ToolBox Tips

by John B. Matthews

Introduction

One virtue of a standard, high-level programming language is to ease the burden of transporting software from one machine to another. While making a program widely available sells well, so does speed; and compiler writers just can't help adding "features" to a standard language. The software engineers who use these tools will generally save optimization for later in the development cycle, but portability is the first thing to be sacrificed. This article will examine some of the techniques and trade-offs in optimizing using Apple's Macintosh Programmer's Workshop (MPW) Pascal compiler.

Register variables

Every central processing unit (CPU) has an assortment of internal memory locations called registers. These registers provide fast access to program variables, but there never seems to be enough of them to go around. Optimizing the allocation of CPU registers typifies the problem of balancing speed verses portability.

Ideally, register allocation should be entirely transparent to the programmer. Most implementations of the C programming language provide for explicit register allocation, but the great diversity of CPU register architectures makes portability a real problem. What's optimal on one machine may be sub-optimal on another. In contrast, the Pascal programming language uses the with…do construct to specify access to fields of records using only field identifiers. By using the with…do construct, the programmer can tell the compiler about complex variable access while still allowing the compiler to optimize register allocation in other respects. Compare the two sample programs test1 and test2 below to see the difference.

```
program test1;
var t: Integer;
begin

t:= thePort^.visRgn^^.rgnBBox.top;

t:= thePort^.visRgn^^.rgnBBox.left
end.
```

```
program test2;
var t: Integer;
begin
with thePort^.visRgn^^.rgnBBox do

begin


t:= top;


t:= left

end
end.
```

To see when the with…do statement will be of benefit, it is helpful to examine the code generated for each of the sample programs. In the code for test1, each access to the boundary of the the port's visRgn involves an identical sequence of instructions to de-

reference the various pointers, handles and fields. In test2, the same reference is calculated only once and stored for future reference in an anonymous global at -6(A5). Clearly this is advantageous, but only when the reference is complex or more than two fields will be accessed within the same procedure.

Code generated for test1.

```
206D 0000         MOVEA.L thePort,A0
2068 0018         MOVEA.L $0018(A0),A0
2050              MOVEA.L (A0),A0
3B68 0002 FFFE MOVE.W   $0002(A0),-$0002(A5)
206D 0000         MOVEA.L thePort,A0
2068 0018         MOVEA.L $0018(A0),A0
2050              MOVEA.L (A0),A0
3B68 0004 FFFE MOVE.W   $0004(A0),-$0002(A5)
```

Code generated for test2.

```
206D 0000         MOVEA.L thePort,A0
2068 0018         MOVEA.L $0018(A0),A0
2050              MOVEA.L (A0),A0
5488              ADDQ.L   #$2,A0
2B48 FFFA          MOVE.L   A0,-$0006(A5)
3B50 FFFE          MOVE.W   (A0),-$0002(A5)
206D FFFA          MOVEA.L -$0006(A5),A0
3B68 0002 FFFE MOVE.W   $0002(A0),-$0002(A5)
```

Inline code

The inline directive provides an even more powerful, though less portable, means of optimization. The body of an inline procedure consists of machine language instructions specified as hexadecimal constants. The function UWord below is a simple example which takes a word size argument and returns its value as an unsigned number. This is handy for doing arithmetic on all sixteen bits of such quantities as Random or certain fields of a volume control block. The corresponding assembly language instructions are shown as comments in the function listing. The first instruction clears all thirty-two bits of register D0; the second pops the word sized argument off the stack into the low sixteen bits of D0; the third stores D0 on the top of the stack where the function result is expected.

```
function UWord(univ n:Integer):LongInt;

inline

$4280, CLR.L D0

$301F, MOVE.W (A7)+,D0

$2E80; MOVE.L D0,(A7)
```

How does inline code help? Usually, calling a procedure causes the compiler to generate code for stacking results (if any), passing arguments and calling the procedure as a subroutine. Moreover, the procedure itself may have substantial code for saving registers and generating a stack frame. With an inline procedure, the code is inserted directly in the calling sequence in a fashion somewhat analogous to an assembly language macro, and without the overhead of a subroutine call or a stack frame. The penalty for this performance increment is larger code size since each call to the procedure causes its code to be replicated inline. Clearly the inline facility is designed for short routines in machine language. MPW Pascal uses it largely for ROM calls (eg.

one word traps) and "glue" routines (eg. calling register based traps).

As another example consider the NumToString procedure defined in Inside Macintosh (IM). NumToString is a register based trap in package seven accessed via selector zero. It is defined there as a procedure with one value parameter (the number to be converted), and one variable parameter (the resulting string). Suppose for coding convenience we want to call NumToString as a function. Declaring such a function in Pascal would require storing the result string in a local variable and copying it back to the caller. For example:

```
function NumToStr(n:LongInt):Str255;
var s: Str255;
begin

NumToString(n,s);

NumToStr:= s
end;
```

This harmless looking little function generates a rather surprising amount of code.

Code for function NumToStr.

```
4E56 FF00 LINK      A6,#$FF00
2F2E 0008 MOVE.L    $0008(A6),-(A7)
486E FF00 PEA       -$0100(A6)
4EBA 0000 JSR       NUMTOSTRING
206E 000C MOVEA.L $000C(A6),A0
43EE FF00 LEA       -$0100(A6),A1
703F        MOVEQ     #$3F,D0
20D9        MOVE.L    (A1)+,(A0)+
51C8 FFFC DBF       D0,*-$0002
4E5E        UNLK      A6
2E9F        MOVE.L    (A7)+,(A7)
4E75        RTS
```

In addition, there is the overhead for a call to the library subroutine NUMTOSTRING.

```
206F 0004 MOVEA.L $0004(A7),A0
202F 0008 MOVE.L    $0008(A7),D0
4267        CLR.W     -(A7)
A9EE        _Pack7
205F        MOVEA.L (A7)+,A0
504F        ADDQ.W    #$8,A7
4ED0        JMP       (A0)
```

The inline function NumToStr below does the job much more efficiently. The argument and the function result are popped into the requisite registers, the correct selector (zero) is pushed, _Pack7 is invoked, and the resulting string pointer is pushed back on the stack. The compiler will take care of allocating space for the string even if the function is called more than once in an expression. Just don't expect recursion to work without a stack frame!

```
function NumToStr(n:LongInt):Str255;

inline

$201F, MOVE.L (A7)+,D0

$205F, MOVE.L (A7)+,A0

$4267, CLR.W -(A7)
```

$A9EE, _Pack7

$2F08; MOVE.L A0,-(A7)

Obviously, highly structured, modular code with many small procedures and functions is easier to optimize, but even short routines are tedious to code inline. The MPW Assembler is an excellent tool for this, but the MPW DumpObj command is a reasonable alternative. The samples of code generation below were created this way. Examining the compiler's output for very simple routines is an excellent way to learn something about 68000 assembly language. With either tool, pay close attention to the stack and parameter passing conventions outlined in the assembly language chapter of IM.

Other considerations

For certain toolbox procedures it takes longer to access the routine via the trap dispatch mechanism than to execute the routine itself. For example given a and b of type Point, a call such as EqualPoint(a,b) can be replaced with the more efficient Boolean expression LongInt(a) = LongInt(b). On the other hand using the existing trap may preserve functionality if the underlying data structures change. BitSet(myHandle, lockBit) is faster than HLock(myHandle), but the latter is less likely to break if Apple moves the lock bit to make room for 32-bit addresses—say in System 7.0.

Boolean expressions can often be simplified for faster execution. The test if a=b then t:= true else t:= false can be shortened to t:= a=b. If one term in a conjunction (using and) is false, then the entire expression will be false no matter what the value of the other terms. Similarly, if one term in a disjunction (using or) is true then the entire expression will be true. With the "short circuit" operators (& for and, | for or), expressions that contribute nothing to the result can be skipped.

Some global optimization is under the influence of compiler directives. For stable code, it is usually possible to turn off range checking $R-. Be sure you haven't inadvertently left overflow checking on $OV+, or turned the peephole optimizer off $W-. If the target machine has the necessary hardware, allow the compiler to generate 68020 code $MC68020+ and 68881 floating point opcodes $MC68881+. The $SC+ directive causes conjunction and disjunction to be treated as the "short circuit" operators described above.

Because optimization can be cumbersome, it's well to consider carefully where the effort will be best spent. Occasionally, simple inspection of the code can pinpoint the weak spots. For more complex code, the MPW Performance Tools can give a picture of where in the code the CPU is spending the most time.

Finally, lengthy execution time may be inherent in the problem or the algorithm chosen to solve it. If the program may be "away" for a while don't forget to put up the watch cursor. During longer delays, the spinning "beach ball" cursor (found in the CursorCtl unit) is an easy way to let the user know about progress. When things really slow to a grind, a dialog with a cancel button lets the user know you care!

[reprinted from the Apple-Dayton Journal, P.O. Box 3240, Dayton, Ohio 45401]