Performing Lisp
Analysis of the FANNKUCH Benchmark

Kenneth R. Anderson
BBN STD,10 Moulton St. Cambridge, MA 02174, KAnderson@bbn.com

Duane Rettig
Franz Inc., 1995 University Ave., Berkeley CA, 94704, Duane@franz.com

ABSTRACT

This paper analyzes the FANNKUCH benchmark, that was discussed on the comp.lang.lisp internet newsgroup during September 1994, and reviews the performance issues underlying it. This benchmark involves operations on integers and vectors of integers so one might expect that Lisp and C versions could have comparable performance. However, the original benchmark suggested that the Lisp version was at least 10 times slower than the C version. While this version appeared to be optimized, several important improvements are possible.

The improved version is between 24 and 116 percent slower than C when run on several Lisp implementations. This can be accounted for by differences in the quality of the compiled code of the inner loops of the benchmark, not by an essential difference between the two languages. The GNU C compiler, `gcc`, produces a loop with a larger overall size (footprint) but with a smaller loop body than the current Lisp compilers. In principle, a Lisp compiler can produce these loops with the same or fewer number of instructions.

It is easy to write benchmarks that make Lisp appear slower than C. However, as with any highly tuned benchmark, a small change can have a profound effect on performance. even in C. For example, replacing `/2` by `>>1` makes a 40% improvement. Also changing the representation of integers and arrays of integers among C's built in types varies the performance by 80%.

HISTORY

This benchmark came out of a thread on comp.lang.lisp in September 1994 originated by Bruno Haible (haible@ma2s2.mathematik.uni-karlsruhe.de). The original post introduced the language Beta to the news group and in passing mentioned an "integer hacking" benchmark that indicated that some Lisp implementations were much slower (50 to 100 times) on the benchmark than C.

Haible describes the benchmark as follows: Take a permutation of {1,...,n}, for example: {4,2,1,5,3}. Take the first element, here 4, and reverse the order of the first 4 elements: {5,1,2,4,3}. Repeat this until the first element is a 1, so flipping won't change anything more: {3,4,2,1,5}, {2,4,3,1,5}, {4,2,3,1,5}, {1,3,2,4,5}, Count the number of flips, here 5. Do this for all n! permutations, and record the maximum number of flips needed for any permutation. The conjecture is that this maximum count is approximated by n*log(n) when n goes to infinity. FANNKUCH" is an abbreviation for the German word "Pfannkuchen", or pancakes, in analogy to flipping pancakes. The first few fannkuch numbers and their corresponding permutation are shown in Exhibit 0.

This is a good benchmark because it can easily be programmed in different languages and only uses common language features like manipulation of small integers and vectors of them. The original Lisp version of this benchmark is shown in Exhibit 1.

Exhibit 0 - The first few FANNKUCH numbers and their permutations.

N	F	Permutation
1	0	(1)
2	1	(2 1)
3	2	(2 3 1)
4	4	(3 1 4 2)
5	7	(3 1 4 5 2)
6	10	(5 6 4 1 3 2)
7	16	(3 1 4 6 7 5 2)
8	22	(6 1 5 7 8 3 2 4)
9	30	(6 1 5 9 7 2 8 3 4)
10	38	(5 9 1 8 6 2 10 4 7 3)
11	51	(4 9 11 6 10 7 8 2 1 3 5)
12	65	(2 6 1 10 11 8 12 3 4 7 9 5)
13	80	(2 9 4 5 11 12 10 1 8 13 3 6 7)

Exhibit 1 - The original Lisp version of the FANNKUCH benchmark.

```
(defun fannkuch-0 (&optional (n (progn
                                  (format *query-io* "n = ?")
                                  (parse-integer (read-line *query-io*)))))
  (unless (and (> n 0) (<= n 100)) (return-from fannkuch-1))
  (let ((n n))
    (declare (fixnum n))
    (let ((perm (make-array n :element-type 'fixnum))
          (perm1 (make-array n :element-type 'fixnum))
          (zaehl (make-array n :element-type 'fixnum))
          (permmax (make-array n :element-type 'fixnum))
          (bishmax -1))
      (declare (type (simple-array fixnum (*)) perm perm1 zaehl permmax))
      (declare (fixnum bishmax))
      (dotimes (i n) (setf (svref perm1 i) i))
      (prog
       ((r n))
       (declare (fixnum r))
       Kreuz
       (when (= r 1) (go standardroutine))
       (setf (svref zaehl (- r 1)) r)
       (decf r)
       (go Kreuz)
       Dollar
       (when (= r n) (go fertig))
       (let ((perm0 (svref perm1 0)))
         (dotimes (i r) (setf (svref perm1 i) (svref perm1 (+ i 1))))
         (setf (svref perm1 r) perm0))
       (when (plusp (decf (svref zaehl r))) (go Kreuz))
       (incf r)
       (go Dollar)
       standardroutine
       (dotimes (i n) (setf (svref perm i) (svref perm1 i)))
       (let ((Spiegelungsanzahl 0) (k 0))
         (declare (fixnum Spiegelungsanzahl k))
    (loop
     (when (= (setq k (svref perm 0)) 0) (return))
     (let ((k2 (ceiling k 2)))
       (declare (fixnum k2))
       (dotimes (i k2) (rotatef (svref perm i) (svref perm (- k i))))))
```

```
 (incf Spiegelungsanzahl))
(when (> Spiegelungsanzahl bishmax)
     (setq bishmax Spiegelungsanzahl)
     (dotimes (i n) (setf (svref permmax i) (svref perm1 i)))))
  (go Dollar)
  fertig)
 (format t "The maximum was ~D.~% at " bishmax)
 (format t "(")
 (dotimes (i n)
  (when (> i 0) (format t " "))
  (format t "~D" (+ (svref permmax i) 1)))
 (format t ")")
 (terpri)
 (values))))
```

Several people studied the benchmark. Lawrence Mayka (lgm@polaris.ih.att.com) produced a properly optimized Lisp version (these and other optimizations are described below). Jacob Seligmann (jacobse@daimi.aau.dk) provided Haible's C version and a Beta version. Several people complained about the old fashioned "spaghetti" style of using a `prog` and `go`. However, it is straight forward to unwind the spaghetti into several nested loops. Another complaint was that the benchmark involved array creation and formatted output in addition to integer and vector operations which might contaminate the results.

The following exhibit shows the time in seconds for running various versions of the fannkuch function, where `N = 9` or `10`, for various Lisp and C implementations on a Sun Sparc 10:

Exhibit 2: Time in seconds for various versions of the Fannkuch benchmark. The rightmost column shows the time relative to `gcc -O2`. Each Lisp used an optimization setting of `(optimize (speed 3) (safety 1))`.

Language	Compile Info	Original N = 9	Final N = 9	Final N = 10	Relative N = 10
C	gcc -O2		1.70	20.60	1.00
C	cc -O2		1.80	22.10	1.07
Lisp	Allegro	13.58	2.00	25.62	1.24
Lisp	CMU	8.49	2.36	27.22	1.32
Lisp	Lispworks	15.29	2.74	32.56	1.58
Lisp	Lucid 2			35.74	1.73
Lisp	Lucid 1	7.38	3.75	44.57	2.16
C	cc		5.90	70.90	3.44
C	gcc		6.40	76.70	3.72

The original Lisp version is about 10 times slower than best C time, while the optimized Lisp version is only between 24 and 116% slower. While the original Lisp version, `fannkuch-0`, looked optimized, there were several important improvements in the optimized version. To understand them, we review how integer and vector objects are represented in Lisp and C.

In C, type information is associated with each variable name at compile time, not with the datum actually stored there. Lisp takes the opposite approach, the type of a datum is associated with the datum itself, not with a variable name (compile time variable declarations are optional). So, in C you can't tell the data without a program, while in Lisp you can. The C approach is referred to as "static typing" while the Lisp approach is referred to as "dynamic typing". However, this is only from the point of view of the variables. From the point of view of the data, it is the other way around.

C's numeric types closely match what can be easily manipulated by computer hardware. There is an `int` type which is a machine integer, and several modifiers such as "`short`", "`long`",and "`unsigned`" that can be used to describe subsets of machine integers. For example, the integers used in the C version of the benchmark are declared to be "`unsigned int`", or unsigned integers of the standard word size. Integer operations simply overflow and integer division is truncation.

In Lisp, an object is typically represented by a machine word-sized "object description" that contains some information about the object's identity and type. The actual representation is implementation specific, see [Gudeman] for the range of possibilities. Thus when describing such details we refer to a typical implementation, not a specific one. For example, on a 32-bit RISC machine, two to four of the low order bits could provide type information, and the remaining high order bits could provide either the object itself, or a pointer to the actual object. For a small, "immediate object", such as a fixnum, character, or short-float, the object itself can fit entirely in the data portion of its description. For a larger "indirect" object, such as a bignum, or array, the data field of the description is a pointer to the object itself.

Lisp supports infinite precision integers, and integer division produces infinite precision ratios. The abstract type `integer` is typically implemented using two different representations, `fixnum` for small integers and `bignum` for large integers.

A two bit type tag of `#b00` is commonly used for fixnums because it has several advantages. Such a fixnum looks like a machine integer multiplied by four. Thus adding one to a fixnum is the same as adding four to its machine integer representation. This is convenient for indexing through an array of word-sized elements in a byte addressed machine. Also, RISC machines provide instructions for doing operations on such integers that provide some automatic type checking.

In Lisp, the main performance consideration when using integers is to use fixnums whenever appropriate. Also, declaring a variable to be a fixnum can allow the compiler to use machine instructions rather than costlier generic operations. However, declaring arguments to the functions `+`, `-`, `*`, and `truncate` to be fixnums is not enough since there are cases where each function could produce a bignum given only fixnum arguments. One must either declare the output of the functions to be a fixnum, such as `(the fixnum (- k i))` , or provide a more restrictive declaration on the arguments. The function `deftype` is convenient for this. For example, since all integers used in this benchmark are between 0 and 100:

```
(deftype small () '(integer 0 100))
```

A final point about integer arithmetic is that the function / produces a ratio while `truncate` produces an integer. Be careful with this when converting software from C to Lisp. For example, `5/3` in C produces `1`, while `(/ 5 3)` in Lisp produces `5/3`. While this is mathematically correct, accidental use of ratios can make an algorithm significantly slower.

Now, we can understand several of the improvements that can be made to the original version of the benchmark:

> **IMP1:** Operations on fixnum arguments should be declared to return fixnum results when appropriate. Alternatively, declare variables to be a subset of the fixnum range.

The second problem has to do with `dotimes`. It is obvious from the definition of `(dotimes (i N) ...)` where `N` is a positive fixnum, that `i` is also a positive fixnum between 0 and `(- N 1)`. Unfortunately, not all compilers can deduce this fact, so to be safe you should declare it so. One can use a specialized `dotimes` macro for this purpose. Without such a declaration, more general, but slower, increment and comparison operations are done.

IMP2: Declare the index and limit of `dotimes` to be fixnums when they are.

To understand the next performance issue, we need to review arrays. In C, an array is simply a pointer to a block of memory. Each element of the block is of the same data type and occupies same amount of space.

Lisp provides several types of arrays. Fixed length arrays are referred to as type `simple-array`. Other array types can be variable length or be displaced on top of another array. Generally, an array element can be an object of any type (actually any object description), however, elements can be restricted to be of certain types, such as `(unsigned-byte 32)`, an unsigned thirty two bit integer, or `single-float`, a single precision floating point number.

Two reasons for using such arrays are 1) Lisp can refer to such objects without heap allocating (bignums or single-floats in the cases mentioned above), and 2) such arrays can be used to interface with other languages, such as C.

`Fannkuch-0` creates array of type `(simple-array fixnum (*))`. Such an array is like an int array in C, each array element is stored as a machine integer. However, the array is accessed using `svref` which should only be used on arrays of type `(simple-array t (*))`, also know as type `simple-vector`. While Lisp warns about this declaration conflict, it treats the array as a `simple-vector`. Since the routine never returns the arrays involved, the bug goes unnoticed.

To be consistent, one should use either a `(simple-array fixnum (*))` or a `simple-vector`. However, each has a slightly different effect on performance. When a fixnum array is used the compiler can use the fact that each element is a fixnum. However, the array elements are stored as machine integers. This requires an extra shift instruction when the array is accessed.

Using a simple-vector will avoid this overhead. However, when storing a fixnum into a simple-vector, one should be sure that the compiler knows that the value being stored is a fixnum. The reason for this has to do with the generational garbage collector used in some Lisp systems, also referred to as a ephemeral garbage collector, or EGC. Such garbage collectors focus their attention on recently created objects under the assumption that they are likely to become garbage. This can substantially reduce the total effort involved in garbage collection. However, a reference from an old object to a new one must be recorded. While this can be done efficiently, this overhead can be avoided when an immediate object, such as a fixnum is stored.

To summarize, consideration of arrays leads to the following improvements:

IMP3: Use `svref` only on simple-vectors. Better yet, declare the array and use `aref`.

IMP4: For an array of fixnum's use type `simple-vector` when dealing only with Lisp, and use type `(simple-array fixnum (*))` when dealing with another language, like C.

IMP5: When storing a `fixnum` into a `simple-vector` be sure the compiler knows that the value being stored is a fixnum.

A major functional difference between the Lisp and C versions of the original benchmark is that the Lisp version has `(ceiling k 2)` while the C version has the hand optimized equivalent `(k+1)/2`, which is optimized by the compiler to `(k+1)>>1` since `k` is an unsigned int. (This `/2 -> >>1` optimization makes a 40% improvement in the C version.) While Lisp provides a ceiling function that works for one required and one optional argument, C only provides one for a single double argument, `ceil`. Using `ceil` instead of the optimized version makes the C version of the benchmark 2.88 times slower. There is a similar effect in Lisp. The time difference between

the different Lisp's on the original benchmark is largely due to their treatment of `ceiling`. It is simply not possible for either a Lisp or a C compiler to produce code as good as the hand optimized version which uses information not shared with the compiler.

Profiling tools can be used to identify such bottle necks. For example, the profiler in Allegro Common Lisp show that at least 60% of the time of the original benchmark was spent inside `ceiling`.

IMP6: Identify performance bottlenecks using profiling tools and minimize their effect, for example by inlining.

The algorithm generates each of the `n`! permutations of `n` integers and computes the number of flips. It would be better, of course, to generate the permutations in a way that poorer ones can be immediately eliminated from further consideration. For example, the heuristic, "The best permutation can not start with `1` or end with `n`", leads to a simple check that saves about nineteen percent of the effort.

IMP7: Thinking about the algorithm can lead to improvement.

RESULTS OF THE IMPROVEMENTS

Exhibit 3 and 4 show the Lisp and C version of comparable benchmarks. Both versions were written in similar styles to allow easy comparison. The Lisp version uses macros to help in type declaration. Such heavy handed declarations are not required by most Lisp's but makes porting the benchmark easier. Lisp macros such as `rotatef` are avoided, as are C constructs such as ++ or `register` declarations.

These improvements lead to a dramatic improvement in the Lisp performance on this benchmark. Lisp is within 24% and 116% of the best C time.

So why are the Lisp versions slower? Lisp and C implementations produce essentially the same number of machine instructions, 106 for cc, and 108 for Allegro, for example (see the leftmost column of Exhibit 5). Exhibit 3 includes counts of the number of times key blocks of code are entered (referred to as "Weights" in Exhibit 5). The three innermost loops account for over 80% of the execution time.

Exhibit 3. Fannkuch benchmark in Lisp.

```
(deftype small () '(integer 0 101)))
(defmacro small (a) `(the small ,a))
(defmacro s+ (a b) `(small (+ (small ,a) (small ,b))))
(defmacro s- (a b) `(small (- (small ,a) (small ,b))))
(defmacro s> (a b) `(> (small ,a) (small ,b)))
(defmacro s= (a b) `(= (small ,a) (small ,b)))
(defmacro sref (a i) `(small (svref ,a ,i)))
(defmacro setfs (a b) `(setf ,a (small ,b)))
(defmacro dotimess ((i n) &body body)
   (dotimes (,i ,n) (declare (type small ,i)) ,@body))
```

```
(defun fannkuch (n perm perm1 zaehl permmax)
  (declare (optimize (safety 0) (speed 3) (space 0) (debug 0))
           (type simple-vector perm perm1 zaehl permmax)
           (type (integer 1 100) n))
  (dotimess (i n)
   (setfs (sref perm1 i) i))                          ; FILL-I
  (let ((bishmax -1)
        (r n))
    (loop                                             ; 1.00
      (loop                                           ; 1.72 KREUZ
        (when (s= r 1) (return))
        (let ((i (s- r 1)))
          (setfs (sref zaehl i) r)
          (setq r i)))
      (when (not (or (zerop (sref perm1 0))
                     (let ((i (s- n 1)))
                       (s= (sref perm1 i) i))))
        (dotimesS  (i n)                              ; 0.81
         (setfs (sref perm i) (sref perm1 i)))        ; 8.11 COPY
        (let ((Spiegelungsanzahl 0)
              (k 0))
          (loop                                       ; 6.39 COUNT
            (when (s= (setq k (sref perm 0)) 0) (return))
            (let ((k2 (the small (ash (s+ k 1) -1))))
              (dotimess (i k2)
               (let* ((temp (sref perm i))            ; 14.73 FLIP
                      (j (s- k i)))
                 (setfs (sref perm i) (sref perm j))
                 (setfs (sref perm j) temp))))
            (setq Spiegelungsanzahl (s+ Spiegelungsanzahl 1)))
          (when (s> Spiegelungsanzahl bishmax)
            (setq bishmax Spiegelungsanzahl)
            (dotimesS (i n)
             (setfs (sref permmax i)(svref perm1 i))))))
      (loop                                           ; 1.72
        (when (s= r n) (return-from fannkuch-10 bishmax))
        (let ((perm0 (sref perm1 0)))
          (let ((i 0))
            (loop                                     ; 4.44 SHIFT
              (if (s= i r) (return))
              (let ((k (s+ i 1)))
                (setfs (sref perm1 i) (sref perm1 k))
                (setq i k))))
          (setfs (svref perm1 r) perm0))
        (when (s> (setfs (sref zaehl r) (s- (sref zaehl r) 1)) 0)
          (return))
        (setq r (s+ r 1))))))
```

Exhibit 4. Fannkuch benchmark in C.

```
#define ASMALL unsigned int
#define SMALL  unsigned int
#define Length 100

long fannkuch(n, Perm, Perm1, Zaehl, PermMax)
     SMALL n;
     ASMALL Perm[], Perm1[], Zaehl[], PermMax[];
{
  long BishMax=-1, Spiegelungsanzahl;
  SMALL r, i, k;
  for (i=0; i<n; i++)
    Perm1[i]=i;
  r=n;
  while(1) {
    while (r != 1) {
      Zaehl[r-1]=r;
      r=r-1;
    }
    if (! (Perm1[0] == 0 || (i=n-1, Perm1[i] == i))) {
      for (i = 0; i < n; i = i + 1)
        Perm[i] = Perm1[i];
      Spiegelungsanzahl=0;
      while (!((k=Perm[0]) == 0)) {
        SMALL k2=(k+1)>>1;
        for (i = 0; i < k2; i = i + 1) {
          SMALL temp = Perm[i];
          Perm[i] = Perm[k - i];
          Perm[k - i] = temp;
        }
        Spiegelungsanzahl = Spiegelungsanzahl + 1;
      }
      if (Spiegelungsanzahl > BishMax) {
        BishMax = Spiegelungsanzahl;
        for (i = 0; i < n; i = i + 1)
          PermMax[i] = Perm1[i];
      }
    }
    while(1) {
      if (r == n) return(BishMax);
      { SMALL Perm0;
        Perm0 = Perm1[0];
        i = 0;
        while (i < r) {
          k = i + 1;
          Perm1[i]=Perm1[k];
          i = k;
        }
        Perm1[r]=Perm0;
      }
      if ((Zaehl[r] = Zaehl[r]-1) > 0) break;
      r=r+1;
    }
  }
}
```

Several loops were disassembled to compare the quality of the compiled code, as shown in Exhibit 5. The row labeled "Weights" show the relative importance of each loop in the computation, and the column labeled "Dot" shows the dot product of the weights and the lines of code. It correlates well with relative performance, except for Lucid, which will be discussed below.

Exhibit 5: Lines of code for the fannkuch function and several loops.

	Fannkuch	**Fill-i**	**Shift**	**Copy**	**Flip**	**Kreuz**	**Dot**
cc	106	5	9	6	8	6	217
gcc -O2	117	5	8	5	10	5	232
Allegro	108	5	8	7	12	6	279
CMUCL	123	7	9	8	14	10	328
Lispworks	132	9	11	12	18	9	427
Lucid	139	6	9	8	13	8	310
Weight		**0.00**	**4.44**	**8.11**	**14.73**	**1.72**	

The bodies of the loops are smaller for C than for Lisp, while the overall footprint of a loop is smaller for Lisp than for C. Take a loop like `(dotimes (i n) ...)`. In a simple C-like assembly language, the loop is coded as something like: (Each line corresponds to a RISC machine instruction, except that `if <condition> goto <address>` corresponds to two instructions.)

```
     GCC loop                Typical Lisp loop

     i = 0                       i = 0
     if (i >= N) goto end        goto test
        ...                  top  ...
top  ...                         i = i + 4
     i = i + 1               test if (i < N) goto top
     if (i < N) goto top
end
```

The difference is a matter of style, rather than language. The gcc compiler does a test and starts executing the body of the loop, while the typical Lisp code jumps to the test which is at the end of the loop body.

The number of instructions in the body of the loop varies between the different Lisps. The Lisps with the higher instruction counts are typically due to one or more redundant instructions that could be removed. There is no reason such deficiencies could not be corrected.

The loop body for `fill-i`, for example, shows the influence of the two languages:

```
   In C:                          In Lisp:
  for (i=0; i< N; i = i + 1)    (dotimes (i N)
   p[i] = i;                     (setf (aref p i) i))

      Compiled C                 Compiled Lisp

1       i = 0                        i = 0
2       goto test                    goto test
3 top   temp = i << 2           top  temp = i + ARRAYOFFSET
4       p[temp] = i                  p[temp] = i
5       i = i + 1                    i = i + 4
6 test  if (i < N) goto top     test if (i < N) goto top
```

The loops have been written in the same style for ease of comparison. While both take the same number of instructions, there are two differences. Lisp uses fixnums, while C uses machine integers. We can see the effect of this in line 5 where `i` is incremented by `4` rather than by `1`, as in the C version. In C, where machine integer are used, an `int` must be left shifted before it can be used as an array offsets (alternatively, another counter incremented by one could be used).

On the other hand, when accessing a Lisp array, one must account for the array type tag. This is done by adding an offset (`ARRAYOFFSET`) to the index to remove the tag. On CISC computers, like the M68000's, the offset and index can be added to the array pointer in one instruction, but on a RISC machine, two instructions are required.

While `ARRAYOFFSET` is a loop constant, it is generally not pulled out of the loop if an interrupt could occur during the loop. If a GC occurs, it must be possible to recognize all pointers to an array as Lisp values, so that the pointers can be changed if the array changes location. Removing the `ARRAYOFFSET` removes the tag information. On the other hand,when a loop is noninterruptable, the compiled Lisp code could be smaller than the compiled C code.

One final issue is register versus stack allocation of variables. In a function call on a SPARC, the first six arguments are passed in registers, while any additional arguments are passed on the stack. Internal to a subroutine, 24 registers may be used (after a `save` instruction). An argument on the stack must be moved to a register before its value can be used. Thus an important optimization is to move a stacked variable into a register whenever possible. Unfortunately, Lucid Common Lisp does a register analysis, but it is suboptimum here. Important variables are kept on the stack, reducing its performance further. Unintuitively, performance can be improved by 20% by moving the inner count loop as a separate subroutine (Referred to as "Lucid 2" in Exhibit 2.). An earlier version of Allegro had a register allocation problem here as well, and the difference between the two C implementations is due to register/stack usage differences.

CONCLUSIONS

Lisp provides freedom that can be valuable during software development, but can get in the way of delivering performance if one is not careful. It might seem that recoding in C is a good way to deliver performance. However, it is the recoding process itself, and the analysis of perfomance details that is important, not the language. Today's Lisp compilers provide advice that helps identify performance pitfalls. Such advice would have identified most of the improvements described above. Also in CMU Common Lisp, with low optimization settings, declarations are verified at runtime rather than simply trusted, while at higher settings, they are used to produce optimized code. This allows a fluid software development style where performance issues can be addressed at various stages.

IMP8: Heed compiler advice.

C requires the programmer to provide explicit information from the beginning, so it might seem straightforward to write efficient C here. However, C is not without performance pitfalls either. Using the floating point version of `ceiling`, rather than a hand coded one makes a factor of three difference in performance and changing "`/2`" to "`>>1`" makes a 40% improvement. The hand optimized version might be fairly obvious here, but isn't in general [Bentley].

Other choices a C programmer could make could have performance effects that are difficult to predict [Bentley]. For example, the original C version of the benchmark used register declarations which reduced performance by a few percent. Also, by varying the definitions of `ASMALL` and `SMALL` among the types `char`, `short`, `int`, `unsigned short`, and `unsigned int` the performance of the algorithm ranges from 15% faster to 65% slower than the time in Exhibit 2. The fastest combination is not particularly obvious:

```
#define ASMALL char
#define SMALL  unsigned int
```

Gabriel [Gabriel, 85] teaches us that benchmarking without analysis is bogus. This benchmark is a perfect example. Had one accepted the original performance numbers without further analysis one would conclude that Lisp was significantly slower than C, and that certain implementations of Lisp were significantly slower than others. Both conclusions are false. It is easy to make Lisp look slow relative to C, and one Lisp implementation look slower than another simply by steering the benchmark away from one implementation's pitfalls and toward another's, intentionally or otherwise.

How much difference should one expect between Lisp and C? This benchmark is quite sensitive to the number of instructions generated for the inner loops. The data shows that Lisp compilers can be improved here. (Three improvements were made to Allegro Common Lisp in the process of analyzing this benchmark.) Lisp itself provides little if any additional overhead in this benchmark as both integer operations and array manipulation are comparable in both languages. A benchmark with a different set of loop weights might have shown more comparable results. For example a uniform weighting would produce differences of less than 40% between the different language implementations. Other benchmarks suggest that Lisp programs can be quite comparable, even faster than C programs [Anderson]. It is reasonable to expect that Lisp should be within 20% or less of the C time for benchmarks like this one [Baker]. Anything worse might indicate a problem to be corrected, as was done here.

REFERENCES

Anderson, Kenneth R. "Courage in Profiles", Lisp Users and Vendors Conference, (1994).

Baker, H.G. "Critique of DIN Kernel Lisp Definition Version 1.2", *Lisp and Symbolic Computation,*, 4,4 (march 1992), 371-398.

Bentley, Kernighan and VanWyk, "An elementary C cost model", Unix Review, 9,2, p. 38-48.

Gabriel R.P.,*Performance and Evaluation of Lisp Systems*.. MIT Press, Cambridge, MA, 1985.

Gabriel R.P., "Lisp: Good News, Bad News, How to Win Big", AI Expert, June, 1991, p. 31-39.

Gudeman, David. "Representing Type Information in Dynamically Typed Languages", TR 93-27, Dept. Computer Science, University of Arizona, Tucson, 1993. FTP from ftp.cs.arizona.edu reports/1993/TR93-27.ps.Z.