OPTIMIZATION OF SPL AND FORTRAN PROGRAMS

John Machin

Campbell & Cook Computer Services

Melbourne, Australia

CONTENTS

1. INTRODUCTION

1.1 Scope

This paper discusses general principles and specific techniques for making SPL and FORTRAN programs use less CPU time on HP3000 computers.

There are three things which affect the CPU speed of a program:

(a) Hardware: Once one has purchased a particular machine, nothing can be done to increase its CPU speed.

(b) Manufacturer-supplied compilers and run-time libraries: The quality of compiler-generated code can have a potent effect on the speed of a program, as can the efficiency or otherwise of the library routines it calls. Later sections of the paper highlight language features and compiler features which necessarily execute slowly, in order that the programmer might avoid them, where possible. Other features, where the necessity is improbable or dubious, are listed for temporary avoidance by the programmer until the suggested compiler enhancements are implemented.

(c) User-written code: Given that suitable algorithms have been chosen, the way in which the user programs them can be of considerable consequence.

1.2 Terminology

To avoid much repetition, abbreviations are used for the names of the four main numerical data types, as shown below:

Abbreviation	Expansion	SPL Equivalent	FORTRAN Equivalent
SI	short integer	integer	integer[*2]
LI	long integer	double	integer*4
SR	short real	real	real
LR	long real	long	double precision

1.3 Basis

Statements made in this paper in relation to code emission by the compilers are based on the Athena (1918) software release.

Timings were obtained on an HP3000 Series III.

1.4 Background: The Machine Architecture

1.4.1 It is of course stating the obvious to say that the HP3000 architecture, at the "machine instruction" level, is scarcely traditional. Many of the features of the machine ease considerably the task of generating machine code for medium-complexity languages such as SPL and FORTRAN. Some features however cause difficulty in emitting code, and this difficulty sometimes lead to suboptimal code.

1.4.2 The low limits on direct data addresses (e.g. DB+255,Q+127) can cause problems with programs requiring many variables. This leads to an extra level of indirection in FORTRAN programs (see "MORECOM" and "Indirect Indirection" later) and surgery in SPL programs.

1.4.3 The BR (branch) instruction post-indexes (like all other memory-reference instructions) when it should pre-index. The code generated for computed GO TO, CASE and SWITCH statements is cute but is about 3 times as much as would otherwise be necessary.

1.4.4 The elegant simplicity of the stack machine is rudely violated by the instructions which handle long reals. Far from the zero-address "stackops" used for arithmetic on other data types, these instructions operate on the addresses of one operand (negate), two operands (compare), or three operands (add, subtract, multiply, divide). Given this startling departure from orthogonality, together with the fact that there are no specific instructions for loading or unloading the stack four words at a time, the number of references to long reals in later sections of this paper should cause little surprise.

1.4.5 The low limits on direct code addresses (e.g. P-255,P+255 for LOAD,BR,MTBA etc and P-31,P+31 for the test-and-branch instructions) cause two problems in code generation:

(a) Is a branch to be direct or indirect?
If indirect, where should the indirect word be placed?

(b) Where should constants be placed?

Some solutions to these problems are good; occasionally however, an indirect branch is used unnecessarily, and a constant is dumped immediately (with a branch around it) instead of being carried forward.

1.5 Background: The Compilers

Neither the SPL compiler nor the FORTRAN compiler is represented by Hewlett-Packard as being an optimizing compiler. However, on perusing a check list of optimization techniques described in the literature, one finds that some of these are used (at least partially) and that the use of others is obviated by the machine architecture.

It is generally accepted that of the languages available on the HP3000, SPL is the most "efficient", closely followed by FORTRAN. Looking at directly comparable features of the two languages, it is found that sometimes SPL generates better code, and other times FORTRAN does. Both compilers would benefit from a cultural exchange.

2. GENERAL OPTIMIZATION PRINCIPLES

2.1 Do it at compile-time

While it may look better to code:

```
     PARAMETER PI = 3.14159...
     ...
     CIRCUM = 2.0 * PI * RADIUS
     (or the SPL equivalent)
```

it will run faster if you write

```
PARAMETER TWOPI = 6.28318...
CIRCUM = TWOPI * RADIUS
```

Neither compiler will simplify expressions involving constants; if you write

```
A:= 1 + 1;
```

that is exactly what you get.

2.2 Do it only as often as needed

2.2.1 Both compilers perform limited elimination of common sub-expressions within a statement. This is done only with respect to subscripted array references. The seemingly wider scope stated by Splinter [1] (expressions in parentheses) does not prevail.

The method used is to load the index register once only with the value required for the offset into the array(s).

Three qualifications must be met for the compilers to perform this optimization:

(a) The subscript expressions must be lexically identical.

(b) The array(s) must not be long real.

(c) (SPL only) The subscript "expressions" can only be simple variables or constants.

Examples of FORTRAN statements where the elimination is done are:

```
A(I + J) = B(I + J) + C(I + J)
D(I + 3, J + 5) = E(I + 3, J + 5)
```

No elimination is done in:

```
A(I + J) = B(J + I)   (not lexically same)
K = (I + J) * (I + J) (not in array reference)
```

2.2.2 Where there are common sub-expressions not fitting the above criteria, time and code-space can be saved by using a temporary variable.

2.2.3 Invariant expressions can be moved outside loops. This seems obvious, almost too trivial to mention, but such cases can be "hidden" by the high level language: see section 3.1.1.

2.3 Do it in the registers

2.3.1 **The HP3000 architecture does not offer quite the same scope (or the necessity!) for optimizing the use of registers as does a machine of the "umpteen general purpose registers" variety.**

2.3.2 The index (X) register has a limited arithmetic capability, and may also be used as temporary storage. Of course the X register is used for several things other than array subscripting, and so it is dangerous to merely equate some name to the X register and write code as though an ordinary variable was involved.

In particular the use of the X register in and around statements involving long reals is perilous.

```
E.g. LONG A, B;
     LONG ARRAY C(0:10), D(0:10);
     INTEGER X = X;
     ...
     A:= B;  << SETS X TO 1
                IF B IS A PROCEDURE ARGUMENT BY
                 REFERENCE>>
     ...
     C(X):= D(X); << CHANGES VALUE OF X >>
```

2.3.3 The assignment operator can be used in SPL to replace a load from memory with a faster stack-duplicate operation.

Instead of

```
C:= D + E;
A:= B + C;
```

write

```
A:= B + (C:= D + E);
```

Warning: for long reals, the compiler generates worse code for the latter case.

2.4 Don't do it at all

2.4.1 When you write

```
FOR I:= J  STEP K UNTIL L DO .....
```

the SPL compiler must generate code which checks whether the loop is to be entered at all.

When you write

```
FOR I:= 0 UNTIL 9 DO .....
```

the loop body must be entered, but the compiler still goes through the motions.

If this latter loop is nested within others, this is a waste of time, which can be saved (together with 2 or 3 words of code) by writing

```
FOR* I:= 0 ....
```

2.5 Don't do it with a procedure call

2.5.1 As is well known, there is a reasonable overhead involved in a call to a procedure in the current segment, and a greater overhead in a call to a procedure in another segment (especially if the called segment is not present in memory).

While splitting a program into procedures or subroutines aids greatly in structuring a program, care should be taken to avoid frivolous procedure calls.

2.5.2 The SPL subroutine, although offering a somewhat more austere environment than a procedure, has the advantage of faster invocation and faster return.

It may sometimes be worth the waste of code space to change a procedure into a subroutine and include it in each calling procedure.

The following "dirty trick" allows a single copy of a subroutine to be shared by several procedures in the same segment:

(a) Include the source of the subroutine ("SHARED'SUB") in an "initialising" procedure.

(b) The initialising procedure should include:

```
   SUB'ADDR:= @SHARED'SUB;
where  SUB'ADDR is global.
```

(c) Invocation of the shared subroutine is done by

```
<< stack arguments, if required >>
TOS:= SUB'ADDR;
ASSEMBLE (SCAL 0);
```

2.5.3 Less obvious than explicit procedure calls (coded by the programmer) are implicit procedure calls generated by the compilers.

Usually there is a rationale for an implicit procedure call: the language feature is not directly supported by the microcode, and in-line code would take up too much space.

2.5.4 In Fortran, all "basic external functions" are handled by procedures. Turning to the "instrinsic functions", which also look like function calls at the source level, we find that some of them are in fact handled by in-line code. Almost all the numerical functions are in this category; among the exceptions are AINT, JDINT, DDINT, AMOD, and the MAX/MIN family.

Of course the MAX and MIN procedures cater for a variable number of arguments; but there is a case for using in-line code for the frequent case of two short integer arguments. The code currently generated for J = MAX (K, L) is

```
LOAD  K
LOAD  L
LDI   2
PCAL  MAX0'
STOR  J
```

whereas in-line code would take only two more words:

```
LOAD  K
LOAD  L
DDUP, CMP
BGE   P + 2
XCH,  NOP
DEL,  NOP
STOR  J
```

In a degenerate case such as J = MAX (K, J), the programmer can instead write

```
IF (J.GT.K)  J = K
```

which is better than the in-line code above, especially if the probability of J exceeding K is low.

2.5.5 In both SPL and FORTRAN, procedures are generally used for exponentiation. The exception is that FORTRAN emits in-line code for the exponentiation of short integers and short reals to the short integer powers 1, 2, 3 and 4.

This means that, contrary to the advice given by H-P [3], it is better to write A**2 than A*A, where A is short (integer or real). The converse applies when A is long (integer or real).

It is curious that optimization of the unlikely expression A**1 is done (not very well: B = A**1 generates e.g.

```
LDD Q + 1, I; NOP, NOP; STD Q + 2, I)
```

whereas no effort is made with the equally unlikely expression A**0.

2.6 Be wary of long reals

2.6.1 As mentioned earlier, the non-stack nature of the instructions for handling long reals makes life hard for compiler writers, occasionally leading to the emission of rather peculiar code.

2.6.2 In the code generated for A = B + C, the FORTRAN compiler loads the address of A last instead of first, then does CAB, CAB to put it into the right place. The SPL compiler avoids this waste of time and code space.

2.6.3 The FORTRAN compiler always uses the MOVE instruction for simple assignments, and usually achieves reasonable code, e.g. nine machine instructions for A(J) = B(J). On the oth?r hand the SPL compiler eschews the MOVE instruction and in desperately trying to simulate 4-word loads and stores, requires 22 instructions to encode A(J):= B(J)!!

2.6.4 As the long real machine instructions work with addresses, not values on the stack, it follows that when expressions force the compilers to put temporary results on the stack (the natural method with other data types), the results will be sub-optimal.

One way of avoiding this is to use variables to hold often used constants.

It is better to write

```
     ONE = 1D0
     ...
     A= A + ONE
     B= B + ONE
```

than

```
     A= A + 1D0
     B= B + 1D0
```

Another method is to simulate the code which would be emitted by a compiler for a 3-address machine:

Instead of

```
     A= B + C * D
```

write

```
     TEMP = C * D
     A= B + TEMP
```

2.7 <u>Avoid data-type conversions</u>

2.7.1 Unlike SPL where the programmer must explicitly code a type conversion, FORTRAN automatically emits type conversions in "mixed-mode" expressions. As these conversions take time and code-space, they should be avoided where possible.

2.7.2 Particularly wasteful is the habit common to some programmers of using short integer constants in an otherwise real expression. The code generated by A= A + 1 runs at about 70% of the speed of that generated by A = A + 1.0.

2.7.3 Conversion from long integer to long real and vice versa requires a procedure call; all others are done in-line.

2.7.4 Intriguing code is generated by the FORTRAN compiler for the conversion from long integer to short integer.

The instructions used to do it on the stack are:

```
DTST, NOP
DASL 16
DEL, NOP
```

followed by a test for overflow which is not done in SPL. The SPL compiler uses only one word of code instead of the 3 above:

```
DTST, DELB.
```

2.8 Multiply instead of dividing

2.8.1 Multiplication is faster than division, so it should be substituted where possible.

2.8.2 Care should be used when replacing division by multiplication when a constant is involved. For example 10.0 can be represented exactly as a short real, but 0.1 cannot be. Coding A=B*0.1 instead of A=B/10.0 may result in loss of precision.

2.9 Exploit special hardware features

2.9.1 Testing for a true or false value is actually reduced by the machine to a test for odd or even. Consequently we may obtain a test for parity in the guise of a "logical" test.

In SPL, the condition

```
I MOD 2 = 1
```

can be re-written as

```
LOGICAL (I)
```

and in FORTRAN, the similar condition can be re-written as BOOL (I).

2.9.2 In SPL the construct I <= J <= K uses the CPRB (compare range and branch) instruction and it is better to use this than

```
I <= J AND J <= K.
```

However, when the lower bound is zero, it is much faster to use

```
    LOGICAL (J) <= LOGICAL (K)
than 0 <= J <= K.
```

2.9.3 The hardware condition code is not affected merely by testing it, nor by branches. Where a logic path is required for each of the results of a comparison (<, +, >), the test does not need to be performed twice.

Instead of

```
    IF I = J THEN ...
    ELSE IF I > J THEN ...
    ELSE ...;
```

write

```
    IF I = J THEN ...
    ELSE IF > THEN ...
    ELSE ...
```

2.9.4 The CMPB (compare bytes) instruction can be induced to report the residual count of uncompared bytes, as well as the addresses of the unequal bytes and the condition code.

A classic case is scanning off trailing blanks. Although it is easier and clearer to write

```
WHILE LEN > 0 AND BUF(LEN-1) = " " DO LEN:= LEN-1;
```

the following code runs much faster:

```
IF BUF(LEN-1) = " " THEN BEGIN
    IF BUF(LEN-2) = BUF(LEN-1), (1-LEN), 0 THEN;
    LEN:= - TOS;
    DDEL;
END;
```

2.10 Avoid unnecessary memory references

The standard practice of the FORTRAN compiler and the normal usage of the SPL programmer is to address arrays indirectly through a pointer. To obtain the contents of an array element, the contents of the pointer cell must first be obtained.

2.10.2 The SPL programmer can avoid this, when sufficient primary address space is available, by coding "=DB" or "=Q" in the array declaration. As only the "zero'th" element of the array must be in the direct address range, one large array may use direct addressing.

2.10.3 The FORTRAN compiler makes no attempt to use direct addressing, even in the simple case when all the local arrays and variables would fit in the range (Q+1, Q+127).

Although optimal allocation of addresses might require n! iterations where n is the number of local arrays and variables, some optimization would be better than none.

3. OPTIMIZATION TOPICS SPECIFIC TO FORTRAN

3.1 Multi-dimensioned arrays

3.1.1 Given the declaration

```
DIMENSION A(3, 4, 5), AX(60)
EQUIVALENCE (A, AX)
```

when the programmer writes

```
    DO 100 K = 1, 5
100 T = T + A(I, J, K)
```

the effect is as though the following had been written:

```
    DO 100 K = 1, 5
100 T = T + AX(((K - 1) * 4 + J - 1) * 3 + I)
```

Obviously part of the offset calculation need be done once only, before the loop is entered.

It is possible to recode this as:

```
    IX = (J - 5) * 3 + I
    DO 100 K = IX + 12, IX + 60, 12
100 T = T + AX(K)
```

Such rewriting is of course error-prone. Having the compiler do it would be preferable, but this is one of the more complex optimization algorithms.

3.1.2 Where one or more of the subscripts is a constant, no cognisance is taken of this, and the effect is ludicrous. For example, when the user writes A(1, 1, 1), the code emitted is as though he had written

```
     AX(((1 - 1) * 4 + 1 - 1) * 3 + 1)
```

instead of AX(1)!!

There is a glaring need for compiler enhancement in this case.

3.1.3 The programmer should evaluate carefully his perceived need for multi-dimensioned arrays. Often such an array can be replaced entirely by an array of one dimension, or an array of one dimension can be equivalenced to it and used for some of the manipulations required (especially those which operate on every element of the array).

3.1.4 One special case is where an array has two dimensions, but in references to the array, one of the subscripts is always constant. This array should be split up into several arrays of one dimension.

For example:

```
DIMENSION CASH(3, 10)
...
NET = CASH(1, J) - CASH(2, J) - CASH(3, J)
```

is better written as:

```
DIMENSION SALES(10), EXPENSES(10), TAX(10)
...
NET = SALES(J) - EXPENSES(J) - TAX(J)
```

This is much clearer as well as much more efficient.

3.2 MORECOM

3.2.1 As mentioned earlier, the limit of 255 on direct DB-relative addressing causes problems with FORTRAN programs with many variables and arrays in COMMON.

The compiler option $CONTROL MORECOM was introduced to alleviate this problem.

When this option is in effect, instead of one word of primary DB being allocated to point to each variable and array in COMMON, one word is allocated to point to each COMMON block. Consequently indexing must be used to address variables within COMMON.

3.2.2 Assume the following declarations:

```
REAL A, B, C
INTEGER I, J
COMMON/BLK/I, A, J, B
```

Without the MORECOM option in effect, the statement C = B will generate code such as:

```
     LDD   DB + 3, I
     STD   Q + 1
```

which is as good as one will ever get.

With MORECOM, however, the same statement will generate:

```
     LDXI  2
     LDD   DB + 0, I, X
     STD   Q + 1
```

The variable B above is at an even offset (4) from the start of the block - this allows the use of double-word indexing in the LDD instruction.

Things can be worse: the variable A is at an odd offset (1) from the start of the block - what happens is this:

```
     LDXI      1
     LOAD      DB + 0, I, X
     INCX,     NOP
     LOAD      DB + 0, I, X
     DTST,     NOP
     STD       Q + 1
```

To add insult to injury, the DTST instruction above is quite redundant.

3.2.3 A similar effect is observable when arrays are used; however when the offset is zero, the (longer) code for the odd case is used!!

Timings are shown below for repeating A(I) = B(I) 5,000 times, where A and B are REAL arrays.

Addressing for A & B	Time (ms)	Ratio to "standard"
No MORECOM	56	1.00
MORECOM,offset even,>0	90	1.61
MORECOM,offset odd	154	2.75

With one-word variables and arrays (INTEGER*2, LOGICAL) and four-word variables and arrays (DOUBLE PRECISION), the parity of the offset is irrelevant; it just takes more code and more time when the MORECOM option is used.

3.2.4 When the use of the MORECOM option is unavoidable, considerable time and code space can be saved by ensuring that the offsets of REAL and INTEGER*4 variables are even. A straightforward way of doing this is to list all the REAL and INTEGER*4 variables and arrays first in the COMMON statement. Offset parity in existing COMMON blocks can be checked by perusing the output of the compiler MAP option.

3.2.5 A better solution would have been to allocate two DB-primary words per common block, one pointing to the 1st word in the block and the other to the 2nd word. Thus any offset in the block would be even with respect to one of the pointers. Unfortunately this would have restricted the maximum number of common blocks to only 127 instead of 254!

3.3 $INTEGER*4

3.3.1 As the manual says, this option forces all integer variables and arrays (other than those explicitly declared INTEGER*2) and all integer constants to be INTEGER*4. It is likely to be used in the early stages of converting a FORTRAN program from another machine.

3.3.2 Use of this option can cause gross waste of code space and data space and considerable increase in execution time.

Consider the following example:

```
      REAL A(10), T
      T = 0.0
      DO 100 I = 1, 10
100   T = T + A(I)
```

Without the use of the option, the variable I and the constants 1 and 10 are INTEGER*2, and the loop is controlled by the efficient MTBA instruction which increments I, tests it against 10, and branches back to the start of the loop, all in one hit.

When $INTEGER*4 is in effect, I, 1 and 10 are INTEGER*4, and the loop is controlled by code which is slow for t o reasons:

(a) each function of the MTBA instruction has to be simulated separately

(b) LI arithmetic is slower than SI.

3.3.3 Once a converted program has been made to run correctly, the following steps should be taken:

(a) Remove the $INTEGER*4

(b) Insert IMPLICIT INTEGER*4 (I-N)

(c) Determine which variables and arrays do not need to be INTEGER*4 and declare them explicitly as INTEGER*2.

(d) Determine which constants need to be INTEGER*4 and append the letter "J" to them.

3.4 Character Variables

3.4.1 The following declarations are used in the discussion:

```
CHARACTER  A*80, B*1(80), C*(10), D*10
EQUIVALENCE (A, B, C), (B(11), D)
INTEGER*2 I, J, N
CHARACTER S*(N), T*1, U*1
```

3.4.2 The code generated for references to character variables of constant size (e.g. A), and for references to constant substrings (e.g. A [11:10]) is generally efficient. Some special cases are:

(a) Assignment oi character variables of size 1 is done by the same sort of code as is used for wider variables.

E.g. T = U is done by

```
LOAD  Q + 1
LOAD  Q + 2
LDI   1
MVB   3
```

instead of

```
LDB  Q + 2, I
STB  Q + 1 , I
```

and T = " " is done by

```
LOAD Q + 1
BR   P + 2
%020000
LRA  P - 1
LSL  1
LDI  1
MVB  PB, 3
```

instead of

```
LDI  %40
STB  Q + 1, I
```

(b) When a shorter string is assigned to a longer string (e.g. C = T), the balance of the longer string is blanked by calling the procedure BLANKFILL'. If one really needs the blanking done, an alternative is:

```
C [1:1] = T
C [2:9] = "         "
```

This will run faster, but takes more code, and if one counts too few blanks, BLANKFILL' will still be called!

(c) When the position part of the substring is not 1, code must be emitted to generate the offset from the start of the variable. Thus it is faster to use D than A[11:10]. The offset is calculated once only (using rather cunning code) in the subprogram prologue; the trade-off is the extra word taken for a pointer to the start of D.

3.4.3 If the size of a string is variable (e.g. S), or variable substrings are used (e.g. A[I:J]), external procedures are used not only to perform the operation required, but also for paternalistic error checking which cannot be turned off.

Where only the position part of a substring is variable (e.g. A [I:10]), it may be possible to avoid the dreaded PCALs by equating to a character array. E.g. it is better to use B(I) than A[I:1].

3.4.4 When reference is made to a character array of more than one dimension, the element offset is calculated as described in Section 3.1.1; then this is multiplied by the element size to get the byte offset. (The multiplication uses slow unsigned arithmetic (LMPY, DELB), because the byte offset could exceed 32K). Thus offset calculation for a character array of n dimensions is as complicated as that for an integer (say) array of n + 1 dimensions. The exception is where the character element size is 1; multiplication by 1 is avoided.

3.5 Indirect Indirection

3.5.1 A Fortran main program or subprogram can run more slowly than expected if it has many local variables.

All local variables and arrays must be addressed relative to the Q register, and the direct range is +1 to +127.

Space in this range is allocated according to the following priorities:

(1) One word as a pointer to each array, character variable or variable mentioned in a DATA statement (Compilation will fail if there are more than 127 words required.)

(2) One word for each INTEGER*2 and LOGICAL variable.

(3) One word as a pointer for each DOUBLE PRECISION variable.

(4) Two words for each INTEGER*4 and REAL variable.

3.5.2 Once the total of the above allocations exceeds 127, one location (typically Q + 1) is allocated as a pointer to an "extension area". Then the remaining variables are addressed as though the extension area were an array and they were elements of the array. The offset into this pseudo-array is shown in the output from the compiler MAP option.

The effect on the execution speed is apparent from the code generated:

Variable J's address in the MAP is Q + 23.

The code required for J = 0 is

ZERO, NOP; STOR Q + 23.

Variables K's address in the MAP is Q + 1, I, %11.

The code required for K = 0 is

ZERO, NOP; LDXI %11; STOR Q + 1, I, X.

3.5.3 Fortunately the problem does not seem to be compounded by parity problems with REAL and INTEGER*4 variables (as it is with MORECOM); there is no language-imposed ordering requirement (as there is with variables in COMMON) and in observed cases the compiler allocates the two-word variables first, so that they are at even offsets in the pseudo-array.

3.5.4 Unfortunately when some variables will fit in the primary area and others would not, the compiler has no way of knowing which will be used more frequently than others, so it can happen that a variable used as a DO index can end up in the secondary area.

Several things can be done by the programmer to alleviate the problem:

(1) Split the subprogram.

(2) Use EQUIVALENCE to equate less frequently used variables to elements of arrays (one array for each data-type).

(3) Put some variables into COMMON. If MORECOM is required, the <u>less</u> frequently used variables should be put into COMMON. While this is the easiest to write, it will typically waste stack space.

3.6 <u>$CONTROL BOUNDS</u>

Use this option, if you must, while debugging, but be sure to remove it for production running.

This option generates a procedure call for each subscripted array reference.

3.7 The Formatter

3.7.1 We have the authority of Splinter [1] saying that the implementation of the Formatter is inefficient; on top of this it should be realised that each READ or WRITE statement involves an overhead of two procedure calls, together with one procedure call for each variable, each array, and each iteration of a DO-implied list.

3.7.2 Unformatted I/O merely avoids the conversion to external form; it does not avoid all those procedure calls. It is often worth the effort involved in using the file system instrinsics instead of unformatted I/O.

3.7.3 Further details on the diseconomy of using the Formatter are given by Green [4].

3.8 $CONTROL INIT

3.8.1 Use of this compiler option causes all local variables and arrays to be cleared to zero during the subprogram prologue. The code used to do this is quite efficient: one block move is done to clear variables and arrays with fixed bounds, and another is done if there are any arrays with dynamic bounds, to clear them.

3.8.2 It follows that if any arrays, or more than a few variables, are to be cleared at the start of a subprogram, it is much better to use $CONTROL INIT than to write explicit statements, especially for the arrays.

3.9 Use of SPL Routines

3.9.1 SPL allows access to all the features of the machine, and can thus be used to perform operations which can be expressed only clumsily, if at all, in FORTRAN. As access to an SPL routine from FORTRAN necessitates a procedure call, the time saved within the SPL routine needs to be worthwhile.

3.9.2 As recommended in [3], a frequent choice for an excursion into SPL is usage of the MOVE and MVB instructions for initializing or assigning arrays en masse. Appendix A shows an example of how to obtain leverage from the investment in a PCAL by allowing many arrays to be cleared at once.

4. MISCELLANEOUS

4.1 Slow programs: prevention

4.1.1 Ensure that the best data structures and algorithms have been chosen; it is pointless to "bit-twiddle" with the X register if you are bubble-sorting a 10,000 element array.

4.1.2 When writing the program, bear in mind the language features which can cause a problem, and avoid them in frequently executed code.

4.1.3 Draw a diagram showing which procedures call which other procedures, and arrange the segmentation to minimize calls which cross segment boundaries.

4.2 Slow programs: cure

4.2.1 Obtain a PMAP of the program and establish the reason for each external procedure reference. This should be easy for procedure names which do not contain an apostrophe: you explicitly coded the procedure call.

4.2.2 However procedures whose names contain an apostrophe are likely to be called implicity by the compiler. The Compiler Library Manual will give you an idea of what the procedure is for. The PMAP will tell you one subprogram in each segment which is calling the procedure. If you still cannot match up the procedure name with the language feature it encodes, it is possible to decompile the calling segment, find the actual references, and tie these back to the source (via the PMAP and the LOCATION output of the compiler). Then, if desired, the source can be modified to avoid the procedure call.

4.2.3 If the problem cannot be traced to one or more external procedure calls, several options are open:

(a) Review the segmentation

(b) Read your source again carefully.

(c) Ensure debugging statements are in-operative.

(d) Re-run the program with calls to e.g. PROCTIME inserted at salient points.

4.3 Know your machine

4.3.1 For high-level-language programmers interested in learning more about what happens behind the scenes, a starting point is to read sections of the System Reference Manual.

This will give an overview of how the machine works at the machine code level. The Machine Instruction Set Manual should be used as a reference for particular machine instructions.

4.3.2 Specific details as to the implementation of language features can be obtained by compiling sample source programs with the MAP option (and, for FORTRAN, the LOCATION option), prepping with the PMAP option, and then decompiling the object program. The programs DECOMP (in the Contributed Library) and EMDISASM (on the Orlando swap tape) may be used for this purpose.

The INNERLIST option in the SPL compiler is often useful; however as its output is produced before code generation is complete, the result can occasionally be misleading.

4.4 Future shock

4.4.1 As stated in section 1.3, this paper is based on observation of the behaviour of the Athena versions of the SPL and FORTRAN compilers. It is hoped that future versions of the compilers will generate better code. It is likely that when an enhancement is made, the code generated by the compiler for a particular language feature will be better than that generated for the "work-around" the programmer has used in the interim.

4.4.2 It may be found useful for the programmer to set up a jobstream to compile, prepare, and decompile a program (or suite of programs) which exercise the language features and their work-arounds. This jobstream may then be run after a software update and its output compared with previous results.

REFERENCES

[1] E. Splinter: "Optimizing FORTRAN IV/3000" in HP3000 Users Group 1977 International Meeting Proceedings.

[2] C. Morris: "FORTRAN Optimization" in Journal of the HP General Systems Users Group, Volume 1 No. 6 March/April 1978.

(Not quoted in this paper).

[3] Hewlett-Packard Company: "Program Optimization", Appendix F of "FORTRAN/3000 Reference Manual", Edition 1, Update 3.

[4] R. M. Green: "HP3000 - Optimizing Batch Jobs" in Hewlett-Packard General Systems Users Group 1981 International Meeting (Orlando, Florida) - Proceedings.

APPENDIX A: CLEARMANY PROCEDURE

A.1 Purpose

This procedure will clear one or more arrays with one call.

A.2 Calling sequence

Given that n arrays are to be cleared:

The (2i-1)th argument is the address of the ith array.
The (2i)th argument is the number of words to be cleared in the ith array.
The (2n+1)th argument is n, the number of arrays.

Limits: 1 <= n <=29

Example:

```
INTEGER*2 SIA(100), SIB(M)
DOUBLE PRECISION LR(10,10)
...
CALL CLEARMANY (LR,400,SIB,M,SIA(51),50,3)
```

Warning:

The FORTRAN compiler will complain if there are two or more calls in one subprogram and the arguments do not agree in type and number.

A.3 Source

```
Procedure clearmany (n);
   integer n;
begin
   integer fence, i;
   integer pointer arg, len, block;
   instrinsic quit;
   <<start of argument checking>>
   if not (1<=n <=29) then quit(1);
   push (Q);
   fence:= tos-5-2*n;
   @arg:= @n;
   for* i:= 1 until n do begin
     @arg:= @arg-2; @len:=arg(1);
     if logical(arg) > logical(fence) then quit (2);
     if logical(@len) > logical(fence) then quit (3);
     if not (1<= len <= (fence-arg-1)) then quit (4);
   end;
   <<end of argument checking>>
```

```
   @arg:=@n;
   for * i:= 1 until n do begin
     @arg:=@arg-2;
     @block:= arg;
     @len:= arg (1);
     block:=0;
     move block(1):=block,(len-1);
   end;
   tos:= %31400 + 2 * n + 1;
   assemble (XEQ O);
end;
```