Chapter 1

Introduction

This is the final report of the Stanford Lisp Performance Study, which was conducted by the author during the period from February 1981 through October 1984. This report is divided into three major parts: the first is the theoretical background, which is an exposition of the factors that go into evaluating the performance of a Lisp system; the second part is a description of the Lisp implementations that appear in the benchmark study; and the last part is a description of the benchmark suite that was used during the bulk of the study and the results themselves.

This chapter describes the issues involved in evaluating the performance of Lisp systems and is largely a reprint of the paper "Performance of Lisp Systems" by Richard P. Gabriel and Larry Masinter. The various levels at which quantitative statements can be made about the performance of a Lisp system are explored, and examples from existing implementations are given wherever possible. The thesis is that benchmarking is most effective when performed in conjunction with an analysis of the underlying Lisp implementation and computer architecture. Some simple benchmarks which have been used to measure Lisp systems examined, as well as some of the complexities of evaluating the resulting timings, are examined.

Performance is not the only—or even the most important—measure of a Lisp implementation. Trade-offs are often made that 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. Implementation details 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.

The technical issues and trade-offs that determine the efficiency and usability of a Lisp implementation will be explained in detail; though there will appear to be a plethora of facts, only those aspects of a Lisp implementation that are the most important for evaluation will be discussed. Throughout, the impact of these issues and trade-offs on benchmarks and benchmarking methodologies will be explored.

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

1.1 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. The following levels are 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 and this leads 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, while 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.

1.1.1 *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 a CPU with a basic clock rate of 50 nanoseconds to run a Lisp system faster than the same architecture 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, the instruction buffering and decoding, and the size of data paths. The most important of these facilities will be described in the rest 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 accurately measure the total performance by failing to demonstrate 'normal' locality. Hence, one would expect the small-benchmark methodology to

---

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

[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.

tend to result in optimistic measurements, since small programs have 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—without a relatively high bandwidth for a given CPU speed, there will not be an effective utilization of that CPU. As an extreme case, consider a 50-nanosecond machine with 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 because a significant number of the total machine cycles go into things, such as condition testing and instruction dispatch, that are not data-path limited.

Many other subtle aspects 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 appropriately on these 64-bit quantities 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.

1.1.2 *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.

At the 'instruction level' Lisp is more complex than 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—that is, they share the same implementation strategy. For example, there are two distinct strategies for implementing free/special variables in Lisp—*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. 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 that operation exclusively and that time the execution in one of several ways. The operations will be discussed before the benchmarking techniques.

1.1.2.1 *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 variable as an argument, referencing a constant, and referencing lexical and global variables.

Typically, bound variables are treated as lexical variables by the compiler. The compiler is free to assign a lexical variable to any location (or more prop-

[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 that can be lambda-bound. In this report the terms *lexical* or *local* will be used for nonglobal, nonfluid variables, *global* for global variables, and *free/special* for global and fluid variables.

erly, to assign any location the name of the lexical variable at various times). Typical locations for temporaries, both user-defined and compiler-defined, are the registers, the stack, and memory. Since Lisp code has a high proportion of function calls to other operations, one expects register protection considerations to mean that temporaries are generally stored on the stack or in memory. In addition, since many Lisp programs are recursive, their code must be re-entrant and, hence, must be read-only. This argues against general memory assignment of temporaries. Consequently, most lexical variables are assigned to the stack in many Lisp implementations. Variables that are in registers can be accessed faster than those in memory, although cache memories reduce the differential.[4]

Compilation of references to constants can be complicated by the fact that, depending on the garbage collection strategy, the constants can move. Thus, either the garbage collector must be prepared to relocate constant pointers from inside code streams or the references must be made indirect through a reference-table. Sometimes, the constants are 'immediate' (i.e., the bits can be computed at compile time). On some systems, constants are in a read-only area, and pointers to them are computed at load time. Immediate data are normally faster to reference than other kinds, since the operand-fetch-and-decode unit performs most of the work.

1.1.2.2 *Free/Special Variable Lookup and Binding*

There are two primary methods for storing the values of free/special variables: *shallow binding* and *deep binding.* Deep binding is conceptually similar to ALIST binding; ⟨variable name, value⟩ pairs are kept on a stack, and looking up the value of a variable consists of finding the most recently bound ⟨variable name, value⟩ pair. Binding a free/special variable is simply placing on the stack a new pair that will be found before any previous pairs with the same variable name in a sequential search backwards along the variable lookup path (typically this is along the control stack).

A shallow-binding system has a cell called the *value cell* for each variable. The current value of the variable with the corresponding name is always found

[4] And, in fact, on some machines the cache may be faster than the registers, making some memory references faster than register references. A good example is the KL-10, where, unlike KA-10, it is slower to execute instructions out of registers and to fetch registers as memory operands than it is to perform those operations from the cache.

there. When a variable is bound, a ⟨variable name, old value⟩ pair is placed on a stack so that when control is returned beyond the binding point, the old value is restored to the value cell. Hence, lookup time is constant in this scheme.

The performance profiles for free/special lookup and binding are very different depending on whether you have deep or shallow binding. In shallow-binding implementations, times for function call and internal binding of free/special variables are inflated because of the additional work of swapping bindings. On some deep-binding systems, referencing a dynamically bound variable (which includes all variable references from the interpreter) can require a search along the access path to find the value. Other systems cache pointers to the value cells of freely referenced free/special variables on top of the stack; caching can take place upon variable reference/assignment or upon entry to a new lexical contour,[5] and at each of these points the search can be one variable at a time or all/some variables in parallel. 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 multi-processing 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 (in-

---

[5] A *lexical contour* is the real or imaginary boundary that occurs at a LAMBDA, a PROG, a function definition, or at any other environment construct. This terminology is not universal.

[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]

voking 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 demand 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 just variable references can be difficult because a compiler can make decisions that may not reflect intuition, such as optimizing out unreferenced variables.

1.1.2.3 *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: 1) building a stack frame, moving addresses of computed arguments into that frame, placing a return address in it, and transferring control; and 2) 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 be done either by the caller or the callee, or

---

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

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.

Function call and return time are grouped together because every function call is normally paired with a function return. It is possible for a function to exit via other means, for example, via the nonlocal exits such as RETFROM in INTERLISP and THROW in MacLisp. As it searches for the matching CATCH, THROW does 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 either 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. Upon function entry, numeric code '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]), there may be an impact on how the register allocator can operate, and this may cause memory bottlenecks. If they are all on the stack

(Lisp machine, SEUS [Weyhrauch 1981]), a count of the number of values that must be examined must be made at various times. Sometimes an implementation may put multiple values in heap-allocated storage. This could severely degrade performance.

Timing function calls has several pitfalls that should be noted as analyses such as the ones given above can be misleading. 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 zero 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 involve 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 involve either 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.

Franz Lisp is a good example of an implementation with several types of function-calling mechanisms. 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 and hence reduces 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

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

[10] As contrasted with *Debuggabilly*, the music of hayseed hackers.

a function with some arguments and applies the function to those arguments. In Franz, if normal function call is time 1.0 on a function-call-heavy benchmark (TAK', described below) running on a Vax 11/780, slow function call is 3.95, 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 be performed, and this adds 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, and the number of values. The timing must be done over a range of all of these parameters, with each being duly noted.

1.1.2.4 *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.

In addition to CONS cells, several implementations provide other basic data structures that 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.

---

[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.

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

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 two 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 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 garbage collection with each operation (such as CAR/CDR/RPLACA/RPLACD) on the data type performing a few steps of the process. Others delay garbage collection until there are no more

free objects or until a threshold is reached. Garbage collection will be discussed in detail in a subsequent section.

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 that 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 and finding another cell to which the forwarding pointer will point and which will contain the old CAR and the new CDR. All this could bring 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, on a machine with CDR-coding it is possible to construct a program that performs many RPLACDs and that by doing so will show the machine to be much worse than expected (where that expectation is based on other benchmarks).

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

1.1.2.5 *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 and the following are 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, in 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 register, 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.

1.1.2.6 *Arithmetic*

Arithmetic is complicated because Lisp passes pointers to machine format numbers rather than passing machine format numbers directly. Converting to and from pointer representation is called *boxing* and *unboxing*, respectively. Boxing is also called *number*-CONS*ing*.

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 all positive and negative single-precision fixed-point numbers consisting of 31 bits of data are both immediate data and machine format integers with their tags in place.[13] Thus, unboxing of these numbers 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 to create a pointer to it rather than to heap-allocate it; hence, arithmetic in which 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

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

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 tetraword, for instance), and the times to operate on each format may vary. When evaluating the numeric characteristics of a Lisp, it is important to study the architecture manual to see how arithmetic is done and to know whether the architecture is fully utilized by the Lisp.

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 to suit its own needs, only a few caveats will be mentioned.

Different rounding modes in the floating-point hardware can cause the '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 the best performance out of a stock-hardware Lisp, one must compile with declarations that are as detailed 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.

1.2 Lisp Operation Level

Simple Lisp 'operations,' i.e., simple, common subroutines such as MAPCAR, ASSOC, APPEND, and REVERSE, are located above the instruction level. Each is used by many user-coded programs.

If a benchmark uses one of these operations and if one implementation has coded it much more efficiently than another, then the timings can be influenced more 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 the loop is 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, which is 9.7 instructions per iteration, or roughly $\frac{2}{3}$ the number of original instructions. It pays 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 nontermination of a MAP on a nonlist.

1.3 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 collector, the interpreter, the file system, and the compiler.

1.3.1 *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, that can slow down some operations.

Interpreters usually are carefully handcoded, and this handcoding can make a difference of a factor of two. Having interpreter primitives in microcode may help, but in stock hardware this handcoding 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 is used unless there is a lambda-contour-entry penalty. For example, when a lambda is encountered in S-1 Lisp, 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; 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 these 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)
```

Where declarations of numeric type are used in compiled code, 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 its arguments. The discrepancy is about 20% from ZEROP to =.

The analysis is that ZEROP takes only one argument, and so the time spent managing arguments is substantially smaller. Once the argument is obtained, a type dispatch and a machine-equality-to-0 are 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; next, machine equality of the values is 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.

1.3.2 *File Management*

The time spent interacting with the programming environment itself has become an increasingly important part of the 'feel' of a Lisp, and its importance should not be underestimated. There are three times when file read time comes 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 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 megabit per second speed range (examples are 3-megabit Ethernet, 10-megabit 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.

1.3.3 *Compiler*

Lisp compiler technology has grown rapidly in the last 15 years. Early recursive-descent compilers generated simple and 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 essentially eliminate 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>,@<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?" 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.

1.4 The Art of Benchmarking

Benchmarking is a black art at best. Stating the results of a particular benchmark on two Lisp systems usually causes people to believe that a blanket statement ranking the systems in question is being made. 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 that meets their requirements or will be able to tune their programming style to the performance profile.

1.4.1 *Know What is Being Measured*

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

Consider the following 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))))))
```

If used as a benchmark, what does this function measure? Careful examination shows that function call, simple arithmetic (small-integer arithmetic, in fact), and a simple test are all that this function performs; no storage allocation is done. In fact, when applied to the arguments 18, 12 and 6, this function 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 4 megahertz 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 maintain this ratio, one would expect the MC68000 to be 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 of the results of a number of benchmarks that have been run leads 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 as follows:

```
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; 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 (called TAK'_i) of identical functions that 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 do these 100 functions destroy the effect of the cache. The answer is that they do 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. Both benchmarks were run 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 a size that is significantly beyond that 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, which was 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 slowdown of more than a factor of 9 in the overall execution speed of Lisp.[14]

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

[14] 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.

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 that called the one that is currently invoked.

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

Previously it was 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′ shown 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 the coding technique therefore cannot be number-size dependent (up to BIGNUMs).

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

1.4.2 *Measure the Right Thing*

Often a single operation is too fast to time directly, and therefore must be performed many times; the total time is then used to compute the speed of the operation. Although simple loops to accomplish this appear straightforward, the benchmark may be mainly testing 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 i)`.

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).

1.4.3 *Know How the Facets Combine*

Sometimes a program that performs two basic Lisp operations will combine them nonlinearly. 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 that are of interest to that audience and that are large enough so that the combinational aspects of the problem domain are reasonably unified. 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 them as arrays whose elements are other arrays, or it might 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. The history of the address space that is being used for the timing—how many CONSes have been done and how full the atom hash table is, for example—can make a difference.

1.4.4 *Personal Versus Time-shared Systems*

The most important and difficult question associated with timing a benchmark is exactly how to time it. This is a problem, particularly when one is comparing a personal machine to a time-shared machine. Obviously, the final court of appeal is the amount of time that a person has to wait for a computation to finish. On time-shared machines one wants to know 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.

Time-sharing systems often do background processing that 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 time-sharing 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 that can be disabled, such as monitoring the keyboard and the network. 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. The effect of load averages on elapsed time on other machines has not been tested.

The quality of interaction is an important consideration. In many cases the personal machine provides a much better environment. But in others, the need for high absolute performance means that a time-shared system is preferable.

1.4.5 *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, and 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, which was 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.[15]

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