburgh, Pa (August 1976).

[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).

[Clar79a] Clark, D. W., "Measurements of Dynamic List Structure in Lisp," *IEEE Transactions on Software Engineering* **5**(1) pp. 51-59 (January 1979).

[Cohe81a] Cohen, J., "Garbage Collection of Linked Data Structures," *Computing Surveys* **3**(13) pp. 341-367 (September 1981).

[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).

[Deer85a] Deering, M. F., "Architecture for AI," *Byte* **10**(4) pp. 193-206 (April 1985).

[Deut73a] Deutsch, L. P., "A Lisp Machine with Very Compact Programs," *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, (August 1973).

[Deut76a] Deutsch, L. P. and D. G. Bobrow, "An Efficient, Incremental, Automatic Garbage Collector," *Communications of the ACM* **19**(9) pp. 522-526 (September 1976).

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

[Deut80a] Deutsch, L. P., "ByteLisp and its Alto Implementation," *Conference Record of the 1980 ACM Lisp Conference*, pp. 231-242 (1980).

[Dijk78a] Dijkstra, E. W., L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly Garbage Collection: An Exercise in Cooperation," *Communications of the ACM* **21**(11) pp. 966-975 (November 1978).

[Fate78a] Fateman, R. J., "Is a Lisp Machine different from a Fortran Machine?," *ACM SIGSAM Bulletin* **12**(3) pp. 8-11 (August 1978).

[Feni69a] Fenichel, R. and J. Yochelsen, "A Lisp Garbage-Collector for Virtual Memory Computer Systems," *Communications of the ACM* **12**(11) pp. 611-612 (November 1969).

[Fode79a] Foderaro, J. K., *The Franz LISP Manual*, University of California, Berkeley (1979).

[Fode81a] Foderaro, J. K. and R. J. Fateman, "Characterization of VAX Macsyma," *Proceedings of the 1981 Symposium on 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).

[Gabr84a] Gabriel, R. P. and J. McCarthy, "Queue-based Multi-processing Lisp," *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 25-43 (August 1984).

[Gabr85a] Gabriel, R. P., *Performance and Evaluation of Lisp Systems*, MIT Press (1985).

[Gott83a] Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Machine," *IEEE Transactions on Computers* C-32(2) pp. 175-189 (February 1983).

[Gris78a] Griss, M. L. and R. R. Kessler, "Reduce/1700: A Micro-coded Algebra System," *Proceedings of the 11th Annual Microprogramming Workshop*, pp. 130-138 (1978).

[Guzm81a] Guzman, A., "A Heterarchical Multi Microprocessor Lisp Machine," *Proceedings of the IEEE Workshop on CAPAIDM*, pp. 309-317 (November 11-13, 1981).

[Hals81a] Halstead, R. H. Jr., "Architecture of a Myriaprocessor," *Digest of Papers of 1981 SPring Compcon*, pp. 299-302 (February 1981).

[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).

[Harb81a] Harbison, P., R. Levin, and W. A. Wulf, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill (1981).

[Haya83a] Hayashi, H., A. Hattori, and H. Akimoto, "Alpha: A High-Performance Lisp Machine Equipped with a New Stack Structure and Garbage Collection System," *Proceedings of the 10th International on Computer Architecture*, pp. 342-348 (June 1983).

[Hewi80a] Hewitt, C., "The Apiary Network Architecture for Knowledgeable Systems," *Proceedings of the 1980 Lisp Conference*, pp. 107-117 (1980).

[Hill85a] Hillis, W. D., "," in *The Connection Machine*, The MIT Press Series in Artificial Intelligence (1985).

[Jone82a] Jones, M. A., "A Comparison of LISP Specifications of Function Definition and Argument Handling," *Sigplan Notices* 17(8) pp. 67-73 (August 1982).

[Kell79a] Keller, R. M., G. Lindstrom, and S. Patil, "A Loosely Coupled Applicative Multi-Processing System," *AFIPS NCC Proceedings* 48 pp. 613-622 (1979).

[Kell80a] Keller, R. M., "Divide and CONCer: Data Structuring in Applicative Multiprocessing Systems," *Conference Record of the 1980 ACM Lisp Conference*, pp. 196-202 (1980).

[Li85a] Li, K. and P. Hudak, "A New List Compaction Method," Yale University Research Report (February 1985).

[Lieb83a] Lieberman, H. and C. Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects," *Communications of the ACM* 26(6) pp. 419-429 (June 1983).

[Mart78a] Marti, J. B., A. C. Hearn, M. L. Griss, and C. Griss, "Standard LISP Report," *University of Utah Symbolic Computation Group Report No. 60*, (1978).

[Mart80a] Marti, J. B., "Compilation Techniques for a Control-Flow Concurrent Lisp System," *Conference Record of the 1980 ACM Lisp Conference*, pp. 203-207 (1980).

[Mart83a] Marti, J. B. and J. P. Fitch, "The Bath Concurrent Lisp Machine," *Proceedings of EUROCAL 83, European Computer Algebra Conference*, pp. 78-90 (March 1983).

[Mart71a] Martin, W. A. and R. J. Fateman, "The MACSYMA System," *Proceedings of the 2nd ACM Symposium on Symbolic and Algebraic Manipulation*, pp. 23-25 (1971).

[Matt70a] Mattson, R. L., J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal* 2(9) pp. 78-117 (1970).

[McCa60a] McCarthy, J., "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I," *Communications of the ACM* 3(4) pp. 184-195 (April 1960).

[McCa78a] McCarthy, J., "History of Lisp," *Sigplan Notices* 13(8) pp. 217-223 (August 1978).

[Mead80a] Mead, C. A. and L. A. Conway, *Introduction to VLSI Systems*, Addison Wesley (1980).

[Mins73a] Minsky, N., "Representation of Binary Trees on Associative Memories," *Information Processing Letters*, (2) pp. 1-5 (1973).

[Moha84a] Mohammed-Ali, A., "Object-Oriented Storage Management and Garbage Collection in Distributed Processing Systems," Ph. D. Dissertation, Royal Institute of Technology, Stockholm, Sweden (December 1984).

[Moon74a] Moon, D. A., "MACLISP Reference Manual," Project MAC Technical Report, MIT, Cambridge, Mass. (1974).

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

[Mose69a] Moses, J., "The Function of FUNCTION in Lisp," *Sigplan Notices*, (June 1969).

[Myer82a] Myers, G. J., *Advances in Computer Architecture*, John Wiley & Sons, Inc. (1982).

[Olss83a] Olsson, O., "The Memory Usage of a LISP System: The Belady Life-Time Function," *Sigplan Notices* 18(12) pp. 112-119 (December 1983).

[Padg83a] Padget, J. A., "The Ecology of Lisp or the Case for the Preservation of the Environment," *Proceedings of EUROCAL 83, European Computer Algebra Conference*, pp. 91-100 (March 1983).

[Patt81a] Patterson, D. A. and C. H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 443-457 (May 1981).

[Ples82a] Pleszkun, A. R., "A Structured Memory Access Architecture," Ph. D. Dissertation, University of Illinois, Urbana-Champaign, Il. (August 1982).

[Pond83a] Ponder, C., "... but will RISC run LISP??" (a feasibility study)," University of California, Berkeley Technical Report (August 1983).

[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).

[Prin80a] Prini, G., "Explicit Parallelism in Lisp-Like Languages," *Conference Record of the 1980 Lisp Conference*, pp. 13-18 (August 1980).

[Ram85a] Ram, A. and J. H. Patel, "Parallel Garbage Collection Without Synchronization Overhead," *Proceedings of the 12th Annual Symposium on Computer Architecture*, pp. 84-90 (July 1985).

[Rees82a] Rees, Jonathan A. and Norman I. IV Adams, "T: A Dialect of Lisp, or, Lambda: The Ultimate Software Tool," *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, (August 1982).

[Road83a] Roads, C. B., "3600 Technical Summary," *Symbolics Corp., Cambridge, MA*, (February 1983).

[Sans82a] Sansonnet, J. P., M. Castan, C. Percebois, D. Botella, and J. Perez, "Direct Execution of Lisp on List Directed Architectures," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 132-139 (March 1982).

[Scho67a] Schorr, H. and W. Waite, "An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures," *Communications of the ACM* **10**(8) pp. 501-506 (August 1967).

[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).

[Stee75a] Steele, G. L. Jr., "Multiprocessing Compactifying Garbage Collection," *Communications of the ACM* **18**(9) pp. 495-508 (September 1975).

[Stee78a] Steele, G. L. Jr. and G. J. Sussman, "The Art of the Interpreter or, The Modularity Complex," MIT AIL Memo No. 453 (May 1978).

[Stee84a] Steele, G. L. Jr. et al, *Common Lisp Reference Manual,* Digital Press (1984).

[Sugi83a] Sugimoto, S., K. Agusa, K. Tabata, and Y. Ohno, "A Multi-Microprocessor System for Concurrent Lisp," *Proceedings of the 1983 International Conference on Parallel Processing*, pp. 135-143 (June 1983).

[Suss75a] Sussman, G. J. and G. L. Jr. Steele, "Scheme: An Interpreter for Extended Lambda Calculus," MIT AIL Memo No. 349 (December 1975).

[Swan77a] Swan, R. J., S. H. Fuller, and D. P. Siewiorek, "Cm* -- A Modular Multi-Microprocessor," *AFIPS Conference Proceedings 46*, (1977).

[Teit75a] Teitelman, W., "INTERLISP: Interlisp Reference Manual," Xerox PARC Technical Report, Palo Alto, Ca (1975).

[Weis67a] Weisman, C., *LISP 1.5 Primer,* Dickenson Press (1967).

[Weiz63a] Weizenbaum, J., "Symmetric List Processor," *Communications of the ACM* 6(9) pp. 524-544 (September 1963).

[Weng79a] Weng, K-S, "An Abstract Implementation for a Generalized Data Flow Language," MIT Laboratory for Computer Science, Rep. No. TR-228 (1979).

[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).

[Yama81a] Yamamoto, M., "A Survey of High Level Language Machines in Japan," *Computer*, pp. 68-78 (July 1981).