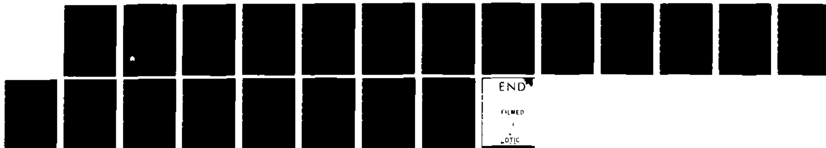
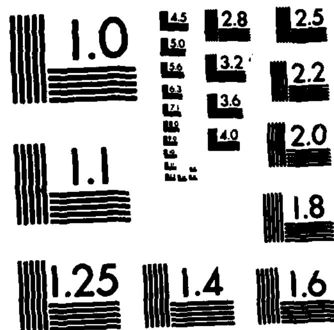
AD-A125 209 EFFICIENCY CONSIDERATIONS FOR C PROGRAMS ON A VAX (TRADEMARK) 11/780(U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE C J VAN WYK ET AL. 1/1

UNCLASSIFIED 04 AUG 82 CMU-CS-82-134 N00014-76-C-0370 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

CMU-CS-82-134

Efficiency Considerations for C Programs on a VAX 11/780

Christopher J. Van Wyk
Bell Laboratories
Murray Hill, New Jersey 07974

Jon L. Bentley
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Peter J. Weinberger
Bell Laboratories
Murray Hill, New Jersey 07974

DEPARTMENT
of
COMPUTER SCIENCE

DTIC FILE COPY

Carnegie-Mellon University

This document has been approved for public release and sale; its distribution is unlimited.

83 03 02 015

CMU-CS-82-134

Efficiency.Considerations for C Programs on a VAX[1] 11/780

Christopher J. Van Wyk
Bell Laboratories
Murray Hill, New Jersey 07974

Jon L. Bentley[2]
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Peter J. Weinberger
Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Typical C programs are "efficient enough" for most applic..ons. If they are not, judicicious improvement of the algorithms and data structures often can improve performance enough. Sometimes, even better performance is needed, and one must manipulate the C program in ways that depend on the language and the machine on which the application is running. In this paper we compare the efficiency of some C constructs and discuss some methods of improving the performance of C programs running on VAX 11/780's.

August 4, 1982

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distr
Avai
Dist
A

[1]VAX is a trademark of Digital Equipment Corporation.

[2]This research was supported in part by the Office of Naval Research Contract N00014-76-C-0370.

Efficiency Considerations for C Programs on a VAX* 11/780

Christopher J. Van Wyk

Bell Laboratories
Murray Hill, New Jersey 07974

Jon L. Bentley

Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Peter J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

C [3] was designed by Dennis Ritchie in the early 1970's as a programming language for the DEC PDP-11. Some of its features were designed with an eye to their efficient realization in compiled code, and a substantial amount of folklore has grown up about how to write fast C programs. Since the VAX 11/780 is becoming popular, we investigated timings for basic operations and the effects of a few program transformations on C program performance.

2. Caveat Lector

A few of the techniques we report may be useful generally, but most should not be applied unless the subject programs are "truly needy." The techniques make the code harder to understand, and may lock the programmer into a particular choice of data structures and algorithms too early in program development. Therefore, to achieve an efficient program, one should take the steps outlined below:

(1) Design the program robustly.

(2) Monitor its performance, and, if necessary, improve the data structures and algorithms in time-consuming parts of the code. (Execution timing [1] and statement counting [5] may be useful for this step.)

(3) Monitor its performance again, and improve the code in ways independent of the system and the language. [2]

(4) Monitor its performance yet again, and if (and only if!) the program still runs too slowly, apply these techniques to functions that use most of the time.

We cannot emphasize too strongly that steps (1) and (2) are much more important than steps (3) and (4). Nevertheless there will be times when one must resort to machine-dependent program transformations to improve performance.

We ran our tests on a VAX 11/780 with a floating-point accelerator. Each C program was compiled and optimized under Berkeley UNIX† version 4.1. Readers running on a different system should try our tests on their machines; our results may apply "more or less" to their systems, but one of the lessons we hope readers will draw from this paper is that intuition in matters of efficiency is often faulty.

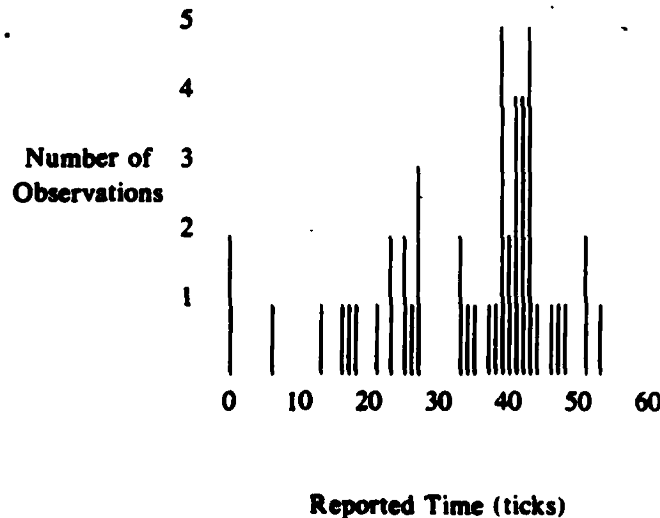
* VAX is a trademark of Digital Equipment Corporation.
† UNIX is a Trademark of Bell Laboratories.

Finally, those who want to try the more exotic of these suggestions should, at the least, isolate the time consuming portions of their programs and monitor them before and after changes, to be certain that they *have* improved performance. The simplest way to time a program `a.out` is to give the command `time a.out`; this will run the program and tell how much real time, user time, and system time it took. To find out in which functions the program is spending most of its time, one can compile the program with the `-p` option on the C compiler, run it, then give the command `prof`; this produces a table that tells how often each function was called and how much time was spent in it. Peter Weinberger's system [5] produces a listing showing how many times each statement is executed.

Readers who want or need to apply these techniques in earnest should become familiar with their machine's assembly language. The assembly version of a C program may be obtained by invoking the C compiler with the -S option. The VAX Architecture Handbook may also be useful.

3. Caveat Timer

It is hard to get repeatable timing measurements. First, the system function *times*(2) gives times in "ticks"—sixtieths of a second—so one must time a large number of iterations of simple statements to get any numbers at all. Second, the way *times*() works is to wake up every sixtieth of a second and charge whatever process is active at that moment with the whole sixtieth of a second. When there are other users on the machine, it is possible to be charged far too much or far too little because of this. For example, here are the times reported for fifty trials of a loop that finds the largest and smallest elements of an array of size 100,000.



It seems inappropriate to apply statistical analysis to such data. Instead, we waited for an opportunity to run the tests described later in this paper alone on a machine.

If you cannot get a machine to yourself, you can still get an idea of the relative efficiency of two programs. If the time reported for executing one program is consistently twice that reported for another, the first is probably faster. Alas, a difference of ten or twenty percent may not be significant, even on an unloaded machine.

For example, we timed 100,000 iterations of two statements ten times. Here are the reported times (in ticks):

```
i--    24  18  23  25  25  22  25  23  25  24
i-=1   20  23  22  22  22  21  22  23  22  22
```

This suggests it is about 10% better to use the second form than the first. Moreover, the

superiority of the assignment operator over the corresponding increment or decrement operator held for all types of `i`, which leads to a simple rule of thumb.

This seemed too wonderful to be true, so we looked at the assembler code for the two tests. It was *identical*. The difference arose because the second statement fell on an even eight-byte boundary in memory, while the first fell across an eight-byte boundary. Thus, the 100,000 iterations of the first statement required twice as many memory fetches as those of the second. So much for our rule of thumb.

On the other hand, timing revealed that declaring `char` and `short` variables `register` never made much difference. It turns out Berkeley UNIX ignores `register` declarations of `char` and `short` variables. Always read the documentation before you start experimenting.

4. Primitive Operations

We hold the numbers gathered from our tests suspect, so we have relegated them to the appendix. Here we present some principles gleaned from the data.

Keep in mind that these observations apply to VAX 11/780's running Berkeley UNIX. Some of them could become obsolete because of changes in the microcode, the operating system, or the C compiler.

1. It is always more expensive to operate on variables of type `char` or `short` than on variables of type `int`. Use of `char` and `short` variables should be limited to where the program's meaning requires it or where space is limited.

2. Because of C's conversion rules, operations on variables of type `float` are much more expensive than on those of type `double`. The observation above about trading off space for time holds here also. Note too that floating-point operations are much more expensive than integer operations.

3. Converting among the integer types is relatively inexpensive.

4. It is better to use assignment operators, as in `a += b`, than to write out the expression (as `a = a + b`). Bear in mind, however, that the right operand of an assignment operator is converted to the type of the left operand before doing the operation. Thus `i *= 0.5` for integer `i` will store zero in `i`, because 0.5 is converted to integer zero before doing the multiplication.

6. For arithmetic operations, addition and subtraction are cheapest, multiplication is somewhat more expensive, and division is very costly. For integers in registers, multiplication requires about 25% more time than addition or subtraction; one division takes about three times the time for one multiplication.

7. Function calls are about as expensive as five additions. But increasing the number of parameters to a function does not increase the cost of calling it appreciably.

8. Operations on integer variables are faster if they are in registers. A possible exception to this is the cost of calling a function that uses register variables; even here, the difference is not clear.

5. Some Simple Loop Optimizations

If a section of code accounts for a large percentage of the total run time of a program, chances are it includes loops. In this section, we investigate some simple transformations that may be applied to speed up the execution time of loops. The programs are not necessarily of intrinsic interest, but we hope to illustrate the kinds of transformations that can be useful.

We started with this code to find the sum of twice the elements in an array `x` of `ARRSIZE` (= 100,000) `ints`:

```
sum = 0;
for (i = 0; i < ARRSIZE; i++)
     sum += 2*x[i];
```

This loop executes in 810 microseconds. The code generated is the same whether we use a for-loop, a while-loop, or a do-while loop, and where we place the increment operation, so we didn't test such variants further.

We don't need to double each element: we can sum the array elements and double the result.

```
sum = 0;
for (i = 0; i < ARRSIZE; i++)
     sum += x[i];
sum *= 2;
```

This simple change cuts the run time down to 590 ms, saving 27% of the run time. This is an example of moving code out of loops.

Next we wrote the loop in terms of pointers rather than array accesses:

```
sum = 0;
hi = x + ARRSIZE;
for (p = x; p < hi; p ++)
     sum += *p;
sum *= 2;
```

This loop executes in 660 ms.

The increase may surprise readers who expect programs that use pointers to be faster than those that use arrays, but the explanation is almost as simple as the remedy. The for-statement on the array index is compiled into a very fast loop instruction*, which increments by one and tests whether to jump back to the loop start. The version with pointers is not compiled into such a fast instruction. We can improve matters, however, by declaring variable `sum` and the loop variables (`i` in the array version, `high` and `p` in the pointer version) to be register variables. The corresponding timings for the three programs above are 620 ms, 410 ms, and 320 ms. In other words, placing loop variables in registers is important to speeding execution, especially if the loop variables are pointers.

For our next example, we chose the problem of calculating the minimum and maximum elements of an array. We filled the array with random numbers, wrote two simple two-argument functions, `Max` and `Min`, and wrote the code this way:

```
big = small = x[0];
for (i = 1; i < ARRSIZE; i ++)
     big = Max(big,x[i]);
for (i = 1; i < ARRSIZE; i ++)
     small = Min(small,x[i]);
```

All of `i`, `big` and `small` are declared to be register variables, in light of our experience above. This code requires 5 seconds to execute. (Note: `ARRSIZE` is still 100,000.)

Since the timing table in the appendix suggests that it costs a lot to call a function that does only a little, we defined `Max` and `Min` as macros:

```
#define   Max(x,y)   ((x)>(y)?(x):(y))
#define   Min(x,y)   ((x)<(y)?(x):(y))
```

(Note that this is dangerous if the arguments to `Max` and `Min` are themselves operations with side effects, as they would be if they involved, say, increment or decrement operators.) This simple change reduced the cost of the loop to 1600 ms, saving a factor of more than three.

---

* the VAX `aoblss` instruction

Even this, however, does more work than necessary: big and small are always assigned a value, even if that value is their old value. So we rewrote the code with the evaluations of Max and Min in-line:

```
big = small = x[0];
for (i = 1; i < ARRSIZE; i ++)
     if (x[i] > big)
          big = x[i];
for (i = 1; i < ARRSIZE; i ++)
     if (x[i] < small)
          small = x[i];
```

This loop requires 1100 ms for execution.

Fusing the two loops is an obvious way to avoid incurring the overhead of the loop twice.

```
big = small = x[0];
for (i = 1; i < ARRSIZE; i ++) {
     if (x[i] > big)
          big = x[i];
     if (x[i] < small)
          small = x[i];
}
```

This loop takes only 840 ms: we've broken the one second barrier.

The loop body refers to x[i] two or three times. If we put the value of x[i] into a register variable t, then we can save a couple of array references at the cost of an assignment.

```
big = small = x[0];
for (i = 1; i < ARRSIZE; i ++) {
     t = x[i];
     if (t > big)
          big = t;
     if (t < small)
          small = t;
}
```

This loop requires 740 ms.

Having written the maximum and minimum calculations in line and fused the loops, we can write the algorithm so it tests pairs of elements against the current maximum and minimum values. (If ARRSIZE were odd we could just set big and small to the value of x[0] and start the loop with i set to one.)

```
if (x[0] < x[1]) {
     big = x[1];
     small = x[0];
} else {
     big = x[0];
     small = x[1];
}
for (i = 2; i < ARRSIZE; i++) {
     t = x[i];
     s = x[++i];
     if (s < t) {
          if (s < small)
               small = s;
          if (t > big)
               big = t;
     } else {
          if (t < small)
               small=t;
          if (s > big)
               big = s;
     }
}
```

The execution time of this loop is 590 ms.

Following a suggestion of John Reiser's [private communication], we rearranged the loop to count down to zero:

```
if (x[ARRSIZE-1] < x[ARRSIZE-2]) {
     big = x[ARRSIZE-2];
     small = x[ARRSIZE-1];
} else {
     big = x[ARRSIZE-1];
     small = x[ARRSIZE-2];
}
for (i = ARRSIZE-3; --i >= 0; ) {
     t = x[i];
     s = x[--i];
     if (s < t) {
          if (s < small)
               small = s;
          if (t > big)
               big = t;
     } else {
          if (t < small)
               small = t;
          if (s > big)
               big = s;
     }
}
```

This loop takes only 520 ms.*

Finally, we tried replacing the array references by pointer operations:

* It is compiled into the VAX sobgeq instruction instead of an aoblss.

```
if (x[0] < x[1]) {
     big = x[1];
     small = x[0];
} else {
     big = x[0];
     small = x[1];
}
high = x + ARRSIZE;
for (p = x+2; p < high; p ++) {
     t = *p;
     s = *++p;
     if (t > s) {
          if (t > big)
               big = t;
          if (s < small)
               small = s;
     } else {
          if (s > big)
               big = s;
          if (t < small)
               small = t;
     }
}
```

(The counting down technique mentioned above is special only when counting down to zero, so we didn't use it here.) The execution time of this loop is 460 ms.

Summary Table

Change	Time (ms)	Difference (ms)
Max and Min Functions	5000	
		-3400
Max and Min Macros	1600	
		-500
Max and Min In-line	1100	
		-260
Fusing two loops	840	
		-100
Replacing two array references by one assignment	740	
		-150
Avoiding unnecessary tests	590	
		-70
Counting down to zero	520	
		-60
Pointers	460	

In this section we examined mostly machine-independent program transformations—moving code out of loops, combining nearby loops, rewriting calls in line, and eliminating common subexpressions. These modifications saved a factor of more than 7 in the max/min example. Only after all these improvements did we perform the machine dependent modifications of reversing the direction of the loop and performing array operations using pointers.

6. Searching

In this section, we tried to speed up two kinds of searching in a sorted table. The table contained 5,000 elements, and the costs we report are for the execution of a loop that searched for each element once.

First we examined a sequential search routine.

```
seq1 (x, v, n)
register VECTYPE x;
VECTYPE v[];
register int n;
{
     register int i;
     for (i = 0; i < n && v[i] != x; i ++)
          ;
     if (i == n)
          return (-1);
     else
          return (i);
}
```

This routine took 55 seconds of user time and 190 ms of system time to find each element of the table once.

The first transformation we made would be valuable on unsorted tables as well: store the value being sought at the end of the table, making the bounds check on `i` unnecessary: (Note that array `v` should have room for `n+1` elements.)

```
seq2 (x, v, n)
register VECTYPE x;
VECTYPE v[];
register int n;
{
     register int i;
     v[n] = x;
     for (i = 0; v[i] != x; i ++)
          ;
     if (i == n)
          return (-1);
     else
          return (i);
}
```

This simple change cut the user time down to 50 seconds, and the system time down to 170 ms.

Finally, we created a pointer version of the sequential search routine.

```
seq3 (x, v, n)
register VECTYPE x;
VECTYPE v[];
register int n;
{
     register VECTYPE *cur, *high;
     high = v + n;
     *high = x;
     for (cur = v; *cur != x; cur ++)
          ;
     if (cur == high)
          return (-1);
     else
          return (((unsigned) cur - (unsigned) v)/(sizeof (VECTYPE)));
}
```

This version required only 36 seconds of user time and 120 ms of system time.

We turned our attention to binary search, starting with this recursive version:

```
bin1 (x, v, lo, hi)
register VECTYPE x;
VECTYPE v[];
register int lo, hi;
{
     register mid;
     if (lo > hi)
          return (-1);
     mid = (lo + hi)/2;
     if (x < v[mid])
          return (bin1 (x, v, lo, mid-1));
     else if (x > v[mid])
          return (bin1 (x, v, mid+1, hi));
     else
          return (mid);
}
```

The loop using this search, still finding each element once, executes in 2.7 seconds. The system time is once again negligible. Notice that this change of algorithm gave better than a factor of 12 improvement in running time over the best sequential search routine above. Often a change in the algorithm or data structure can do more for performance than many iterations of the machine-dependent techniques that are the subject of this paper.

We rewrote the recursive version as an iterative program:

```
bin2 (x, v, n)
VECTYPE   x, v[];
int n;
{
     register int low, high, mid;

     low = 0;
     high = n - 1;
     while (low <= high) {
          mid = (low + high)/2;
          if (x < v[mid])
               high = mid - 1;
          else if (x > v[mid])
               low = mid + 1;
          else return (mid);
     }
     return (-1);
}
```

This search requires only 1.1 seconds, so we saved almost 60% of the execution time.

Following a suggestion of Satish Desai [private communication], we replaced division by two by a right shift by one. This is valid because the index into the array cannot become negative.

```
bin3 (x, v, n)
VECTYPE   x, v[];
int n;
{
     register int low, high, mid;

     low = 0;
     high = n - 1;
     while (low <= high) {
          mid = (low + high)>>1;
          if (x < v[mid])
               high = mid - 1;
          else if (x > v[mid])
               low = mid + 1;
          else return (mid);
     }
     return (-1);
}
```

This change decreased the loop execution time to 700 ms, saving almost 40% of the run time. Notice that this implies that the division took almost half the time in the earlier loop.

Changing to pointers worked well in the past, so we tried it here:

```
bin4 (x, v, n)
register VECTYPE x, v[];
int n;
{
     register VECTYPE *low, *high, *mid;

     low = &v[0];
     high = &v[n-1];
     while (low <= high) {
          mid = (VECTYPE *) ((((unsigned) low + (unsigned) high) >> 1) & ((~0)^3)
          if (x < *mid)
               high = --mid;
          else if (x > *mid)
               low = ++mid;
          else return (((unsigned) mid - (unsigned) &v[0])/(sizeof (VECTYPE)));
     }
     return (-1);
}
```

The search loop now takes 870 ms, so we have made the execution time greater by changing array operations to pointer operations. Since this code is also harder to understand, it is not worth mak-

ing this change to one's search routine.

Summary Table

Sequential Search

Version	Usr Time (seconds)	Difference (seconds)	Sys Time (ms)	Difference (ms)
`seq1`	55		190	
		-5		-20
`seq2`	50		170	
		-14		-50
`seq3`	36		120	

Binary Search

Version	Time (ms)	Difference (ms)
`bin1`	2700	
		-1600
`bin2`	1100	
		-400
`bin3`	700	
		+170
`bin4`	870	

These measurements also tell us when it becomes worthwhile to consider using binary search on a sorted table. Say the average sequential search for an element requires looking at $c_1 n$ table elements, and the average binary search requires $c_2 \log_2 n$ table accesses. Using the fastest times above for sequential and binary search, we find that $c_1=7.3$ and $c_2=5.7$. The equation $c_1 n = c_2 \log_2 n$ is satisfied when n is around 40, so we should use sequential search unless the table will be larger than that.

7. Input/Output Operations

Kernighan and Plauger[4] have noted that most programs spend most of their time in input and output. Almost all programs should start out their lives using the standard package for input/output operations (see manual page *stdio*(3)). If the program is worth speeding up, and input/output proves to be the bottleneck, some of the techniques reviewed herein may help. We investigated two common operations—counting the lines in a file and copying a file—to see what kinds of improvements could be obtained by replacing calls that rely on *stdio.h* by system-dependent functions.

We used the file /usr/dict/words (more than 201,000 bytes on our system), arranged into lines of about eighty characters, as input to each version of line-counting function. The first version simply checks each character to determine whether it is a newline:

```
lncntl()
{
register FILE *infile;
char c;
int numlines;
     if (!(infile = fopen (FILENAME, "r")))
          exit (1);
     numlines = 0;
     while ((c = getc(infile)) != EOF) {
          if (c == '\n')
               numlines ++;
     }
     fclose (infile);
     return (numlines);
}
```

This function took an average of 590 ms of system time and 1980 ms of user time to count the lines in our modified version of /usr/dict/words.

Variations of this program that put **numlines** in a register, that used functions **fgets()** or **fread()**, or that placed sentinel newlines at the end of the buffer either took more time or made little difference in performance.

Al Aho [private communication] suggested that we use the system function **read()**. (See the manual page for *read*(2).)

```
lncnt2()
{
register int infile;
register int numlines, i, numchars;
     if (!(infile = open (FILENAME, 0)))
          exit(1);
     numlines = 0;
     while ((numchars = read (infile, buf, BUFSIZE)) > 0)
          for (i = 0; i < numchars; i ++) {
               if (buf[i] == '\n')
                    numlines ++;
          }
     close (infile);
     return (numlines);
}
```

This function takes 580 ms system time and only 990 ms user time.

Using a sentinel newline at the end of the buffer reduces this to 550 ms system time and 700 ms user time.

```
lncnt3()
{
register int infile;
register int numlines, i, numchars;
     if (!(infile = open (FILENAME, 0)))
          exit(1);
     numlines = 0;
     do {
          numchars = read (infile, buf, BUFSIZE);
          buf[numchars+1] = '\n';
          for (i = 0; i <= numchars; i ++) {
               for ( ; buf[i] != '\n'; i ++)
                    ;
               numlines ++;
          }
          -- numlines;
     } while (numchars == BUFSIZE);
     close (infile);
     return (numlines);
}
```

Having examined the relative merits of different means of input, we turned our attention to a program that copies its input to its output. Our first copy function used `getc()` and `putc()`:

```
copy1()
{
register FILE *infile;
char c;
     if (!(infile = fopen (FILENAME, "r")))
          exit (1);
     while ((c = getc(infile)) != EOF)
          putc(c,outflds);
     fclose (infile);
}
```

This function took 740 ms system time and 3 seconds user time to copy /usr/dict/words.

Writing this function using `fgets()` and `fputs` or `fread()` and `fwrite` made it take more time. Using the system functions `read()` and `write()` improved the performance to 660 ms system time and a negligible amount of user time:

```
copy2()
{
register int infile;
register numchars;
     if (!(infile = open (FILENAME, 0)))
          exit(1);
     while ((numchars = read (infile, buf, BUFSIZE)) > 0)
          write (outbfds, buf, numchars);
     close (infile);
}
```

We have seen that the functions provided by *stdio.h* are hard to improve on without resorting to system calls. Several of our "improvements" actually resulted in longer run time for programs. This illustrates well the importance of monitoring to be certain that changes represent improvements.

8. Observations

In Section 5 we squeezed a factor of two out of the summation loop, and improved the maximum and minimum finder by a factor of ten. The later changes—like counting down instead of up and switching to pointers—are obviously system-dependent. It is harder, though, to decide whether the earlier steps were changes in algorithm or system-independent changes. Was moving the multiplication by two out of the loop at the beginning of Section 5 a change in algorithm or a change in coding? What kind of modification was the elimination of an unnecessary test in the second problem?

Programs are not made faster in one step. In the maximum and minimum finder, the fused loop (fourth step) was six times better than the program we started with. For many applications it would be enough to improve the performance of a much-used function by a factor of six, especially because the code at this stage remains clear. But we know that if we needed to, we could get almost another factor of two improvement by making more complicated, less portable, changes.

A clearly algorithmic change was the switch from sequential to binary search in Section 6. The slowest binary search is almost fifteen times faster than the fastest sequential search. But the fastest binary search is another factor of four better. It involves a system-dependent bit operation, but it is much clearer than the pointer version (which turned out to be slower). Nevertheless, the fast sequential search algorithm could be useful for searching short sorted lists, as the discussion at the end of the section suggests.

With regard to Section 7 we note that the superior speed of `read()` and `write()` over the functions in `stdio.h` has not led us to rewrite all of our programs using them. The system calls are less convenient to use since they do not scan the input or format the output. But if we were writing a program where performance was critical and input/output was a bottleneck, we would certainly consider using them.

We end as we began, with a warning that language- and machine-specific changes to programs are not for universal, or even wide, application. *It is much more important that programs be clear so that they can be understood and modified.* But when program speed is an issue, there are many ways to find out where the time is being spent and how to reduce it.

9. Remarks

These tests are arranged so that others can take them and run them on their own machines. Please let us know if you would like a copy.

Thanks to Andy Koenig for letting us use `rabbit` alone for about two days before letting other users on.

Thanks also to Al Aho, Brian Kernighan, and Tom Szymanski for their comments on drafts of this paper.

10. References

1. *UNIX Programmer's Manual*, University of California, Berkeley (1981). Seventh edition. Virtual VAX—11 version. 4.1 BSD.
2. Jon Louis Bentley, *Writing Efficient Programs*, Prentice-Hall (1982).
3. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall (1978).
4. Brian W. Kernighan and P. J. Plauger, *Software Tools in Pascal*, Addison-Wesley (1981).
5. P. J. Weinberger, *Dynamic statement counting: a manual for users and owners*, Bell Laboratories (1982).

Appendix 1. Timings of Primitive Operations on a VAX 11/780

There are so many combinations of kinds of variables and operators that we wrote a program to generate this part of the tests. The workhorse of the tests is a macro:

```
repeat    10 times {
record times on usr and sys clocks
execute the statement of interest 100,000 times
record time elapsed since last look at clocks
}
```

Thus, we get ten measurements of how long it takes to do some primitive operation 100,000 times. The numbers we report below represent the average of these ten measurements. In every case, the system time was negligible—no more than two ticks—so we will say no more about it in this section.

Each column of this table is headed by a variable type, which may be one of six C types: char, short, int, register int (Rint), float, double. The entry in the left column describes the operation that was timed. Lower case letters in the left column denote different variables of the type named at the top of the corresponding column. Upper case letters denote variables of a particular type, usually `I` for integer.

Here are the timings in milliseconds for tests compiled and optimized under Berkeley UNIX version 4.1 and run on a VAX 11/780 with floating-point accelerator.

	char	short	int	Rint	float	double
a = -b	590	590	445	336	948	651
a = ~b	593	590	443	339		
I = !a	609	615	609	528	609	653
RI = !a	566	570	568	485	570	611
a ++	423	423	421	336	1013	718
++ a	421	421	421	338	990	716
a +=	378	381	379	338	949	720
a = a+1	546	545	421	338	951	716
a --	378	378	423	338	1011	718
-- a	419	423	381	336	993	718
a -=	423	423	379	338	973	716
a = a-1	548	548	441	336	949	716
a=b%c	1898	1898	1749	1496		
a=b>>c	843	845	653	486		
a=b<<c	779	778	593	506		
a=b&c	736	738	525	400		
a=b^c	654	651	506	336		
a=b|c	696	696	443	338		
C = a	428	463	466	421	759	949
S = a	439	423	464	421	770	948
I = a	441	443	421	379	654	841
RI = a	401	400	379	339	609	803
F = a	738	738	738	676	464	546
D = a	633	631	634	568	609	654
a+=b	464	466	463	339	504	718
a=a+b	696	695	464	333	911	718
a-=b	423	423	464	338	506	716
a=a-b	695	696	443	336	861	675
a*=b	803	800	508	420	506	886
a=a*b	821	824	504	485	906	886
a/=b	1605	1600	1328	1244	823	1368
a=a/b	1601	1605	1375	1266	1286	1416

	char	short	int	Rint	float	double
a%=b	1876	1880	1794	1500		
a=a%b	1899	1899	1728	1494		
a>>=b	843	843	611	529		
a=a>>b	906	906	615	486		
a<<=b	821	821	545	506		
a=a<<b	779	783	611	464		
a&=b	738	734	508	379		
a=a&b	740	740	563	376		
a^=b	464	463	423	336		
a=a^b	696	700	424	336		
a¦=b	421	421	421	339		
a=a¦b	695	695	463	338		
a=b?c:d	738	741	676	528	908	1098
if	588	593	590	483	633	858
a=b+c	696	696	486	338	908	738
a=b-c	651	656	443	335	991	784
a=b*c	823	823	548	485	928	928
a=b/c	1603	1624	1350	1265	1244	1368
I=a<b	695	696	696	525	781	970
I=a>b	615	611	613	529	696	886
I=a<=b	681	675	671	548	759	949
I=a>=b	675	675	675	548	759	970
I=a==b	729	716	695	546	781	968
I=a!=b	611	609	611	528	696	886
I=a&&b	759	759	758	678	759	845
I=a¦¦b	696	695	700	696	695	736
f(a)	2050	2048	2008	1983	2218	2276
f(a,b)	2130	2130	2133	2155	2488	2550
f(a,b,c)	2300	2300	2341	2239	2805	2933

In all cases `f()` is the empty function. One hundred thousand calls on the empty function with no parameters takes 1941 ms.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER CMU-CS-82-134	2. GOVT ACCESSION NO. A125209	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) EFFICIENCY CONSIDERATIONS FOR C PROGRAMS ON A VAX 11/780		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) JON L. BENTLEY CHRISTOPHER J. VAN WYK, Bell Labs Murray Hill, NJ PETER J. WEINBERGER, Bell Labs Murray Hill, NJ		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0370
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA. 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE August 4, 1982
		13. NUMBER OF PAGES 19
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Approved for public release; distribution unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)