IN-62
117757
p. 36

NASA Technical Memorandum 103907

Analysis of a Benchmark Suite to Evaluate Mixed Numeric and Symbolic Processing

Bharathi Ragharan and David Galant

(NASA-TM-103907) ANALYSIS OF A BENCHMARK SUITE TO EVALUATE MIXED NUMERIC AND SYMBOLIC PROCESSING (NASA) 36 p

N92-32886

Unclas

G3/62 0117757

August 1992

National Aeronautics and
Space Administration

NASA Technical Memorandum 103907

Analysis of a Benchmark Suite to Evaluate Mixed Numeric and Symbolic Processing

Bharathi Ragharan and David Galant, Ames Research Center, Moffett Field, California

August 1992

NASA
National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

1.0 SUMMARY

The suite of programs that formed the benchmark for a proposed advanced computer is described and analyzed. The features of the processor and its operating system that are tested by the benchmark are discussed. The computer codes and the supporting data for the analysis are given as appendices.

2.0 INTRODUCTION

This benchmark program suite, chosen to test the Symbolic VHSIC (Very High-Speed Integrated Circuit) Multiprocessor System, provides a measure of the symbolic and numeric computational capabilities of a system, especially in the absence of actual applications. This benchmark suite was chosen to test system behavior resulting from a combination of language implementation, operating system, and system hardware. It does not test the raw speed of the hardware nor any particular component of the system. Note that the benchmark programs actually presented were tested on uniprocessor systems and may not be strictly appropriate for efficient parallel processing.

The individual programs of the suite and their performance on various systems are discussed. The tested hardware systems are Symbolics 3675, Vax 8800, Compaq 386, MIPS, Sun-4 and IIM45000. Program descriptions and performance analyses are given, and the static and dynamic counts of the functional level are discussed. Timing results obtained from each system are analyzed to determine the effects of machine architecture and optimization levels on the behavior of the programs. The timing results are **not** intended as criteria for measuring the relative merits of the hardware systems used. Conclusions regarding the selection of programs appropriate to test a given system are given. Also, methods to incorporate an application into a benchmark program suite are discussed.

The benchmark program suite consists of a Symbolic Set (ref. 1), using specifically Boyer, Browse, traverse, and Triangle; a Numeric Set, consisting of Whetstone, Linpack, Cfft2dm, and Cholskym; AutoclassII; NASA/JSC Ada programs (ref. 6); and the University of Michigan Ada programs (ref. 7). Except for the last two programs, all programs are implemented in both Lisp and Ada. Descriptions of these programs are provided in section 2. The Common Lisp version of the programs are generic, in that they do not include specific type declarations or optimizations. The programs are run in each hardware system's default (commercially available) execution environment (to study the program behavior) as opposed to any specially provided custom or optimized environment. The benchmark suite and associated programs can be obtained from the second author.

The authors would like to thank the following team members for their contributions to this effort: Gloria J. Davis, James Gibson, Yuan-kwie Liu, Pushpa Majumdar, Ellen Ochoa, and Nancy Sliwa. The Ada version of AutoclassII has been developed by Gloria J. Davis, and the Ada version of the numeric set has been developed by James Gibson. We extend our thanks to Rodney Swab and Charles Fry for reviewing and providing expert opinion and comments.

3.0 MACHINE DESCRIPTIONS

Descriptions are given to provide understanding of the machines, not for relative comparisons.

In generic Lisp machines, a certain number of bits in the address (format shown in table 3.1) are reserved to represent data types. These bits are called tags and the machines are classified as tagged architecture machines. Non-tagged architecture machines are called stock hardware machines.

Table 3.1. Tagged format

data-type	object-reference
<-tag->	<---object--->

3.1 The Symbolics 3675

The Symbolics 3675 is a 36-bit tagged architecture Lisp machine running Symbolics Common Lisp, herein referred to as S C-lisp, which is Common Lisp with some extensions. It uses a microcoded 36-bit TTL processor with a clock time of about 200 ns. The tagged format has three fields: compact Lisp code, data type, and object reference. An object reference is defined as a conceptual object, which is used to reference a Lisp object.

Table 3.2. The Symbolics tagged format

compact-Lisp code	data-type	object-reference
<-------------tag------------->		<---object--->

A compact Lisp code technique called CDR coding—a type of tagged, compact list representation—is used for internal list representations. This technique reduces the memory requirement for list construction by as much as half. Therefore, any non-tagged architecture requires as much as twice the memory as a tagged architecture for list computation.

The data-type bits contain information about the type of the object. Type checking is supported by the hardware and performed in parallel with the instruction execution.

The machine has a demand-paged word-addressable virtual memory, which makes its main memory a large cache. Random access takes three machine cycles, and sequential access takes one machine cycle. The compiler can specify sequential access, where appropriate, to speed up memory access. Array referencing takes only a single instruction. Normally, one-dimensional array access takes approximately ten machine cycles, and multidimensional array access takes three more cycles per dimension. Array bounds are checked on every access.

A single-precision, floating-point number is represented as an immediate datum in a single-machine word with type information in the tag field. If both operands of an arithmetic operator are single-precision and the floating-point accelerator is installed, results are computed in a couple of cycles.

The architecture uses stacks to get operands, to stash results, to pass arguments to functions, to return values, and to make local data references. Stack management tasks are handled by processor hardware. There are no general purpose registers at the macro instruction level. High-speed stack groups provide efficient execution of function calls and recursion. The stack access time is one cycle.

3.2 DEC 8800

The DEC 8800 is a 32-bit stock hardware machine. It consists of the console subsystem, a primary CPU, a memory subsystem with one to eight memory arrays, a power system, and one to two VAX bus interconnect adapters. The system is interconnected through a synchronous backplane bus called the VAX 8700/8800 memory interconnect. This provides the system with a communication path between the CPU's, memory, and the adapters attached to the VAX bus interconnects. It runs Lucid Common Lisp, Allegro Common Lisp, and Telesoft Ada under the VMS operating system. The architecture supports a stack that occupies one half of the memory address space. Lisp, which needs both a large stack and a large address space, uses this stack.

3.3 Compaq 386/20e

The Compaq 386 is a personal computer built around the Intel 386 processor. The Compaq 386/20e contains a 20 MHz Intel 386 processor, a 20 MHz Intel 80387 Floating-Point coprocessor, and a 20 MHz Intel 82385 Cache Memory Controller with a 32-kilobyte cache of high-speed static memory. The total system memory is 8 megabytes of dynamic RAM using a proprietary 32-bit bus. Mass storage is a 110-megabyte hard disk drive with average access time of less than 25-milliseconds. It runs Lucid Common Lisp and Alsys and DDCI Ada under the Unix operating system.

3.4 SUN-4

The Sun-4 is a SPARC (Scalable Processor ARChitecture) (ref. 4) machine. It has an Integer Unit (IU), an optional SPARC Floating-Point Unit (FPU), a memory management unit (MMU), an optional virtually addressed cache, an optional VME bus interface and several control registers (ref. 4). Computations are register intensive, and memory access is through load and store instructions, via the cache. Computational instructions obtain their data from registers and immediate fields in instructions and put their results in registers.

The SPARC architecture defines a set of 32-bit instructions, a set of registers, how the registers work, and how traps and interrupts are handled. The instruction categories are memory reference, multiprocessor, arithmetic/logic, tagged, special, and control transfer (ref. 13). Tagged instructions provide support for languages that can benefit from operand tags. Tags are used to classify data and pointers differently and to perform legal operations on the data and pointers. They assume 30-bit, left-justified signed integers and use the least significant two bits of a word as tag.

The Sun-4 workstation that used to run these programs has the Fujitsu implementation of the SPARC design. It runs Allegro Common Lisp under the Unix operating system. The tag field in the instruction format contains information about the data-type and fixed-point numbers, which are both immediate data and machine format integers. When a floating-point instruction is decoded by the integer chip, it signals the floating-point coprocessor to start execution immediately. Integer and floating-point operations without dependencies execute concurrently. This increases the execution speed, but time is spent to dispatch to the floating-point coprocessor. SPARC uses a 32-bit byte-addressing format. The chip uses two major buses for accessing the cache: a 32-bit address-bus and a 32-bit bi-directional data bus for performance improvement.

3.5 IIM45000

The Integrated Inference Machine is a generic 40-bit (32 bits of data, 8 bits of tag) tagged-architecture Lisp machine running Integrated Inference Machine Common Lisp with extensions under DOS operating system. It has 40 megabytes of memory and is referred to as IIM.

3.6 MIPS M/2000

MIPS M/2000 (Microprocessor without Interlocking Pipeline Stages), referred to as MIPS, is a 32-bit RISC (ref. 12) computer system. It is a multiuser system which runs IBUKI Common Lisp and Verdix Ada under the Unix operating system. It uses a 25 MHz clock R3000 processor based on the MIPS RISC architecture. The execution environment of the R3000 is provided by the R3200 CPU board which allows the processor to operate at maximum efficiency.

The major functional components of the R3200 CPU are:

1. Full-custom R3000 CMOS VLSI processor.
2. Optional full-custom R3010 CMOS VLSI floating-point accelerator (FPA) coprocessor.
3. 64-K instruction cache and 64-K data cache.
4. A master-only interface to the system bus (VMEbus). The CPU is also a system controller and an interrupt handler.
5. A high-speed private memory interface to ECC (Error Correction Circuitry) protected memory.

The R3000 RISC (Reduced Instruction Set Computer) processor uses 32-bit data, address, and instructions. The processor contains 32 general purpose registers, which are 32 bits wide. The processor also contains an instruction set and interface support for three external coprocessors, each with up to 32 registers. Two other functional areas on the R3000 processor are a cache control unit and a fully associative 64-entry Translation Lookaside Buffer (TLB) for fast access to a 4-gigabyte virtual address space.

The large instruction and data caches reduce memory bandwidth requirements, while increasing processor performance. The R3200 CPU board provides two high-speed, direct-mapped caches: a 64-K instruction cache and a 64-K data cache. The caches collectively provide an aggregate maximum instruction/data bandwidth of 200 megabyte per second over the CPU cache bus. The cache RAMs have 20 ns access times. Both caches provide parity checking on tags and data. The CPU board provides a separate, dedicated, high-speed bus (called the Private Memory Bus), for an unobstructed access to the main memory.

4.0 EXECUTION ENVIRONMENT CONSIDERATIONS

Elements of a program's execution environment influences its runtime efficiency. Specific influential factors include the file system (i.e., file access such as read and print, and the operating system interface), the compiler, and automatic storage reclamation, also known as garbage collection. Another factor is the ability of the compiler to optimize execution speed.

4.1 Programming Languages

This section describes the implementation features of Lisp necessary to understand the benchmark programs.

4.1.1 Lisp implementation– Since the data type (ref. 2) of an object can be changed during runtime, Lisp supports runtime type checking. This increases computational complexity. Type information can be encoded in the pointer to the object or in the object itself.

The two types of object reference are *immediate* and *pointer*. An immediate object contains the object within its object-reference field. Examples of these are fixnums, single-float, and characters. A pointer object reference contains an address in its object-reference field. Some Lisp objects that require pointers are bignums, arrays, lists, hash tables, and structures. Roughly speaking, immediate object references are used for small objects, whereas pointer objects are used for large objects.

When the type is encoded in the pointer, and if there are not enough bits to describe the subtype of an object in the pointer, the main data types such as atoms and cons's are encoded in the pointer and the subtype is encoded in the object. A cons is the object that points to the head and tail of a list. Sometimes the pointer itself represents a type. In such cases, memory is partitioned into segments and each contains a data type. A single data type can span several segments. This technique is called the BiBOP (Big Bag Of Pages) scheme.

A quoted object evaluates to itself and is considered to be of type "constant." Some systems represent these constants as immediate data when the value can be determined at compile time, others put constants in a read-only area, and pointers to them are computed at load time. Immediate data are normally faster to reference than other kinds of data.

The function *eq* simply compares two pointers. Two objects are *eq* if they address the same memory location. The function *equal* uses *eq* to compare symbols.

In some implementations, arrays contain only pointers to actual values stored in the array. Thus, actual value references are indirect. Accessing an array element uses several index instructions, and array indices are recalculated on every access and store operation. In the absence of tagged hardware, type checking is done through software, and computing array indices takes a considerable amount of time during execution.

Arithmetic in Lisp is handled by passing *pointers* to machine format numbers, rather than by passing machine format numbers directly. Converting to and from pointer representation is called boxing and unboxing, respectively. Referencing a number using a pointer is called number CONSing. This representation, which is required to make numbers into Lisp objects, causes slow execution. The speed of arithmetic in Lisp also depends on boxing and unboxing techniques and the ability of the compiler to minimize these operations. Some ways of achieving speed are:

A. Represent all single-precision, fixed-point numbers (consisting of 31 bits of data) as immediate machine format integers and perform range checks after arithmetic operations.

B. Allocate numbers in the machine format on two stacks: one, for fixed-point numbers, called FXPDL (Fixed-point Push Down List); the other, for floating-point numbers, called FLPDL (Floating-point Push Down Lisp). Arithmetic operations can use these stacks as temporary storage, eliminating unnecessary number CONSing.

C. Pre-allocate a range of normally used fixed-point numbers on a stack, thereby reducing the number CONSing procedure into a simple addition of adding the base index of this stack table to the original number. Pre-CONSing of a certain range of small integers, called the small-number scheme, reduces boxing of numbers in this range to adding the number to the base address of the table.

On microcoded machines, runtime type checking is done almost in parallel by the hardware. On stock hardware implementations, code must be emitted to perform type checking. Therefore, on stock hardware implementations of Lisp, type-specific arithmetic operations contribute to the speed of arithmetic operations. For this purpose, some Lisp implementations have arithmetic functions that operate directly on type-specific data. Unboxing all relevant numbers and performing as many computations in machine representation format as possible is also advantageous. This is called open-compiling or open-coding.

4.1.2 Ada– Ada is a programming language designed by the United States Department of Defense. It has considerable expressive power for a wide application domain. The language is a modern modular algorithmic language with the usual control structures, along with the additional capability to define types and sub-programs. In addition, it has features for real-time programming and exception handling. Both application-level and machine-level input-output are defined. Thus, systems programming, which requires precise control over the representation of data and access to system-dependent properties, is also possible in Ada.

Lisp programs can be translated into Ada rather directly. A list in Lisp is represented as a linked list of records in Ada. The linked list is created using the *new* allocator. The property list that every Lisp symbol carries is represented as a record component in Ada. This increases the complexity of manipulating objects pointed to by the components of records. For floating-point and complex arithmetic computations, an Ada basic mathematical function library has been developed (ref. 11). The numeric benchmark programs—Whetstone, Linpack, Cfft2dm, Cholskym—and AutoclassII require this library.

4.2 Compiler

Clearly, the compiler is a very important factor in determining the runtime behavior of a program. However, since we had to accept the compilers available on the various machines, we can say very little of importance about this. Interpretation of the effects of different compiler implementations are spread throughout section 8.

4.3 Garbage Collection

The fundamental data manipulated by Lisp are *object references* called pointers. The object reference contains the address of the object (ref. 2) in memory. It is used to collect objects that are no longer useful and to free the memory used by them. On tagged architecture machines (section 2.6.1), tag bits indicate if a memory word is an object reference or data. On stock hardware, the tagged word format is emulated through software. This requires decoding an object reference into tag and true address, creating some overhead (ref. 14). Some overhead is also incurred in encoding these object references.

The runtime effect of garbage collection (gc) depends on the memory size. On stock hardware machines, Lisp normally starts with a default memory size that is expanded dynamically to a predefined limit. The default setting usually can be altered, by memory management strategies provided by the Lisp systems. The effect of gc on an individual benchmark is discussed in the section dedicated to that benchmark (section 3).

5.0 COMPARISON OF LISP LANGUAGE IMPLEMENTATIONS

5.1 IBUKI Common Lisp

IBUKI Common Lisp is based on a kernel written in the C language. The high-level Common Lisp functions and the IBUKI Common Lisp compiler are written in Common Lisp, using the primitives in the kernel. Some amount of machine-dependent, assembly language code is present in some IBUKI implementations for improved performance. These routines include parts of bignum multiplication and division, and bit-table manipulation used by the gc. The IBUKI Common Lisp system includes an optimizing Common Lisp compiler.

The IBUKI Common Lisp compiler has three phases. The first phase generates a c-file that contains the C code for the Lisp source code, an h-file that contains declarations used by the c-file, and a data-file that contains Lisp constants. The second phase invokes the native C compiler to produce a code file from the c-file. Finally, the data-file is appended to the code file to obtain an object file, which can be loaded. Hardware dependent optimizations, such as register allocations and peephole optimizations, are left to the C compiler.

IBUKI Common Lisp does not support immediate data. Every object is represented as a cell in the heap area. Each cell consists of several 32-bit words. The first word is common to all data types. Half the word is the type indicator and the other half is used as a mark bit for the gc. A CONS cell consists of three words, and a fixnum cell consists of two words. Array headers and compiled function headers are represented as fixnum cells. Array elements and compiled code are placed elsewhere in the memory. Array elements are represented in one of six ways, depending on the type of the array. General array elements are cell pointers: 32-bit integers for fixnums; 32-bit floating-point numbers for short-floats; 64-bits for long-floats; 1-bit for bit and bit-vector; and, 8-bit code for strings.

Stacks are used for function calls, arguments, and value passing. A C language Control Stack called C stack is used for type-specified operations to improve performance. The C stack can be accessed more efficiently than other stacks, such as the value stack, which is the "main stack" of IBUKI Common Lisp. On the C stack, arguments and values passed, values of lexical variables allocated, and temporary values saved may be represented as immediate data, sometimes referred to as *raw data*, instead of as pointers to heap allocated cells. Some of the built-in Common Lisp arithmetic functions, such as +, -, 1+, 1-, *, floor, mod, /, and expt can operate on raw data. This improves the performance of arithmetic computations.

Certain Lisp objects, such as fixnums and characters, may be represented by their value. Cells of small fixnums and cells of characters are pre-allocated in fixed locations. Symbol print names and string bodies are usually allocated in relocatable pages (explained below) and moved to the heap when an object file is created. The size of the cell is determined by the implementation type of the object. For objects of some of the implementation types, such as array, the cell is simply a header of the object. The body is allocated separately from the cell and is managed differently. Memory space occupied by the body of such an object is called a block.

Memory space is divided into two parts that occupy contiguous space in the memory: the heap area and the relocatable area. The heap is divided into pages, where each page holds cells of the same type (BiBOP technique). Cells allocated in the heap are not gc. Blocks allocated in the heap are called contiguous blocks. Thus each page in the heap is either a page for cells of a specific type, or a contiguous block. Blocks allocated in the relocatable area are called relocatable blocks.

5.2 Lucid Common Lisp

The Lucid Common Lisp design for a general-purpose processor is based on emulation of features of special-purpose Lisp processors. Some of interest are a tagged pointer encoding scheme, a function-to-function calling protocol permitting dynamic redefinition, and packing of frequently used data, in particular CDR-coding—a kind of tagged, compact list representation. A general-purpose processor often has the following machine features: RISC-like (ref. 12) instruction sets, numerous general-purpose registers usable for address indexing, byte-aligned addressing on a memory bus of at least 32 bits, a large address space, and a large real memory available for Lisp process's working set (greater than 4 megabytes). Lucid has additional constraints, in that the compiled code is made read-only and position-independent, and virtually all of the Lisp system should be written in Lisp itself.

This compiled code read-only constraint is imposed to gain maximal sharing of memory segments in a time-shared system. It also improves paging performance, and it provides a clear separation of the memory segments that the gc must scan. Another positive consequence of this constraint is the clear separation of a compiled function into a sequence of executable machine instructions and a sequence of linkage cells to data and other environment. That the Lisp system is written in Lisp ensures that porting this system to a new hardware architecture is reduced to only porting machine and operating system specifics of the Lisp system.

Because Lisp depends on runtime typing, data encoding is an important issue. Lucid Common Lisp uses a tagged-pointer scheme for data encoding to address very large virtual memories, accelerate some of the small integer arithmetic, and list pointer chasing operations. Thus every address that is a pointer is divided into two parts, a tag and a true address. The tag bits specify whether the true address part is an immediate datum, or a virtual memory address. Some small integers, characters, and internal state markers for gc are encoded this way because the information contained in a single datum is less than the number of bits allocated in the pointer for the address part. A pointer is four bytes, while a CONS cell requires eight bytes. CONS cells are the smallest objects allocated. Thus, at least seven out of every eight byte-level addresses are unused, freeing three address bits to represent primary tags.

Two of the eight possible primary tags are used to represent immediate even and odd fixnums. These fixnums are 30 bits long. The lower order bit of the numeric value coincides with the higher order bit of the 3-bit tag. This alignment enables a level of performance for many Lisp programs equal to that of conventional languages.

An array contains one word of header information, some sub-type information and an element length count. The header information is useful for runtime typing and permits the gc to scan memory segments linearly, parsing them into various Lisp data types.

If the arguments to primitive Lisp functions are located in general-purpose registers, then the component selection of a compound object such as a CONS cell, and simple fixnum arithmetic operations are accomplished, in one or at most two reduced instruction set-like instruction cycles. A FLPDL (section 2.3.1) is used for floating-point operations. Similar simple sequence of instructions accomplish element selection for objects with indexable entries like vectors and strings. The two most common type tests performed in code selection are CONSP and FIXNUMP. These types are among the primary tags. The operations on these data are highly optimized.

In Lisp, function definitions and some globally defined data can be changed during runtime. This gives Lisp its dynamic modularity. A symbol's (ref. 2) function value cell contains its current function definition. The mapping from the symbolic name of a function to the start address of its code is always available and every function call to the symbol has to go *indirectly* through this cell. Some recursive function calls are optimized. Such recursive function calls jump directly to the start of the compiled code rather than through the symbol's function value cell.

A CDR-coding technique that uses two bits facilitates compact storage of linked cells. In this method, the four combinations are used to designate normal, next, and nil. Memory consumption for list construction is reduced by at least half using this technique.

5.3 Allegro Common Lisp

Typing is done via the BiBOP scheme with 512-byte pages. Pointers of different types are allocated to different pages. There may be many pages for each type. All numbers are boxed. CONS cells are eight bytes. Because CDRs are more frequently referred than CARs, the CDR is stored first, which permits a CDR operation to be done with less indirect addressing.

6.0 DESCRIPTION OF THE BENCHMARK SUITE

6.1 The Numerical Benchmarks

6.1.1 Whetstone– This synthetic program is designed to measure the performance of a system executing a scientific program. The mix of operations are integer arithmetic, floating-point arithmetic, array referencing, branching statements, function and subroutine calls, and standard mathematical functions. The main program encompasses a set of eleven modules, each with its own iteration count number to control the number of times it is executed. The counters can be varied to produce different mixes of instructions. The Ada version used in this study was published in *Ada Letters* (ref. 5).

6.1.2 Linpack– The Linpack benchmark solves an N × N system of linear equations by Gaussian elimination with partial pivoting. The matrix consists of random numbers in the interval (0,1), and the right-hand side is chosen to make the vector $[1, 1,\ldots 1, 1]^T$ the solution. The program has three phases: initialization of the matrix and the right-hand side vector, copying the matrix to the appropriate working space, and multiple decomposition-solution phase for two differently sized matrices. The original program, written in Fortran, was obtained from the NAS project at NASA Ames and translated into Common Lisp and Ada (ref. 8).

This program measures the performance of a system executing regular access to memory in numeric computation, which includes integer and some floating-point arithmetic. The problem size is N = 100, and the arrays are 200 × 200 and 201 × 200.

6.1.3 Cfft2dm– Cfft2dm is a two-dimensional Fast (Discrete) Fourier Transform. This is a typical intermediate step in solving an elliptical partial differential equation on a rectangular grid. The program initializes the array to be transformed with single-precision and complex floating-point numbers. The array is transformed in place, and then inverse transformed in place again. In theory, the final result is the initial array. The original program was obtained from the NAS project at NASA Ames (ref. 8) and the Fortran code was translated into Common Lisp and Ada.

This program measures the performance of a system executing single-precision and complex single-precision, floating-point numbers and matrix computations.

6.1.4 Cholskym– Cholskym is typical of the kernel calculations of Computational Fluid Dynamics codes in the aerospace industry. It is the solution of a large number of independent sets of linear equations with banded and symmetric matrices and multiple right-hand sides. Only the upper band is stored (by diagonals). As it is implemented, it favors machines with vector pipes running Fortran.

The program consists of an initialization phase, which assigns elements of the matrices and the right-hand sides from uniformly distributed random numbers in (0, 1). Then, a loop is executed that copies the matrices to working storage and that solves the systems of equations. The program prints an error value and the number of floating-point operations that are used to confirm proper execution of the program (ref. 8).

This program measures the performance of a system executing single-precision and floating-point arithmetic operations, in which the precision of the floating-point representation matters immensely.

6.2 The Symbolic Benchmarks

6.2.1 Boyer– The Boyer benchmark is a theorem-proving program. It tries to prove that a particular logical statement is a tautology (true by virtue of its logical form alone). The first part of the program, Boyer-setup, uses a list of axioms to set up property lists of symbols. The second part of the program, Boyer-test, rewrites the logical statement into a canonical form, which is a nested "IF" statement, using the property list of each symbol. The axioms are used as production rules. To prove that the logical statement is a tautology, a simple tautology checker is invoked.

Boyer performs a large number of list-structure manipulations, a moderate number of function calls, and the property list operation GET. A property list is implemented as a memory cell, containing a list with an even number of elements. Hence, the above operations measure the performance of list implementation.

6.2.2 Browse– The Browse benchmark is designed to perform a mixture of operations in proportions very similar to those in real expert systems. The basic operation is to search a data base of objects, identifying all those objects that satisfy some predicate. The data base of objects is implemented as property lists. The objects contain patterns called "descriptors." The predicate is that a set of search patterns matches the descriptors. A simple pattern matcher is defined to this end. Exhaustive matching is done by matching all search patterns against all descriptors, regardless of the results of any individual matches.

There are many list-structure-manipulating operations to test the performance of list implementation in Lisp. A random number generator, which performs operations similar to many done in compilers, AI systems, and other large systems, is defined. The object type used in computation is "character." Therefore, type checking reduces to tag extraction and pointer comparison in parallel, or address comparison in the BiBOP scheme. The primary computations that Browse performs are accessing property lists and their elements, comparing characters and CONSing symbols. Accessing property lists is address computation and character comparison. CONSing, which requires memory allocation, is used in this program. This triggers gc. Browse is a strictly sequential search program, designed with tail recursion, which may be collapsed by compilers into iteration for fast execution.

6.2.3 Traverse– The Traverse benchmark is designed to measure the performance to be expected from the *defstruct* facility. This allows the user to create and use aggregate data types with named elements. To do the measurements, a directed graph of nodes is built and traversed. Each node is a *defstruct* with ten slots (components): a backpointer to parents, pointers to sons, a serial number of the nodes, a mark field, and six other slots to hold any information needed. The program is structured into two sections: initialization and Traverse.

This program is an example of what might be termed as "pointer chasing." Several levels of indirection are involved in accessing the value of an object embedded in a *defstruct* construct that represents structures. The initialization, which creates structures, requires memory allocation. Random linkage of these structures is done by rearranging pointers. Object access in a random distribution of pointers is disorderly. A compiler can rebuild the list and make the random distribution straightforward again.

6.2.4 Triangle– The general purpose of this program is an exhaustive search represented as a tree of possible moves, where each node of the tree signifies a decision about the next move. The possible moves can be represented as *if then* rules, which allow the program to be implemented in rule-based shells and toolkits. Because the moves are implemented as elements of one-dimensional arrays, the Triangle benchmark mainly tests one-dimensional array references. This benchmark has been developed as a differentiator between one-dimensional and two-dimensional array references. The performance of one-dimensional array references are measured.

The main operations this program performs are as follows: access the first array location specified by an index and get the value (this value is the index of the second array); get the value at this index; check if this value equals to 1 or 0. On large systems, this program does not trigger gc and a dynamic memory allocation is not required.

6.3 AutoclassII

AutoclassII is a classification program that uses Bayesian least squares estimation theory to determine data classes. Its general principles are given in reference 9.

This is a real application program that uses data from the spectra of stars. This particular data base is set up to execute floating-point operations. It provides a measure of array referencing and floating-point arithmetic operations in real applications.

6.4 JSC Real-Time Ada Programs

The JSC Real-Time Ada programs are designed to measure the execution time for those features of the Ada language important for the implementation of real-time programs. The features measured are tasking, exceptions, and branching statements (*if* and *case*). The programs measure the amount of time to execute individual language statements, or groups of statements under various conditions. This approach is called microscopic benchmarking or feature measurement.

The programs use the dual-loop method for measuring a language feature. The feature to be measured is placed inside a loop, known as the test loop, and executed many times. The total execution time of the test loop is measured by reading the real-time clock immediately before and after the loop. To account for the loop overhead, a control loop that is identical to the test loop, except for the feature to be measured, is timed for the same number of repetitions as the test loop. The time for the feature is then calculated by subtracting the control loop time from the test loop time and dividing this number by the number of repetitions. The loops must be designed carefully, so that the only difference between the control and test loops is the feature to be measured. Also, care must be taken to ensure that an optimizing compiler does not modify either loop, thereby distorting the comparison.

To ensure proper measurement of the language features, the order in which the program units are compiled is important. To defeat possible loop optimization, which would introduce errors into the measured timings, the benchmarking_global_support package body, which contains a set of procedures named similarly to stable_call_to_a_remote_procedure, must be compiled last. One or more of these procedures is always called as a part of the control and test loops. If the bodies of these procedures are compiled last, an optimizing compiler will be unable to place the executable code inline within the loop and will distort the basis for feature measurement. The features of the language that are measured by the program fall into several categories. Several different tests are performed in each of the following categories:

1. Time to read the real-time clock.
2. Time to create, activate, and terminate a task.
3. Time to perform a rendezvous.
4. Time to raise and handle an exception.
5. Time to execute a branch instruction.

The program was obtained from the Johnson Space Center's Avionics Division. A few changes were made in the program as received from Johnson Space Center (ref. 6). The macroscopic tests were eliminated, and only the microscopic tests were used in the SVMS version.

6.5 University of Michigan Real-Time Ada Programs

The purpose of the University of Michigan (UM) Real-Time Ada programs is to measure the execution time for elements of the Ada language. These features include procedure calls, allocation of memory and variables, rendezvous, tasking, delays, exceptions, and time calculations. Each program measures the performance of one distinct language feature under various conditions. Several techniques are used to minimize interference from external or operating system functions and to defeat compiler optimization that would affect the measured results. The original set of benchmarks consisted of about 325 separate tests. Because many of these overlap with the JSC Real-Time Ada programs, a small subset consisting of 18 tests was selected.

The features of the language that are measured by the programs fall into several categories (ref. 7). A variety of different tests are performed in the categories shown below:

1. Clock function analysis.
2. Procedure calls.
3. Dynamic variable allocation.
4. Time arithmetic.

7.0 DESCRIPTION OF THE EVALUATION PROCESS

Each of the benchmark programs were compiled on each machine using the default settings of the compiler present. The programs were then executed and timed. For the Lisp programs, the compiled code was executed from a clean environment; that is, from a newly started Lisp environment to avoid any side effects from earlier activity in the environment. A sub-study was to investigate the effects of optimization and increased memory allocation on the runtime of the Lisp numeric benchmarks on stock hardware using the Lucid Lisp compiler on the DEC 8800. This compiler does respect type declarations.

Additionally, a profiler program was written in Lisp to profile the Lisp programs. A profiler is a program that provides a trace of another program's execution pattern. It reads a program, tabulates the static count of the functions used, and writes a profiled program with statements inserted at appropriate places to monitor program execution. The profiled program can then be compiled and executed to obtain dynamic counts of the execution pattern and operation types actually executed. Static and dynamic counts are useful for applying optimization techniques. The profiler developed for the results presented in this report was designed to monitor user defined functions and the two commonly used loop controls, the *cond* and *if* special forms.

8.0 EVALUATION RESULTS AND ANALYSIS

8.1 Whetstone

8.1.1 Profiler analysis– The profiler dynamic count reveals that the major computations are array referencing and floating-point operations. There are approximately a half million array references, one million floating-point operations, and a quarter-million integer operations. The main module spends time in integer computation and subroutine calls (150,000). Two subroutines do array referencing and floating-point operations, and a third performs only floating-point operations.

In Lisp, array referencing is accomplished by pointers to the actual value. Index bound checking and recalculation, and access to floating point numbers, are very time consuming. The type declarations and inline coding of functions speed up execution.

8.1.2 Program execution– Timing results are shown in table 8.1. The primary effects on behavior are number representation, type checking, array referencing, and the function calling scheme (ref. 1) (section 2.3); a secondary factor is gc because floating-point operations create intermediate objects and thus garbage. A large memory reduces gc. A floating-point coprocessor also improves performance.

Representation of numbers as immediate data eliminates boxing and unboxing thereby reducing the number of instructions to execute numeric operations. Data type checking using the tagged instruction format requires fewer instructions when type checking is done by hardware in parallel than are required when type checking is emulated through software. Indirect array referencing and array bounds checking on every access takes several instructions. Program execution is accelerated on systems where only one instruction is required for array accessing.

Table 8.1. Timings (in seconds) of the Lisp version of Whetstone

Machine name		Symbolics	MIPS R2000	Compaq 386/20e	IIM	SUN4	DEC8800	
Language		S C-Lisp	IBUKI C-Lisp	Lucid C-Lisp	C-Lisp	Allegro C-Lisp	Allegro C-Lisp	Lucid C-Lisp
Whetstone	e	17.54	72.00	78.87		30.55	71.00	35.83
	r	17.29	34.00	70.01	14.80	30.48	71.00	34.84

e = elapsed real time in seconds. r = machine run time (cpu) in seconds.

Execution of the Ada version of Whetstone is summarized in table 8.2. The significant contributing factors are the Ada compiler and the processor.

The Verdix Ada compiler was able to produce code that can run efficiently on the MIPS. The behavior on the DEC 8800 is comparable with MIPS. Ada on the Symbolics did not produce efficient code. It is difficult to quantify the code generated by Alsys and DDCI compilers on Compaq 386. A better way to understand their behavior would be to use these compilers on other systems to study the execution. The i386 processor and i387 floating-point coprocessor on the Compaq 386 have a considerable effect. Using a structured language like Ada, a program with a mix of scientific operations can execute well on stock hardware. The Symbolics Ada compiler does not produce efficient code so that the relative performance is not really comparable.

Table 8.2. Timings (in seconds) of the Ada versions of Whetstone

Machine name	Symbolics	MIPS R2000	Compaq 386/20e		DEC 8800
Language	Symbolics Ada	Verdix Ada	Alsys Ada	DDCI Ada	Telesoft Ada
Whetstone	53.60	2.67	14.28	19.4	3.90

8.2 Linpack

8.2.1 Profiler analysis– This program executes three and a half million array referencing operations. Three million operations are in one iteration block; the remainder are distributed throughout the program.

8.2.2 Program execution– The timing results for the linpack's execution are shown in table 8.3. The factors that influence the execution are integer computation and array operations that require memory access.

This benchmark shows that Lisp programs that perform array referencing and floating-point operations execute differently on stock hardware and on Lisp machines, because of indirect array referencing and boxing and unboxing of numbers. Executing this program provides a measure of these factors of a system.

Table 8.3. Timings (in seconds) of the Lisp version of Linpack

Machine name		Symbolics	MIPS R2000	Compaq 386/20e	IIM	SUN4	DEC8800	
Language		S C-Lisp	IBUKI C-Lisp	LUCID C-Lisp	C-Lisp	Allegro C-Lisp	AllegroC-Lisp	LUCID C-Lisp
Linpack	e	138.07	460.00	827.30		398.5	549.00	657.26
	r	130.00	448.52	809.12	80.00	384.2	549.00	657.70

e = elapsed real time in seconds. r = machine run time (cpu) in seconds.

The execution of the Ada version of the program depends on the compiler and the processor (table 8.4). The Verdix Ada running on the MIPS executes this mix of operations effectively. The Symbolics Ada compiler is not very effective. The strong typing in Ada language contributes to efficient execution of array referencing operations on stock hardware machines.

Table 8.4. Timings (in seconds) of the Ada version of Linpack

Machine Name	Symbolics	MIPS R2000	Compaq 386/20e		DEC 8800
Language	Symbolics Ada	Verdix Ada	Alsys Ada	DDCI Ada	Telesoft Ada
Program linpack	299.6	29.0	71.5	80.0	50.0

Table 8.5. Timings (in seconds) of the Lisp version of Linpack with optimization

Optimization	Machine	Elapsed Time	CPU time	GC time	Speed up
Generic	DEC 8800	657.0	657	170	1.0
With declarations	Lucid	157.0	157	47	4.2
With declarations and memory expansion	Common Lisp	81.0	81	0.0	8.0

8.3 Cfft2dm

8.3.1 Profiler analysis– The program spends considerable time in two tail-recursive functions, which execute addition and subtraction of complex arithmetic operations. This conforms with the program analysis. A function is tail-recursive if the value returned by the function is the value of the recursive call within the function.

8.3.2 Program execution– In this program, the majority of computations require memory allocation because temporary objects are created during computation. These temporary objects immediately become garbage when the old value is no longer needed. A significant amount of time is spent in gc to free memory. Therefore the behavior of the program is affected by the amount of memory and the gc algorithm in a system.

Excluding gc time, the program executes similarly on all the systems (table 8.6). The results can be explained from an analysis of the gc times. The IIM has a large memory and a very efficient gc algorithm. The time spent for gc is 50 sec on the SUN-4, 593 sec on the DEC 8800 running Lucid Common Lisp, and 69 sec running Allegro Common Lisp. On the Compaq 386, there were many gcs. On the Symbolics, 57 seconds are spent doing page faults, page breaks, and other routines and gc was not measured.

Including gc time, the program executes very differently (table 8.6). The behavior of Lisp programs that require dynamic memory allocation while executing complex single-precision, floating-point arithmetic depends on the size of the memory and the gc strategy.

This program can be used to obtain a combined measure of both gc and complex single-precision, floating-point arithmetic computations of a system.

Table 8.6. Timings (in seconds) of the Lisp version of Cfft2dm

Machine name		Symbolics	MIPS R2000	Compaq 386/20e	IIM	SUN4	DEC8800	
Language		S C-Lisp	IBUKI C-Lisp	Lucid C-Lisp	C-Lisp	Allegro C-Lisp	Allegro CLisp	Lucid C-Lisp
Cfft2dm	e	358.40	211.00	991.45		315.6	589.07	890.81
	r	305.40	210.00	972.01	44.00	310.6	538.00	857.34

e = elapsed real time in seconds. r = machine run time (cpu) in seconds.

The behavior of the Ada version of this program is similar to that of the Whetstone and Linpack Ada versions. The program performed well in Verdix Ada running on the MIPS and poorly in Symbolics Ada (table 8.7).

Table 8.7. Timings (in seconds) of the Ada version of Cfft2dm

Machine Name	Symbolics	MIPS R2000	Compaq 386/20e		DEC 8800
Language	Symbolics Ada	Verdix Ada	Alsys Ada	DDCI Ada	Telesoft Ada
Program Cfft2dm	320.7	8.67	45.26	64.0	25.60

Table 3.8. Timings (in seconds) of the Lisp version of Cfft2dm with optimization.

Optimization	Machine	Elapsed Time	CPU time	GC time	Speed up
Generic	DEC 8800	614.3	264.0	592.8	1.0
With memory expansion	Lucid	371.0	297.0	74.0	1.7
With declarations and memory expansion	Common Lisp	276.0	205	71.0	2.2

8.4 Cholskym

8.4.1 Profiler analysis– There are approximately ninety thousand random number computations that mainly involve integer computation. In addition, there are two million array references and two million floating-point additions, subtractions, and multiplications of array elements. Therefore, the program spends most of its time doing array referencing and floating-point computation.

8.4.2 Program execution– The factor that determines the execution of the initialization phase of the program is the random number generator function used in the program. This is reflected in the execution of the initialization phase of the program (table 8.9). The random number generator uses a 32-bit integer format. When run on some of the stock hardware, Lisp uses a 29-bit integer format. Hence, these random numbers are bignums on these machines and their generation requires CONSing. On machines with 32-bit integer format, these are fixnums or single-floats.

The major factors that affect the execution of the loop are number representation, type checking, arithmetic, and array referencing operations. On stock hardware machines, different Lisp implementations behave similarly. The combined effect of these factors have similar impact on the execution. On the Lisp machines such as the IIM and Symbolics, microcoded arithmetic operations (which take more instructions than built-in arithmetic functions described in section 2.3.1) seem to offset the favorable execution factors like array referencing.

The loop is a clean test of floating-point computation, so this program can be used as a good measure of the floating-point performance of a system, because there is no user defined function calling overhead.

Table 8.9. Timings (in seconds) of the Lisp version of Cholskym

Machine name		Symbolics	MIPS R2000	Compaq 386/20e	IIM	SUN4	DEC8800	
Language		S C-Lisp	IBUKI C-Lisp	Lucid C-Lisp	C-Lisp	Allegro C-Lisp	AllegroCLisp	Lucid C-Lisp
Cholskym	e	271.0	113.0	434.8		126.0	145.1	136.43
loop	r	269.0	112.0	419.8	194.8	82.0	137.8	126.00
Cholskym	e	118.0	70.0	152.5		231.9	400.0	418.54
initialization	r	99.0	70.0	150.8	30.0	212.1	412.0	389.44

e = elapsed real time in seconds. r = machine run time (cpu) in seconds.

For Ada, an optimizing compiler, strong typing and a fast hardware configuration are the major contributing factors of execution. Clearly, for floating-point computations, Lisp is inefficient compared with Ada, and the difference in performance is enormous (table 8.10).

Table 8.10. Timings (in seconds) of the Ada version of Cholskym

Machine Name	Symbolics	MIPS R2000	Compaq 386/20e		DEC 8800
Language	Symbolics Ada	Verdix Ada	Alsys Ada	DDCI Ada	Telesoft Ada
Program cholskym	s 50.0	s 6.75	d 27.90 s 25.05	d 32.0 s 26.0	s 12.20

d = double precision. s = single precision

Table 8.11. Timings (in seconds) of the Lisp version of Cholskym with optimization

Optimization	Machine	Elapsed Time	CPU time	GC time	Speed Up
Generic	DEC 8800	634.0	459.5	102.6	1.0
With declarations	Lucid	107.0	91.0	16.1	5.1
With declarations and memory expansion	Common Lisp	104.0	88.13	0.0	5.1

8.5 Boyer

8.5.1 Profiler analysis– The dynamic counts indicate that Boyer spends time executing recursive function calls and CONSing symbols and lists. Lists in Lisp are recursive in nature. Dynamic counts confirm the program's main goal: to test the list implementation. These tail-recursive function calls can be transformed into iterations for faster execution.

8.5.2 Program execution– This program executes similarly on the SUN-4, IIM, MIPS, DEC 8800, and more slowly on the Compaq 386 and Symbolics (table 8.12).

The function *equal* is used to compare a symbol and a constant. Atoms and CONS are the only types this program uses. This reduces type checking considerably in the BiBOP scheme. A large amount of CONSing is done, which is typical of list operations in Lisp. Because this requires constant allocation of new memory cells, a large memory changes the behavior of this program, because garbage collection may not be required during execution. Because this program has a number of tail-recursive function calls, a Lisp implementation that performs tail-recursion optimization improves execution.

Table 8.12. Timings (in seconds) of the Lisp version of Boyer

Machine name		Symbolics	MIPS R2000	Compaq 386/20e	IIM	SUN-4	DEC8800	
Language		Symbolics C-Lisp	IBUKI C-Lisp	Lucid C-Lisp	C-Lisp	Allegro C-Lisp	AllegroCLisp	Lucid CLisp
Boyer	e	31.0	13.0	27.3		7.54	8.70	10.75
	r	25.49	7.4	11.31	6.0	5.918	8.30	9.88

e = elapsed real time in seconds. r = machine run time (cpu) in seconds.

The behavior of the Ada version of Boyer depends on creating and searching large linked lists. The timing results are given in table 8.13. Because the list of axioms is large and requires a large memory, this program did not execute on the Compaq 386 because of memory constraints. Program restructuring is required to run on the Compaq 386. Further, CONSing also requires memory allocation. Memory allocation is done continuously during the execution of the program. This indicates that systems with a large memory are required to run programs with large linked-list operations. Therefore, Ada may not be efficient for performing list operations similar to Lisp list operations.

Table 8.13. Timings (in seconds) of the Ada version of Boyer

Machine Name	Symbolics	MIPS R2000	DEC 8800
Language	Symbolics Ada	Verdix Ada	Telesoft Ada
Program Boyer	808.6	23.0	57.0

8.5.3 Effect of optimization– Memory expansion speeds up execution by a factor of 2.7. Memory size has a great impact on the type of operations Boyer performs, for example, CONSing and function calls. Because symbols are the only type of objects used in computation, type declarations have no effect. The timing results are shown in table 8.14.

Table 8.14. Timings (in seconds) of Lisp version of Boyer with optimization

Optimization	Machine	Elapsed time	CPU time	GC time	Speed up
Generic	DEC 8800	10.75	9.9	2.6	1.0
memory expansion	Lucid C-Lisp	4.9	4.8	0.0	2.7

8.6 Browse

8.6.1 Profiler analysis– Dynamic counts indicate that this program spends most of the time executing one block of code, testing for the atomic property, performing about two hundred thousand character comparisons, and CONSing. The program has one tail-recursive call and many recursive function calls. This confirms the program analysis.

8.6.2 Program execution– An efficient compiler, a large memory allocation, a fast memory referencing scheme, and optimization of recursive function calls are the major influences on the execution of this program. Character comparison and type checking also influence the execution.

Table 8.15. Timings (in seconds) of the Lisp version of Browse

Machine name		Symbolics	MIPS R2000	Compaq 386/20e	IIM	SUN4	DEC8800	
Language		Symbolics C-Lisp	IBUKI C-Lisp	Lucid C-Lisp	C-Lisp	AllegroC-Lisp	Allegro C-Lisp	Lucid C-Lisp
program	e	18.52	13.00	22.53		6.24	24.38	12.53
Browse	r	11.71	7.6	14.63	6.8	6.12	21.00	11.15

e = elapsed real time in seconds. r = machine run time (cpu) in seconds.

The performance of the Lisp version of Browse is better than that of the Ada version (table 8.16). Character comparisons and extensive search operations proceed through linked lists. This program requires dynamic memory allocation and generates garbage. Without gc, a large memory is needed to execute Browse. On the Compaq 386, because of memory constraints, this program did not run. These results indicate that a gc algorithm is necessary in Ada, and Lisp is a better choice for executing these types of operations.

Table 8.16. Timings (in seconds) of the Ada version of Browse

Machine Name	Symbolics	MIPS R2000	DEC 8800
Language	Symbolics Ada	Verdix Ada	Telesoft Ada
Program Browse	856.4	56.0	66.0

8.6.3 Effect of optimization– The effect of type declaration is insignificant. However, with memory expansion, the performance improves considerably, because of decreased gc. Therefore, a large memory is an advantage for performing pattern matching and exhaustive sequential search operations. Timing results are given in table 8.17.

Table 8.17. Timings (in seconds) of Lisp version of Browse with optimization

Optimization	Machine	Elapsed time	CPU time	GC time	Speed Up
Generic	DEC 8800	12.5	11.6	3.9	1.0
With declarations	Lucid	12.0	11.2	3.9	1.0
With declarations and memory expansion	Common Lisp	8.6	7.6	0.0	1.5

8.7 Traverse

8.7.1 Profiler analysis– This program executes more than three million tail-recursive function calls, performs logical comparisons, and switches the logical sense of the object's value embedded in the structure.

8.7.2 Program execution– The factors responsible for the performance are accessing *defstruct* objects in random distribution and handling of tail-recursive function calls. Though memory access in the SUN-4 is very effective, this is not reflected in the performance of this program (table 8.18). It follows that Allegro Common Lisp does not handle *defstruct* constructs very well. This is also true for DEC 8800 running Allegro Common Lisp; furthermore, this program executes better on the DEC 8800 running Lucid Common Lisp. Symbolics does not optimize tail-recursion calls.

The analysis of the timing results shows that a measure of *defstruct* construct implementation can be obtained using this program.

Table 8.18. Timings (in seconds) of the Lisp version of Traverse

Machine name		Symbolics	MIPS R2000	Compaq 386/20e	IIM	SUN4	DEC 8800	
Language		Symbolics C-Lisp	IBUKI C-Lisp	LucidC-Lisp	C-Lisp	Allegro C-Lisp	Allegro C-Lisp	Lucid C-Lisp
program	e	40.05	24.00	22.53		32.12	97.8	21.9
Traverse	r	39.70	24.30	22.53	16.00	30.96	97.8	21.9

e = elapsed real time in seconds. r = machine run time in seconds.

The compiler and the hardware architecture strongly influence execution of the Ada version (table 8.19). The record type in Ada, which is equivalent to the *defstruct* construct in Lisp, is used to create the nodes of the graph. The operations are accessing the components of records and testing and setting their values. The Ada version executes these operations well. Symbolics Ada is not very effective. A program in Ada or Lisp, executing structure operations, can perform well on both stock hardware and on fast systems.

Table 8.19. Timings (in seconds) of the Ada version of Traverse

Machine Name	Symbolics	MIPS R2000	Compaq 386/20e		DEC 8800
Language	Symbolics Ada	Verdix Ada	Alsys Ada	DDCI Ada	Telesoft Ada
Program traverse	153.0	30.0	21.47	---	37.0

8.7.3 Effect of optimization– Memory expansion has no effect, since Traverse does not generate garbage to be collected. There are some fixnum operations, and type declaration of these fixnum provides some performance improvement. The timing results are summarized in table 8.20.

Table 8.20. Timings (in seconds) of the Lisp version of Traverse with optimization

Optimization	Machine	CPU time	GC time	Speed Up
Generic	DEC 8800	22	0.0	1.0
With declarations	Lucid	19	0.0	1.1
With declarations and memory expansion	Common Lisp	19	0.0	1.1

8.8 Triangle

8.8.1 Profiler analysis– One main function is called recursively about six million times. This function performs array referencing, and comparing and changing the array values. This code is fairly straightforward.

8.8.2 Program execution– The factors that influence the performance are array referencing and type checking. The timing results obtained (table 8.21) are mainly due to the efficiency of array indexing and access. Additionally, on stock hardware, type checking through software takes time.

A Lisp program doing one-dimensional array access can execute well if the system, stock hardware, or Lisp machine has efficient array referencing and compiler array operation code generation.

Table 8.21. Timings (in seconds) of the Lisp version of Triangle

Machine name		Symbolics	MIPS R2000	Compaq 386/20e	IIM	SUN4	DEC8800	
Language		Symbolics C-Lisp	IBUKI C-Lisp	Lucid C-Lisp	C-Lisp	AllegroCLisp	Allegro CLisp	Lucid C-Lisp
program	e	125.63	159.00	331.48	65.20	77.00	252.93	234.6
Triangle	r	125.63	158.21	331.48		75.3	247.00	234.4

e = elapsed real time in seconds. r = machine run time (cpu) in seconds.

Contributing factors for the Ada version are the compiler and the system hardware (table 8.22). The Ada version of the program executes better than the Lisp version because of strong typing in Ada.

Table 8.22. Timings (in seconds) of the Ada version of Triangle

Machine Name	Symbolics	MIPS R2000	Compaq 386/20e		DEC 8800
Language	Symbolics Ada	Verdix Ada	Alsys Ada	DDCI Ada	Telesoft Ada
Program triangle	425.2	86.0	84.0	102.0	92.0

8.8.3 Effect of optimization– Triangle performs integer comparisons and array references. Arrays are declared as simple vectors and integers as fixnums. For arithmetic operations, type declarations are essential to speed up execution. The dynamic bytes CONSed remain constant. There is no temporary creation or destruction of objects to cause gc. Timing results are given in the table 8.23.

Table 8.23. Timings (in seconds) of the Lisp version of Triangle with optimization

Optimization	Machine	Elapsed time	CPU time	GC time	Speed Up
Generic	DEC 8800	234.6	234	0.0	1.0
With declarations	Lucid	52.4	52	0.0	4.5
With declarations and memory expansion	Common Lisp	52.4	52	0.0	4.5

8.9 AutoclassII

8.9.1 Profiler analysis– The program spends most of its time in array referencing and executing floating-point operations, while collecting and computing weights. The profiler results show that the number of computations are directly proportional to the product of the number of cycles, partitions, variable types, and data elements. The results referred to are for 1 and 2 cycles with 531 cases of data, and for 2 cycles with 50 cases of data. Each case has 103 elements of data. In the function collect-weights for one-cycle-one-partition and one data element, the block of code executes

95 times, because there are 95 floating-point elements out of 103 in each case of data. Hence, this program provides the count for array references and floating-point operations, for a given execution set up. The functions *MAX+, log-gamma, log-gamma-dataum-prob, sigms-sq,* and *safe-exp* can be coded inline for faster execution, because some time is spent in these functions.

8.9.2 Program execution– The factors that affect the execution speed of floating-point arithmetic in Lisp are type checking, the number representation scheme, presence of a floating-point coprocessor, gc strategy, and memory. This program generates a considerable amount of garbage, due to number CONSing. A moderate amount of time is spent in gc. A good gc algorithm and a large memory will decrease the gc time. The indirect Lisp array referencing, emulated through software, affects the program execution.

For Lisp programs performing a large number of array referencing and floating-point operations, essential features are immediate number representation, fast array accessing, good gc algorithm, and large memory.

Table 8.24. Timings (in seconds) of the Lisp version of AutoclassII

Machine name		Symbolics	Compaq 386/20e	IIM	SUN4	DEC8800	
Language		S C-Lisp	Lucid C-Lisp	C-Lisp	Allegro C-Lisp	Allegro C-Lisp	LucidC-Lisp
c 2, n 50	e	48.14	252.72		168.346	340.59	234.84
	r	47.81	235.90	29.10	162.684	294.44	229.82
c 1, n 531	e	203.96	2895.67		847.58	1750.8	1939.38
	r	203.23	2844.13	131.3	843.16	1616.4	2017.37
c 2, n 531	e	315.93	4805.91		1402.84	2922.4	3192.27
	r	305.93	4719.04	203.0	1396.80	2612.8	3320.89

c = cycle. n = number of data elements

The Ada version of AutoclassII is a good program to measure the floating-point performance of stock hardware. The factors that affect the execution are the compiler and the hardware. Strong typing in Ada allows the program to perform better than the Lisp version (table 8.25). On the MIPS, floating-point operations are fast. The DEC 8800 floating-point accelerator is relatively slow. The Compaq 386 is a smaller system compared with the others.

Table 8.25. Timings (in seconds) of the Ada version of AutoclassII

Machine name	Symbolics	MIPS R2000	Compaq386/20e		DEC8800
Language	Symbolics Ada	Verdix Ada	Alsys Ada	DDCI Ada	Telesoft Ada
c 2, n 50	630.00	36.00	95.57	102.0	75.65
c 1, n 531	3867.00	210.00	536.00	--	497.20
c 2, n 531	---	340.00	839.98	--	657.55

c = cycles. n = number of data elements

8.9.3 Effect of optimization– As the program was written, the only major effect was memory expansion because of decreased gc. In general, a real application program executing a large number of floating-point operations needs data type declarations, immediate floating-point representation, and a large memory for fast execution. The timing results are provided in table 8.26.

Table 8.26. Timings (in seconds) of the Lisp version of AutoclassII with optimization

Optimization	Machine	Elapsed Time	CPU time	GC time	Speed Up
Generic	DEC 8800	3194	3320	1826	1.0
With declarations	Lucid	3539	3569	2129	1.0
With declarations and memory expansion	Common Lisp	1793	1661	216	2.0

9.0 OTHER CONSIDERATIONS

Besides factors discussed in earlier sections, two other considerations are important. First, the actual design of the benchmark programs, and second, optimization, both in program coding and compiler optimizations.

9.1 Design of Benchmarks

Benchmark programs provide the only means to conduct objective evaluation of systems. These programs can be developed, or they can be selected from existing programs. Benchmark programs can be implemented to reflect different approaches to solve a problem. To select appropriate programs, knowledge about the system implementation is required.

Selecting a known application or an existing program as a benchmark requires a description and analysis of the program, a profile of the program to determine its features, and a careful elimination of features not desired, without sacrificing the effect. The AutoclassII is an example of an application used as a benchmark. Several types of programs, based on a particular need, can be developed to test an implementation.

A well-defined statement of the problem to test an implementation can be used. In that case, the problem needs to be short, not very complex, and capable of running on several systems. It can be implemented as a program in different ways and should be profiled to determine the implementation features it actually tests. The result is analyzed to fine tune as appropriate. Profiling a program is an effective way of determining the features it tests.

The Symbolic Triangle program and the Boyer program (discussed in sec. 3) belong to this category. The Triangle program can be implemented either as an array referencing program, in rule-based shells and tool kits such as KEE (Knowledge Engineering Environment), or in Lisp and Prolog as a rule-based deductive system. The general problem is a well-defined exhaustive search. The general problem of the Boyer program is also well defined, and the implementation it tests is CONSing and function calls.

For a comprehensive coverage of the important computational functions, large programs that have the combination of characteristics of the problem are useful. To develop these programs, all the required characteristics to be tested are selected and combined into a program. The program is profiled to get the distribution and is modified as required. The Whetstone program is a combination of several scientific computational features.

A composite program having a set of features is desirable when a combined effect of the features is required. A combination of different problems can be used to implement different features, or a single problem can also be selected and implemented to reflect different features. However, no composite program can capture all implementation features.

A set of programs, each measuring specific features of the implementation is necessary, for applications planning, where a detailed knowledge of the performance of individual features is required. This involves isolation of features to be measured, achieving measurement, accuracy, and repeatability. A more accurate measurement means eliminating underlying operating system interface from time slice, paging, etc. The University of Michigan and the JSC Ada programs are examples of this type of programs (refs. 6 and 7).

9.2 Optimization

The compiler plays a major role in optimizing a Lisp program to decrease execution time. Additionally, several lisp coding guidelines can be applied to improve the runtime speed of lisp programs. The standard for Common Lisp does not require optimization. Thus, optimization could not be used to compare the various versions of Lisp used. Instead, a study of the effect of optimization was performed on a DEC 8800 running Lucid Common Lisp. Some guidelines for lisp coding for runtime speed optimization are:

1. Use optimization declarations to emphasize speed.

   An optimization declaration controls the type and amount of optimization a compiler can perform. The variables are:

   - the speed at which compiled code runs,
   - the level of space the compiled code needs,
   - the speed at which the code is compiled, and
   - the level of safety (error checking) retained during compilation.

The default values for optimization that define the execution environment and their effects are shown in table 9.1. The integer value of these variables represents the level of optimization.

Table 9.1. **Default settings of system implementations**

Common Lisp	Compilation Speed	Space	Speed	Safety	Effects
Symbolics	unknown	1	3	1	Compiler ignores these declarations
Lucid	0				Allows use of optimizations that may decrease compilation speed.
		0			Imposes no size constraint on the compiled code.
			3		Turns off all the restrictions that affect speed.
				1	Indicates that functions with fixed number of arguments are checked on entry for correct number of arguments.
Allegro	Automatic				Defined by the combination of other three factors.
		1			Turns off in-line coding of safe access functions, which may decrease the size of compiled code.
			1		Does not turn off any restrictions that affect speed to preserve robustness of the compiled code.
				1	Enables argument count and interrupt checking.
IBUKI	0				Not defined.
		0			Imposes no constraint on the size of the compiled code, which means that the compiled code may be larger and faster.
			3		Turns off optimization switch of the C compiler that affects speed.
				0	No run time error checking.

2. Specify data types of arguments and returned values of Lisp expressions.

   A declaration is a statement that supplies information about a Lisp program to the Lisp environment. These advise the compiler so it may produce faster, efficient code. This particularly applies to type declarations, which specify the data types of the values of Lisp expressions, and it eliminates type checking.

3. Use explicit type declarations for floating-point and fixnum operations to increase speed.

   Type declarations added to arithmetic operations can make the operations significantly faster by reducing type checking and type dispatching overhead of a function call. For fixnum type arguments and values, the compiler can directly code applications of arithmetic operators, making fixnum arithmetic fast. The local variables that are declared as floating-point type are allocated in a special block on the stack or in registers. These variables are not gc.

4. Use simple arrays and simple vectors to increase array access efficiency.

    The element type and the number of dimensions of the array should be declared.

5. Code simple functions in-line.

    This is a request to the compiler to generate machine language code for a function to eliminate function calling overhead.

6. Use macros, loop unrolling, and tail-recursive functions.

    A tail-recursive function is a function that calls itself as the last operation. The body of a tail-recursive function can be converted into iteration; this eliminates the need to preserve the execution environment of previous calls. Also, if the iteration is straightforward, the body of the block is replaced with several copies of the body. This is called loop unrolling, and reduces the overhead of looping.

10.0 SUMMARY OF RESULTS

Knowing what is being measured requires a thorough examination of the program. Such knowledge can be obtained from running the program through the profiler to obtain static and dynamic counts of functions used, determining the operations it mainly executes, and analyzing execution time. Applying optimizing techniques enables study of what is required to improve the performance.

In this study to evaluate the performance of Ada and Lisp programs, it is evident that Lisp is not the right choice for numeric computation, and Ada is not the right choice for performing Lisp list-like operations.

For programs written in Lisp, the important aspects for effective performance are a large memory and the declaration of types, which is especially true for numeric operations. Immediate representation of numeric data and a fast array referencing scheme are also essential. By incorporating these features, a Lisp program can be made to execute efficiently on stock hardware. Lisp is not a good choice for extensive scientific computations.

10.1 Ada Version Implementation of the Symbolic Set

Programs written in Ada, which involve large linked-list operations executing in a dynamic environment, perform poorly and may fail to execute. Boyer requires the creation of a large linked-list of records that requires memory allocation. The CONSing performed also requires memory allocation. Almost always, the problem of gc arises with CONSing. Implementing gc increases the complexity of the computation. This program did not run on the Compaq 386 because available memory was insufficient for representing the large linked list. The program can be restructured to run on a Compaq 386 by breaking the large linked list into several small ones and relinking them.

This will not solve the problem entirely when the majority of computation involves memory allocation, and a system can crash because of insufficient memory. A large number of linked list operations and CONSing demand a large memory. In this program, although there are character comparison operations, it amounts to testing the memory size using the *new* allocator. Therefore, this program is not a good measuring tool of any Ada features.

Browse, which is similar to the Boyer category, uses small linked lists of records. It performs a large number of CONSes while searching through these linked lists in the exhaustive pattern matching. A considerable amount of garbage is generated. This program also did not execute on the Compaq 386 because of insufficient memory. A gc routine needs to be written to execute on the Compaq 386. This is a difficult task, so Ada is not a practical choice for performing exhaustive pattern matching using Lisp list-like operations. A relative performance measure of memory allocation and a search of the linked list of records can be obtained by running this program on systems with a large memory.

Triangle executes one-dimensional array references and recursive function calls. Recursive function calls require a large, high-speed stack for efficient execution. The stack operations are system dependent. This program provides a measure of the combined effects of these operations.

Traverse initially allocates memory for creating an initial linked list of records, generates garbage while creating a randomly distributed linked list out of the initial list, and uses recursive function calls to Traverse through this randomly distributed list where it accesses the components of records. As recursive function calls involve stack operations that are system dependent, it is difficult to quantify the performance of any one feature of Ada. However, a measure of the combined effect of accessing records in a random distribution and recursive function calls can be obtained by timing the traversal section only.

Ada programs of the numeric set provide a good measure of their respective features in an implementation.

10.2 Lisp Version Implementation of the Numeric Set

The optimization results of the Numeric Set benchmark programs (tables 8.3, 8.5, 8.8, and 8.11) lead to the conclusions:

1. A compiler that heeds declarations saves significant time by speeding up calculations through elimination of type checking, and also significantly reduces gc time as well. This is because many intermediate calculations are done in the stack, rather than as general objects in dynamic memory.

2. Declarations improve complex number arithmetic relatively little because of the creation of temporary results in dynamic storage.

3. Expanding the memory has a profound effect on the overall execution, more so than including declarations.

When the programs Whetstone, Linpack, and Cholesky are run with memory expanded before execution, they do not perform (or spend time doing) gc. If run with expanded memory and all the type declarations, Linpack, Cholesky, and Whetstone can provide a measure of the respective features they are designed to test in a Lisp implementation. Despite memory expansion and declarations, the Cfft2dm program still consumes time doing gc. This, with creation of temporary results from complex arithmetic operations, among other Lisp system overhead, makes an exclusive measure of complex number arithmetic nearly impossible. However, Cfft2dm still provides an overall measure of these arithmetic operations in a Lisp implementation.

REFERENCES

1. Gabriel, R. P.: Performance and Evaluation of Lisp Systems. MIT Press 1985.

2. Steel, G. L., Jr.: Common Lisp: The Language. Digital Press, 1984, pp. 11-53.

3. Moon, D. A.: Architecture of the Symbolics 3600. IEEE 12th Intl.Symp.on Computer Architecture, June 1985, pp. 76-83.

4. Garner, Robert B.; et al.: The Scalable Processor Architecture(SPARC). Sun Microsystems, Inc. 2550 Garcia Avenue, Mountain View, CA 94043.

5. Harbaugh, Sam; and Forakis, John A.: Timing Studies Using a Synthetic Whetstone Benchmark. Ada Letters, Vol. IV, No. 2, Sep/Oct 1984.

6. Dimorier, Keith L.: ADA Software Benchmarking. CSDL-R-2067, NASA, May 1988.

7. Clapp, R. M.; et al.: Toward Real-Time Performance Benchmarks for Ada. RSD-TR-12-86, University of Michigan, Department of Electrical Engineering and Computer Science, Center for Research on Integrated Manufacturing.

8. Baily, D. H.; and Barton, J. T.: The NAS Kernel Benchmark Program. NASA TM-86711, August 1985, NASA Ames Research Center, Moffett Field, CA 94035.

9. Cheeseman, P.; et al.: An Approximation to Bayesian Classification. Information Sciences Division, NASA Ames Research Center, Moffett Field, CA 94035.

10. Kleiman, S. R.; and Williams, D.: SunOS on SPARC. Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043.

11. Galant, David C.: Basic Mathematical Function Libraries for Scientific Computation. NASA TM-102256, NASA Ames Research Center, Moffett Field, CA 94035.

12. Katevenis, M.: Reduced Instruction Set Computer architectures for VLSI. Ph.D dissertation, Computer Science Div., University of California, Berkeley, 1983.

13. Agrawal, Anant; et al.: SPARC: An ASIC Solution for High Performance MIcroprocessor. Sun Microsystems, Mountain View CA.

14. Sobalvarro, Patric G.: A lifetime-Based Garbage Collector for Lisp Systems on General Purpose Computers. B.S. Thesis, Electrical Engineering and Computer Science MIT, Supervisor Robert H. Halstead, Jr.

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE August 1992	3. REPORT TYPE AND DATES COVERED Technical Memorandum

4. TITLE AND SUBTITLE Analysis of a Benchmark Suite to Evaluate Mixed Numeric and Symbolic Processing	5. FUNDING NUMBERS 506-59-31
6. AUTHOR(S) Bharathi Ragharan and David Galant	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Ames Research Center Moffett Field, CA 94035-1000	8. PERFORMING ORGANIZATION REPORT NUMBER A-92023
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	10. SPONSORING/MONITORING AGENCY REPORT NUMBER NASA TM-103907

11. SUPPLEMENTARY NOTES

Point of Contact: David Galant, Ames Research Center, MS 269-3, Moffett Field, CA 94035-1000; (415) 604-4851

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified — Unlimited Subject Category 62	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The suite of programs that formed the benchmark for a proposed advanced computer is described and analyzed. The features of the processor and its operating system that are tested by the benchmark are discussed. The computer codes and the supporting data for the analysis are given as appendices.

14. SUBJECT TERMS Benchmarking computer systems, Computer systems performance evaluation, Lisp machine evaluation, Benchmark analysis	15. NUMBER OF PAGES 36
	16. PRICE CODE A03

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT