CMU-CS-81-116

Writing Efficient Code[1]

Jon Louis Bentley
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

27 April 1981

Abstract -- The most important step in making a software system efficient is the proper selection of data structures and algorithms; many papers and textbooks have been devoted to these topics. Most discussions, however, neglect another important activity: that of writing machine-independent efficient code. This paper examines a set of techniques for accomplishing that step. We will examine those techniques both in an abstract setting and in their application to a real program, where they led to a speedup of a factor of over six. Because these techniques should be employed rarely, an important part of this paper is describing exactly when one should (and should not!) use them.

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

Table of Contents

close to optimal.

> **The Nearest Neighbor Heuristic:** Choose an arbitrary starting point, and then repeatedly visit the closest unvisited point to the current point until all points have been visited. When this is accomplished, close the tour by returning to the starting point.

Figure 1c shows the Nearest Neighbor tour of the point set in Figure 1a; the starting point of the tour is circled. For details on the performance of this heuristic, see Bentley and Saxe [1980].

In this section we will study a Pascal procedure that implements the heuristic. The initial version of the procedure has a running time of approximately $47.0N^2$ microseconds on a PDP-KL10; it therefore required approximately 47 seconds to construct the tour of a one-thousand city set. That program was used in two distinct applications. In the first it was run several dozen times per day on a PDP-KL10, and therefore consumed about half an hour of CPU time per day. In the other it was run only about a dozen times per day, but on a machine that was only about half as fast as a PDP-KL10; it also required about half an hour per day of CPU time. In both applications, it was *tremendously* desirable to decrease the times, so we are justified in expending energy in trying to do so.

As we try to reduce the run time, we must keep an important ground rule in mind.

> The purpose of this exercise is to make changes that will decrease the running time of the Pascal program on most systems; therefore we cannot make changes that exploit a particular feature of the compiler we happen to be using.

As we study the Pascal program we will not examine the machine code that the compiler generates; our only view of the program's speed will be by using the built-in RunTime function (for details on how the times were gathered see Appendix I). The Pascal compiler used for this experiment[3] does little optimization, so the computation that we see in the source code accurately reflects the machine code from which times were collected. To make sure that the timings are not merely an artifact of the particular compiler and hardware, the programs of this section were also implemented in a different language on a different computing system, and produced a similar set of relative timings (details on those timings can also be found in Appendix I).

We will now investigate a series of Pascal programs that implement the Nearest Neighbor Heuristic for the Traveling Salesman Problem. As we do so, the reader should attempt to improve each successive program before reading the next. (And if you find any more improvements, please communicate them to the author!)

---

[3]The compiler used was the Pascal compiler on the Carnegie-Mellon University Computer Science Department PDP-KL10 (Arpanet Host CMUA), which is a derivative of the Hamburg Pascal compiler.

2.1. A Sequence of Pascal Code Fragments

The first subroutine for implementing the nearest neighbor heuristic for the traveling salesman problem is shown in Fragment A1. It assumes several external definitions: the type PtPtr is an integer in the range 1..MaxPts, where MaxPts denotes the maximum possible number of points in the plane. The points themselves are stored in an array PtArr[PtPtr], whose elements are records with the two real components X and Y. The number of points currently stored in the array is stored in the integer variable NumPts; we shall often refer to NumPts as N, since it is the problem size. The subroutine Dist is passed two PtPtr's, and returns the Euclidean distance between the two points (the code for the function will be shown in Fragment A3).

The operation of procedure ApproxTSTour is straightforward. The only data structure it uses besides the array PtArr is an array Visited[PtPtr] of boolean; Visited[I] is true if and only if point I has already been visited in the tour. The routine's first action is to initialize every element of that array to false and then choose the first point to be visited as PtArr[NumPts] (the last point in the array). It then goes into a loop in which it selects the next NumPts – 1 points on the tour. To select each point it finds the closest point not yet visited, and then makes that point the current point. The writeln statements produce a description of the tour on the output file; they were not actually present in the version used for the timings and will not henceforth be shown in the program fragments. The program is simple; excluding lines that contain only begin, end or writeln statements, it contains only thirteen executable lines of code.

```
procedure ApproxTSTour;
    var I, J: PtPtr;
        Visited: array [PtPtr] of boolean;
        ThisPt, ClosePt: PtPtr;
        CloseDist: real;

    begin
    (* Initialize unvisited points *)
    for I := 1 to NumPts do
        Visited[I] := false;

    (* Choose start point as NumPts *)
    ThisPt := NumPts;
    Visited[NumPts] := true;
    writeln('First city is ', NumPts);

    (* Main loop of nearest neighbor heuristic *)
    for I := 2 to NumPts do
        begin
        (* Find nearest unvisited point to ThisPt*)
        CloseDist := maxreal;
        for J := 1 to NumPts do
            if not Visited[J] then
                if Dist(ThisPt, J) < CloseDist then
                    begin
                    CloseDist := Dist(ThisPt, J);
                    ClosePt := J
                    end;
        (* Report closest point *)
        writeln('Add edge from ', ThisPt, ' to ', ClosePt);
        Visited[ClosePt] := true;
        ThisPt := ClosePt
        end;
    write('Add edge from ', ThisPt, ' to ', NumPts)
    end;
```

Fragment A1. Original code.

The main for loop of the program is executed N – 1 times, and contains an inner loop that is itself executed N times; the total time required by the program will therefore be dominated by a term proportional to N^2. The Pascal running time of Fragment A1 was observed to be approximately $47.0N^2$ microseconds (details on measurements of the running time can be found in Appendix I.)

We will now modify the program to increase its speed. As we do so, we should concentrate on the inner loop (for J := 1 to NumPts do ...), because statements in that loop are executed $N^2 - N$ times, while all other statements are executed at most N times. The first thing that we might notice is that the real result of Dist(ThisPt,J) can be calculated twice for each distinct value of J in the inner loop. We will instead calculate it just once, store it, and then use that stored value twice. Since this can cut

down the number of distance calculations by a factor of two, we might expect it to cut the run time of the program almost in half. The modified code is shown in Fragment A2; it stores Dist(ThisPt,J) in the real variable ThisDist. All the lines that have been changed from Fragment A1 are marked with a vertical bar.

```
begin                                                              |
ThisDist := Dist(ThisPt, J);                                       |
if ThisDist < CloseDist then                                       |
    begin
    CloseDist := ThisDist;                                         |
    ClosePt := J
    end
end                                                                |
```

Fragment A2. Store ThisDist.

When I first made this change I was eagerly waiting to see a factor of almost two squeezed out of the runtime, and I was shocked to see it drop from $47.0N^2$ microseconds only to $45.6N^2$ microseconds! After I observed these times, though, it was easy to explain what had happened. The then clause of the inner if statement is executed very rarely, so in Fragment A1 the subroutine Dist was usually called only once per loop! Specifically, we can show analytically that the average number of times that the then branch is executed when there are M points left unvisited is

$$H_M = 1 + 1/2 + 1/3 + 1/4 + \ldots + 1/M,$$

which is called "the M-th harmonic number" and is approximately equal to the natural logarithm of M (H_{1000} is about 7.5). For a more detailed analysis of this fascinating combinatorial problem, see Section 1.2.10 of Knuth [1968]. Empirical observations that confirm this analysis can be found in Appendix I. This example illustrates a common experience in writing efficient code: optimizations that we expect to lead to a big time savings often make but a small difference. Even though this improvement did not yield a great time savings, it did identify an important part of the problem: computing the distances between pairs of points.

Is there any way we can improve the distance calculation procedure Dist shown at the top of Fragment A3? Unfortunately it appears that we cannot; that procedure expresses the mathematical definition of Euclidean distances very succinctly. We can, however, solve a different problem more efficiently: since all we ever do (in this procedure) is compare the relative magnitude of two distances, we do not need to take the square root of the sum of the squares before we return the result. That is, we can compare the squares of the distance to decide which point is closer; this relies on the monotonicity of the square root function. The resulting code is shown at the bottom of Fragment A3.

```
function Dist(I,J: PtPtr): real;
    begin
    Dist := sqrt(sqr(PtArr[I].X-PtArr[J].X) +
                 sqr(PtArr[I].Y-PtArr[J].Y))
    end;

function DistSqrd(I,J: PtPtr): real;
    begin
    DistSqrd := sqr(PtArr[I].X-PtArr[J].X) +
                 sqr(PtArr[I].Y-PtArr[J].Y)
    end;

...
ThisDist := DistSqrd(ThisPt, J);
...
```

Fragment A3. Remove square roots.

This improvement does indeed lead to a substantial time savings: while Fragment A2 required $45.6N^2$ microseconds, Fragment A3 requires only $24.2N^2$ microseconds. This difference is almost a factor of two. Since removing $\binom{N}{2} = N(N-1)/2$ square roots saved $21.4N^2$ microseconds, we can deduce that each square root required about 43 microseconds.

There is still one glaring deficiency in the organization of the program. Suppose we are solving the thousand-city problem and we have only ten unvisited cities; how do we find the closest city to ThisPt? We look at all one thousand cities, only to find for most of them that they have already been visited. It would be more efficient for us to keep track of the unvisited cities in a more direct way, so we could ignore the visited cities after having visited them. This is accomplished in Fragment A4, which is a complete rewrite[4] of Fragment A1, incorporating the changes of Fragments A2 and A3. The array UnVis contains integer pointers (that is, PtPtr's) to unvisited members of PtArr; specifically, the unvisited cities can always be found in UnVis[1..HighPt]. The overall structure of the routine is almost unchanged. The initialization is somewhat different, and the structure of the inner loop is very different: the for statement runs from 1 to the current number of points (NumPts), and no if test is required. When the closest point has been identified it is swapped with the point in UnVis[HighPt]; this maintains the invariant condition that all unvisited cities can be found in UnVis[1..HighPt].

[4]This is the most substantial change we will make to the Pascal program, and indeed the most substantial *kind* of change that falls under the heading of "writing efficient code". This change might be better classified as a selection of data structures.

```
procedure ApproxTSTour;
    var
        I: integer;
        UnVis: array [PtPtr] of PtPtr;
        ThisPt, HighPt, ClosePt, J: PtPtr;
        CloseDist, ThisDist: real;

    procedure SwapUnVis(I, J: PtPtr);
        var Temp: PtPtr;
        begin
        Temp := UnVis[I];
        UnVis[I] := UnVis[J];
        UnVis[J] := Temp
        end;

    begin
    (* Initialize unvisited points *)
    for I := 1 to NumPts do
        UnVis[I] := I;

    (* Choose start vertex as NumPts *)
    ThisPt := UnVis[NumPts];
    HighPt := NumPts-1;

    (* Main loop of nearest neighbor tour *)
    while HighPt > 0 do
        begin
        (* Find nearest unvisited point to ThisPt *)
        CloseDist := maxreal;
        for I := 1 to HighPt do
            begin
            ThisDist := DistSqrd(UnVis[I],ThisPt);
            if ThisDist < CloseDist then
                begin
                ClosePt := I;
                CloseDist := ThisDist
                end
            end;
        (* Report this point *)
        ThisPt := UnVis[ClosePt];
        SwapUnVis(ClosePt,HighPt);
        HighPt := HighPt-1
        end
    end;
```

Fragment A4. Convert boolean array to pointer array.

The result of this change is to decrease the running time from $24.2N^2$ microseconds to $21.2N^2$ microseconds; it cut the loop overhead in half and eliminated testing, but introduced a level of indirect addressing (through the array UnVis) that was not previously present.

From this point on we will concentrate on the inner for loop, which is responsible for almost all of the time. From our knowledge that the if test is rarely successful, we can deduce that most of the run time is spent in the subroutine DistSqrd. To reduce its time, we will rewrite its body in line, which eliminates the overhead of the procedure calls. We then observe that some invariant expressions are reevaluated each time through the loop (namely, the array indexing of PtArr[ThisPt]), so we instead assign those outside the loop to the real variables ThisX and ThisY. The resulting code is shown in Fragment A5; its running time is $14.0N^2$ microseconds (a reduction of $7.2N^2$ from Fragment A4).

```
(* Find nearest unvisited to ThisPt *)
ThisX := PtArr[ThisPt].X;
ThisY := PtArr[ThisPt].Y;
CloseDist := maxreal;
for I := 1 to HighPt do
    begin
    ThisDist := sqr(PtArr[UnVis[I]].X-ThisX)
                + sqr(PtArr[UnVis[I]].Y-ThisY);
    if ThisDist < CloseDist then
        begin
        ClosePt := I;
        CloseDist := ThisDist
        end
    end;
```

Fragment A5. Rewrite procedure in line and remove invariants.

We can now see precisely where the time of the program is spent. When M cities are unvisited, it calculates ThisDist M times, makes M comparisons with CloseDist, and then executes the then branch H_M times, on the average. Since the H_M term is too small to worry about (remember, it is 7.5 out of 1000) and all M comparisons seem necessary, we had better concentrate on calculating ThisDist. It contains two terms; is there some way we can reduce them to one? Such a reduction is shown in Fragment A6: we first compute the x-distance from the I-th point to ThisVert, and if that alone is greater than CloseDist, then we need not examine the y-distance. (Because the second term is positive, it can only increase ThisDist.)

```
ThisDist := sqr(PtArr[UnVis[I]].X-ThisX);
if ThisDist < CloseDist then
    begin
    ThisDist := ThisDist + sqr(PtArr[UnVis[I]].Y-ThisY);
    if ThisDist < CloseDist then
        begin
        ClosePt := I;
        CloseDist := ThisDist
        end
    end
```

Fragment A6. Delay computing y-distance.

Fragment A6 will be faster than Fragment A5 if the x-distance alone is usually sufficient to discard the point from consideration. A heuristic analysis suggests and experimental evidence confirms the conjecture that the number of times the y-distance must be considered is only about $2.25M^{1/2}$, where M is the current number of points; thus for 1000 points, only about 70 need have their y-values examined. (Details on the number of y-values examined can be found in Appendix I.) The empirically observed running time of Fragment A6 confirms the efficacy of the approach: it reduced the running time from the $14N^2$ microseconds of Fragment A5 to $8.2N^2$ microseconds.

Fragment A6 appears to be the best we can do with the current structure, so we are going to have to be really sneaky to squeeze out any more time. We know that most of the time is going to computing a difference and product of real numbers; is there any way to reduce that?[5] We can now use the fact that we know that integer arithmetic is faster than real arithmetic on many machines, and convert all of the arithmetic from real to integer. The reader should complain that the cost difference is there for a good reason: real arithmetic solves a different problem! Henceforth we can advertise this program as giving only an approximate version of the approximate nearest neighbor tour, but we can deduce from a larger context that the approximation will not be far off (we will not go into the details here).

The specific mechanism of Fragment A7 is to copy the points in PtArr (which we will assume have each coordinate between 0 and 1) to the array IntArr in which each coordinate is in 1..10000. We then perform all operations in this integer domain. Fragment A7 defines types SmallInt (for the Small Integer coordinates) and BigInt (for the sum of squares of differences of coordinates). The resulting program is shown in Fragment A7; its run time is $7.5N^2$ microseconds, a reduction of $.7N^2$ microseconds (or about ten percent).

---

[5] One way to reduce the cost is to replace the multiplication by taking an absolute value (we then have the x-distance itself rather than its square) and compare that to the distance to ClosePt (not the square of that distance). On the particular system used for this test, the cost of computing an absolute value is as much as the cost of a square, so we did not follow this path. It would have been beneficial on machines without fast multipliers.

```
procedure ApproxTSTour;
    type
        SmallInt = 0..10000;
        BigInt = 0..10000000000;
        IntPoint = record
            X, Y: SmallInt
            end;
    var
        I: integer;
        UnVis: array [PtPtr] of PtPtr;
        ThisPt, HighPt, ClosePt, J: PtPtr;
        CloseDist, ThisDist: BigInt;
        ThisX, ThisY: SmallInt;
        IntArr: array [PtPtr] of IntPoint;

    procedure SwapUnVis(I, J: PtPtr);
        *** Unchanged ***

    begin
    (* Build IntArr *)
    for I := 1 to NumPts do
        begin
        IntArr[I].X := round(PtArr[I].X * 10000);
        IntArr[I].Y := round(PtArr[I].Y * 10000)
        end;

    *** The remainder of the code is changed as follows.
    *** The builtin function "maxreal" is replaced by "10000000000".
    *** References to PtArr are replaced by IntArr.
    end;
```

Fragment A7. Convert reals to integers.

It is important to realize that the fact that real arithmetic happens to be slower than integer arithmetic is machine-dependent. The above change could actually slow the program's performance on some architectures, while on other machines (especially minis and micros) it could lead to a savings of one or two orders of magnitude. Regardless of that possible savings, though, Fragment A7 has opened to us another opportunity for time savings: we can now remove the level of indirection imposed by the UnVis array. Since we copied over our new version of the points into IntArr, we can now permute those to keep all the unvisited cities in IntArr[1..HighPt], and do away entirely with the array UnVis. (Note that we did not before have the freedom to alter the values in PtArr.) The resulting code is shown in Fragment A8; its running time is $6.9N^2$ microseconds, which is another improvement of about ten percent.

```
(* Find nearest unvisited to ThisPt *)
ThisX := IntArr[ThisPt].X;
ThisY := IntArr[ThisPt].Y;
CloseDist := 10000000000;
for I := 1 to HighPt do
    begin
    ThisDist := sqr(IntArr[I].X-ThisX);
    if ThisDist < CloseDist then
        begin
        ThisDist := ThisDist + sqr(IntArr[I].Y-ThisY);
        if ThisDist < CloseDist then
            begin
            ClosePt := I;
            CloseDist := ThisDist
            end
        end
    end;
```

Fragment A8. Remove UnVisited array.

We now have a fast program. All that usually happens in each iteration of the inner loop is an array access, a subtraction, a multiplication, and a comparison, all of which seem necessary. The only overhead that does not perform a useful service is that of the for statement itself, which we will now try to eliminate.[6] There are two aspects of the for statement: it increments I and it tests to see whether I equals the termination value. Since it seems hard to avoid the cost of incrementing I (although we will see later that it can be done!), we will try to make the second function faster by finding a better way to test for termination of the loop. The approach taken in Fragment A9 exploits the fact that ThisPt is stored in position HighPt + 1. Because the distance from ThisPt to itself is always zero, it would always be assigned as its own closest point, so we can put the test for loop termination into that part of the code that is executed only H_M times on the average. The one other change in the program is that we must be careful to ensure that points of distance zero from ThisPt are indeed assigned as ClosePt; that involves changing a ">=" to a ">" in two places. The performance of Fragment A9 is $6.8N^2$ microseconds, a savings of $.1N^2$ microseconds over the time of Fragment A8, or less than a two percent reduction.

---

[6] W. A. Wulf [1981] has pointed out that if our goal were to make the program compile to extremely fast object code under most slightly-optimizing compilers, then we should follow a different path at this point. Namely, we could decrease the loop time by counting down to zero (instead of up to HighPt) and by testing at the end of the loop (using a repeat...until statement); both of these transforms result in source code that is usually compiled much more efficiently. The fact that IntArr is a vector of records can complicate both indexing through the array and the cost of accessing; we could reduce those costs by changing IntArr to XArr and YArr.

```
        I := 0;
StartLoop:
            I := I+1;
            ThisDist := sqr(IntArr[I].X-ThisX);
            if ThisDist > CloseDist then goto StartLoop;
            ThisDist := ThisDist + sqr(IntArr[I].Y-ThisY);
            if ThisDist > CloseDist then goto StartLoop;
            if I >= HighPt then goto EndLoop;
            ClosePt := I;
            CloseDist := ThisDist;
            goto StartLoop;
EndLoop:
```

Fragment A9. Put loop control inside inner test.

There are two important facts to note about Fragment A9. The first is that the program does specify less computation than Fragment A8 -- it does much less loop control. The second fact is that many compilers would produce substantially faster code from Fragment A8 than Fragment A9 -- they "know about" for loops and can compile them quite efficiently.

2.2. An Assembly Program

We saw in the introduction to this section that the nearest neighbor heuristic for approximate traveling salesman tours is important in many applications. Because of this, it was worthwhile to improve the run time of a program implementing the heuristic; we have concentrated so far in this section on doing so by reorganizing the computation in the Pascal language. This is often "good enough", but in certain applications we might need a program that is faster yet. When this is the case, we have at least one further hope: we can recode the algorithm in assembly code to utilize the full potential of the underlying machine architecture. Fragment A10 shows how the code of Fragment A9 can be translated into (a slightly augmented version of) IBM System/360-370 Assembler language.

```
Registers: CloseDist, ThisDist, I, ThisX, YDist

           L     ThisX,MemThisX
           L     CloseDist,PosInf
           LA    I,Array-8
StartLoop  LA    I,8(I)                increment I by one record
           L     ThisDist,0(I)         ThisDist := (X[I]-ThisX)^2
           SR    ThisDist,ThisX          *
           MR    ThisDist,ThisDist       *
           CR    ThisDist,CloseDist    if ThisDist > CloseDist
           BH    StartLoop               then goto StartLoop
           L     YDist,MemThisY        YDist := (Y[I]-ThisY)^2
           S     YDist,4(I)              *
           MR    YDist,YDist             *
           AR    ThisDist,YDist        add YDist to XDist, giving total
           CR    ThisDist,CloseDist    if ThisDist > CloseDist
           BH    StartLoop               then goto StartLoop
           C     I,EndPtAdd            if I>=EndPtAdd
           BNL   EndLoop                 then goto EndLoop
           ST    I,ClosePtAdd          ClosePt := I
           LR    CloseDist,ThisDist    CloseDist := ThisDist
           B     StartLoop             goto StartLoop
EndLoop    EQU   *
```

Fragment A10. Rewrite to assembly code.

This program is almost an exact transliteration of the inner loop of Fragment A9. It assumes that the array of points is stored as N consecutive (x,y) pairs of fullwords (that is, 32-bit words aligned on a four-byte boundary). The variables ThisX and ThisY from Fragment A9 are assumed to be originally present in memory location MemThisX and MemThisY. All of the arithmetic in the program is carried out as 32-bit integers, but that could easily be changed to real numbers.

The primary activity of the main loop of the program is easy to trace: it is centered entirely in the six lines of code starting at the line labeled "StartLoop". At that line the register variable I is incremented by eight bytes to point to the next point to be tested. The x-value of that point is loaded into the register variable ThisDist in the next line of code, and the two lines after that subtract ThisX from the x-value and square the difference. The fifth line then compares the squared difference to CloseDist, and the sixth line does a conditional branch that is almost always taken (that is, all but about $2.25M^{1/2}$ times when M points are left).

A simple experiment was conducted to compare the speed of the assembly code of Fragment A10 with the speed of the code a typical compiler produces from Fragment A1. Fragment A1 was

compiled on an IBM System/370 under the Pascal/VS compiler[7], and the compiled code and the code of Fragment A10 were both assigned time costs according to the methodology described by Knuth [1971] (in which one time unit corresponds roughly to seven-tenths of a microsecond on an IBM System/360 Model 67). Fragment A10 had a dominant term of $6.5N^2$ time units, while the compiled code had a dominant term of $110.833N^2$; Fragment A10 is over 17 times faster than the compiled code.[8] The Pascal compiler used in this experiment seemed to achieve roughly the same level of optimization as the PDP-10 Pascal compiler used in the previous experiments.[9]

2.3. What Have We Learned?

We have devoted a great deal of effort in this section to a relatively small piece of code. Before going on to study the general principles underlying this example, we should pause for a moment to review what we have learned in the exercise.

The first thing that we saw was that this fragment was located in the bottleneck of a system and that it was indeed worthwhile to improve its running time. It is important that the techniques of this section be applied only to a bottleneck in an inefficient system.

We then studied the computation as embodied in a series of Pascal procedures. We started with a simple and correct program and performed a set of *transformations* that

- preserve the *correctness* of the program,
- usually increase the *length* and decrease the *readability* of the program text, and
- decrease the *run time* of the program.

---

[7] That compiler is IBM Program Number 5796-PNQ (an Installed User Program) and is described in the "Pascal/VS Programmer's Guide" (IBM Publication Number SH20-6162-0). The program was compiled under Pascal/VS Release 1.0 on an IBM System/370 at the University of Texas at Austin on 10 April 1981.

[8] The dominant term of $110.833N^2$ time units can be apportioned among the various activities in the innermost loop of Fragment A1 as follows. Loop cost of "for J := 1 to NumPts": $10N^2$; testing "if not Visited[J]": $6.5N^2$; cost of square root in procedure Dist: $42.5N^2$; other cost of procedure Dist: $42.833N^2$; comparing result of Dist to CloseDist: $9.0N^2$.

[9] The point of this subsection is that after using the techniques of writing machine-independent efficient code (which gave a speedup factor of over seven), careful hand-translation into assembly code can make the resulting program even faster (by over a factor of two). If we are willing to use extremely clever techniques to exploit the full potential of the underlying architecture (and recall that that topic is explicitly beyond the scope of this paper), then we can achieve even greater speedups. Fragment A10 requires $6.5N^2$ time units under Knuth's model. We can use the IBM System/360-370 LPR instruction to replace the MR instruction (of cost 6) with a unit-cost absolute value operation; this reduces the program run time to $4.5N^2$ units. A different strategy checks for the x-value being in a currently valid range by explicitly comparing the x-value to lower and upper bounds. Implementing the strategy by the instruction sequence LA, C, BH, C, BL has average time of $3.25N^2$ time units. Steele [1981] has found a way to implement bounds checking with instructions intended to implement loops in $2.75N^2$ time units (the sequence was LA, L, BXH, BXLE). Loop unrolling can be used to remove most of the cost of the LA instruction, giving an average run time of $2.25N^2$ units. These techniques are incredibly machine dependent and result in extremely messy code, but they do yield a program over 2.88 times faster than Fragment A10 and over 49 times faster than the code compiled from Fragment A1.

The transformations and the transformed programs are summarized in Table 1. One can easily see that the two major time improvements in the program were A2 -> A3 (removing square roots) and A4 -> A6 (computing distances in line with delayed calculation of y-distance). The transformations leading from Fragment A6 to Fragment A9 were not as "clean" as the previous transformations, and had much less impact on the running time (the time of Fragment A9 is about twenty percent less than the time of Fragment A6).

Fragment Modification	Time/N^2 Time	Change
A1.	47.0	
Store ThisDist		1.4
A2.	45.6	
Remove square roots		21.4
A3.	24.2	
Convert boolean array to pointer array		3.0
A4.	21.2	
Rewrite procedure inline and remove invariants		7.2
A5.	14.0	
Delay computing y-distance		5.8
A6.	8.2	
Convert reals to integers		0.7
A7.	7.5	
Remove unvisited array		0.6
A8.	6.9	
Put loop control inside inner test		0.1
A9.	6.8	

Table 1. Summary of program improvements.

Although the final three transformations did not greatly decrease the Pascal running time, they did pave the way for an elegant and efficient assembly program. Transform A6 -> A7 (converting reals to integers) increased the storage of the program and made only a slight difference in time; it was included primarily as a didactic tool to avoid the intricacies of real arithmetic in the assembly code. It did open the way for A7 -> A8 (removing the UnVisited array), which reduced the overhead in the resulting assembly code. Transform A8 -> A9 (loop control inside the inner test) led to an efficient inner loop in the assembly code. The part of the inner loop that is usually executed contains only six lines of assembly code. Calculations showed that for large values of N, the assembly code of Fragment A10 is over seventeen times faster than the code a typical compiler produced from the Pascal Fragment A1. This efficient program was easy to code from Fragment A9; it would be extremely difficult to achieve an assembly program with comparable efficiency by coding directly from Fragment A1.

1. Introduction

Research on the issue of efficiency in software systems has come from two primary directions. On the "low end", work has focused on compilers that produce code that is as good as that produced by experienced assembly coders (see, for instance, Wulf *et al* [1975]). On the "high end", researchers have examined the problems of designing efficient algorithms and data structures. The implicit understanding in both camps has been that the two endeavors together cover the complete range of activity needed to produce efficient programs.[2]

The thesis of this paper is that the above understanding is false. In particular, I propose that there is an intermediate activity between those two extremes that is necessary in the design of an efficient program. This activity, which I call "writing efficient code", takes as input a high-level language program that incorporates efficient algorithms and data structures, and produces as output a program in the same high-level language that is suitable for compilation into extremely efficient machine code. The operations undertaken at this level are too complex for most current and foreseeable compilers, yet are at so low a level as to be "beneath" most work on algorithms and data structures.

Practitioners have long worked at the level of writing efficient code, yet there is little written about the subject in most discussions of efficiency in software systems. The purpose of this paper is to describe this activity to software engineers interested in efficiency. In Section 2 we will study the activity in its application to a subroutine that arose in a real system; by manipulating the code we can decrease the program's run time by a factor of more than six. In Section 3 we will take a more systematic view of the field, describing both a set of rules for making code more efficient and a methodology for applying those rules. Sections 2 and 3 provide two orthogonal views of the same subject; the reader may read them in either order to suit his taste. Section 4 contains a brief survey of other work, and conclusions are offered in Section 5.

This paper assumes that the reader has an extensive background in reading and writing code in a high-level computer language. The tools that we will examine are like the surgeon's scalpel: they are very useful when applied in the right circumstances but disastrous if applied inappropriately. Their proper application must therefore be grounded in much programming experience. A background in

---

[2] It is interesting to note that representatives from the two extremes have upon occasion attached different weights to the relative importance of the two endeavors. Baase [1978, p. 27] writes in her algorithms text that "since the total execution time is of the same order of magnitude as the number of basic operations done, we are justified in counting basic operations and ignoring bookkeeping and implementation details". Wulf *et al* [1975, p. 124] observe in their description of an optimizing compiler that "all the fancy optimization in the world is not nearly as important as careful and thorough exploitation of the target machine".

algorithms and data structures, assembly language programming, and techniques for analyzing the correctness and efficiency of programs would be useful in reading parts of this paper, but it is not necessary. In particular, the paper never assumes much background along these lines for more than a few paragraphs at a time, so the reader not versed in these areas will not suffer a great deal.

2. An Example

In this section we will study the (in)famous *Traveling Salesman Problem*: our input is a set of N points in the plane (which we often think of as cities), and we must produce as output a minimal-length tour of the points. A tour of the cities is defined to be a list of the cities in which each city appears exactly once; the length of the tour is the sum of the distance from the first city to the second plus the distance from the second to third and so on, finally ending with the distance from the N^{th} city to the first. A set of points is shown in Figure 1a, and their traveling salesman tour is shown in Figure 1b. This problem arose in the context of scheduling a mechanical plotter to draw marks representing approximately 1100 points: we would like to plot the points in an order that does not waste much time moving from one mark to the next.

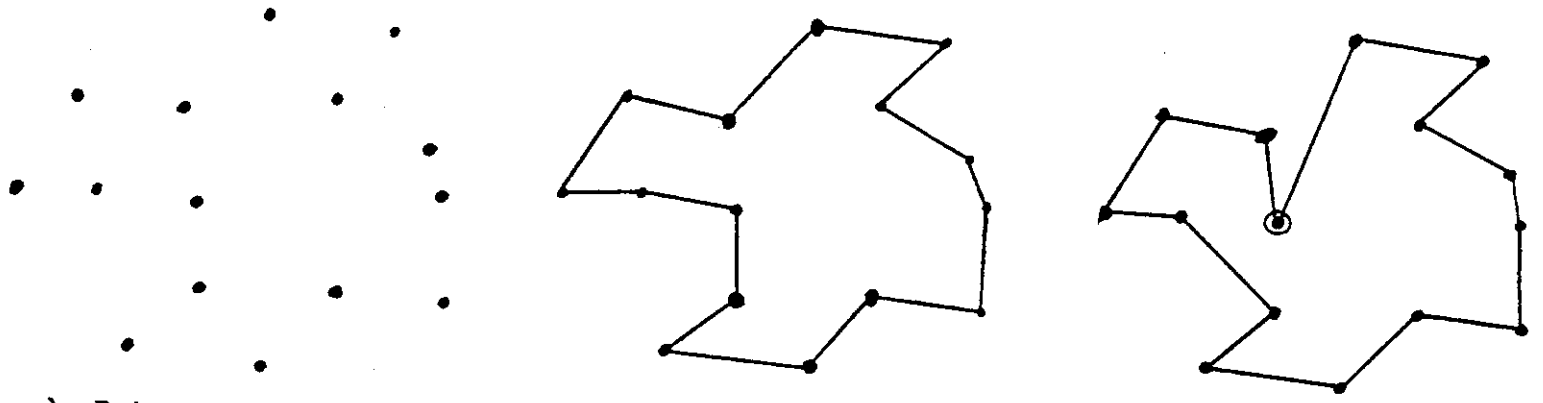

Figure 1. A point set and two tours.

The problem of finding a travelling salesman tour of a set of points with absolutely minimal total length has been studied for almost a hundred years and is still an open problem; many think that some day we will be able to prove that one cannot efficiently find an optimal tour. (See, for example, Lewis and Papadimitriou [1978].) We will therefore be concerned with finding a relatively good, if not absolutely optimal, tour; this is ideal in the above application, and appropriate in many other applications. The following heuristic algorithm is known to give tours whose lengths are usually quite

It is important to keep in mind the purpose of studying the assembly code. It was not to illustrate any sophisticated assembly coding techniques; Fragment A10 is a straightforward translation of Fragment A9. Rather, the reason was to show the interaction of machine-independent and machine-dependent coding techniques: even if our original goal had been to implement the assembly program, we would have been wise to perform the high-level changes before considering low-level issues.

The above discussion has been in a rather abstract context; we will now discuss the problem in the two concrete applications mentioned at the start of this section. In the first application we had to write a Pascal program that was executed on thousand-city problems several dozen times per day over a period of a few weeks; we used the program of Fragment A6 (the nature of some of the inputs would not allow the conversion from reals to integers). The changes reduced the run time from approximately half an hour per day to less than five minutes per day. In the second application we had to develop a Fortran program that would be executed on thousand-city problems around a dozen times per day over approximately a one-year period. Because the Pascal if statements had to be implemented as gotos in the available Fortran, the basic structure of Fragment A9 was used; again, the reals were not converted to integers. In both applications the original clean code was left in the program, along with documentation showing how the dirty program was derived from the clean program.

To summarize this section, our work on the Nearest Neighbor Heuristic has shown us the following.

- An increase of over a factor of six in the speed of a particular Pascal program.
- An efficient assembly language implementation of the Pascal program that is seventeen times faster than the object code produced by a typical compiler.
- A methodology for increasing a program's speed while preserving its correctness: transformations at the source program level.

3. A Set of Tools

In Section 2 we saw one particular example of writing efficient code in great and gory detail. In this section we will take a different view, and study a number of general principles. In Subsection 3.1 we will consider exactly when one should apply these tools, when they are best left alone, and when other tools are more appropriate for the job at hand. With this as background, we discuss some techniques for modifying data structures in Subsection 3.2 and techniques for modifying code in Subsection 3.3. Subsection 3.4 then provides a retrospective view of the techniques.

- As a very simple last resort, it is usually easy ... access a global system clock. To measure the time taken by a routine one merely stores

3.1. When To Bother

The efficiency of a program is secondary when compared to the program's correctness: it is nice if a program is fast, but it is essential that it does what it claims to. For this reason among many others, efficiency should usually be a minor concern during the design and development of a program. Rather, the primary concerns of the programmer during the early part of a program's life should be the overall organization of the programming project (facing the issues described by Brooks [1975]) and producing disciplined and correct code (using the techniques of Kernighan and Plauger [1978], for instance). Unless one knows in advance with certainty that efficiency will be a major concern, efficiency should be almost ignored at this stage in the design[10]; Kernighan and Plauger [1978, Chapter 7] present a number of programs that underscore the point that "premature optimization is the root of all evil". Indeed, statistics given by Kernighan and Plauger [1976] show that most of the programs in their book spend the vast majority of their time (upwards of seventy percent) performing input and output; any optimization of their computing time would have little impact on the overall efficiency of those programs. Deliberately ignoring efficiency early in the program's life should greatly increase the chances of achieving on schedule a correct and easily maintained program. Furthermore, in many contexts, that clean program is often "efficient enough" for the particular application.

Suppose, though, that you proceed as above and rapidly produce a correct and readable program that takes an enormous amount of time -- what do you do then? The obvious next step is to modify the parts of the program you "know in your heart" are consuming all the time. The only problem with this approach is that programmers are notoriously bad at guessing what parts of the program are the real resource hogs! The correct next step, therefore, is to instrument the program to gather data on what parts of the program take all the time, and then focus one's attention on those parts. There are a number of different ways by which statistics on program usage can be gathered, including the following

Once the programmer has identified the resource hogs in his program, it is time for him to bring to bear his two primary programming tools in the never-ending battle against slow programs.[11] The first tool is the field of *data structures*. Brooks [1975] has eloquently stated the importance of data structures: "representation is the essence of programming". By reorganizing the representation of data, one can often drastically reduce the time required to operate on it. We will return to the issue of data structures in Subsection 3.2. The second tool is the field of *algorithm design*. By changing the underlying technique used to solve a problem, one can often achieve tremendous savings in time. For instance, changing a sequential search subroutine to binary search will reduce the number of probes required to search a sorted N-element table from about N/2 to just $\log_2 N$; for N = 1000, this is a reduction from 500 to 10. Bentley [1979] provides a survey of how proper algorithm design can lead to similar savings for many problems.

the code, it was spending almost seventy percent of its time in the system's memory allocator!

Further investigation revealed that most of this was used in allocating one particular kind of node (more than 68,000 times, with the next most popular node being allocated around 2,000 times). I added the minimal bookkeeping necessary to keep my own queue of free nodes of this particular kind, and reduced the memory allocator's share of the time to about thirty percent.

There were three benefits of this change:

1. less time in the allocator (it's a circular list with a roving pointer),
2. less memory fragmentation (our allocator doesn't compact), and
3. now that the statistics aren't overwhelmed by the allocator's share, I can find places that need to be sped up with sophisticated algorithms or data structures.

On the other hand, it would not be worth my time to provide my own bookkeeping for every kind of node I allocate, so I save programming effort on another front.

To make a long story short, by spending a few hours monitoring his program and then making a small change, Van Wyk was able to reduce the program's run time to about forty-five percent of what it was previously. (We will see in the next section that Van Wyk's modification can be viewed as an instance of Space-For-Time Rule 3 and Procedure Rule 2.)

3.2. Modifying Data Structures

In this section we will study techniques that increase the efficiency of a program by slightly modifying the program's data structures. This topic is included for completeness, but will not be emphasized as much as the section on modifying code because these techniques are often taught in courses on data structures.

The best changes to make to a data structure are, of course, those that reduce both the program's time and space. A host of data structure texts (see, for example, Knuth [1968, 1973] and Standish [1980]) describe many sophisticated structures that can replace their simpler cousins and thereby reduce both the time and space requirements of a program. Throughout the remainder of this section, though, we will take a much more microscopic view of data structures, and try to find the best way to implement a particular structure once it has been chosen to represent our data. At this low level there are few changes that reduce both program time and space; most changes trade one resource for the other.

We will start our study by investigating four techniques that decrease a program's time requirements by increasing the amount of space the program uses. In general, they trade space for time by storing redundant information. The first such rule is the following.

Space-For-Time Rule 1 -- Data Structure Augmentation: The time required for

> common operations on data can often be reduced by augmenting the structure with extra information or by changing the information within the structure so that it can be accessed more easily.

The example of Section 2 provides two instances of such changes. In changing Fragment A3 to A4 we reduced the time spent in scanning bits by increasing the storage required to one pointer per word (rather than one bit). In removing the indirect addressing of Fragment A7 we duplicated an array in Fragment A8. Such changes are common when dealing, for instance, with linked lists. If we know that there are going to be many changes near the end of the list, then we can augment the structure with an explicit pointer to the end of the list. Likewise, if we know that we will often access the predecessor of an element, then instead of searching for the predecessor from the front, we can use a doubly linked list and access the predecessor immediately. In Loop Rules 2 and 3 we will see a kind of augmentation that involves adding a "sentinel node" to a data structure.

The next tradeoff is perhaps the one used most commonly with the greatest resulting savings in time.

> **Space-For-Time Rule 2 -- Store Precomputed Results:** The cost of recomputing an expensive function many times can often be reduced by computing the function only once and storing the results in a table. Subsequent requests for the function are then handled by table lookup rather than by computing the function.

We already saw a very simple application of this rule as we changed Fragment A1 to A2 by computing the distance between points only once and storing it in the variable ThisDist (which can be viewed as a one-element table). A more interesting application occurs if we have a program that repeatedly computes Fibonacci numbers, as happens, for instance, in Fibonaccian search (see Knuth [1973, Section 6.2.1]). Recall that the mathematical definition of the Fibonacci numbers is given recursively as

> Fib(1) = 1, Fib(2) = 1, and
> Fib(N) = Fib(N-1) + Fib(N-2) for N > 2;

the first eight Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, 21. It is quite easy to translate the above recursive definition into the Pascal-like subroutine of Fragment B1.

```
function Fib(N: integer): integer;
    var A, B, C, I: integer;
    begin
    if N<1 or N>Max then return 0;
    if N<=2 then return 1;
    A := 1;  B := 1;
    for I := 3 to N do
        begin
        C := A + B;
        A := B;
        B := C
        end;
    return C
    end;
```

Fragment B1. Fibonacci numbers.

Note that subroutine Fib returns 0 if its input parameter N is less than 1 or greater than the upper limit Max. If this subroutine were in the time bottleneck of a program, then we could replace it by a table defined as

```
var Fib:  array [1..Max] of integer;
```

and replace each call of Fib(N) by the simple access to Fib[N], assuming that we have ensured that N is in 1..Max. (We will return to this approach when we study Space-For-Time Rule 4.) Bird [1980] discusses storing precomputed results in the general context of recursive programs, and Lisp programmers will recognize this technique in MEMO functions.

The next rule is an extension of the previous rule that has many applications throughout computer science.

> **Space-For-Time Rule 3 -- Caching:** Data that is accessed most often should be the easiest to access.

This rule is used in computer architecture, for instance, by having a cache in the memory system that stores words that have been recently accessed. When a request arrives for a particular word, the memory system first checks to see if the desired word is in the cache, in which case it can be returned immediately, without the need for the costly address mapping and access to main memory. The same idea is used in searching sequential lists by moving each item as it is found closer to the front of the list; items that are retrieved often tend to be near the front of the list, so they are found quickly (see Knuth [1973, Section 6.1]). Jalics [1977, p. 140] describes an application in which merely remembering the last item retrieved from a table was sufficient to answer 99% of the queries and reduced the time in those cases from 2004 instructions to 4 instructions. Van Wyk's storage allocator described in Section 3.1 is another nice example of caching: he uses a special scheme for the most commonly used kind of node and a general scheme for the rest.

As a more sophisticated application of caching, consider the problem of implementing a computerized dictionary so a program can ensure that every word in an English manuscript file is indeed in the dictionary. The huge size of the language (some dictionaries contain half a million words) dictates that most words must be stored in a secondary memory, but caching (for instance) the one thousand most frequently used English words in main memory would make accesses to the secondary store rare for many documents. A strategy similar to this was used by Peterson [1980] in a spelling correction program.

The next rule is often used in conjunction with Space-For-Time Rule 2 (Storing Precomputed Results).

> **Space-For-Time Rule 4 -- Lazy Evaluation:** The strategy of never evaluating an item until it is needed avoids evaluations of unnecessary items.

Note that this rule counters the well-known proverb by advising the programmer "never do today what you can put off till tomorrow". For a simple example of lazy evaluation, we will return to the problem of computing Fibonacci numbers that we studied under Space-For-Time Rule 2. We saw there that a subroutine that computes a Fibonacci number can be replaced by a program that first computes all possible desired numbers and then stores then all in a table. This technique, though, does much unneeded work if we never access more than the first few Fibonacci numbers. Fragment B2 reduces that cost of initially evaluating all possible numbers by evaluating each Fibonacci number once and only once, *as it is needed*.

```
function Fib(N: integer): integer;
    var I: integer;
    static TopGood: integer (initially 2);
           GoodFibs: array [1..Max] of integer (initially [1,1]);
    begin
    if N<1 or N>Max then return 0;
    if N>TopGood then
        begin
        for I := TopGood+1 to N do
            GoodFibs[I] := GoodFibs[I-1] + GoodFibs[I-2];
        TopGood := N
        end;
    return GoodFibs[N]
    end;
```

Fragment B2. Lazy evaluation of Fibonacci numbers.

In the above fragment the static variables TopGood and GoodFibs have the invariant relation that GoodFibs[1..TopGood] always contains the first TopGood Fibonacci numbers.

To illustrate a more subtle application of lazy evaluation, we will consider two examples. The first was supplied by Al Aho [1980], who had constructed a very efficient table-driven program to locate a given pattern in a long string. The size of the table was a fast growing function of the length of

pattern, but after the table had been built, the string could be processed very quickly. Unfortunately, his program spent approximately thirty seconds merely building the table. When he replaced that by a lazy evaluation of the table (that is, his program evaluated each table element as it was needed), the run time of the entire program was less than half a second.

A second application of lazy evaluation was supplied by Brian Kernighan [1981]. When he monitored the TROFF program (a Bell Laboratories document formatter), he found that approximately twenty percent of the run time of the program was devoted to calculating the width of the current line after each input character, and also observed that the width was rarely accessed. He therefore changed the program to store the current line, and calculated the width from that representation on the few occasions that the width was needed. This change was quite easy to incorporate into the program, and reduced its run time by twenty percent.

We will now study two techniques that decrease a program's space requirements; they both trade time for space by recomputing information from compact representations.

> **Time-For-Space Rule 1 -- Packing:** Dense storage representations often decrease storage costs by increasing the time required to store and retrieve data.

The classical example of packing is the representation of integers. On the IBM System/360-370, for example, integers stored as character strings require eight bits per decimal digit, but can be read and written to external media without change. The "packed decimal" format requires four bits per digit, while a binary representation requires only approximately 3.3 bits per decimal digit. These three representations illustrate three levels of packing: none at all, an intermediate level, and an optimal packing.

For a more sophisticated example of packing we will consider a data structure that arose in a geopolitical database. The bulk of main storage of the (512K 16-bit word) computer was devoted to storing a collection of approximately 10,000 records, each of which contained 36 integers. Upon inspection we found that the vast majority (over 99%) of records had all 36 fields in the range 0..1000. We could therefore use only 10 bits to store each integer in all of those records, which allowed us to put three integers in two 16-bit words, and reduce the size of the record from 36 to 24 words. (The few records with greater integers were marked with a flag and stored in the old format.) Note that this method greatly increases the time required to access each integer, but reduces the storage requirement of a not-inefficiently-coded system to about two thirds of its previous size.

The literature contains many variants on the basic theme of packing. Peterson [1980, pp. 56 ff.] uses packing to store three characters into one 16-bit word to represent a dictionary in little space.

Morris [1978] shows that one can count very large numbers of events in very small registers if one is willing to trade accuracy for space. Knuth [1968, Exercise 2.2.4-18] describes a kind of packing that allows a linked list with a single pointer per node to be traversed either front-to-back or back-to-front. Knuth [1973] uses packing on page 401 to represent a set of primes succinctly, and on page 444 and in Exercise 6.3-42 to compress text.

It is important to remember that when we pack data we decrease the space required to store data but usually increase both the time required to access it and the space to store the program that manipulates it. There are horror stories of programmers who decreased their program's space by thousands of words by *unpacking* small tables -- the data space they saved by packing was much less than the code space devoted to manipulating the packed data! An application of packing that avoids this pitfall is to pack data in files on secondary storage. This decreases the storage required by the file, the time required to read and write files, and the time required to translate the data between internal and external format. Applying this technique, Laird [1981] found that by storing a data structure of floating point numbers in a packed binary format he was able to read it 80 times faster than when it was stored in character representation.

The final data structure rule we will examine reduces not the data structure space of a program, but rather the space devoted to storing the description of the program itself.

> **Time-For-Space Rule 2 -- Interpreters:** The space required to represent a program can often be decreased by the use of interpreters in which common sequences of operations are represented very compactly.

This rule is applied in the development of all large systems, with the motivation not of producing efficient code but rather of producing understandable code; this is the idea underlying the refinement of a program into subroutines! If we have an action that is done in many different parts of the program (perhaps with some minor changes), we describe it once as a subroutine and then call it many times (perhaps with parameters to describe the changes). This use of subroutines decreases the storage cost of the program by slightly increasing the time cost (through the procedure call mechanism and the generality of parameters); we will see in Subsection 3.3.3 that these costs can often be avoided.

There are many examples of more complex interpreters. It is typical, for instance, to perform the lexical analysis of a program text by a finite state machine (FSM). Although an FSM can be implemented directly in a programming language with either while loops or goto's, they are often easier to implement by a small FSM interpreter (about a dozen lines of code) that executes the FSM defined in a two-dimensional table. Although the table-driven interpreter is (minutely) slower than a

directly-coded FSM, it offers many advantages: it is easier to define, code, prove correct, and maintain, and it requires less memory. For details on this approach, see, for instance, Wulf, Shaw, Hilfinger and Flon [1981, Chapters 1 and 19]. Brooks [1975, pp. 102-103] describes an application in which an interpreter saved the day by trading a little time for much space. For an excellent discussion of the applications and construction of interpreters, see Knuth [1968, Section 1.4.3].[12]

In this section we have seen a number of ways of trading space and time against one another. It is important to emphasize that although the descriptions in this section were in particular directions (time for space or space for time), they can usually be applied in the opposite direction also. For instance, Space-For-Time Rule 2 (Store Precomputed Results) can be used to reduce space at the cost of time by recomputing results rather than storing them. The tradeoffs we described above were presented in their typical directions, but all of them can be reversed.

3.3. Modifying Code

In this section we will study techniques for increasing the speed of code. The techniques involve making local transformations that are almost machine independent. That point is important to emphasize: we are not concerned here with the best way to accomplish a particular operation on a particular machine. It is a compiler's job to implement a certain computation on a particular architecture; it is our job to give to the compiler a clean initial computation.

This section is divided into four parts, each of which discusses modifying a different kind of computation. The four types are loops, logic, procedures and expressions. Several of the rules appear in slightly altered form in more than one place, so it is important to realize that the classification imposed by the sections is not meant to be absolute, but rather to be a guide for someone trying to speed up a particular piece of code.

3.3.1. Loop Reorganization

The most often-used efficiency rules deal with loops for the simple reason that the time hogs in most programs involve loops. (It is awfully hard for code to take a lot of time if it isn't executed a lot, and the most common -- although not the only -- way to be executed a lot is to be in a loop.) Because of this fact, we will first study efficiency rules that deal with loops. Our approach in this section will be

---

[12] An important interpreter in some applications is the machine code provided by the underlying computer architecture. For instance, in system sorting routines it is typical for the user to specify a sophisticated method for comparing two records. It would be remarkably time consuming to refer to that specification each time two records are compared, so many sorting routines compile the specifications into code and them jump to the code to compare two records. This very messy approach of compiling tables into machine code is occasionally useful to exploit the full potential of the underlying machine.

to study individually six rules that each reduce some particular cost of a loop; at the end of the section we will return to view the six rules as a collection.

The first efficiency rule deals with repeated computation in loops.

> **Loop Rule 1 -- Code Motion Out Of Loops:** Instead of performing a certain computation in each iteration of a loop, it is better to perform it only once, outside the loop.

The reason for this rule is simple: by incurring the cost of the computation just once outside the loop, we avoid incurring it many times inside the loop. We saw this in the transformation from Fragment A4 to Fragment A5: instead of evaluating PtArr[ThisPt].X and PtArr[ThisPt].Y each time through the loop, we evaluate them only once and store them in the variables ThisX and ThisY. A similar but more substantial savings can be achieved in the following program, whose purpose is to multiply each element of X[1..N] by $e^{sqrt(pi/2)}$.

```
for I := 1 to N do
    X[I] := X[I] * exp(sqrt(Pi/2));
```

Fragment C1. Multiply elements of an array.

Instead of repeatedly performing the expensive division, square root, and exponentiation each time through the loop, we can perform it only once, as in the following code.

```
Factor := exp(sqrt(Pi/2));
for I := 1 to N do
    X[I] := X[I] * Factor;
```

Fragment C2. Evaluate Factor once outside the loop.

Not only is this code usually faster, it also makes the purpose of the loop more transparent. We should note, however, that this particular efficiency rule is easy to implement mechanically, and many compilers already perform this transformation on the code they produce.[13]

The next rule is almost never implemented by a compiler, because it involves a real (though usually local) change to the computation performed by the program.

> **Loop Rule 2 -- Combining Tests:** An efficient inner loop should contain as few tests as possible, and preferably only one. The programmer should therefore try to simulate some of the exit conditions of the loop by other exit conditions.

We used exactly this rule to reduce the cost of termination checking as we transformed Fragment A8 to Fragment A9. An oft-cited application of this rule deals with the following sequential search

---

[13] An important caveat in Loop Rule 1 is that this transformation can actually increase the run time of a program by moving code out of a loop that is executed zero times. Baskett [1978] describes a simple way that compilers can avoid this pitfall; we will see an application of that method in the insertion sort program at the end of this section.

program.[14]

```
I := 1;
while I <= N cand X[I] <> T do
    I := I+1;
if I <= N then
    (* The search is successful; T = X[I] *)
    Found := true
  else
    (* The search is unsuccessful; T is not in X[1..N] *)
    Found := false
```

Fragment D1. Sequential search in an unsorted table.

This loop searches through the array X looking for the value T, and terminates in one of two ways: it either finds T in X[I], or it runs out of valid values of I to investigate. Although it might seem that these two cases really are distinct and that both must be handled in the loop, the following program cleverly simulates the action of "running out of values" by "finding the desired element".[15]

```
X[N+1] := T;
I := 1;
while X[I] <> T do
    I := I+1;
if I <= N then
    Found := true
  else
    Found := false
```

Fragment D2. Add sentinel to end of table.

Note that this version of the program is potentially much faster than the previous version: it contains only half as many tests. Knuth [1973, Section 6.1] reports that this change reduces the run time of a carefully coded Mix program from ~5C to ~4C, where C is the number of comparisons made. The program does have one serious problem, though: what about the old value of X[N + 1]? We might have just clobbered an important element of the array, or (even worse), the array X might contain only N elements and we just generated an array index out of bounds. This modification to the program can therefore only be incorporated if we are careful to ensure that the position is indeed valid and modifiable. Notice that this change increases the program's speed by decreasing its robustness.

Fragment D2 illustrates a very common application of Loop Rule 2 (and Space-For-Time Rule 1 -- Data Structure Augmentation) to searching data structures: to avoid testing whether we have

---

[14] This program uses McCarthy's conditional and operator abbreviated as *cand*. To evaluate A cand B, we first test A and then test B only if A is true. This is necessary in Fragment D1 to avoid accessing X[N + 1] during the last iteration of an unsuccessful search.

[15] Note that the last four lines of the program could be replaced by the assignment "Found := I <= N". The program was presented in its current form to facilitate processing the search element after it is found; such processing can replace the assignment "Found := true".

exhausted a structure, we can augment the structure with a *sentinel* at the boundary in which we place the object for which we are searching. In a binary search tree, for instance, we could replace all nil pointers by pointers to a sentinel node. When we search, we first place the search object, T, in the sentinel node and proceed as usual. When we find T we then test whether it was in a real node of the tree or the sentinel node. Knuth [1973] reports in Exercise 6.2.2-3 that this change decreases the run time of a Mix binary search tree program from ~6.5C to ~5.5C, where C is the number of comparisons made. He used a similar technique in Exercise 5.2.1-33 to decrease the times of two sorting programs from ~9B to ~8B and from ~7B to ~6B, in Exercise 6.4-12 to decrease the run time of a hashing inner loop from ~5C to ~4C, and in Exercise 6.1-4 to decrease the run time of searching a linked list. On page 160 Knuth describes how sentinels can be used to make a merge program simpler while slightly increasing its run time.

Loop Rule 2 can also be used in many other ways. For instance, Fragment E1 performs a sequential search in a sorted table, and was claimed to be more expensive than Fragment D1 because the former makes three comparisons per loop (two of X[I] to T and one to implement the for loop), while the latter makes only two.

```
for I := 1 to N do
    begin
    if X[I] = T then
        begin Found := true;  goto Done end;
    if X[I] > T then
        begin Found := false; goto Done end;
    end;
Found := false;
Done:
```

Fragment E1. Sequential search in a sorted table.

We can immediately notice that the two comparisons made in the begin-end block are similar, and replace them by the statement "if X[I] >= T then goto Done", and set Found accordingly outside the loop. With that change we can also convert the for loop to a while loop, which results in Fragment E2.

```
I := 1;
while I <= N cand T < X[I] do
    I := I+1;
if I <= N cand T = X[I] then
    Found := true
  else
    Found := false
```

Fragment E2. Combine the two comparisons of T to X[I].

With the above code it is easy to put a sentinel at the end of the table (just as in Fragment D2), which results in Fragment E3.

```
X[N+1] := T;
I := 1;
while T < X[I] do
    I := I+1;
if I <= N cand T = X[I] then
    Found := true
  else
    Found := false
```

Fragment E3. Add T to end of table.

Because this is a sorted array, we could also have implemented the sentinel by putting the highest possible key value at the end of the table. It is interesting to note that although Fragment E1 was criticized for making fifty percent more comparisons than Fragment D1, the slightly modified version of Fragment E3 makes only half as many!

The next rule allows us to eliminate some of the overhead in extremely small loops.

> **Loop Rule 3 -- Loop Unrolling:** A large cost of some loops that are only a few lines long is in modifying the loop indices. That cost can often be reduced by "unrolling" the loop.

As an example of a loop in which most of the expense is devoted to index overhead, consider Fragment F1, which places in Sum the sum of the elements of X[1..10].

```
Sum := 0;
for I := 1 to 10 do
    Sum := Sum + X[I]
```

Fragment F1. Compute the sum of X[1..10].

In each iteration of the loop there is only one "real" operation (the addition), but quite a bit of overhead (adding 1 to I and comparing I to 10). That overhead is eliminated entirely in the following code.

```
Sum := X[1] + X[2] + X[3] + X[4] + X[5]
       + X[6] + X[7] + X[8] + X[9] + X[10]
```

Fragment F2. Unrolled sum of X[1..10].

We now have just nine additions and no other loop overhead.

Loop unrolling often decreases program run times dramatically. When a mini- or microcomputer's multiply instruction is implemented in software rather than in hardware, unrolling the main loop can easily decrease the subroutine's time by thirty percent. Unrolling is also commonly used in system numerical routines such as square root and exponentiation. Instead of testing for convergence at each iteration, a numerical analyst can prove that it will take at most k iterations and then unroll the loop k times.

So far we have only discussed unrolling a loop that is executed a constant number of times; the technique can also be extended to general loops that are executed the variable N times. To unroll

such a loop k times, we repeat k copies of the code in the main loop, and then test in the control part whether we are within k of the end of the loop. We must take special care to handle the end values properly.

For an example of variable-length loop unrolling, we will return to Fragment E3 (which performs a sequential search in a sorted table). Two operations are performed in each iteration of the loop: T is compared to X[I] and I is incremented by one; thus a large share of the loop's cost is devoted to the none-too-productive process of incrementing. We can decrease that cost in the following code by unrolling the loop five times.[16]

```
X[N+1] := T;
I := 1;
loop
    if X[I]   >= T then begin Last := I;    break end;
    if X[I+1] >= T then begin Last := I+1;  break end;
    if X[I+2] >= T then begin Last := I+2;  break end;
    if X[I+3] >= T then begin Last := I+3;  break end;
    if X[I+4] >= T then begin Last := I+4;  break end;
    I := I+5
  repeat;
if Last <= N cand T=X[Last] then
    Found := true
  else
    Found := false
```

Fragment E4. Loop-unrolled sequential search in a sorted array.

Whereas before we had only one "real" operation (comparing an element of X to T) for every "bookkeeping" operation (adding one to I), we now have a ratio of five real operations for every bookkeeping operation. This technique can be applied with any value other than five; we trade program size for run time. (One might complain that in the above example we must, for instance, add 3 to I to access X[I + 3]; most compilers, though, implement the instruction with 3 as a compiled offset from the base I.) One can use exactly this technique to unroll the inner loop k times in the assembly program A10 of Section 2.2; this would remove the incrementing instruction "LA I,8(I)" from the loop and replace the Load instruction with "L ThisDist,j(I)", where j = 0,8,16,...,(k-1)·8.

To give a feeling for the difference that loop unrolling can make in real programs we will note several examples. Knuth [1973] shows in Section 6.1 that unrolling a sequential search loop k times decreases its running time from ~4C to ~(3 + 1/k)C, where C is the number of comparisons made. In Exercise 6.2.1-11, Knuth uses unrolling to reduce the run time of a uniform binary search of an N-

---

[16] In Fragment E4 we use the language construct loop...repeat, which has the semantics of repeating the code it contains in a loop until a break statement is reached, at which time the program resumes execution at the next statement following the loop...repeat. This kind of loop construct is often useful in unrolling loops. Note that if a particular language does not offer this construct, then its effect can be synthesized by disciplined use of goto statements.

element table from ~8.5 $\log_2$ N to ~4.5 $\log_2$ N. Dongarra and Hinds [1979] present empirical timings that show that unrolling extremely tight Fortran loops on high-performance machines can often increase their speed by a factor of up to two. Sedgewick [1975, Appendix A] uses loop unrolling to reduce the Mix time of a Quicksort implementation from ~10.63 N ln N to ~9.57 N ln N (see also Sedgewick [1978]).

All of the loops that we have seen so far have the property that the maximum number of iterations is known before the first iteration (some were known even at compile time). We will now consider a different kind of example: taking the sum of a linked list of integers. If we use the standard representation of linked lists, then the loop must access the next node of the list and compare it to nil for each addition. Although we cannot remove the cost of accessing the next node, we can reduce the cost of comparison to nil by augmenting the list with a special kind of sentinel node at the end. That node has the value of zero and a link field that points to the node itself; we then unroll the loop k times, and test for nil every k iterations. Note that we might make up to k unnecessary iterations of the loop, but adding zero to the sum will not change the final result. A "self-pointing" sentinel can often be used in other applications to unroll "run-time unknown"-length loops.

We turn now to a special kind of loop unrolling whose purpose is not to reduce the cost of indexing but rather to reduce the need for trivial assignments (that is, assignments of the form I: = J, where I and J are both simple variables).

> **Loop Rule 4 -- Transfer-Driven Loop Unrolling:** If a large cost of an inner loop is devoted to trivial assignments, then those assignments can often be removed by repeating the code and changing the use of variables. Specifically, to remove the assignment I: = J, the subsequent code must treat J as though it were I.

The above statement of this rule is quite vague; we will now illustrate its use by studying two examples in detail. The reader interested in a more detailed study of this technique is referred to the fascinating paper of Mont-Reynaud [1976].

As our first example of transfer-driven unrolling, we will again consider Fragment B1, which computes Fibonnaci numbers. Recall that the program's only loop consists of a for statement that contains one assignment (involving an addition) and two trivial assignments. We can remove both of those trivial assignments by modifying the code as shown in Fragment B3.

```
function Fib(N:integer):integer;
    var A,B,I: integer;
    begin
    if N < 1 or N > Max then return 0;
    if N <= 2 then return 1;
    A := 1; B := 1;
    for I := 1 to (N div 2) - 1
        begin
        A := A+B
        B := B+A
        end;
    if not even(N) then
        B := B+A;
    return B
    end;
```

Fragment B3. Loop-unrolled Fibonacci numbers.

The invariant of the loop is that before the first assignment is executed, A contains the $2I-1^{st}$ Fibonacci number and B contains the $2I^{th}$; it is easy to prove the program correct using that invariant. Note that while Fragment B1 used a loop control and two trivial assignments for every "real" operation of addition, Fragment B3 involves only half a loop test for every addition, and that fraction can be reduced by loop unrolling.

The second example of transfer-driven unrolling that we will study inserts a new node named ThisNode into a sorted linked list whose elements contain both a Link and Value field. To ease programming (and increase the speed of the loop), we will assume that the list has been augmented to contain sentinel nodes at the head and tail whose values are, respectively, less than and greater than all keys. The code for inserting ThisNode into the list pointed to by Anchor is shown in Fragment G1.

```
P := Anchor;
Q := P↑.Link;
while Q↑.Value <= ThisNode↑.Value do
    begin
    P := Q;
    Q := Q↑.Link
    end;
ThisNode↑.Link := Q;
P↑.Link := ThisNode;
```

Fragment G1. Insert ThisNode in a sorted linked list.

This is a standard operation on linked lists in which P is always "one step behind" Q. Note that a substantial percentage of the code in the inner loop is devoted to the trivial assignment P:=Q. That can be removed by unrolling the loop two times and changing their roles so that they "leap frog" over one another, first Q in front of P and then P in front of Q. This modification is reflected in Fragment G2.

```
P := Anchor;
repeat
    Q := P↑.Link;
    if Q↑.Value <= ThisNode↑.Value then
        begin
        ThisNode↑.Link := Q;
        P↑.Link := ThisNode;
        break
        end
    P := Q↑.Link;
    if P↑.Value <= ThisNode↑.Value then
        begin
        ThisNode↑.Link := P;
        Q↑.Link := ThisNode;
        break
        end
  loop;
```

Fragment G2. Remove trivial assignment.

Note that the above code makes only one assignment (involving a Link field) and one test for each node visited; Fragment G1 involved an extra trivial assignment statement.

To illustrate the impact of transfer-driven loop unrolling we will again turn to several extremely-well coded Mix programs of Knuth [1973]. In Exercise 5.2.1-33 he shows that a change similar to unrolling Fragment G1 to achieve G2 reduces the Mix time from ~6B to ~5B. Exercise 5.2.4-15 reduces the time of a merge sort from ~10N lg N to ~8N lg N, and on page 426 he reduces the time of a binary tree search from ~7.5C to ~6.5C. The application of this technique to Fibonaccian search can be found on pp. 415 and 416. Knuth [1971, p. 124] uses this technique to increase the speed of a binary search on an IBM System/360 by a factor of more than 2. Another instance of this technique can be found in Knuth [1968, Exercise 1.1-3].

We will now study an efficiency rule that is not appropriate for programs in a high-level language, but can often be used to speed up an inner loop in assembly code or in unstructured languages such as Fortran.

> **Loop Rule 5 -- Unconditional Branch Removal:** A fast loop should contain no unconditional branches. An unconditional branch at the end of a loop can be removed by "rotating" the loop to have a conditional branch at the bottom.

As an example of this rule, we will consider the typical low-level implementation of the statement while C do S shown in Fragment H1.

```
Loop:   if not C then goto End;
        S;
        goto Loop;
End:
```

Fragment H1. Typical translation of while C do S.

If the code for C and S is extremely small, then the cost of the unproductive goto can be a substantial amount of the time spent in the loop. That cost is removed in the following fragment.

```
        goto Test;
Loop:   S;
Test:   if C then goto Loop;
End:
```

Fragment H2. Efficient translation of while C do S.

Note that this translation contains a new unconditional goto outside the loop but has no unconditional branch within the loop (it also avoids inverting the value of C, which can save time in many implementations). This transformation can be applied to loops other than while; for more details on this transformation, see Baskett [1978]. Knuth [1973] uses this technique in all of the inner loops in the text; a particularly interesting example can be found in the organization of Program 6.2.2T.

It is important to emphasize that the above rule usually need not and should not be applied in a high-level language -- many compilers recognize loop constructs as special cases and compile them very efficiently. By "optimizing" the code one runs the risk of the compiler not recognizing the loop, which results in a slower and more obscure program. When applied to a very small loop in a low-level language, though, this technique can often reduce run time by ten or twenty percent.[17]

The next loop rule that we will see is based on the same idea as car-pooling: if two sets of operations are performing operations on the same set of values, then why shouldn't they "share the ride" through those values?

> **Loop Rule 6 -- Loop Fusion:** If two nearby loops operate on the same set of elements, then combine their operational parts and use only one set of loop control operations.

The application of this rule reduces the loop overhead without impairing the "real" computation that is being performed. It can often be used when two nearby loops operate on the same data structure for unrelated purposes, at the price of confusing the code.

So far we have viewed the efficiency rules for loops as acting in isolation; we will now take a moment to view the rules as a collection. The following list gives the number and name of each of the rules, and briefly describes what unnecessary computation it eliminates from loops.

1. **Code motion out of loops:** eliminates repeated computation.
2. **Combining tests:** reduces the number of tests.

---

[17] Although I have tried to avoid entirely low-level language tricks specific to given machines, there is one trick that is so common as to be worth a brief note. Because on many machines it is easy to compare a value to zero, it is often advantageous to restructure a tight loop to "count down to zero" to facilitate a more rapid termination test (see Knuth [1968, p. 148, Note 10]). Indeed, many computer architectures provide single instructions that implement counting loops (such as the IBM System/360-370 BXLE instruction and the PDP-11 SOB instruction).

3. **Loop unrolling:** reduces costs of indexing.
4. **Transfer-driven unrolling:** reduces the number of trivial assignments.
5. **Unconditional branch removal:** removes the unconditional branch at the bottom of the loop.
6. **Loop fusion:** shares the cost of loop overhead.

Each one of the rules eliminates a different kind of unnecessary computation, and together they can eliminate almost all excess baggage from a loop. We will see in Logic Rule 2 a technique for eliminating unnecessary iterations of loops, and in Expression Rule 2 a technique for simplifying the kind of computation in loops.

As an example of how the above six rules work together on a single loop, we will consider the classical insertion sorting program for arranging the elements of an array in nondecreasing order shown in Fragment I1 (for more information on insertion sorting, see Knuth [1973, Section 5.2.1] or Sedgewick [1975, Chapter 1]).

```
for I := 2 to N do
    begin
    J := I;
    while J > 1 cand X[J] < X[J-1] do
        begin
        Swap(X[J],X[J-1]);
        J := J-1
        end
    end
```

Fragment I1. Insertion sort.

The above program is easy to read and prove correct; in most applications, the program should be left in exactly the above state. If sorting is in the program's time bottleneck, though, and this is the best sorting procedure to use (and it is for small values of N), then we are justified in devoting a great deal of energy to improving the program.

The first improvement we should make is to write the call of the Swap procedure in line, which results in the following fragment.

```
for I := 2 to N do
    begin
    J := I;
    while J > 1 cand X[J] < X[J-1] do
        begin
        T := X[J];
        X[J] := X[J-1];
        X[J-1] := T;
        J := J-1
        end
    end
```

Fragment I2. Swap procedure written in line.

We do this both to eliminate the cost of the procedure call and to allow further time reductions. We will now try to apply Loop Rule 1 to move repeated computation out of the loop. Careful inspection of the code shows that the variable T is repeatedly being assigned and then storing the same value; we can remove those assigns and stores by rewriting the code as shown in Figure I3.

```
for I := 2 to N do
    begin
    J := I;
    T := X[I];
    while J < 1 cand T > X[J-1] do
        begin
        X[J] := X[J-1];
        J := J-1
        end;
    X[J] := T
    end
```

Fragment I3. Move operations on T out of loop.

(Note that if the while loop is usually executed zero times, then this code will take longer than Fragment I2; Knuth [1973] shows how this pitfall can be avoided in Exercise 5.2.1-10 and Program 5.2.2Q, Step 9.)

We next try to apply Loop Rule 2 and combine tests. The inner while loop contains two tests that can easily be reduced to one by placing a sentinel in the zeroth position of the table; the modified code is shown in Fragment I4.

```
X[0] := MinusInfinity;
for I := 2 to N do
    begin
    J := I;
    T := X[I];
    while T < X[J-1] do
        begin
        X[J] := X[J-1];
        J := J-1
        end;
    X[J] := T
    end
```

Fragment I4. Add a sentinel at X[0].

To measure the benefit of the above transformations Fragments I1 through I4 were implemented in Pascal on a PDP-10 (using the same compiler and processor used in the experiment of Section 2). Insertion sort is known to require time proportional to N^2. In ten runs each on five hundred random elements, the constants for the four fragments were estimated to be 10.17, 6.43, 4.03, and 3.32 microseconds, respectively (that is, Fragment I1 required approximately $10.17N^2$ microseconds, on the average). Note that the cumulative effect of the three transformations is to speed the program up

by over a factor of three.[18]

We can still apply further loop rules to this program. Loop Rule 3 (Loop Unrolling) can be used to reduce the cost of the instruction J := J - 1 by unrolling the loop some fixed number of times; because that change is so straightforward, we will not show it here. Loop Rule 4 (Transfer-Driven Unrolling) is not applicable to this code because it contains no trivial assignments. Likewise, Loop Rule 5 (Removing Unconditional Branches) is not applicable because we are coding in a high-level language, but it would certainly be used in any efficient low-level language implementation of the code.

It is interesting to study Loop Rule 6 (Loop Fusion) as it relates to Fragment I4. On the one hand, it appears not to be applicable because there is only one loop in the program. On the other hand, though, we can view it as having been applied already: the two tasks of finding where to place X[I] and then placing it there might logically be divided into two loops, but our code already performs both tasks with the overhead of only one loop!

3.3.2. Logic Reorganization

In this section we will study techniques that can decrease the cost of code that is devoted to logic. In particular, these techniques will focus on efficiency problems that arise when evaluating the program state by making various kinds of tests. They all take a clean piece of code and massage it so that it is less clear but (we hope) more efficient; in other words, they sacrifice clarity and robustness for speed of execution.

The first rule for manipulating logic will arise again in a similar context as Expression Rule 2.

> **Logic Rule 1 -- Exploit Algebraic Identities:** If the evaluation of a logical expression is costly, replace it by an algebraically equivalent expression that is cheaper to evaluate.

For instance, instead of testing whether "sqr(X) > 0" in an inner loop, we could just as easily test "X<>0" (because the square of a number is greater than zero if and only if that number is not zero). Similarly, we could use deMorgan's laws to change the test "~A and ~B" to "~(A or B)"; the latter might involve one less negation. In general, we could use many techniques of switching theory to

---

[18] Kernighan and Plauger [1978, pp. 131-133] study an "efficient" interchange sort that was presented in a programming text and show that a simple version of the same idea not only requires only half as many lines of Fortran code but is also about thirty percent faster on randomly generated data. Their simple program was conceptually somewhere between Fragments I1 and I2; its Pascal transliteration was eight lines long and had a running time of $7.10N^2$ microseconds. Kernighan and Plauger present this as an illustration of the principle to "keep it simple to make it faster". It is interesting to note that Fragment I4 contains only twelve lines of (relatively simple) code to achieve a speed increase of a factor of two over their program. Although this longer and faster program might appear to violate the letter of their maxim, it does follow its spirit: the more complex approach to efficiency they describe results in a program that requires thirty percent more time, while our approach of applying simple and well-understood transformations results in a program that is faster by more than a factor of two.

minimize the work required by boolean functions. Operations at this level are, however, extremely dependent on the compiler and the underlying machine, so one must be careful that a clever "optimization" along these lines does not fool one's compiler into generating slower code!

There are more substantial applications of Logic Rule 1 that will reduce the run times of many programs on most compilers and machines. For instance, in changing Fragment A2 to A3 we used "strength reduction" to remove a square root. In particular, we wished to compute a boolean variable telling whether a new point was closer than the best point so far. We exploited the fact that square root is a monotone increasing function to show that "A > B" if and only if "sqrt(A) > sqrt(B)", which allowed us to remove the square root from the test. This algebraic technique can often be used to avoid a function evaluation when we are concerned only about the relative ordering of a pair of objects (though it can increase bookkeeping). Knuth [1973, Exercise 6.2.1-23] shows how a different algebraic identity can be used to reduce ternary comparisons to binary and thereby decrease the cost of a single comparison in comparison-based approaches to searching.

The next rule for dealing with logic allows us to avoid unneeded work after we have already gleaned enough information to make a decision.

> **Logic Rule 2 -- Short-circuiting Monotone Functions:** If we wish to test whether some monotone nondecreasing function of several variables is over a certain threshold, then we need not evaluate any of the variables once the threshold has been reached.

A common application of this rule is in the evaluation of simple boolean formulas. In many languages, for instance, if we wish to evaluate "A and B", we can write "A cand B", which evaluates A and then evaluates B only if A is true. This avoids the evaluation of B if A is false, which can represent a substantial time savings. (In the ADA language, this feature is explicitly called "short-circuiting".) For a more sophisticated application of the rule, consider determining whether there are any negative elements in an array of reals. The most naive (and perhaps cleanest) approach sets the boolean FoundNegative originally to false, and then goes through the array and sets FoundNegative to true if it observes that a given real is negative. It is only slightly more work to modify the loop to terminate precisely at that point, because we can then accurately report that the array does contain a negative.

Logic Rule 2 was essentially the idea we used in transforming Fragment A5 to A6 by calculating the y-distance between a pair of points only after ensuring that their x-distance alone was not sufficient to discard them from consideration. We will now generalize that idea by examining the problem of determining whether the sum of the real numbers in X[1..N] is greater than the given real CutOff; we will assume that the reals in X are known to be positive (in Fragment A6, for instance, the corresponding values were squares and therefore nonnegative). A straightforward program for this

task is shown in Fragment J1; the boolean variable Greater is true if and only if the sum of the elements of X is greater than CutOff.

```
Sum := 0;
for I := 1 to N do
    Sum := Sum + X[I];
Greater := Sum > CutOff
```

Fragment J1. Sum first then compare.

If it is known that CutOff is usually less than the sum of the first few values of X, then Fragment J2 is a faster means of accomplishing the same task.

```
I := 1;
Sum := 0;
while I <= N and Sum <= CutOff do
    begin
    Sum := Sum + X[I];
    I := I+1
    end;
Greater := Sum > CutOff
```

Fragment J2. Compare as we sum.

If the cost of comparing Sum to CutOff is relatively high, or if the probability that the loop will be terminated early is relatively low, then Fragment J2 can be slower than Fragment J1. The two fragments are extremes along a spectrum in which we trade the work of additional comparisons for the expected benefits of early termination of the loop. A middle element of that spectrum is shown in Fragment J3, in which we perform two additions for every comparison.

```
I := 1;
Sum := 0;
if odd(N) then
    begin
    I := 2;
    Sum := X[1]
    end;
while I < N and Sum <= CutOff do
    begin
    Sum := Sum + X[I] + X[I+1];
    I := I+2
    end;
Greater := Sum > CutOff
```

Fragment J3. Add twice for each compare.

Note the careful preprocessing necessary to apply unrolling to the above loop.

Logic Rule 2 is especially powerful when dealing with loops that evaluate monotone logical functions. It is usually easiest to write such loops so that they iterate over their entire range of values, and they should be written this way originally. If we later find that a certain such loop is in a time bottleneck of the program, then we can modify it to terminate early, by disciplined use of loop exiting

constructs (either break statements or even disciplined and documented use of gotos). Depending on where the "jump to threshold" usually occurs, this technique can usually save a factor of two or more on loops that evaluate this particular kind of logic. This strategy gives us the best of both worlds: all loops in our programs are initially designed with a clean and straightforward structure, and then the critical loops are modified in an understandable way from understandable code (as opposed to being monuments to extreme cleverness from the beginning!).

The next logic rule reduces the running time of a program by rearranging the sequencing of tests.

> **Logic Rule 3 -- Reordering Tests:** Logical tests should be arranged such that inexpensive and often successful tests precede expensive and rarely successful tests.

This rule has the corollary that when a series of nonoverlapping conditions is sequentially evaluated until one is true, the inexpensive and common conditions should be evaluated first and the expensive and rare conditions should be evaluated last. As an example of this corollary, we will consider Fragment K1, which is a pseudo-Pascal function that returns an integer code that describes the type of the character it was passed.

```
function CharType(X: char): integer;
    begin
    CharType :=
        if X = ' ' then 1
          else if ('A' <= X and X <= 'I') or ('J' <= X and X <= 'R')
                or ('S' <= X and X <= 'Z') then 2
          else if '0' <= X and X <= '9' then 3
          else if X = '+' or X = '/' or X = '-' or X = ',' or X = '('
                or X = ')' or X = '=' then 4
          else if X = '*' then 5
          else if X = '"' then 6
          else 7
    end;
```

Fragment K1. A character recognizer.

Fragment K1 was used to process every character read by a compiler for a Fortran-like language on an IBM System/360; the seven integers respectively denote blank, letter, digit, operator, asterisk, quote and other. (It is because of "holes" in the EBCDIC character code that the test for letter is so complicated). Although the above presentation is somewhat clearer, the code actually used in the compiler was similar to that shown in Fragment K2.

```
function CharType(X: char): integer;
    begin
    CharType :=
        if X = ' ' then 1
          else if X = '*' then 5
          else if X = '"' then 6
          else if '0' <= X and X <= '9' then 3
          else if ('A' <= X and X <= 'I') or ('J' <= X and X <= 'R')
                or ('S' <= X and X <= 'Z') then 2
          else if X = '+' or X = '/' or X = '-' or X = ',' or X = '('
                or X = ')' or X = '=' then 4
          else 7
    end;
```

Fragment K2. Order of tests changed.

Fragment K2 will be faster than the previous version if there are enough occurrences of asterisks, quotes and digits to merit their earlier testing. For a precise mathematical formulation of this corollary of Logic Rule 3, see Knuth [1973, Exercise 6.1-16]. Knuth [1973, Program 6.2.2T] orders the tests in a binary tree search program (lines 10 and 11) to reduce its running time from ~7C to ~6.5C.

Logic Rule 3 has many applications other than performing a sequential series of tests; for instance, it encourages us to "push an expensive test inside a cheaper test". This is exactly what we did in transforming Fragment A8 to Fragment A9; we "pushed" the loop control test inside the necessary test for being a new minimum. This was exactly the same idea underlying the sentinels in Loop Rule 2: we push a test for loop control inside a test on the data structure. Knuth [1973, Exercise 5.2.3-18] uses this idea to reduce the running time of a heapsort program from $\sim 16N \log_2 N$ to $\sim 13N \log_2 N$ time units. A more sophisticated kind of "pushing one test inside another" is to perform an expensive yes-no check by first running a cheaper algorithm that usually returns yes or no but sometimes returns "maybe" -- only in the latter case do we have to perform the more expensive test. An example of such an approximate test can be found in Bentley, Faust and Preparata [1981].

Logic Rule 4 is an application of Space-For-Time Rule 2 (Store Precomputed Results) to the domain of logic.

> **Logic Rule 4 -- Precompute Logical Functions:** A logical function over a small finite domain can be replaced by a lookup in a table that represents the domain.

A simple application of this rule can replace the rather complicated and very slow CharType procedure of Fragment K2 by the more elegant and clean program of Fragment K3.

```
function CharType(X: char): integer;
    begin
    CharType := TypeTable[ord(X)]
    end;
```

Fragment K3. Character recognition by table lookup.

This program determines the type of character X simply by looking at the appropriate entry in a 256-element table (the number of characters in the EBCDIC character code); the function "ord" is used in Pascal to convert a character to its integer rank. Peterson [1980] used precisely this method to classify letters in a spelling correction program. This change results in slightly more space (we have less code but a new table), but is much faster; trading that space for time is wise if much time was spent in Fragment K2. Kernighan [1981] reports that when translating the programs of Kernighan and Plauger [1978] from Ratfor to Pascal, he observed that between 30 and 40 percent of the run time of some of the resulting Pascal programs was spent in character recognition by a subroutine like Fragment K2. In an application such as that, the 256-element table would be well worth its space!

Logic Rule 4 has many faces. Sometimes we use it to replace a long chain of if-then-else if-then-else statements by a single case statement; clever compilers then choose an optimal strategy for implementing the case statement in the assembly code, and often generate a table. Knuth [1968, Exercise 1.3.2-9] describes how assembly language coders can implement multiway branches effectively as a jump table. He uses that technique in Exercise 1.3.2-9 to test for validity of a certain field of an assembly code instruction, in Exercise 1.3.2-23 to prepare graphical output, and on pp. 200-204 to implement an interpreter.[19] If we were evaluating a function of six boolean variables, we could replace the function evaluation by a lookup in a sixty-four element table.

A powerful application of this technique was used by David Moon [1981] in the design of a PDP-8 simulator (which was designed to run on a PDP-10 but was never actually implemented). Because the PDP-8's memory words are just twelve bits wide, there are only 2^{12}, or 4096, different instructions. Moon observed that instead of taking the time to interpret each instruction at run time, we could precompute the actions of all possible instructions, and store them in a 4096-element table. This led to an extremely efficient inner loop in the simulator: we execute a single instruction, using the program counter as an index into the instruction table. Most of the PDP-8 instructions could be emulated by a single PDP-10 instruction, and those that couldn't had a jump to subroutine in their position in the table.

In the final logic rule we will see how the time required to read and write boolean variables can be eliminated by "storing them in the program counter".

> **Logic Rule 5 -- Boolean Variable Elimination:** We can remove boolean variables from a program by replacing the assignment to a boolean variable V by an if-then-else statement in which one branch represents the case that V is true and the other represents the case that V is false.

---

[19] See especially the paragraph starting at the bottom of page 200 and the first sentence on page 204.

As an instance of the above rule, we will consider Fragment L1.

```
V := LogicalExp;
S1;
if V then
    S2
  else
    S3
```

Fragment L1. Code with boolean variable V.

We could replace the above example by the code in Fragment L2, as long as the boolean variable V is used nowhere else in the program.

```
if LogicalExp then
    begin
    S1;
    S2
    end
  else
    begin
    S1;
    S3
    end
```

Fragment L2. Boolean variable V removed.

The resulting code is larger by the size of S1 (because it is now repeated twice), but is slightly faster. Knuth [1974, pp. 284-285] shows how boolean variable elimination can be used in the partitioning phase of a Quicksort program to reduce the total run time by about 25 percent. Knuth [1973, Program 6.2.3A] uses a similar technique to eliminate a variable over {-1,0,1} in a program for manipulating balanced binary search trees.

3.3.3. Procedure Reorganization

So far we have improved the efficiency of a program by making local changes to small pieces of code. In this section we shall take a different approach by leaving the code alone and instead modifying the underlying structure of the program as it is organized into procedures.

The first procedure rule that we will study is essentially the dual of Time-For-Space Rule 2 (Interpreters); we pay in program space to buy program run time.

> **Procedure Rule 1 -- Collapsing Procedure Hierarchies:** The speeds of the elements of a set of procedures that (nonrecursively) call themselves can often be reduced by rewriting procedures in line and binding the passed variables.

The simplest application of this rule is that subroutine calls in time bottlenecks should be written in line; this is exactly the method we used in transforming Fragment A4 to A5. This achieves two kinds of savings: we avoid the cost overhead of the procedure call and it often opens the way for further optimizations (as in Fragments I2 through Fragment I4).

Many languages provide means whereby certain subroutines are always expanded in line (as opposed to being invoked through a subroutine call); this mechanism goes under names such as macros and in-line procedures. Scheifler [1977] studies the benefit this operation can have in entire programs, and observes that "in programs with a low degree of recursion, over 90 percent of all procedure calls can be eliminated, with little increase in the size of compiled code and a small savings in execution time." This operation is especially important if a system has been designed using data abstraction methodologies in which most accesses to data are done with a procedure call; if the substitutions are made mechanically, then we have a clean program with rapid execution time.

Procedure Rule 1 need not always instantiate all procedures into in-line code; as in many tradeoffs, we can often choose a middle between two extremes. For instance, it might be cleanest to design a particular piece of code with one subroutine with five variables called from ten places. We could then replace that with ten different in-line instantiations, as one extreme. A more moderate approach might involve replacing the one subroutine with three subroutines that have, say, just two parameters each, and each much faster than the single subroutine.

Procedure Rule 1 takes a nicely structured program and unstructures it for the sake of speed[20]; because of this, some people have deduced that efficiency and clean modularity cannot peacefully coexist. Although that deduction might appear valid on the surface, a deeper analysis shows that quite the opposite is in fact true. An important cost in most programs is the space they require, and a clean module structure usually reduces that space (as we saw in Time-For-Space Rule 2 -- Interpreters). Recall that monitoring programs usually shows that a very small percentage of the code accounts for a very large percentage of the run time; at that point we know which are the expensive data structures in the system, and can then modify them. If the accesses to the structures had been spread throughout the system, then there is no way we could have isolated the resource hogs. When we finally do collapse the hierarchy, we can do so in an orderly way (often by changing procedures to macros), so the programmers later involved in the project can still see the highly structured code, even though it is compiled into something less clean.

The next rule for procedures is related to Logic Rule 3 (Reordering Tests); it formalizes Allen Newell's [1981] maxim that "almost always is almost always as good as always".

Procedure Rule 2 -- Exploit Common Cases: Procedures should be organized to

---

[20] Because it is machine-dependent and therefore outside the scope of this paper, we have left unmentioned one of the most important applications of collapsing procedure hierarchies. It is often profitable to "push" common operations in a system down into the operating system, microcode, or even special-purpose hardware. This is difficult to accomplish in most systems, but it can sometimes yield substantial time reductions.

handle all cases correctly and common cases efficiently.

Jalics [1977, p. 137] used this technique in a routine to calculate Julian dates. He observed that 90% of the calls to the routine had the same date as the previous call, so in those cases he returned the previous answer without recomputing it. We saw an application of this rule in Space-For-Time Rule 3: caching data allows us to handle all accesses to it correctly and common accesses efficiently. The basic mechanism for implementing Procedure Rule 2 is simple: we have two routines that accomplish the same end. One is slow but handles all cases correctly; the other handles only special cases but does so very quickly. There are several ways by which we can ensure that the proper procedure will be executed at run time.

1. All calls in the body of the program are to the general procedure. The general procedure contains a prelude that checks the input, and calls the special procedure when it is appropriate. This localizes knowledge of the special procedure to the general procedure itself, but incurs an added cost at run time.
2. All calls in the body of the program are to the special procedure if we can deduce at compile time that it is appropriate; otherwise they are to the general procedure. This saves run time, but destroys program modularity by spreading knowledge of the special procedure throughout the entire program.
3. An intermediate approach has all calls refer to the procedure through a compile-time macro; if that macro can determine at compile time that the special procedure can be called then it is, otherwise it calls the general procedure. This approach has the advantages of both the above schemes: maximal efficiency is achieved, and knowledge about the special procedure is still localized (in the macro).

The first approach is the easiest to implement in most languages and achieves most of the time savings possible, the third approach can squeeze out a little more time, and the second approach should almost never be used (because it pays too much a cost in maintainability to buy too little in time).

An important application of Procedure Rule 2 is to observe when a particular subroutine is being used in a certain way. For instance, it might be natural to write a subroutine to access the I^{th} element of a sequence of elements, and then ask sequentially for the first, second, third, up to the N^{th} elements. For most representations of sequences (including linked lists, trees, and usually even arrays), it would be preferable to make a new procedure to access the next element. A different approach would use Space-For-Time Rule 3 and cache the most recently accessed element in the hope that the next element will be near it; this approach retains state in a procedure across calls. (For a general discussion of retaining state across calls, see Scherlis [1980, p. 5].) This application of Procedure Rule 2 is often appropriate when dealing with input and output; it would encourage us, for instance, to have operating system primitives that read or write entire records instead of forcing the user to access each byte individually through an operating system call. There are two benefits of this

strategy: we can avoid the cost of many procedure calls and avoid recomputing state across those calls.

Procedure Rule 2 encourages us to reduce time by developing specialized procedures; it is sometimes the case, though, that we can reduce time by developing more general procedures. This fact is so startling that Polya [1945, p. 121] refers to it as the "Inventor's Paradox" and states it as "the more general problem may be easier to solve". Experienced programmers can usually recall seeing the paradox applied in many circumstances; for instance, it is often more effective in terms of both coding effort and run time to have one procedure for searching tables in a program rather than to have a separate procedure for each table in the program. By having only one routine we can devote a great deal of energy to making it very fast; the payoff of that work is then realized each time the single procedure is called. We also are using the inventor's paradox to achieve efficiency when we have one very efficient (but very general) system sort routine rather than have each programmer write a custom sort for each new application. The inventor's paradox has long been realized as an important tool for maintaining correct and maintainable code; it is important not to let Procedure Rule 2 make us forget that the inventor's paradox can also help us make efficient code.

The next procedure rule can reduce the overhead cost of input and output in many applications.

> **Procedure Rule 3 -- Coroutines:** A multiple-pass algorithm can often be turned into a single-pass algorithm by use of coroutines.

This technique has been discussed in detail by Knuth [1968, pp. 194-196]. He observes that if program A reads file 1 and writes file 2 sequentially, and is then followed by program B which reads file 2 sequentially and writes file 3, then we can avoid any use of file 2 by keeping both programs A and B in core and allowing them to communicate as coroutines. This will be faster than the sequential version (because we reduced the costly I/O operations), but requires more space (both for data and code). An important point, however, is that programs are often easier to design if we think of them as multiple-pass algorithms and then implement them later as single-pass algorithms. The Unix operating system has made coroutines a fundamental concept in the form of its *pipes* and *filters* (see Ritchie and Thompson [1978, Sections 5.2 and 6.2]).

Recursive procedures (that is, procedures that can call themselves) are powerful expressive tools that can greatly ease the design and implementation of a correct program. That lesson was made clear to me when I translated a dozen-step iterative algorithm that I had written (which resulted in about 80 lines of code) into a four-step recursive algorithm (which took just 30 lines of code). Unfortunately, this expressive power is not entirely without cost, and many implementations of recursion are rather slow. For this reason, a great deal of work has been devoted to transformations

that increase the speed of recursive programs. (See, for instance, Burstall and Darlington [1976], Bird [1980], Darlington and Burstall [1977], Knuth [1974], and Standish *et al* [1976].) Because that literature is so extensive, we shall mention here only a few transformations on recursive procedures, and those only briefly.

Procedure Rule 4 -- Transformations on Recursive Procedures: The run time of recursive procedures can often be reduced by applying the following transformations.

- Code the recursion explicitly by use of a program stack. This can sometimes reduce costs induced by the system structure, but is often slower than using the system procedure calls.

- If the final action of a procedure P is to call procedure Q, replace that call by a goto to Q. That goto can often be transformed into a loop. Knuth [1974, p. 281] has a detailed discussion of this rule.

- If a procedure contains only one recursive call on itself, then it is not necessary to store the return address on the stack (see Knuth [1974, pp. 281-282]).

- It is often more efficient to solve small subproblems by use of an auxiliary procedure, rather than by recurring down to problems of size zero or one. This technique was used by Sedgewick [1978] to reduce the time Quicksort used in sorting small subfiles and by Friedman, Bentley and Finkel [1977, pp. 220-221] to reduce the search time of a data structure by a factor of almost two.

This above list only briefly scratches the surface of techniques for increasing the efficiency of recursive programs; programmers who use recursion often should certainly read the articles mentioned above.

The final procedure rule we will study is the most nitty-gritty.

Procedure Rule 5 -- Parallelism: A program should be structured to exploit as much of the parallelism as possible in the underlying hardware.

This rule of course takes on gigantic proportions when we are designing a program to be executed on a multiprocessor architecture, but it usually surfaces whenever we scrutinize a computing system. For instance, in Expression Rule 5 we will see that we can exploit the width of a computer word to perform several operations at once. Clever compilers often use the fact that operations set condition codes as a side effect to avoid redundant tests (see, for instance, Russell [1978]); this exploits parallelism at a microscopic level. Many architectures provide "block move" operations that allow us to move many adjacent words in a single instruction (for instance, the IBM System/360 has LDM, STM, and MVC instructions). Parallelism is usually present in the operation of the CPU and I/O devices; in some systems one can move work from the CPU to the channels or device controllers (for instance, by using CCW programs in the IBM System/360 family). Very high performance computers such as the IBM System/360 Model 91 and the CDC 7600 have many functional units in the central

processor that operate in parallel; knowledge about their detailed characteristics can allow us to utilize that parallelism more efficiently.

It is important to keep in mind that all of the above techniques for exploiting parallelism are hard to apply and are quite peculiar to the architecture of the computer on which the program is executed; one should therefore be quite reluctant to use them. Occasionally, however, they can be used to increase the speed of a program dramatically. Kulsrud *et al* [1978] provide a fascinating example of the exploitation of a highly parallel architecture; they describe an implementation of Quicksort on a CRAY-1 that can sort 800,000 elements in less than 1.5 seconds. They employ many general techniques for efficiency (including proper algorithm and data structure selection and very careful assembly coding), many techniques of writing efficient code (including loop unrolling, caching the recursion stack, and special treatment of small subfiles), and many techniques specific to highly parallel machines (including chaining operations, careful instruction buffering, and overlapping the execution of independent activities).

3.3.4. Expression Reorganization

In this final section on code modifications we shall study techniques that reduce the time devoted to evaluating expressions. It is important to emphasize that many of these techniques are applied by even relatively simple compilers, and our attempts to help them produce more efficient code can actually make the object code slower.

The first expression rule that we will see is essentially an extension of Loop Rule 1 (Code Motion Out of Loops). That rule told us to move computation out of a loop, performing a given computation only once rather than many times. The following expression rule treats the many executions of a program as though they were a loop.

> **Expression Rule 1 -- Compile-Time Initialization:** As many variables as possible should be initialized before program execution.

One application of this rule is usually called "constant propagation"; if we have the statement

```
const X = 3; Y = 5;
```

in a Pascal program, then the compiler should replace an instance of X*Y later in the program by the constant 15. As a more substantial application of this rule, we can return to Fragment K3, which classified characters by using a large table. Peterson [1980] initializes the table in his program by a procedure that contains four do statements and five assignment statements; initializing the table at compile time would result in a slight increase in the speed of the program, and, more importantly, less code.

This application is typical of a much larger class of applications of Expression Rule 1. Many programs spend much time reading in data that is unchanged between runs and processing it into tables that are then used for the particular run. Much of the processing and reading time can be avoided by building a new program that processes the input data into an intermediate file which can then be read and processed more quickly. Laird [1981] used this technique in a program that spent 120 seconds processing data that was unchanged from run to run, and then less than three seconds processing the data for the given run. A new program processed the unchanged data into an intermediate file (represented in the packed form we studied in Time-For-Space Rule 1) in 120 seconds; his primary program could then read that intermediate file in less than a second. Thus the time required by his program dropped from over 120 seconds to less than four, for a speedup of over a factor of thirty.

The next expression rule arose in a slightly altered form as Logic Rule 1.

> **Expression Rule 2 -- Exploit Algebraic Identities:** If the evaluation of an expression is costly, replace it by an algebraically equivalent expression that is cheaper to evaluate.

For instance, it would often be beneficial to replace the expression "ln(A) + ln(B)" (where ln is the subroutine for computing natural logarithms) by the algebraically equivalent expression "ln(A*B)". This particular example raises the important point that sometimes we write expressions in a certain way because of the properties of real arithmetic on digital computers, and often the laws of algebra do not apply. Simple applications of Expression Rule 2 are easy to mechanize, and many compilers do quite well at this. (Some Fortran compilers, for instance, will observe that if the two terms SIN(X)**2 and COS(X)**2 are added together in an expression, then they can be replaced at compile time by the constant 1!). With this rule, just as with Logic Rule 1, it is important to avoid "optimizations" that lead to our compilers producing slower code.

An important application of Expression Rule 2 is recognizing when special cases of an expression can be evaluated in a more efficient way than by applying the general rule. For instance, instead of evaluating X**2 by a general routine for raising powers, we could merely multiply X by itself (note that this transformation might actually be achieved by Procedure Rule 1 -- Collapsing Procedure Hierarchies). Similarly, on binary machines we can efficiently multiply or divide by powers of two by shifting left or right (as long as the number is in a proper range, which is often easy to verify). Knuth [1973, p. 408] used this method to reduce the run time of a binary search program from $\sim 26 \log_2 N$ to $\sim 18 \log_2 N$.

An important kind of algebraic identity is quite useful in increasing the speed of loops. Consider, for instance, the loop "for I := 1 to N do ...", and suppose that we evaluated the expression I*J inside

the loop (where J is an integer). The straightforward way of implementing this requires a multiplication in each iteration of the loop, while a more clever implementation can keep the last value of the expression and just add J to get the next. This operation is an instance of a technique called *strength reduction* and is intimately related to techniques for manipulating *induction variables*. For a discussion of these techniques in a general setting, see Aho and Ullman [1977, Section 12.2].

Expression Rule 3 helps us avoid redundant work in evaluating expressions.

> **Expression Rule 3 -- Common Subexpression Elimination:** If the same expression is evaluated twice with none of its variables altered between evaluations, then the second evaluation can be avoided by storing the result of the first evaluation and using that in place of the second evaluation.

This is exactly the rule we used to achieve the (unexpectedly small) time savings as we transformed Fragment A1 to Fragment A2. This rule can be viewed as an application of Space-For-Time Rule 2, where we are now storing recomputed results rather than precomputed results; we already saw an application of this rule in Loop Rule 1, as we eliminated the subexpression from a loop that was common to all iterations. Many compilers are very good at recognizing and exploiting common subexpressions (see, for instance, Aho and Ullman [1977, Sections 14.2 and 15.6] or Wulf *et al* [1975]), and this technique is usually best left to the compiler.

The next expression rule is very similar to Loop Rule 6 (which encouraged us to let similar loops share their overhead of loop control).

> **Expression Rule 4 -- Pairing Computation:** If two similar expressions are frequently evaluated together, then we should make a new procedure that evaluates them as a pair.

The hope of the above rule is that "two can live as cheaply as one". Knuth [1971, p. 116] observes that while sine and cosine each require 110 time units to evaluate (where a time unit is approximately .7 microseconds on an IBM System/360 Model 67), a routine that returns both the sine and the cosine of a value requires only 165 time units. We therefore get the second trigonometric function for just half price, after we have purchased the first! A similar phenomenon occurs in finding the minimum and maximum elements of an N-element set; while either alone requires N - 1 comparisons to find, both together can be found in at most 3N/2 comparisons (see Knuth [1973, Exercise 5.3.3-25]). Note that this rule is similar in spirit to Procedure Rules 1 and 2 -- in both cases we reorganize the division of the computation into procedures.

The final expression rule speeds up a program by utilizing the parallelism inherent in the word width of the underlying machine; it is therefore related to Procedure Rule 5 (Parallelism).

> **Expression Rule 5 -- Exploit Word Parallelism:** Use the full word width of the underlying computer architecture to evaluate expensive expressions.

This rule is used in the implementation of set operations as bit strings; when we OR two 32-bit sets together giving as output their 32-bit union, we are performing 32 operations in parallel. There are many applications of this rule; almost all of them, however, are best implemented in assembly language. Reingold, Nievergelt and Deo [1977, Section 1.1] nicely describe several algorithms for counting the number of one bits in a computer word; the first algorithm takes B operations in a B-bit word, while the final algorithm takes only approximately $\log_2 B$ operations. Beeler, Gosper, and Schroeppel [1972] describe a number of similar algorithms for a host of problems on computer words; see especially Items 167, 169, and 175. All of these techniques have the same motivation in Procedure Rule 5: there is parallelism inherent in the word widths of the underlying data paths and registers, and the algorithms go out of their way to make sure that none of it is wasted in any operation.

3.4. Summary of the Rules

In the previous subsections of this section we have seen a number of tools for writing efficient code. In this subsection we shall take a brief moment to review those tools and consider their application as a collection.

The most important thing to remember about their application is the point made in Subsection 3.1: we should almost never apply them in the first design of a program, and rarely apply them to clean although somewhat slow code to yield messy but fast code. Knuth's [1971] statistics make this point quantifiable: he observed (and many have since verified) that in most programs about four percent of the code of most programs usually accounts for fifty percent of the run time. We saw in Subsection 3.1 that this fact helps us concentrate our search for efficiency on that critical four percent; we will now consider two deeper implications of the statistics.

- If our goal in increasing the efficiency of the program is to increase the speed as much as possible in a short period of time, then we should identify the largest time sinks and focus our work on those. Using the rules of this paper often enables us to decrease the time of many of the critical regions by up to an order of magnitude. If we achieve such a speedup, then we will have increased the efficiency of our system by a factor of about two (because the remaining 96 percent of the code accounted for about half of the original run time).
- If our goal is to increase the efficiency of our entire system by a factor of ten, then it will be very difficult to accomplish. The first step is the one mentioned above: we identify the initial time sinks and reduce those; this might reduce the total system time by a factor of two. We then instrument the resulting program, and try to reduce the time spent by the new resource sinks, and iterate. The hope of this strategy is that the system will have a very jagged time profile at each iteration; that is, we hope that no matter how many improvements we have made, that most of the run time is still concentrated in a small percentage of the code.

The above points put our rules in context: they are often quite useful to reduce dramatically the run time of a costly piece of code (say, by an order of magnitude). Several such changes can have a tremendous effect on the speed of an entire system (say, up to a factor of two), but it is difficult to have an impact of much more than that.

Now that we know the effect these rules can have on an entire system, we should consider the mechanics of using them. There are three steps in applying the rules.

1. **Identify the code to be changed.** The above discussion showed that we should identify the code to be changed by monitoring the program and working on the parts that are taking the majority of the time.

2. **Choose a rule and apply it.** Once we know what code to change we see if any of our rules can be used to change it. The rules have been presented in groups as they relate to different parts of programs; we should therefore identify whether the expense of our program is going to data structures, loops, logic, procedures or expressions, and then search in the appropriate list for a candidate rule. When we apply a rule we should make sure that the application preserves the correctness of the program; this is usually done by applying the spirit, if not the actual formulas, of program verification.

3. **Measure the effect of the modification.** The first transformation we saw in Section 2 (removing the common subexpression from Fragment A1) was typical of many changes we make: it appeared that it would increase the program's speed by a factor of two but in fact it gave less than a three percent improvement. Even if we believe that we understand the effect of a transformation by reasoning alone, it is usually quite beneficial to support that analysis with observation; we often find that we are quite mistaken!

Each of the above steps plays a crucial role in yielding a correct and efficient program, and none of the steps should be skipped in applying the rules. The rules themselves are summarized in Table 2; we will now briefly discuss each class of rules as a set.

Modifying Data Structures

Trading Space-For-Time
1. Data structure augmentation
2. Store precomputed results
3. Caching
4. Lazy evaluation

Trading Time-For-Space
1. Packing
2. Interpreters

Modifying Code

Loops
1. Code motion out of loops
2. Combining tests
3. Loop unrolling
4. Transfer-driven loop unrolling
5. Unconditional branch removal
6. Loop fusion

Logic
1. Exploit algebraic identities
2. Short-circuiting monotone functions
3. Reordering tests
4. Precompute logical functions
5. Boolean variable elimination

Procedures
1. Collapsing procedure hierarchies
2. Exploit common cases
3. Coroutines
4. Transformations on recursive procedures
5. Parallelism

Expressions
1. Compile-time initialization
2. Exploit algebraic identities
3. Common subexpression elimination
4. Pairing computations
5. Exploit word parallelism

Table 2. Summary of the rules.

If monitoring the program shows that a certain data structure is a primary user of a scarce resource, then we should use the rules of Subsection 3.2 to make that structure more efficient. At the time we modify the structure we should know whether space or time is dearest, and then trade the cheaper commodity for the more expensive. Although each rule was expressed in terms of trading one resource for the other, by reversing each we can effect the trade in the opposite direction.

If we find that the primary resource bottleneck is the time spent in a certain loop (as we often do), then we should carefully apply the loop rules of Section 3.3.1 to remove every possible piece of excess baggage from the loop. Although each of the six loop rules typically reduces the run time only by ten or twenty percent, when they are carefully applied together to a single loop it is not uncommon to see speedups of factors of three or more.

The logic rules of Section 3.3.2 should be brought to bear when the time spent in evaluating program state is in the system bottleneck. Logic Rules 1 and 5 sometimes shave a very small percentage from the system run time, but sometimes fool compilers into producing slower object code. Logic Rules 2 and 3 are applicable less often, but can sometimes be used to cut in half the time

required by some loops. Logic Rule 4 is perhaps the most powerful of all: we can frequently eliminate most of the time spent in evaluating a logical function simply by precomputing all possible outcomes.

The procedure rules of Section 3.3.3 make the most global of the changes that we have seen. Procedure Rules 1 and 2 are the most frequently applied and often yield substantial speedups. Procedure Rules 3 and 4 are extremely powerful in certain special cases (if we have a multiple-pass or recursive program). Procedure Rule 5 is the most nitty-gritty: if we know a great deal about the parallelism in the underlying hardware, then we can exploit it.

The expression rules of Section 3.3.4 should usually be brought to bear only as a last resort. They are often done by a compiler, they are perhaps the easiest to apply incorrectly, they rarely yield enormous speedups, and their application can result in slower object code. Occasionally, though, they can be used to shave ten percent here or twenty percent there.

4. A Survey of Related Work

In this section we will briefly survey some of the work that has been done on topics related to writing efficient code.[21] The content of this section will follow the trichotomy discussed in Section 1: we will first examine work related to the "high end" of algorithms and data structures, then work related to the "low end" of optimizing compilers, and finally work at the level of writing efficient code.

Courses in data structures and algorithms are now well-established in most curricula in computer science. Because of the central role played by data structures, expert programmers should be intimately familiar with the material in data structures texts such as Standish [1980]. Knuth [1968, 1973] is an excellent source for all aspects of data structure selection and implementation. Algorithm design and analysis is the subject of several recent texts, including Knuth [1973], Aho, Hopcroft and Ullman [1975], Goodman and Hedetniemi [1977], Reingold, Nievergelt and Deo [1977] and Baase [1978]. For survey articles on the subject, see Lewis and Papadimitriou [1978] and Bentley [1979].

At the other end of the spectrum, there has been a great deal of research on the principles underlying optimizing compilers. For general discussions of these principles, the reader should see Aho and Ullman [1977], Schaeffer [1975], or Waite [1974]. Wulf *et al* [1975] show how these principles can be brought together to form a compiler that produces object code that rivals that of the best assembly language coders. Wulf and Shaw [1980] address the issue of the impact of language

[21] A related topic that we will not investigate in detail is techniques of building efficient hardware. It is interesting to observe that almost all of the rules we have seen for programs have analogs in the domain of hardware.

design decisions on the speed of compiled programs.

There is much less written on the activity of writing efficient code. One line of research at this level goes under the name of "source-to-source program transformations". The goal of that research is to describe precisely a set of transformations at the source language level that preserve program equivalence but increase program speed. The insistence on precise description of transformations has resulted in a set of transformations much more accurately defined than those in this paper, but unfortunately also less powerful. Examples of program transformations can be found in Burstall and Darlington [1977], Darlington and Burstall [1976], Loveman [1977], Scherlis [1980] and Standish *et al* [1976]. Nelson [1981] and Sproull [1981] both use informal source-to-source transformations to write very efficient code; Nelson uses many of the techniques in this paper to reduce the cost of a remote procedure call by over a factor of thirty.

Some programming texts include discussions of writing efficient code. For instance, Goodman and Hedetniemi [1977, Section 4.2] discuss precisely this topic under the title of "implementation efficiency". They mention aspects of Loop Rules 1, 3 and 6, Logic Rule 3, and Expression Rules 2 and 3. Kernighan and Plauger [1976, 1978] describe a number of issues related to writing efficient code; these may be found in the index under the headings "algorithm", "efficiency", "optimization", "running time", and "time complexity", among others.

Jalics [1977] and Smith [1978] are nice introductions to issues of efficiency in data processing software systems. Jalics discusses a number of "high end" issues such as file organization and "low end" issues such as the efficiency of various language constructs. He mentions several of the rules that we have seen in this paper, including Space-For-Time Rule 3, Loop Rule 1, Logic Rule 3, and Procedure Rules 1 and 2. He also provides many concrete examples of increasing the efficiency of real data processing systems. Smith covers in detail many of the important issues in system efficiency. She discusses the "high end" issues of reducing costly input/output operations, reducing paging, and data structure selection, and the "low end" issues of compiler optimization. The techniques of writing efficient code that she discusses includes Loop Rules 1, 2, 3 and 6, Procedure Rule 2, and Expression Rule 2. She addresses a number of important points in applying efficiency improvements, such as selecting the programs to modify and the management of efficiency improvements. She also discusses in detail the improvement of the efficiency of several real systems.

In the transformations of Subsection 2.1 we made changes to the program that we thought would improve performance and then measured the new program to calculate the performance improvement. It would be much more desirable to have an analytic tool that would predict the

performance improvement. One such tool has been described by Shaw [1979]; she gives a set of measurements of the clock times that various Pascal instructions require (on the same compiler and machine used in the experiment of Subsection 2.1). A comparison of the run times derived by her method with the empirically observed run times for seven of the fragments can be found in Appendix I. The analytic predictions were consistently less than the observed times, varying from 94.4% to 73.5% of the observed times. Having these performance statistics for a particular compiler/machine pair allows us to fine-tune our code for that given system with a little analysis replacing a lot of measurement.

There is a treasure-house of information about writing efficient code in the works of Donald Knuth. His series of textbooks (Knuth [1968, 1969, 1973]) are classics in the fields of algorithms and data structures, and are also laden with both examples and principles of writing efficient code. His empirical study of Fortran programs (Knuth [1971]) gave a precise perspective to the activity of writing efficient code; we saw in Section 3.1 that his data allows us to ignore efficiency most of the time and concentrate on it when it really matters. That paper also contains seventeen detailed examples of efficient compilations of fragments of Fortran programs. Knuth [1974] is an excellent study of the question of how programming language design and programming methodologies relate to writing efficient code. It is interesting to note that of the twenty-seven efficiency rules in this paper, fifteen refer explicitly to the works of Knuth! In addition to his own works, many of the Stanford Ph.D. theses and other papers of Professor Knuth's students are invaluable studies in writing efficient code; we have already referred to Sedgewick [1975, 1978], Mont-Reynaud [1976], and Chris Van Wyk.

5. Conclusions

The thesis of this paper is that there is an activity, which we have called writing efficient code, that is an essential part of the engineering activity of producing efficient software. That activity is somewhere "above" the level of optimizing compilers and "below" the level of selecting algorithms and data structures. The goal of this paper has been to equip the reader with the fundamental tools of writing efficient code. To this end, in Section 2 we studied in detail one example that arose in a real application. In Section 3 we took a more systematic view of the endeavor and saw both a set of techniques and the context in which those techniques should be applied. Section 4 then provided a brief survey of work related to writing efficient code.

To give more context to the process of writing efficient code, I propose the following as five steps that are essential in a methodology of building efficient software.

1. The most important issues in the lifetime of a large system are a clean design and

implementation, useful documentation, and a maintainable modularity. The first step in the programming process should therefore be to write the program with a clean design and implementation.

2. If the overall system performance is not satisfactory, then the programmer should monitor the program to identify where the scarce resources are being consumed. This usually reveals that most of the time is used by a few percent of the code.

3. Proper data structure selection and algorithm design are often the key to large reductions in the running time of the expensive parts of the program. One should therefore revise the data structures and algorithms in the critical parts of the code.

4. If the performance of the critical parts is still unsatisfactory, then use the techniques of writing efficient code to recode them. The original code should usually be left in the program as documentation.

5. If additional speed is still needed, then there are courts of last resort, including assembly code, microcode, and special-purpose hardware design.

It is important to keep the techniques of writing efficient code in proper context. If they are used inappropriately, such as in the premature optimization of unmonitored code, then they can reduce a clean system to an incomprehensible mess and sometimes decrease performance as well. On the other hand, when they are applied sparingly under the keen eye of an experienced software craftsman, they can play an important role in building an efficient software system.

Acknowledgments

This paper has benefitted greatly from the contributions of several dozen individuals. The careful comments of Al Aho, Steve Johnson, Elaine Kant, Brian Kernighan, Ed McCreight, John McDermott, Al Newell, Joe Newcomer, Guy Steele, Chris Van Wyk and Bill Wulf have been particularly helpful. Elaine Rich and Bill Trosky went beyond the call of duty in transporting the programs of Section 2 to various computer systems.

References

Aho, A. V. [1980]. Private communication of A. V. Aho of Bell Telephone Laboratories, Murray Hill, NJ, December 1980.

Aho, A. V. and J. D. Ullman [1977]. *Principles of Compiler Design*, Addison-Wesley, Reading, MA.

Aho, A. V., J. E. Hopcroft and J. D. Ullman [1974]. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.

Baase, S. [1978]. *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, Reading, MA.

Baskett, F. [1978]. "The best simple code generation techniques for WHILE, FOR and DO loops," *SIGPlan Notices 13*, 4, April 1978, pp. 31-32.

Beeler, M., R. W. Gosper and R. Schroeppel [1972]. HAKMEM, Artificial Intelligence Memo No. 239, Massachusetts Institute of Technology, February 1972.

Bentley, J. L. [1979]. "An introduction to algorithm design," *IEEE Computer Magazine 12*, 2, February 1979, pp. 66-78.

Bentley, J. L., M. G. Faust and F. P. Preparata [1981]. "Approximation algorithms for convex hulls," to appear in *Communications of the ACM*.

Bentley, J. L. and J. B. Saxe [1980]. "An analysis of two heuristics for the euclidean travelling salesman problem," *Eighteenth Annual Allerton Conference on Communication, Control and Computing*, October 1980, pp. 41-49.

Bergeron, R. D. and H. Bulterman [1975]. "A technique for evaluation of user systems on an IBM S/370," *Software--Practice and Experience*, 5, pp. 83-92.

Bird, R. S. [1980]. "Tabulation techniques for recursive programs," *Computing Surveys 12*, 4, pp. 403-417.

Burstall, R. M. and J. Darlington [1977]. "A transformation system for developing recursive programs," *Journal of the ACM 24*, 1, pp. 44-67, January 1977.

Brailsford, D. F. *et al* [1979]. "Run-time profiling of Algol 68-R programs using DIDYMUS and SCAMP," *SIGPlan Notices 12*, 6, June 1977, pp. 27-35.

Brooks, F. P. [1975]. *The Mythical Man Month: Essays in Software Engineering*, Addison-Wesley, Reading, MA.

Darlington, J. and R. M. Burstall [1976]. "A system which automatically improves programs," *Acta Informatica 6*, pp. 41-60.

Dongarra, J. J. and A. R. Hinds [1979]. "Unrolling loops in FORTRAN," *Software--Practice and Experience 9*, pp. 219-226.

Fitch, G. P. [1977]. "Profiling a large program," *Software--Practice and Experience 7*, pp. 511-518.

Friedman, J. H., J. L. Bentley and R. A. Finkel [1977]. "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software 3*, 3, September 1977, pp. 209-226.

Goodman, S. E. and S. T. Hedetniemi [1977]. *Introduction to the Design and Analysis of Algorithms*, McGraw-Hill, New York, NY.

Jalics, P. J. [1977]. "Improving performance the easy way," *Datamation*, April 1977, pp. 135-148.

Kernighan, B. [1981]. Personal communication of B. Kernighan of Bell Telephone Laboratories, Murray Hill, NJ, February 1981.

Kernighan, B. and P. J. Plauger [1976]. *Software Tools*, Addison-Wesley, Reading, MA.

Kernighan, B. and P. J. Plauger [1978]. *The Elements of Programming Style*, Second Edition, McGraw-Hill.

Knuth, D. E. [1968]. *The Art of Computer Programming, volume 1: Fundamental Algorithms*, Addison-Wesley, Reading, MA.

Knuth, D. E. [1969]. *The Art of Computer Programming, volume 2: Seminumerical Algorithms*, Addison-Wesley, Reading, MA.

Knuth, D. E. [1971]. "An empirical study of FORTRAN programs," *Software--Practice and Experience 1*, 2, pp. 105-133.

Knuth, D. E. [1973]. *The Art of Computer Programming, volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA.

Knuth, D. E. [1974]. "Structured programming with goto statements," *Computing Surveys 6*, 4, December 1974, pp. 261-301.

Kulsrud, H. E., R. Sedgewick, P. Smith and T. Szymanski [1978]. Partition sorting on CRAY-1, SCAMP Working Paper No. 7/78, Institute for Defense Analyses, Princeton, NJ, September 1978.

Laird, J. [1981]. Private communication of J. Laird of Carnegie-Mellon University, March 1981.

Lewis, H. and C. H. Papadimitriou [1978]. "The efficiency of algorithms," *Scientific American*, January 1978, pp. 97-109.

Loveman, D. B. [1977]. "Program improvement by source-to-source transformation," *Journal of the ACM 24*, 1, January 1974, pp. 121-145.

Matwin, S. and M. Missala [1976]. "A simple, machine independent tool for obtaining rough measures of Pascal programs," *SIGPlan Notices 11*, 8, August 1976, pp. 42-45.

Mont-Reynaud, B. [1976]. Removing trivial assignments from programs, Stanford University Computer Science Department Report STAN-CS-76-544, Stanford, California, March 1976.

Moon, D. A. [1981]. Private communication of D. A. Moon of Massachusetts Institute of Technology, March 1981.

Morris, R. [1978]. "Counting large numbers of events in small registers," *Communications of the ACM 21*, 10, October 1978, pp. 840-842.

Nelson, B. J. [1981]. Remote procedure call, Ph.D. Thesis, Carnegie-Mellon University, June 1981.

Newell, A. [1981]. Private communication of A. Newell of Carnegie-Mellon University, March 1981.

Peterson, J. L. [1980]. *Computer Programs for Spelling Correction: An Experiment in Program Design*, Springer-Verlag, New York, NY.

Polya, G. [1945]. *How To Solve It*, Princeton University Press, Princeton, NJ.

Reddy, R. and A. Newell [1977]. "Multiplicative speedup of systems," in *Perspectives on Computer Science*, A. K. Jones (ed.), pp. 183-198, Academic Press, New York, NY.

Reghbati, H. K. [1981]. "An overview of data compression techniques," *IEEE Computer Magazine 14*, 4, April 1981, pp. 71-75.

Reingold, E. M., J. Nievergelt and N. Deo [1977]. *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ.

Ritchie, D. M. and K. Thompson [1978]. "The UNIX time-sharing system," *Bell System Technical Journal* 57, 6, pp. 1905-1930, July-August 1978. (An earlier version is in *Communications of the ACM 17*, 7, pp. 365-375, July 1974.)

Russell, R. D. [1978]. "The PDP-11: A case study of how not to design condition codes," *Proceedings of the Fifth Annual Symposium on Computer Architecture*, pp. 190-194, IEEE and ACM.

Satterthwaite, E. H. [1972]. "Debugging tools for high level languages," *Software -- Practice and Experience* 2, 3, July-September 1972, pp. 197-217.

Schaefer, M. [1973]. *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, Englewood Cliffs, NJ.

Scheifler, R. W. [1977]. "An analysis of inline substitution for a structured programming language," *Communications of the ACM 20*, 9, September 1977, pp. 647-654.

Scherlis, W. [1980]. Expression procedures and program derivation, Ph.D. Thesis, Stanford Computer Science Report STAN-CS-80-818, Stanford, CA, August 1980.

Sedgewick, R. [1975]. Quicksort, Ph.D. Thesis, Stanford Computer Science Report STAN-CS-75-492,

Stanford, CA, May 1975.

Sedgewick, R. [1978]. "Implementing Quicksort programs," *Communications of the ACM 21*, 10, October 1978, pp. 847-857.

Shaw, M. [1979]. A formal system for specifying and verifying program performance, Carnegie-Mellon University Computer Science Technical Report CMU-CS-79-129, June 1979. (A preliminary version of the material in this report can be found in Wulf, Shaw, Hilfinger and Flon [1981, Section 6.5].)

Shaw, M. and W. A. Wulf [1980]. "Toward relaxing assumptions in languages and their implementations," *SIGPlan Notices 15*, 3, March 1980, pp. 45-61.

Sites, R. L. [1978]. "Programming tools: statement counts and procedure timings," *SIGPlan Notices 13*, 12, December 1978, pp. 98-101.

Smith, C. [1978]. "Methods for improving the performance of applications programs," *Computer Measurement Group Transactions 22*, December 1978.

Smith, C. [1980]. "Consider the performance of large software systems before implementations," *Proceedings of the Computer Measurement Group 11*, Boston, MA, December 1980.

Sproull, R. [1981]. "Program transformations and Bresenham's algorithm," in preparation.

Standish, T. A. *et al* [1976]. The Irvine Program Transformation Catalog, Technical Report, Department of Information and Computer Science, University of California at Irvine.

Standish, T. A. [1980]. *Data Structure Techniques*, Addison-Wesley, Reading, MA.

Steele, G. L., Jr. [1981]. Private communication of G. L. Steele, Jr. of Carnegie-Mellon University, April 1981.

Waite, W. M. [1974]. "Code generation," in *Compiler Construction: An Advanced Course*, F. L. Bauer and J. Erckel (eds.), pp. 549-602, Springer-Verlag, New York, NY.

Wulf, W. A. [1981]. Private communication of W. A. Wulf of Carnegie-Mellon University, April 1981.

Wulf, W. A. *et al* [1975]. *Design of an Optimizing Compiler*, American Elsevier Publishing Company, Inc., New York, NY.

Wulf, W. A., M. Shaw, P. Hilfinger and L. Flon [1981]. *Fundamental Structures of Computer Science*, Addison-Wesley, Reading, MA.

I. Details of the Pascal Programs

In Section 2 we studied a sequence of Pascal code fragments for producing Nearest Neighbor Traveling Salesman Tours. This appendix contains some details about the Pascal programs that contained those fragments. In Section 2 we noted that the compiler used for these experiments was the Pascal compiler on the Carnegie-Mellon University Computer Science Department PDP-KL10 (Arpanet Host CMUA), which is a derivative of the Hamburg Pascal compiler. It performs very little optimization, so the computation we see expressed in the source code is very similar to that in the resulting object code. All tests were run with the array bounds checking and debugging features turned off.

In Section 2 we assigned a running time to each fragment of the form K_1N^2 microseconds. Such a time is, of course, merely an approximation; the actual run time of Fragments A1 through A5 is actually of the form

$$K_1N^2 + K_3NH_N + K_4N + o(N),$$

while for Fragments A6 through A9 the run time has the form

$$K_1N^2 + K_2N^{3/2} + K_3NH_N + K_4N + o(N).$$

Because it would be rather laborious and not terribly instructive to calculate all the values of the various K's, we used instead the simple approximation to K_1 of dividing the total run time in microseconds by N^2. Table 3 shows the run times of several experiments; the rows represent the nine fragments and the columns represent values of N from 100 to 1000. Each entry consists of the average run time in seconds over the 95 percent confidence interval for the run time in seconds over the estimated value of K_1.

N Program #	100	200	400	800	1000
1	.4845 (.0075) 48.4	1.8786 (.0060) 47.0			
2	.4533 (.0047) 45.3	1.8249 (.0044) 45.6			
3	.2392 (.0046) 23.9	.9707 (.0030) 24.2			
4	.2118 (.0035) 21.2	.8465 (.0033) 21.2			
5	.1425 (.0051) 14.25	.5578 (.0034) 13.94	2.2241 (.0166) 14.01		
6	.0945 (.0047) 9.45	.3614 (.0049) 9.04	1.3857 (.0036) 8.66	5.3332 (.0179) 8.33	8.2456 (.0159) 8.25
7	.1066 (.0024) 10.7	.3683 (.0021) 9.21	1.3160 (.0058) 8.22	4.9034 (.0125) 7.66	7.5268 (.0113) 7.53
8	.1036 (.0047) 10.4	.3418 (.0040) 8.54	1.2153 (.0041) 7.60	4.5094 (.0081) 7.05	6.9334 (.0139) 6.93
9	.1033 (.0026) 10.33	.3431 (.0041) 8.58	1.2068 (.0047) 7.54	4.4330 (.0077) 6.93	6.7936 (.0081) 6.79

Table 3. Pascal program run times.

Table 3 is ragged due to the extreme expense of using Fragments A1 through A5 on inputs of size greater than 200; recall that Fragment A1 requires approximately 47 seconds on a 1000-point set uniformly distributed on the unit square $[0,1]^2$. Each program was run on ten different data sets for the point sets with 100, 200 and 400 points; on the 800- and 1000-point sets each program was run on five different data sets. The small values of the 95% confidence intervals give us confidence that any statistical error in the table occurs in at most the third digit of the reported times.

Because the above data is only for one compiler on one machine architecture, we might be worried that our estimates of the coefficient K_1 are more artifacts of the particular system than values inherent in the underlying programs. To test this, William J. Trosky transliterated Fragments A1 through A8 from Pascal to C and performed experiments identical to those described above using the C compiler on an HP-1000 computing system. His results are summarized in Table 4; the first column gives the program number, the second column gives the estimate of the coefficient K_1 for the Pascal program (in microseconds), and the third column normalizes that coefficient by dividing it by the coefficient for Fragment A6; the fourth and fifth columns give the corresponding values for the C program. It is satisfying to note that the normalized run times of the Pascal and C programs are remarkably similar. (A C version of Fragment A9 was not available.)

	Pascal Programs		C Programs	
Program #	Coefficient	Normalized	Coefficient	Normalized
1	47.0	5.73	311.8	4.39
2	45.6	5.56	303.8	4.27
3	24.2	2.95	197.4	2.78
4	21.2	2.59	187.2	2.63
5	14.0	1.71	122.0	1.72
6	8.2	1	71.1	1
7	7.5	.91	61.1	.86
8	6.9	.84	59.8	.84
9	6.8	.83	---	---

Table 4. Comparison of run times.

Table 5 presents data on the number of minimal values of CloseDist found by Fragments A1 through A9 (the transformations do not change the expected number of minima). The first column gives N, the number of points in the sets. Tests were run on ten point sets for values of N up to 400, and on five point sets for larger values of N. The second column shows the average number of observed new minima in the point sets, and the next column gives the ninety-five percent confidence interval of that value. The fourth column divides the third column by N; our analysis predicts that to be the sum of the first N harmonic numbers divided by N, or approximately $H_N - 1$, which is shown in the final column. The last two columns show that the observed values were quite close to the predicted values.

N	New Minima	(95% Conf.)	New/N	$H_N - 1$
4	2.4	(.34)	.60	1.083
9	13.4	(1.58)	1.49	1.829
16	36.4	(4.37)	2.28	2.381
25	64.9	(6.37)	2.60	2.816
36	104.2	(8.88)	2.89	3.175
49	161.8	(9.75)	3.30	3.479
64	232.1	(16.47)	3.63	3.744
81	313.4	(13.90)	4.12	3.978
100	422.1	(34.12)	4.22	4.187
144	628.4	(46.96)	4.36	4.550
196	915.3	(47.44)	4.67	4.858
256	1296.6	(64.19)	5.06	5.124
324	1790.2	(92.07)	5.53	5.360
400	2230.1	(90.88)	5.57	5.570
484	2764.4	(136.01)	5.71	5.760
576	3443.4	(148.02)	5.98	5.934
676	4133.2	(316.78)	6.11	6.094
784	4791.4	(180.01)	6.11	6.242
900	5677.4	(237.42)	6.31	6.380

Table 5. Data on new minima.

Table 6 presents data on the efficacy of delaying computing the y-distance in Fragment A6. The first column gives N, the number of points, the second column gives the average number of total y-values calculated during the execution of the program, and the third column gives the 95% confidence interval of the second column. These statistics were gathered on exactly the same point sets used for the statistics of Table 5. The fourth and fifth columns show the average number of y-distances divided by N and $N^{3/2}$, respectively. The fifth column indicates that the total number of y-distances is on the average less than $1.5N^{3/2}$. This fact implies that when M points are left unvisited, $2.25M^{1/2}$ y-distances are calculated on the average (because the sum over all values of M from 1 to N of that value is $1.5N^{3/2}$).

N	Raw	(95% Conf.)	Raw/N	Raw/$N^{3/2}$
4	5.7	(.55)	1.425	.7125
9	27.6	(2.16)	3.033	1.011
16	80.4	(4.58)	5.025	1.256
25	148.5	(4.54)	5.940	1.188
36	268.9	(11.57)	7.469	1.245
49	440.4	(18.73)	8.988	1.284
64	697.5	(21.33)	10.898	1.362
81	978.1	(28.64)	12.075	1.342
100	1347.8	(61.38)	13.478	1.348
144	2368.1	(89.55)	16.445	1.370
196	3753.9	(121.94)	19.153	1.368
256	5666.5	(181.51)	22.135	1.383
324	8462.4	(216.93)	26.119	1.451
400	11150.0	(355.52)	27.875	1.394
484	15016.4	(354.65)	31.026	1.410
576	19619.0	(622.45)	34.061	1.419
676	25196.2	(778.70)	37.272	1.434
784	31452.6	(1205.47)	40.118	1.433
900	38365.0	(676.01)	42.628	1.421

Table 6. Data on y-values tested.

The final experiment on the Pascal fragments compared the empirically observed run times with the analytic estimates given by the technique of Shaw [1979]. The cost of each Pascal operation was determined from Figure 6.3 of Wulf, Shaw, Flon and Hilfinger [1981, p. 165], with the exception of the sqrt function, which was assumed to have a cost of 42.8 microseconds. The results of the experiment are shown in Table 7; the first row of that table says that Shaw's method estimated that Fragment A2 would require $43.05N^2$ microseconds, while it was observed to require $45.6N^2$ microseconds. The final column shows that Shaw's estimates were quite close to the observed times.

Program #	Empirical Time	Analytic Time	Analytic/ Empirical
2	45.6	43.05	.944
3	24.2	21.65	.895
4	21.2	18.85	.889
5	14.0	12.9	.921
6	8.2	7.05	.860
7	7.5	6.1	.813
8	6.8	5.0	.735

Table 7. Analytic predictions of times.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-81-116	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) WRITING EFFICIENT CODE		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jon Louis Bentley		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		12. REPORT DATE April 27, 1981 13. NUMBER OF PAGES 71
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)