Performance of Lisp Systems

Richard P. Gabriel
Stanford University
Lawrence Livermore National Laboratory

Larry M. Masinter
Xerox Palo Alto Research Center

1. Introduction

This paper describes the issues involved in evaluating the performance of Lisp systems. We explore the various levels at which quantitative statements can be made about the performance of a Lisp system, giving examples from existing implementations wherever possible. Our thesis is that benchmarking is most effective when performed in conjunction with an analysis of the underlying Lisp implementation and computer architecture. We examine some simple benchmarks which have been used to measure Lisp systems, and examine some of the complexities of evaluating the resulting timings.

Performance is not the only, or even the most important, measure of a Lisp implementation. Trade-offs are often made which balance performance against flexibility, ease of debugging, and address space.

'Performance' evaluation of a Lisp implementation can be expressed as a sequence of statements about the implementation on a number of distinct, but related, levels. Statements on each level can have an effect on the evaluation of a given Lisp implementation.

Benchmarking and analysis of implementations will be viewed as complementary aspects in the comparison of Lisps: benchmarking without analysis is as useless as analysis without benchmarking.

This paper will explain in detail the technical issues and trade-offs that determine the efficiency and usability of a Lisp implementation; though there will appear to be a plethora of facts, we will only discuss those aspects of a Lisp implementation that we feel are the most important for evaluation. Throughout we will talk about the impact of these issues and trade-offs on benchmarks and benchmarking methodologies.

The Lisp implementations that we will be using for most examples are: INTERLISP-10 [Teitelman;1978], INTERLISP-D [Burton;1981], INTERLISP-Vax [Masinter;1981a] [Bates;1982], PDP-10 MacLisp [Moon;1974], Vax NIL [White;1979], S-1 Lisp [Brooks;1982b], and FRANZ Lisp [Foderaro;1982].

2. Levels of Lisp System Architecture

The performance of a Lisp system can be viewed from the lowest level of the hardware implementation to the highest level of user program functionality. Understanding and predicting Lisp system performance depends upon understanding the mechanisms at each of these levels. We see the following levels as important for characterizing Lisp systems: basic hardware, Lisp 'instructions', simple Lisp functions, and major Lisp facilities.

There is a range of methodologies for determining the speed of an implementation. The most basic methodology is to examine the machine instructions that are used to implement constructs in the language, to look up in the hardware manual the timings for these instructions, and then to add up the times needed.

Another methodology is to propose a sequence of relatively small benchmarks and to time each one under the conditions that are important to the investigator (under typical load average, with expected working-set sizes, etc). Finally, real (naturally occurring) code can be used for the benchmarks.

Unfortunately each of these representative methodologies has problems. The simple instruction-counting methodology does not adequately take into account the effects of cache memories, system services (such as disk service), and other interactions within the machine and operating system. The middle, small-benchmark methodology is susceptible to 'edge' effects: that is, the small size of the benchmark may cause it to straddle a boundary of some sort, leading to unrepresentative results. For instance, a small benchmark may be partly on one page and partly on another, which may cause many page faults. Finally, the real-code methodology, though accurately measuring a particular implementation[1], is not necessarily accurate when comparing implementations. For example, programmers, knowing the performance profile of their machine and implementation, will typically bias their style of programming on that piece of code. Hence, had an expert on another system attempted to program the same algorithms, a different program might have resulted.

Hardware Level

At the lowest level, things like the machine clock speed and memory bandwidth affect the speed of a Lisp implementation. One might expect that a CPU with a basic clock rate of 50 nanoseconds will run a Lisp system faster than the same architecture but with a clock rate of 500 nanoseconds. This, however, is not necessarily true, since a slow or small memory can cause delays in instruction and operand fetch.

Several hardware facilities complicate the understanding of basic system performance, especially on microcoded machines: the memory system, instruction buffering and decoding, and the size of data paths. We will describe the most important of these facilities in the remainder of this section.

Cache memory is an important and difficult-to-quantify determiner of performance. It is designed to improve the speed of programs that demonstrate a lot of locality[2] by supplying a small, high-speed memory that is used in conjunction with a larger, but slower (and less expensive) main memory. An alternative to a cache is a stack buffer which keeps some number of the top elements of the stack in a circular queue of relatively high-speed memory. The Symbolics 3600 has such a PDL buffer.

Getting a quantitative estimate of the performance improvement yielded by a cache memory can best be done by measurement and benchmarking. Lisp has less locality than many other programming languages, so that a small benchmark may fail to measure the total performance well by failing to demonstrate 'normal' locality. Hence, one expects that the small-benchmark methodology would tend to result in optimistic measurements, having atypically higher locality than large Lisp programs.

An instruction *pipeline* is used to overlap instruction decode, operand decode, operand fetch, and execution. On some machines, the pipeline can become blocked when a register is written into and then referenced by the next instruction. Similarly, if a cache does not have parallel write-through then such things as stack instructions can be significantly slower than register instructions.

Memory bandwidth is important, and, without a relatively high bandwidth for a given CPU speed, there will not be an effective utilization of that CPU. In an extreme case, consider a 50-nanosecond machine with

---

[1] Namely, the implementation that the program was developed on.

[2] *Locality* is the extent to which the locus of memory references—both instruction fetches and data references—span a 'small' number of memory cells 'most' of the time.

3-μsec memory and no cache. Though the machine may execute instructions rapidly once fetched, fetching the instructions and the operands will operate at memory speed at best. There are two factors involved in memory speeds: the time it takes to fetch instructions and decode them, and the time it takes to access data once a path to the data is known to the hardware. Instruction pre-fetch units and pipelining can improve the first of these quite a bit, while the latter can generally only be aided by a large cache or a separate instruction and data cache.

Internal bus size can have a dramatic effect. For example, if a machine has 16-bit internal data paths, but is processing 32-bit data to support the Lisp, more microinstructions may be required to accomplish the same data movement than on a machine that has the same clock rate but wider paths. Narrow bus architecture can be compensated for by a highly parallel microinstruction interpreter, since a considerable proportion of cycles of the machine goes into things which are not data-path limited: for example, condition testing and instruction dispatch.

Many other subtle points of the architecture can make a measurable difference on Lisp performance. For example, if error correction is done on a 64-bit quantity, so that storing a 32-bit quantity takes significantly longer than storing a 64-bit quantity, arranging things throughout the system to align data on these 64-bit quantities as appropriate will take advantage of the higher memory bandwidth possible when the quad-word alignment is guaranteed. However, the effect of this alignment is small compared to the above factors.

Lisp 'Instruction' Level

Above the hardware level, the Lisp 'instruction' level includes such things as local variable assignment and reference, free/special[3] variable assignment, binding, and unbinding, function call and return, data structure creation, modification, and reference, and arithmetic operations.

The situation with Lisp is more complex than with a language such as PASCAL because many of the Lisp 'instructions' have several implementation strategies in addition to several implementation tactics for each strategy; in contrast, PASCAL compilers generally implement the constructs of the language the same way—share the same implementation strategy. For example, there are two distinct strategies for implementing free/special variables in LISP, called *deep binding* and *shallow binding*; these strategies implement the same functionality, but each optimizes certain operations at the expense of others. Deep-binding Lisps may cache pointers to stack allocated value cells, and this is a tactic for accomplishing speed in free/special variable lookups.

The timings associated with these operations can be determined either by analysis of the implementation or by designing simple test programs (benchmarks) that contain exclusively that operation and time the execution in one of several ways. Before discussing benchmarking techniques we discuss the operations.

Variable/Constant Reference

The first major category of Lisp 'instruction' consists of variable reference, variable assignment, and constant manipulation. References to variables and constants appear in several contexts, including passing a

[3] In the literature there are several terms used to describe the types of variables and how they are bound in the various implementations. *Global* variables have a value cell that can be set and examined at any lexical level, but cannot be lambda-bound. A *special* variable can sometimes mean a global variable, and sometimes it can mean a *free*, *fluid*, or *dynamic* variable; these synonymous terms refer to a variable that is not lexically apparent, but which can be lambda-bound. In this paper we will use the terms: *Lexical* or *local* for non-global, non-fluid variables, *global* for global variables, and *free/special* for global and fluid variables.

Shallow-binding systems look up and store into value cells, the pointers to which are computed at load time. Deep-binding systems bind and unbind faster than shallow-binding systems, but shallow-binding systems look up and store values faster.[6] Context-switching can be performed much faster in a deep-binding implementation than in a shallow-binding one. Deep binding therefore may be the better strategy for a multiprocessing LISP.[7]

A complication to these free/special problems occurs if a function can be returned as a value. In this case the binding context or environment must be retained as part of a *closure*, and re-established when the closure is invoked. Logically this involves a tree rather than a stack model of the current execution environment since portions of the stack must be retained to preserve the binding environment.

In a deep-binding system changing the current execution environment (invoking a closure) can be accomplished by altering the search path in the tree. In cached systems one must also invalidate relevant caches.

In a shallow-binding system the current value cells must be updated, essentially by a tree traversal that simulates the unbinding and rebinding of variables.

Some shallow-binding Lisps (LISP370, for instance) have a hybrid scheme in which the value cell is treated more like a cache than like an absolute repository of the value and does cache updates and write-throughs in the normal manner for caches.

Some Lisps (the Common Lisp family, for example) are partially *lexical* in that free variables are by default free/special, but the visibility of a bound variable is limited to the lexical context of the binding, unless the binding specifies it as free/special. Lisp compilers assign locations to these variables according to the best possible coding techniques available in the local context rather than demanding a canonical or default implementation in all cases.[8]

As hinted, variable access and storage times can vary greatly from implementation to implementation and also from case to case within an implementation. Timing only variable references can be difficult because a compiler can make decisions that may not reflect intuition, such as unreferenced variables being optimized out.

Function Call/Return

The performance of function call and return is more important in Lisp than in most other high level languages due to Lisp's emphasis on functional style. In many Lisp implementations call/return accounts for about 25% of total execution time. Call/return involves one of two major operations: building a stack frame, moving addresses of computed arguments into that frame, placing a return address in it, and transferring control; and moving arguments to registers, placing the return address on the stack, and transferring control. In addition, function calling may require the callee to move arguments to various places in order to reflect temporary name bindings (referred to as *stashing* below), to default arguments not supplied, and to allocate temporary storage. Furthermore, saving and restoring registers over the function call can either be done by the caller, the callee, or by some cache type of operation that saves/restores on demand [Lampson;1982] [Steele;1979]. As noted in the previous section, function calling can require caching deep-binding free/special variables on the stack.

---

[6] Shallow-binding systems look up and store in constant time. Deep-binding systems must search for the (variable name, value) pairs, and in cached, deep-binding systems this search time may be amortized over several references and assignments.

[7] A shallow-binding system can take an arbitrary time to context switch, and, for the same reason, a deep-binding system can take an arbitrary amount of time to search for the (variable name, value) pairs.[Baker;1978b]

[8] Canonical implementations allows separately compiled or interpreted functions to access free/special variables.

We group function call and return time because, normally, every function call is paired with a function return. It is possible for a function to exit via other means: for example, non-local exits such as RETFROM in INTERLISP and THROW in MacLisp. THROW needs to search for the matching CATCH, doing free/special unbinds along the way (referred to as *unwinding*).

The following two paragraphs constitute an example of the kind of analysis that is possible from an examination of the implementation.

In PDP-10 (KL-10B or DEC-2060) MacLisp a function call is a PUSHJ/POPJ (3 μsec) for the saving and restoring of the return address and transfer of control, a MOVE from memory to register (with possible indexing off the stack—.4–.8 μsec) for each argument up to 5, or a PUSH and maybe a MOVEM (MOVE to Memory—.6 μsec) for each argument when the total number of arguments is more than 5. Function entry is usually a sequence of PUSH's to the stack from registers. Return is a MOVE to register plus the POPJ already mentioned. Numeric code, upon function entry, 'unboxes' numbers (converts from pointer format to machine format) via a MOVE Indirect (.5 μsec) to obtain the machine format number.

Function call without arguments in INTERLISP-10 on a DEC 2060 has a range of about 3 μsec for an internal call in a block (PUSHJ, POPJ) to around 30 μsec for the shortest non-block compiled call (builds a frame in about 60 instructions) to around 100 μsec (function call to a swapped function).

Some Lisps (Common Lisp [Steele;1982], Lisp Machine Lisp [Weinreb;1981]) have multiple values. The implementation of multiple values can have great impact on the performance of a Lisp. For example, if multiple values are pervasive, then there is a constant overhead for marking or recognizing the common, single-value case, and some tail-recursive cases may require that an arbitrary amount of storage be allocated to store values that will be passed on—for example: (prog1 (multiple-values) ...). If some multiple values are passed in registers (S-1 [Correll;1979]) then there may be an impact on how the register allocator can operate, possibly causing memory bottlenecks; if they are all on the stack (Lisp machine, SEUS [Weyhrauch;1981]) then there will need to be a count of the number of values that must be examined at various times. Sometimes an implementation may put multiple values in heap allocated storage, which might severely degrade performance. We have not had much experience evaluating multiple value Lisps, so we cannot comment further on them.

Timing function calls has several pitfalls that must be observed, and analyses such as the ones given above can be misleading without a careful observation of these pitfalls. First, the number of arguments passed may make more than a linear difference. For example, the last of several arguments could naturally be computed into the correct register or stack location, causing 0 time beyond the computation for evaluating the argument. Second, if several functions are compiled together or with cross declarations, special cases can be much faster, eliminating the move to a canonical place by the caller followed by a stashing operation by the callee. In this case, also, complete knowledge of register use by each routine can eliminate unnecessary register saving and restoring. Third, numeric function calls can be made faster given suitable representations of numbers. In MacLisp, as noted, stashing and unboxing can be incorporated into a single instruction, MOVE Indirect. Note that these performance improvements are often at the expense either of type safety or of flexibility (separate compilation; defaulting unsupplied arguments, for instance).

An expression such as:

```
((lambda (x ...) ...) ...)
```

is also an example of a function call even though control is not transferred. If x is a free/special variable, then, in a shallow-binding Lisp, there will be a binding operation

upon entry to the lambda and an unbinding upon exit, even in compiled code; in a deep-binding Lisp caching of free/special variables freely referenced in the body of the lambda may take place at entry. In some Lisps the values of lexical variables may be freely substituted for, so that the code

```
((lambda (x)
   (plus (foo) x)) 3)
```

may be exactly equivalent to

```
(plus (foo) 3)
```

Some machine architectures (e.g. Vax) have special features for making function call easier, although these features may be difficult to use in a given Lisp implementation. For example, on the Vax, the CALLS instruction assumes a right to left evaluation order, which is the opposite of Lisp's evaluation order.

Calls from compiled and interpreted functions must be analyzed separately. Calls from interpreted code involves locating the functional object (in some Lisp implementations this requires a search of the property list of the atom whose name is the name of the function.) Calls from compiled functions either involve the same lookup followed by a transfer of control to the code or a simple, machine-specific subroutine call; usually a Lisp will attempt to transform the former into the latter once the function has been looked up. This transformation is called *fast links*, *link smashing*, or *UUO-link smashing* on various systems. Some Lisps (Vax NIL and S-1 Lisp) implement calls to interpreted code via a heap allocated piece of machine code that simply calls the interpreter on the appropriate function application. Hence, calls to both compiled and interpreted code from compiled code look the same. When benchmarking function calls it is imperative to note which of these is being tested.

The requirement for this function lookup is a result of the Lisp philosophy that functions may be defined on the fly by the user, that functions can be compiled separately, that compiled and interpreted calls can be intermixed, and that when an error or interrupt occurs the stack can be decoded within the context of the error. While link-smashing allows separate compilation and free mixing of compiled and interpreted code, it does not allow for frame retention and often does not leave enough information on the stack for debugging tools to decode the call history.

A good example of an implementation with several types of function calling mechanisms is Franz Lisp. It has *slow function call*, which interprets the pointer to the function for each call.[9] This setting allows one to redefine functions at any time. Franz also has *normal function call*, which smashes the address of the function and a direct machine-level call to that code into instances of calls to that function. This usually disallows free redefinitions, hence reducing the debuggability[10] of the resulting code. Finally Franz has *local function call*, which uses a simple load-register-and-jump-to-subroutine sequence in place of a full stack-frame-building call. Functions compiled this way cannot be called from outside the file where they are defined. This is similar to INTERLISP-10 *block compilation*. A final type of function call is a variant of APPLY, called FUNCALL, which takes a function with some arguments and applies the function to those arguments. In Franz on a function-call-heavy benchmark (TAK', described below), running on a Vax 11/780, if normal function call is time 1.0, slow function call is 3.86, and local function call is .523. FUNCALL for this same benchmark (involving an extra argument to each function) is time 2.05.[11]

In addition, if the formal parameters to a function are free/special, then the binding described earlier must

---

[9] Corresponding to the variable NOUUO being T in MacLisp.

[10] As opposed to *Debuggabilly*, the music of hayseed hackers.

[11] The reason that FUNCALL is faster than the slow-function-call case is that the slow-function-call case pushes additional information on the stack so that it is possible to examine the stack upon error.

be performed, adding additional overhead to the function call.

Direct timing, then, requires that the experimenter report the computation needed for argument evaluation, the method of compilation, the number of arguments, the number of values, and this timing must be done over a range of all of these parameters, each duly noted.

Data Structure Manipulation

There are three important data structure manipulations: accessing data, storing into data, and creating new data. For list cells, these are CAR/CDR, RPLACA/RPLACD and CONS.

Several implementations provide other basic data structures besides CONS cells which are useful for building more complex objects. Vectors and vector-like objects[12] help build sequences and record structures; arrays build vectors (in implementations without vectors), matrices, and multidimensional records; and strings are a useful specialization of vectors of characters.

Further, many Lisps incorporate abstract data structuring facilities, such as the INTERLISP DATATYPE facility, the MacLisp EXTEND, DEFSTRUCT, and DEFVST facilities, and the Lisp Machine DEFSTRUCT and FLAVOR facilities. Several of these, especially the FLAVOR facility, also support Object Oriented Programming, much in the style of SMALLTALK.

The following is an example analysis of CONS cell manipulations.

In INTERLISP-10 on a DEC 2060, times for the simple operations are as follows: CAR compiles into a HRRZ, which is on the order of .5 μsec. RPLACA is either .5 μsec (for FRPLACA) or 40–50 μsec (function call + type test). CONS is about 10 μsec (an average of 20 PDP-10 instructions). MacLisp timings are the same for CAR and RPLACA, but faster for CONS, which takes 5 instructions in the non-garbage collection initiating case.

Creating data structures like arrays consists of creating a header and allocating contiguous (usually) storage cells for the elements; changing an element is often modifying a cell; and accessing an element is finding a cell. Finding a cell from indices requires arithmetic for multidimensional arrays.

In MacLisp, for example, array access is on the order of 5 PDP-10 instructions for each dimension when compiled in-line. For fixed-point and floating-point arrays, in which the numeric data are stored in machine representation, access may also involve a number-CONS. Similarly, storing into an array of a specific numeric type may require an unbox.

In some implementations changing array elements involves range checking on the indices, coercing offsets into array type. Pointer array entries in MacLisp are stored 2 per word, so there is coercion to this indexing scheme, which performs a rotate, a test for parity, and a conditional jump to a half-word move to memory. This adds a constant 5 instructions to the $5n$, where n is the number of dimensions, that are needed to locate the entry. Hence storing into an n-dimensional pointer array is on the order of $5(n+1)$ PDP-10 instructions.

Timing CAR/CDR and vector access is most simply done by observing the implementation. Array access is similar but getting the timings involves an understanding how the multidimension arithmetic is done if one is to generalize from a small number of benchmarks.

A basic feature of Lisp systems is that they do automatic storage management, which means that allocating or creating a new object can cause a garbage collection—a reclamation of unreferenced objects. Hence, object creation has a potential cost in garbage collection time which can be amortized over all object creations. Some implementations do incremental

[12] For instance, *hunks* are short, fixed length, vectors in MacLisp.

garbage collection, with each operation on the data type, such as CAR/CDR/RPLACA/RPLACD, performing a few steps of the process. Others delay garbage collection until there is no more free objects or until a threshold is reached. We will discuss garbage collection more in the section dedicated to the subject.

It is sometimes possible to economize storage requirements or shrink the working-set size by changing the implementation strategy for data structures. The primary compound data structure is the CONS cell, which is simply a pair of pointers to other objects. Typically these CONS cells are used to represent lists, and, for that case, it has been observed that the CDR part of the CONS cell often happens to be allocated sequentially after the CONS. As a compaction scheme and as a strategy for increasing the locality—and, hence, reducing the working-set—a method called *CDR-coding* was developed which allows a CONS cell to efficiently state that the CDR is the next cell in memory. However, doing a RPLACD on such an object can mean putting a forwarding pointer in the old CONS cell, finding another cell to which the forwarding pointer will point and which will contain the old CAR and the new CDR, bringing the cost of this relatively simple operation way beyond what is expected. In a reference-count garbage collection scheme this operation added to the reference count updating can add quite a few more operations in some cases. Therefore, it is possible on a machine with CDR-coding to construct a program that will show that that machine is much worse than expected using RPLACD.

The point is that there is a trade-off between compacting data structures and the time for performing certain operations on them.

Type Computations

Lisp supports a runtime typing system. This means that at runtime it is possible to determine the type of an object and take various actions depending on that type. The typing information accounts for a significant amount of the complexity of an implementation; type decoding can be a frequent operation.

There is a spectrum of methods for encoding the type of a Lisp object with the following as the two extremes: the typing information can be encoded in the pointer or it can be encoded in the object. If the type information is encoded in the pointer, then either the pointer is large enough to hold a machine address plus some tag bits (tagged architecture) or the address itself encodes the type; as an example of the latter case, the memory can be partitioned into segments, and for each segment there is an entry in a master type table (indexed by segment number) describing the data type of the objects in the segment. In MacLisp this is called the *BIBOP* scheme (BIg Bag Of Pages) [Steele;1977a].

In most Lisps types are encoded in the pointer. However, if there are not enough bits to describe the subtype of an object in the pointer, the main type is encoded in the pointer and the subtype is encoded in the object. For instance, in S-1 Lisp, a fixed-point vector has the vector type in the tag portion of the pointer and the fixed-point subtype tag in the vector header. In SMALLTALK-80 and MDL the type is in the object, not the pointer.

In tagged architectures (such as the Lisp Machine [Weinreb;1981]) the tags of arguments are automatically used to dispatch to the right routines by the microcode in generic arithmetic. In INTERLISP-D operations such as CAR compute the type for error-checking purposes. In MacLisp interpreted functions check types more often than compiled code, where safety is sacrificed for speed. The speed of MacLisp numeric compiled code is due to the ability to avoid computing runtime types as much as possible.

Microcoded machines typically can arrange for the tag field to be easily or automatically extracted upon memory fetch. Stock hardware can either have byte instructions suitable for tag extraction or can arrange for other field extraction, relying on shift and/or mask instructions in the worst case. Runtime management of

types is one of the main attractions of microcoded Lisp machines.

The following paragraph is an example analysis of some common type checks.

In MacLisp type checking is about 7 instructions totalling about 7 μsec, while in S-1 Lisp it is 2 shift instructions, totalling about .1 μsec. In MacLisp NIL is the pointer 0, so the NULL test is the machine-equality-to-0 test. In S-1 Lisp and Vax NIL there is a NULL type, which must be computed and compared for. S-1 Lisp keeps a copy of NIL in a vector pointed to by a dedicated vector, so a NULL test is a compare against this entry (an indirection through the register).

Since type checking is so pervasive in the language, it is difficult to benchmark the 'type checking facility' effectively.

Arithmetic

Arithmetic is complicated because Lisp passes pointers to Lisp numbers and not machine format numbers. Converting to and from pointer representation is called *boxing* and *unboxing*, respectively. Boxing is also called *number-CONSing*.

The speed of Lisp on arithmetic depends on the boxing/unboxing strategy and on the ability of the compiler to minimize the number of box/unbox operations. To a lesser extent the register allocation performed by the compiler can influence the speed of arithmetic.

Some Lisps attempt to improve the speed of arithmetic by clever encoding techniques. In S-1 Lisp, for instance, the tag field is defined so that positive and negative single precision, fixed-point numbers, consisting of 31 bits of data, are both immediate data and in machine representation with the tag in place.[13] Unboxing of these numbers, thus, is not needed (though type checking is), but after an arithmetic operation on fixed-point numbers, a range check is performed to validate the type. See [Brooks;1982b] for more details on the numeric data types in S-1 Lisp.

MacLisp is noted for its handling of arithmetic on the PDP-10, mainly because of PDL-numbers and a fast small-number scheme. [Fateman;1973] [Steele;1977b] These ideas have been carried over into S-1 LISP [Brooks;1982a].

A PDL-number is a number in machine representation on a stack. This reduces the conversion to pointer format in some Lisps since creating the pointer to the stack allocated number is simpler than allocating a cell in heap space. The compiler is able to generate code to stack-allocate (and deallocate) a number, and create a pointer to it, rather than to heap-allocate it, hence arithmetic where all boxing is PDL-number boxing does not pay a steep number-CONS penalty. In MacLisp there are fixed-point and floating-point stacks; numbers allocated on these stacks are only safe through function calls and are deallocated when the function that created them is exited.

The small-number scheme is simply the pre-CONSing of some range of small integers, so that boxing a number in that range is nothing more than adding the number to the base address of the table. In MacLisp there is actually a table containing these small numbers, while in INTERLISP-10 the table is in an inaccessible area and the indices, not the contents, are used. The MacLisp small-number scheme gains speed at the expense of space.

The range of numbers that a Lisp supports can determine speed. On some machines there are several number-format sizes (single, double, and tetra-word, for instance), and the times to operate on each format may vary. It is important, when evaluating the numeric characteristics of a Lisp, to study the architecture manual to see how arithmetic is done and to know whether the architecture is fully utilized by the Lisp.

---

[13] The Vax Portable Standard Lisp implementation uses a similar scheme for immediate fixed-point numbers.

A constant theme in the possible trade-offs in Lisp implementation design is that the inherent flexibility of runtime type checking is often balanced against the speed advantages of compile time decisions regarding types. This is especially emphasized in the distinction between microcoded implementations in which the runtime type checking can be performed nearly in parallel by the hardware and stock-hardware implementations in which code must be emitted to perform the type checks. Stock-hardware implementations of Lisp often have type specific arithmetic operations (+ is the FIXNUM version of PLUS in MacLisp), while machines with tagged architectures matched to Lisp processing may not support special type-specific arithmetic operators aside from providing entry points to the generic arithmetic operations corresponding to their names.

With arithmetic it is to the benefit of stock hardware to unbox all relevant numbers and perform as many computations in the machine representation as possible. Performing unboxing and issuing type specific instructions in the underlying machine language is often referred to as *open-compiling* or *open-coding*, while emitting calls to the runtime system routines to perform the type dispatches on the arguments is referred to as *closed-compiling* or *closed-coding*.

A further complicating factor in evaluating the performance of Lisp on arithmetic is that some Lisps support arbitrary precision fixed-point (BIGNUM) and arbitrary precision floating-point (BIGFLOAT) numbers.

Benchmarking is an excellent means of evaluating Lisp performance on arithmetic. Since each audience (or user community) can easily find benchmarks suitable for its own needs, we will only mention a few caveats.

Different rounding modes in the floating-point hardware can cause 'same' Lisp code running on two different implementations to have different execution behavior, and a numeric algorithm that converges on one may diverge on the other.

Comparing stock hardware with microcoded hardware may be difficult since a large variability between declared or type-specific arithmetic is possible in non-generic systems. To get best performance out of a stock-hardware Lisp it is necessary to compile with as detailed declarations as possible. For example, in MacLisp the code:

```
(defun test (n)
       (do ((i 1 (1+ i)))
           ((= i n) ())
           <form>))
```

compiles into 9 loop management instructions when no declarations aside from the implicit fixed-point in the operations, 1+ and =, are given, and into 5 loop management instructions when i and n are declared fixed-point. The 40% difference is due to the increased use of PDL-numbers.

3. Lisp Operation Level

Above the instruction level is that of simple Lisp 'operations', i.e., simple common subroutines such as MAPCAR, ASSOC, APPEND, and REVERSE. Each of these is used by many user-coded programs.

If a benchmark uses one of these operations, and one implementation has coded it much more efficiently than the other, then the timings can be more influenced by this coding difference than by other implementation differences. Similarly, using some of these functions to generally compare implementations may be misleading; for instance, microcoded machines may put some of these facilities in firmware.

For example, consider the function DRECONC, which takes two lists, destructively reverses the first, and NCONCs it with the second. This can be written (without error checking) as:

```
(defun dreconc (current previous)
  (prog (next)
   b
    (cond ((null current) (return previous)))
    (setq next (cdr current))
    (rplacd current previous)
    (setq previous current current next)
    (go b)))))))
```

With this implementation the inner loop compiles into 16 instructions in MacLisp. Notice that NEXT is the next CURRENT, and CURRENT is the next PREVIOUS. If we let PREVIOUS be the next NEXT, then we can eliminate the SETQ and unroll the loop. Once unrolled we can reason the same way again to get:

```
(defun dreconc (current previous)
  (prog (next)
   b
    (cond ((null current) (return previous)))
    (setq next (cdr current))
    (rplacd current previous)
    (cond ((null next) (return current)))
    (setq previous (cdr next))
    (rplacd next current)
    (cond ((null previous) (return next)))
    (setq current (cdr previous))
    (rplacd previous next)
    (go b)))
```

With this definition the (unrolled) loop compiles into 29 instructions in MacLisp, or 9.7 instructions per iteration, or roughly $\frac{2}{3}$ the number of instructions, paying an 80% code size cost.

Such things as MAPCAR can be open-coded, and it is important to understand when the compiler in question codes operations open versus closed. INTERLISP uses the LISTP type check for the termination condition for MAPping operations. This is unlike the MacLisp/Common Lisp MAPping operations which use the faster NULL test for termination. On the other hand, if CDR does not type-check, then this NULL test can lead to non-termination of a MAP on a non-list.

4. Major Lisp Facilities

There are several major facilities in Lisp systems that are orthogonal to the subroutine level, but are important to the overall runtime efficiency of an implementation. These include the garbage collection strategy, the interpreter, the file system, and the compiler.

Garbage Collection

There are three major variations on reclaiming unused storage in Lisp systems, and many minor variations, each of which have different performance profiles.[14]

The 'mark and sweep' variety chases all active pointers, marking all reachable objects as active, and then putting back on the free list any objects that weren't marked. Variations segment the address space in different ways and make optimizations based on arguments about what segments can point to which others. For example, INTERLISP-10 keeps pages marked as 'pure', 'dirty', or 'user' and avoids both sweeping system pages and chasing pointers from 'system' pages which have not been written on.

The '(stop and) copy' variety copies all reachable objects from old-space to new-space, and then swaps old and new; for example, INTERLISP-Vax uses stop-and-copy. In addition to segmentation schemes, there are copying garbage collectors which can work continuously when doing memory references, given some hardware/microcode assist.[Baker;1978a] Continuous garbage collectors eliminate the 'nasty pause' that stop-and-copy garbage collectors suffer.

The 'reference-count' variety counts the number of references to each object, and when a reference count goes to 0, the object can be immediately collected. INTERLISP-D uses a modified reference-count algorithm, where reference counts are kept in a separate hash table rather than with the objects, and the case where reference-count-equals-1 is not stored; in addition, references from the stack (e.g., as the result of a variable reference) and to immediate data (atoms and integers) are not counted. This requires periodically

[14] A good survey of garbage collection algorithms can be found in [Cohen;1981].

sweeping the stack to find pointers from there and then sweeping the reference-count table to find data whose reference count is 0.

Both mark-and-sweep and stop-and-copy suffer in that the cost of collecting garbage is not proportional to the amount of garbage. Let A be the size of the address space, G be the amount of garbage, and R the size of the reachable objects; note that one may have $R + G < A$ because one may decide to garbage collect before storage is exhausted. Then mark-and-sweep is $O(R + A)$, but stop-and-copy is $O(R)$. The various segmentation schemes help, but the heuristics for separating the address space work only in limited contexts. For large programs, garbage collection time is quite significant not because much garbage is being collected but because the active address space is large.

Reference counting schemes suffer in that circular structures are not collected, causing one periodially to have to perform a stop-and-copy operation; this may not in fact be a problem in practice, because it is possible to explicitly control the allocation of circular storage, and because uncollectable garbage is just as likely from sources which are shared among all garbage collection algorithms, e.g., CONSing data onto the property list of a permanent atom.

Since incremental garbage collectors charge some garbage collection time to common operations, we feel that garbage collection time should be amortized over data structure creation.

Measuring garbage collectors is difficult because the time that a particular garbage collection takes depends on factors that are hard to control for; e.g., the amount of garbage, the working-set size, the stack depth, and the atom hash-table population.

Interpreter

Interpreter speed depends primarily on the speed of type dispatching, variable lookup and binding, macro expansion, and call-frame construction. Lexically-bound Lisps spend time keeping the proper contours visible or hidden, so that a price is paid at either environment creation time or lookup/assignment time. Some interpreters support elaborate error correction facilities, such as declaration checking in S-1 Lisp, which can slow down some operations.

Interpreters usually are carefully hand-coded, and this hand-coding can make a factor of two difference. Having interpreter primitives in microcode may help, but in stock hardware this hand-coding can result in difficult-to-understand encodings of data. The time to dispatch to internal routines (e.g., to determine that a particular form is a COND and to dispatch to the COND handler) is of critical importance in the speed of an interpreter.

Shallow binding versus deep binding matters more in interpreted code than in compiled code, since a deep-binding system looks up each of the variables when it are used unless there is a lambda-contour-entry penalty. For example, in S-1 Lisp, when a lambda is encountered, a scan of the lambda-form is performed, and the free/special variables are cached (S-1 Lisp is deep-binding).

The interpreter is mainly used for debugging, and when compiled and interpreted code are mixed the ratio of compiled to interpreted code execution speeds is the important performance measure. Of course, the relative speed of interpreted to compiled code is not constant over all programs, since a program that performs 90% of its computation in compiled code will not suffer much from the interpreter dispatching to that code. Similarly, on some deep-binding Lisps, the interpreter can be made to spend an arbitrary amount of time searching the stack for variable references when the compiler can find those at compile-time (e.g., globals).

Also, one's intuitions on the relative speeds of interpreted operations may be wrong. For example, consider the case of testing a fixed-point number for 0; there are three basic techniques:

```
(zerop n)
(equal n 0)
(= n 0)
```

In compiled code, where declarations of numeric type are used, one expects that ZEROP and = would be about the same and EQUAL would be slowest; this is true in MacLisp. However, in the MacLisp interpreter ZEROP is fastest, then EQUAL, and finally =. This is odd because = is supposedly the fixed-point-specific function that implicitly declares it arguments. The discrepancy is about 20% from ZEROP to =.

The analysis is that ZEROP takes only 1 argument and so the time spent managing arguments is substantially smaller. Once the argument is obtained a type dispatch and machine-equality-to-0 is performed.

EQUAL first tests for EQ, which is machine-equality-of-address, after managing arguments. In the case of equal small integers in a small-number system, the EQ test succeeds. Testing for EQUAL of two numbers beyond the small integer range (or comparing two unequal small integers in a small-number system) is then handled by type dispatch on the first argument, and then machine equality of the value pointed to by the pointers.

= manages two arguments and then dispatches individually on the arguments, so that if one supplies a wrong type argument, they can both be described to the user.

File Management

The time spent interacting with the programming environment itself has increasingly become apparent as an important part of the 'feel' of a Lisp, and its importance should not be underestimated. There are three cases where file read time come into play: when loading program text, when loading compiled code (this code may be in a different format), and when reading user data structures. The time for most of these is in the READ, PRINT (PRETTYPRINT), and filing system, in basic file access (e.g., disk or network management), and in the operating system interface.

Loading files involves locating atoms on the atom hash-table (often referred to as the *oblist* or *obarray*); in most Lisps, this is a hash table. Something can be learned by studying the size of the table, the distribution of the buckets, etc. One can time atom hash-table operations to effect, but getting a good range of variable names (to test the distribution of the hashing function) might be hard, and getting the table loaded up effectively can be difficult.

On personal machines with a relatively small amount of local file storage, access to files may require operation over a local network. Typically these are contention networks in the 1–10 megabyte per second speed range (examples are: 3-megabyte Ethernet, 10-megabyte Ethernet, Chaosnet). The response time on a contention network can be slow when it is heavily loaded, and this can degrade the perceived pep of the implementation. Additionally, the file server can be a source of slowdown.

Compiler

Lisp compiler technology has grown rapidly in the last 15 years. Early recursive-descent compilers generated simple, often ridiculous, code on backing out of an execution-order treewalk of the program. Some modern Lisp compilers resemble the best optimizing compilers for algorithmic languages. [Brooks;1982a], [Masinter;1981b]

Interpreting a language like Lisp involves examining an expression and determining the proper runtime code to execute. Simple compilers eliminate, essentially, the dispatching routine and generate calls to the correct routines with some bookkeeping for values in between.

Fancier compilers do a lot of open-coding, generating the body of a routine in-line with the code that calls it. A simple example is CAR which consists of the instruction HRRZ on the PDP-10. The runtime routine

for this will do the HRRZ and then POPJ on the PDP-10. The call to CAR (in MacLisp style) will look like:

```
move a,<arg>
pushj p,<car>
move <dest>,a
```

And CAR would be:

```
hrrz a,(a)
popj p,
```

if no type checking were done on its arguments. Open-coding of this would be simply:

```
hrrz <dest>,0(<arg>)
```

Other types of open-coding involve generating the code for a control structure in-line. For example, MAPC will map down a list and apply a function to each element. Rather than simply calling such a function, a compiler might generate the control structure code directly, often doing this by transforming the input code into equivalent, in-line Lisp code and then compiling that.

Further optimizations involve delaying the boxing or number-CONSing of numbers in a numeric computation. Some compilers also rearrange the order of evaluation, do constant-folding, loop-unwinding, common-subexpression elimination, register optimization, cross optimizations (between functions), peephole optimization, and many of the other classical compiler techniques.

When evaluating a compiler it is important to know what the compiler in question can do. Often looking at some sample code produced by the compiler for an interesting piece of code is a worthwhile evaluation technique for an expert. Knowing what is open-coded, what constructs are optimized, and how to declare facts to the compiler in order to help it produce code are the most important things for a user to know.

A separate issue is "how fast is the compiler?" and, in some cases, the compiler is slow even if the code it generates is fast: for instance, it can spend a lot of time doing optimization.

5. The Art of Benchmarking

Benchmarking is a black art at best; the normal effect on people of stating the results of a particular benchmark on two Lisp systems is to cause the audience to believe that a blanket statement is being made ranking the systems in question. The proper role of benchmarking is to measure various dimensions of Lisp system performance and to order those systems along each of these dimensions. At that point, informed users will be able to choose a system adequate for the desired purposes or will be able to tune their programming style to the performance profile.

Know What is Being Measured

The first problem associated with benchmarking is knowing what is being tested.

Consider the example of the TAK′ function:

```
(defun tak' (x y z)
       (cond ((not (< y x))
              z)
             (t (tak' (tak' (1- x) y z)
                      (tak' (1- y) z x)
                      (tak' (1- z) x y))))))
```

What does this function measure if used as a benchmark? Careful examination shows that function call, simple arithmetic (small-integer arithmetic, in fact), and a simple test is all that this function performs; there is no storage allocation at all. In fact, this function applied to the arguments 18, 12, and 6 performs 63609 function calls, has a maximum recursion depth of 18, and has an average recursion depth of 15.4.

On a PDP-10 (in MacLisp) this means that this benchmark tests the stack instructions, data moving, and some arithmetic, as we see from the code the compiler produces:

```
tak':                   movei a,(fxp)
push p,[0,,fix1]        push fxp,tt
push fxp,(a)            pushj p,tak'+1
push fxp,(b)            move d,-3(fxp)
push fxp,(c)            subi d,1
move tt,-1(fxp)         movei c,-4(fxp)
camge tt,-2(fxp)        movei b,-5(fxp)
jrst g2                 push fxp,d
move tt,(fxp)           movei a,(fxp)
jrst g1                 push fxp,tt
g2:                     pushj p,tak'+1
move tt,-2(fxp)         push fxp,tt
subi tt,1               movei c,(fxp)
push fxp,tt             movei b,-1(fxp)
movei a,(fxp)           movei a,-3(fxp)
pushj p,tak'+1          pushj p,tak'+1
move d,-2(fxp)          sub fxp,[6,,6]
subi d,1                g1:
movei c,-3(fxp)         sub fxp,[3,,3]
movei b,-1(fxp)         popj p,
push fxp,d
```

One expects the following sorts of results. A fast stack machine might do much better than its average instruction speed would indicate. In fact, running this benchmark written in C on both the Vax 11/780 and a Motorola MC68000 (using 16-bit arithmetic in the latter case), one finds that the MC68000 time is 71% of the Vax 11/780 time. Assuming that the Lisps on these machines would maintain this ratio one would expect that the MC68000 is a good Lisp processor. However, in a tagged implementation, the MC68000 fares poorly, since field extraction is not as readily performed on the MC68000 as on the Vax. An examination of all instructions and the results of a number of benchmarks we have run lead to the conclusion that the MC68000 performs at about 40% of a Vax 11/780 when running Lisp.

As mentioned earlier the locality profile of this benchmark is not typical of 'normal' Lisp programs, and the effect of the cache memory may dominate the performance. Let us consider the situation in MacLisp on the Stanford Artificial Intelligence Laboratory KL-10A (SAIL) which has a 2k-word, 200-nanosecond cache memory and a main memory consisting of a 2-megaword, 1.5-μsec memory and a 256-kiloword, .9-μsec memory. On SAIL, the cache memory allows a very large, but slow, physical memory to behave reasonably well.

This benchmark was run with no load and the result was:

```
cpu time = 0.595
elapsed time = 0.75
wholine time = 0.75
gc time = 0.0
load average before = 0.020
load average after  = 0.026
```

where CPU time is the EBOX time (no memory reference time included), elapsed time is real time, wholine time is EBOX + MBOX (memory reference) times, GC time is garbage collector time, and the load averages are given before and after the timing run, where all times are in seconds and the load average is the exponentially weighted average of the number of jobs in all runnable queues. With no load, wholine and elapsed times are the same.

There are two ways to measure the effect of the extreme locality of TAK': one is to run the benchmark with the cache memory shut off; another is to produce a sequence of identical functions, called TAK'_i, which call functions TAK'_j, TAK'_k, TAK'_l, and TAK'_m with uniform distribution on j, k, l, and m.

With 100 such functions, and no load the result on SAIL was:

```
cpu time = 0.602
elapsed time = 1.02
wholine time = 1.02
gc time = 0.0
load average before = 0.27
load average after  = 0.28
```

which shows a 36% degradation. The question is how well these 100 functions destroy the effect of the cache. The answer is that it does not destroy the effect very much. This makes sense because the total number of instructions for 100 copies of the function is about 3800, and the cache holds about 2000 words. We were able to run both benchmarks with the cache off at SAIL. Here is the result for the single function TAK':

```
cpu time = 0.6
elapsed time = 6.95
wholine time = 6.9
gc time = 0.0
load average before = 0.036
load average after  = 0.084
```

which shows a factor of 9.2 degradation. The 100 function version ran in the same time within a few percent.

Hence, in order to destroy the effect of a cache, one must increase the size of the code to significantly beyond the size of the cache. Also, the distribution of the locus of control must be roughly uniform or random.

This example also illustrates the point about memory bandwidth discussed earlier. The CPU has remained constant in its speed with the cache on or off (.595 versus .6), but the memory speed of 1.5 μsec has caused a factor of over 9 slowdown in the overall execution speed of Lisp.[15]

Some Lisp compilers perform *tail recursion* removal. A tail-recursive function is one whose value is sometimes returned by a function application (as opposed to an open-codable operation). Hence in the function, TAK′, the second arm of the COND states that the value of the function is the value of another call on TAK′, but with different arguments. If a compiler does not handle this case then another call frame will be set up, control will transfer to the other function (TAK′, again, in this case), and control will return only to exit the first function immediately. Hence there will be additional stack frame management overhead for no useful reason, and the compiled function will be correspondingly inefficient. A smarter compiler will re-use the current stack frame, and when the called function returns it will return directly to the function which called the one which is currently invoked.

The INTERLISP-D compiler does tail recursion removal and the MacLisp compiler handles some simple cases of tail recursion; it does no such removal in TAK′.

[15] With the cache off, the 4-way interleaving of memory benefits are abandoned, further degrading the factor of 7.5 speed advantage the cache has over main memory on SAIL.

Earlier we mentioned that MacLisp has both small-number CONSing and PDL numbers. It is true that:

$$\text{TAK}'(x+n, y+n, z+n) = \text{TAK}'(x, y, z) + n$$

and therefore it might be expected that if MacLisp used the small-number CONS in TAK′, and if one chose n as the largest small-integer in MacLisp, the effects of small-number CONSing could be observed. However, the PDP-10 code for TAK′ above demonstrates that it is using PDL numbers. Also, by timing various values for n one can see that there is no significant variation, and that, therefore, the coding technique cannot be number size dependent (up to BIGNUMs).

Hence analysis and benchmarking can be valid alternative methods for inferring the structure of Lisp implementations

Measure the Right Thing

Often a single operation is too fast to time directly, so that it is apparent that the operation must be performed many times; the total time for that is used to compute the speed of the operation. Simple loops to accomplish this appear straightforward, but it might be the case that the benchmark is testing mainly the loop construct. Consider the MacLisp program:

```
(defun test (n)
  (declare (special x)(fixnum i n x))
       (do ((i n (1- i)))
           ((= i 0))
           (setq x i)))
```

which assigns a number to the special variable, x, n times. What does this program do? First, it number-CONSes for each assignment. Second, of the 6 instructions the MacLisp compiler generates for the inner loop, 4 manage the loop counter, its testing, its modification, and the flow of control, 1 is a fast, internal subroutine call to the number-CONSer, and 1 is used for the assignment (moving to memory). So, 57% of the code is the loop management.

```
    push fxp,(a)
g2: move tt,(fxp)
    jumpe tt,g4
    jsp t,fxcons
    movem a,(special x)
    sos fxp
    jrst g2
g4: movei a,'()
    sub fxp,[1,,1]
    popj p,
```

To measure operations that require looping, measure the loop alone (i.e. measure the null operation) and subtract that from the results.

As mentioned in the previous section, even here one must be aware of what is being timed, since the number-CONSing is the time sink in the statement: `(setq x 1)`.

In INTERLISP-D and in S-1 Lisp it makes a big difference whether you are doing a global/free or lexical variable assignment. If `x` were a local variable, the compiler would optimize the SETQ away (assignment to a dead variable).

Know How the Facets Combine

Sometimes a program that performs two basic Lisp operations will combine them non-linearly. For instance, a compiler might optimize two operations in such a way that their operational characteristics are interleaved or unified; some garbage collection strategies can interfere with the effectiveness of the cache.

One way to measure those characteristics relevant to a particular audience is to benchmark large programs of interest to that audience that are of sufficient size that the combinational aspects of the problem domain are unified reasonably. For example, part of an algebra simplification or symbolic integration system might be an appropriate benchmark for a group of users implementing and using a MACSYMA-like system.

The problems with using a large system for benchmarking are that the same Lisp code may or may not run on the various Lisp systems, or the obvious translation might not be the best implementation of the benchmark for a different Lisp system.

For instance, a Lisp without multidimensional arrays might choose to implement then as arrays whose elements are other arrays, or it might be appropriate to use lists of lists if the only operations on the multidimensional array involve scanning through the elements in a predetermined order. A reasoning program that uses floating-point numbers, $0 \leq x \leq 1$ on one system might use fixed-point arithmetic with numbers $0 \leq x \leq 1000$ on another.

Another problem is that it is often difficult to control for certain facets in a large benchmark. For example, the history of the address space that is being used for the timing can make a difference: e.g., the number of CONSes done so far and how full the atom hash-table is.

Personal Versus Timeshared Systems

The most important and difficult question associated with timing a benchmark is exactly how to time it. This is especially a problem when one is comparing a personal machine to a timeshared machine.

Obviously, the final court of appeal is the amount of time that a person has to wait for a computation to finish. On timeshared machines one wants to know both what the best possible time is and how that time varies. CPU time (including memory references) is a good measure for the former, while the latter is measured in elapsed time. For example, one could obtain an approximate mapping from CPU time to elapsed time under various loads and then do all of the timings under CPU time measurement. This mapping is at best approximate since elapsed time depends not only on the load (number of other active users) but also on what all users are and were doing.

Timesharing systems often do background processing which doesn't get charged to any one user. TENEX, for example, writes dirty pages to the disk as part of a system process rather than as part of the user process. On SAIL some system interrupts may be charged to a user process. When using runtime reported by the

timesharing system one is sometimes not measuring these necessary background tasks.

Personal machines, it would seem, are easier to time because elapsed time and CPU time (with memory references) are the same. However, sometimes a personal machine will perform background tasks such as monitoring the keyboard and the network, which can be disabled. On the Xerox 1100 running INTERLISP, turning off the display can increase Lisp execution speed by more than 30%. When using elapsed time one is measuring these stolen cycles as well as those going to execute Lisp.

A sequence of timings was done on SAIL with a variety of load averages. Each timing measures EBOX time, EBOX + MBOX (memory reference) time, elapsed time, garbage collection time, and the load averages before and after the benchmark. For load averages $.2 \leq L \leq 10$, the elapsed time, E, behaved as:

$$E = \begin{cases} C(1 + K(L-1)), & L > 1; \\ C, & L \leq 1. \end{cases}$$

That is, the load had a linear effect for the range tested. We have not tested the effect of load averages on elapsed time on any other machines.

The quality of interaction is an important consideration, and in many cases the personal machine provides a much better environment. In other cases, the need for high absolute performance means that a timeshared system is preferable.[16]

Measure the Hardware Under All Conditions

In some architectures jumps across page boundaries are slower than jumps within page boundaries, and the performance of a benchmark can thus depend on the alignment of inner loops within page boundaries. Further, working-set size can can make a large performance difference; the physical memory size can be a more dominating factor than CPU speed on benchmarks with large working-sets. Measuring CPU time (without memory references) in conjunction with a knowledge of the approximate mapping from memory size to CPU + memory time would be an ideal methodology, but is difficult to do.

An often informative test is to take some reasonably small benchmarks and to code them as efficiently as possible in an appropriate language in order to obtain the best possible performance of the hardware on those benchmarks. This was done on SAIL with the TAK′ function mentioned earlier. In MacLisp the time was .68 seconds of CPU + memory time; in assembly language (heavily optimized) it was .255 seconds, or about a factor of 2.5 better.[17]

6. Conclusions

Benchmarks are useful when the associated timings are accompanied by an analysis of the facets of Lisp and the underlying machine that are being measured. To claim that a simple benchmark is a uniform indicator of worth for a particular machine in relation to others is not proper use of benchmarking. This is why this paper goes to great lengths to explain as many tradeoffs as possible so that the potential benchmarker and the benchmarker's audience are aware of the pitfalls of this exercise.

Benchmarks are useful for comparing performance of Lisp implementations. They help programmers tailor their programming styles and help with programming decisions where performance is a consideration, although we do not recommend that programming style should take a back seat to performance. Benchmarks help identify weak points in an implementation so that

[16] The authors will make no value judgments in either direction: one is working on advanced personal Lisp workstations, and the other is working on Lisp for a large, timeshared multiprocessor.

[17] This factor is not necessarily expected to hold up uniformly over all benchmarks.

efforts to improve the Lisp can focus on the points of highest leverage.

Computer architectures have become complex enough that it is often difficult to analyze program behavior in the absence of a set of benchmarks to guide that analysis. It is often difficult to perform an accurate analysis without doing some experimental work to guide the analysis and keep it accurate; without analysis it is difficult to know how to benchmark correctly.

The final arbiter of the usefulness of a Lisp implementation is the ease that the user and programmer have with that implementation. Performance is an issue, but it is not the only issue.

References

[Baker;1978a]
Baker, H. B. *List Processing in Real Time on a Serial Computer*, Communications of the ACM, Vol. 21, no. 4, April 1978.

[Baker;1978b]
Baker, H. B. *Shallow Binding in Lisp 1.5*, Communications of the ACM, Vol. 21, no. 7, July 1978.

[Bates;1982]
Bates, R., Dyer, D., Koomen, H. *Implementaion of Interlisp on a Vax*, Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, August 1982.

[Brooks;1982a]
Brooks, R. A., Gabriel, R. P., Steele, G. L. *An Optimizing Compiler For Lexically Scoped Lisp*, Proceedings of the 1982 ACM Compiler Construction Conference, June, 1982.

[Brooks;1982b]
Brooks, R. A., Gabriel, R. P., Steele, G. L. *S-1 Common Lisp Implementation*, Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, August 1982.

[Burton;1981]
Burton, R. R, et. al. *Interlisp-D Overview* in **"Papers on Interlisp-D"**, Xerox Palo Alto Research Center, CIS-5 (SSL-80-4), 1981.

[Cohen;1981]
Cohen, J. *Garbage Collection of Linked Data Structures*, ACM Computing Surveys, Vol. 13, no. 3, September 1981.

[Correll;1979]
Correll, Steven. *S-1 Uniprocessor Architecture (SMA-4)* in **"The S-1 Project 1979 Annual Report"**, Chapter 4. Lawrence Livermore National Laboratory, Livermore, California, 1979.

[Fateman;1973]
Fateman, R. J. *Reply to an Editorial*, ACM SIGSAM Bulletin 25, March 1973.

[Foderaro;1982]
Foderaro, J. K., Sklower, K. L. **"The FRANZ Lisp Manual"**, University of California, Berkeley, Berkeley, California, April 1982.

[Griss;1982]
Griss, Martin L, Benson, E. *PSL: A Portable LISP System*, Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, August 1982.

[Lampson;1982]
Lampson, Butler W. *Fast Procedure Call*, Proceedings of the 1982 ACM Symposium on Architectural Support for Programming Languages and Operating Systems, SIGARCH Computer Architecture News, Vol. 10 no.2, March 1982.

[Masinter;1981a]
Masinter, L. **"Interlisp-VAX: A Report"**, Department of Computer Science, Stanford University, STAN-CS-81-879, August 1981.

[Masinter;1981b]
Masinter, L. M., Deutsch, L. P. *Local Optimization For a Compiler for Stack-based Lisp Machines* in **"Papers on Interlisp-D"**, Xerox Palo Alto Research Center, CIS-5 (SSL-80-4), 1981.

[Steele;1977a]
Steele, Guy Lewis Jr. *Data Representations in PDP-10 MacLisp*, Proceedings of the 1977 MACSYMA Users' Conference. NASA Scientific and Technical Information Office, Washington, D.C., July 1977.

[Steele;1977b]
Steele, Guy Lewis Jr. *Fast Arithmetic in MacLisp*, Proceedings of the 1977 MACSYMA Users' Conference. NASA Scientific and Technical Information Office, Washington, D.C., July 1977.

[Steele;1979]
Steele, Guy Lewis Jr., Sussman, G. J. *The Dream of a Lifetime: A Lazy Scoping Mechanism*, Massachusetts Institute of Technology AI Memo 527, November 1979.

[Steele;1982]
Steele, Guy Lewis Jr. et. al. *An Overview of Common Lisp*, Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, August 1982.

[Teitelman;1978]
Teitelman, Warren, et. al. **"Interlisp Reference Manual"**, Xerox Palo Alto Research Center, Palo Alto, California, 1978.

[Weinreb;1981]
Weinreb, Daniel, and Moon, David. **"LISP Machine Manual"**, Fourth Edition. Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, July 1981.

[Weyhrauch;1981]
Weyhrauch, R. W., Talcott, C. T., Scherlis, W. L., Gabriel, R. P.; personal communication and involvement.

[White;1979]
White, J. L., *NIL: A Perspective*, Proceedings of the 1979 MACSYMA Users Conference, July 1979.