sun microsystems

You and Your Compiler

(the handout)

1.0 Preface

Compilers are like dentists … everyone needs them, but the experience is seldom completely pleasant! First one must get the compiler to accept your source … then debug the results … then make it go fast.

For the purposes of this handout

we will concentrate on the last activity, making your code *execute quickly*. One should be very careful to get the right answers first … getting the wrong answer quickly is easy (via. simply print 42), but not challenging enough.

The examples in the following sections happen to be Fortran programs and program fragments (Fortran codes are often old enough to be correct :>), the techniques highlighted are **not** language specific and the lessons apply to SunC, Pascal and Modula.

2.0 The algorithm

For many applications (especially small ones) it is sufficient to simply turn on the optimizer and all is wonderful; the answers are correct, the compilation speed is acceptable, and the execution speed is just fine.

For large applications more work is typically necessary. Basically we apply the scientific method iteratively:

1. Break the code into separate files†.
2. Make sure you are getting the right answers.
3. Compile and link with profiling enabled.
4. Execute for some typical dataset.
5. Generate profile report.
6. Turn up optimization on the most expensive routines.
7. Go to step #2 until happiness is maximized.

†. Actually, we may wind up packing key modules into a single file. See the discussion below.

3.0 Examples

3.0.1 Very simple example

Consider the SDRWAVE benchmark from NIST (*SIAM Newsletter* Vol. 21, Nov 6), it isn't particularly special just 47 modules, and about 4600 lines of code. Much too small to be a "real" application, but easy to play with. The primary purpose of this example is to illustrate how to read the output of gprof. This is by no means a replacement for reading and understanding the documentation!

Being foolhardy, we would be tempted to compile at high optimization levels and take our chances (and to be honest, this is what I did when the code arrived :>) but we will follow the algorithm slavishly for illustrative purposes.

The run without **ANY** optimization and with profiling enabled took 222 seconds (since there is little I/O or other SunOS calls, we will track user time only), the machine employed was a 4/260FPU2 and the compiler was f77v1.2. We compiled with the -pg switch on, enabling us to run gprof

```
gprof sdr > profile.txt
```

Now examination of flat profile (the second half of profile.txt)

```
%time  cumsecs  #call  ms/call  name
 30.4    67.55                  mcount
 26.6   126.5238742470     0.00  .mul
 16.6   163.44                  _sdpsc_
 11.0   187.92                  _sdstp_
  4.4   197.77                  _snrm2_
  3.8   206.29                  _sdcor_
  2.7   212.24                  _utpde_
```

mcount is an artifact of profiling …the fact that it is so expensive indicates that at least some routines are getting called gazzillions of times … and probably perform very little computation per call.

.mul is the "pseudo op" that the compiler generates for integer multiplies. In f77 codes this is nearly always due to compilation without enough optimization. C codes (due to explicit pointer computations) tend to have more .mul computation than f77 codes.

sdpsc is the first user code, so if we were to jump to conclusions, we would simply turn the optimization up here. But since we have the rest of the gprof output we can track down .mul more, if we like

```
                   0.00        0.00       8/4877777     _sdrmol_ [44]
                   0.00        0.00      16/4877777     _icvt [81]
                   0.00        0.00     100/4877777     _sdriv3_ [9]
                   0.00        0.00     344/4877777     _sdstp_ [2]
                   0.01        0.00    8248/4877777     _snrm2_ [3]
                   0.02        0.00   31944/4877777     _bcpde_ [14]
                   0.03        0.00   35937/4877777     _utpde_ [10]
                   0.04        0.00   56722/4877777     _sdpsc_ [1]
                   0.05        0.00   66552/4877777     _sdscl_ [16]
                   0.12        0.00  165816/4877777     _sdntp_ [12]
                   1.61        0.00 2252052/4877777     _func_ [4]
                   1.62        0.00 2260038/4877777     _pde_ [6]
[8]       6.6      3.49        0.00 4877777         .mul [8]
```

This[†] shows us all of the routines which are calling .mul; 11 user routines in all. So we recompile with optimization for these 11, and reprofile.

```
real     0m52.70s
user     0m37.83s
sys      0m1.03s

   %  cumulative    self
 time   seconds   seconds      name
 26.2       9.93     9.93       _sdpsc_ [1]
 20.9      17.83     7.90       _snrm2_ [2]
 20.6      25.62     7.79       _sdstp_ [3]
 12.2      30.26     4.64       _sdcor_ [4]
  7.8      33.20     2.94       _utpde_ [5]
  3.2      34.41     1.21       _sdriv3_ [6]
  2.5      35.34     0.93       _pde_ [7]
  2.1      36.14     0.80       _func_ [8]
  1.6      36.73     0.59         mcount (107)
```

So by turning on optimization for 30% of the program (counting by module), we obtain over a 5x speedup (clever technology, eh?). With profiling removed the elapsed time is very close to user+sys times (and the sys time is reduced to.2s). On this particular code, turning on the optimizer for all the modules made the runtime get a bit worse, by about 4%. We will discuss this phenomena at some length later.

†. The gprof report header explains things in detail. For our purposes it is enough to note that these routines, and only these routines make use of .mul.

For comparison purposes, on a 4/330 with f77v1.3 we obtain a time of 24s; which is 3/4 of a Convex C-120, and 11x a VAX 11/785 running VMS 4.7. Also a run employing doubles instead of floats (64-bit rather than 32-bit precision) ran in 37s. On a 4/75 with f77v1.4 the runtime is a mere 12s.

On a CISC machine, like a Sun3 or a VAX, optimizers can usually only deliver 20-30% speedup on average. On a RISC machine like a SPARC, 3x-5x is quite common.

3.0.2 A somewhat more entertaining example

In the example above, everything went well. We never had a problem with the results, using the optimizer, nor did we need to look at the code involved In this case, we track down a anomaly in the optimizer. Note that this code is an extract from a user's application; its name and origin are of no relevance to this discussion.

For a given application we clocked in at (4/330, f77v1.2 FCS), with all the customary optimization options

```
real  0m29.86s
user  0m29.46s
sys   0m0.18s
```

```
Alas, we had really expected to come in at 18s or less. After recom-
piling and executing with profiling enabled, we found
```

```
%       cumulative  self
          seconds              calls  ms/call  ms/call name

 83.8       28.37    28.37       401    70.75    70.75  _forcen_
 10.4       31.89     3.52                             _pow_ri
  3.6       33.11     1.22       400     3.05     3.05  _corr_
  1.5       33.62     0.51       400     1.28     1.28  _predic_
  0.2       33.69     0.07        82     0.85     0.85  _write
  0.1       33.72     0.03                             mcount
  0.1       33.74     0.02        16     1.25     6.96  _print_
  0.1       33.76     0.02                             _r_sincos_
```

From this profile we would assume that _forcen_ calls the power function (real to an integer power) and that this is unavoidable (r**i).

forcen turned out to be a largish module, and the trouble code was not obvious. So we get a chance to use another tool.

We recompile forcen (and **ONLY** forcen) with the -a option. This used to† override all optimization, and would reduce runtimes to a crawl. We also link with the -a option (this instructs the linker to use the right runtime library).

After the program terminates, we run

tcov forcen.f

†. C1.1, f77v1.4 are the first to not supress optimization.

The result is a file, forcen.tcov. Here is one particularly interesting bit of that file:

```
                        do 200 j = jbeg,108
2316978 ->              xdiff = xi - x0(j)
                        xdiff = xdiff - 2*int(xdiff)
                        ydiff = yi - y0(j)
                        ydiff = ydiff - 2*int(ydiff)
                        zdiff = zi - z0(j)
                        zdiff = zdiff - 2*int(zdiff)
                        rsq = xdiff*xdiff + ydiff*ydiff + zdiff*zdiff
                        if(rsq.gt.rcutu)go to 200
 859674 ->              rsqi = 1.0/rsq
                        r6 = (sigpar*rsqi)**3
```

These are the most often executed code segments. Thus the conclusion is that we are calling the pow_ri function to evaluate x**3 !!! Clearly this is something which could stand to be improved in a future compiler!

For the nonce, we fix the code **ONLY** in the most expensive “traces”... in general one should try to restrict performance kludges only to those places where one must have them.

This simple change† dropped the time down to 19s (30% speedup). While it is something the compiler should have done, it illustrates what some modest hand work can do.

Can we do better?

Well, from the tcov report above, it should be obvious that we did not touch the most executed trace at all. The code looks clean, and nothing stands out. For lack of anything intelligent, we try removing the int() expressions. The result is now clearly wrong, but the runtime is 10s. This suggests that there lurks a second performance improvement. Before plunging into the assembly code, we replace the 2*int() with 2.0*int(). This drops the runtime to 16s.

Why?

We reduce the code in question to a simpler form:

```
subroutine(x,y)
real,x,y
y = 2*int(x)
return
end
```

We now compile with the -S option to obtain

†. r6 = sigpar*rsqi ; r6=r6**r6*r6

```
.seg"text"! [internal]
.proc0
.global_sub_
_sub_:
!#PROLOGUE# 0
!#PROLOGUE# 1
save%sp,-72,%sp
ld[%i0],%f0
fstoi%f0,%f0
st%f0,[%sp+LP1]
ld[%sp+LP1],%i0
sll%i0,1,%o1
st%o1,[%sp+LP1+4]
ld[%sp+LP1+4],%f2
fitos%f2,%f2
st%f2,[%i1]
ret
restore
LP1=64
.seg"data"! [internal]
.align8
```

```
.seg"bss"! [internal]
.seg"text"! [internal]
.proc0
.global_sub_
_sub_:
ld[%o0],%f0
fstoi%f0,%f0
sethi%hi(L1D12),%o2
ld[%o2+%lo(L1D12)],%f3
fitos%f0,%f0
fmuls%f0,%f3,%f0
st%f0,[%o1]
retl
add%g0,%o2,%o0
.seg"data"! [internal]
.align8
.align4
L1D12:
.single0r2.0
.seg"bss"! [internal]
```

In the case on the left, the real value is moved from the FP regs to the integer regs. This hits stack memory. In the "real" program this data motion is the slow up. In this smaller example, the stack hit causes leaf node optimization to fail.

*On the right we see pretty good code. No wasted data motion. Optimal code for 2*int(x) is a conversion followed by an fadd.*

As it happens, 2*x generates optimal code also. The **ONLY** case which is "broken" is the one employed in this code. This is due to the optimizer being careful, in that the obvious transformation† results in different results in the case of integer overflow! Alas, moving from the FP regs, to the INT regs and back, is very expensive! But getting the right answer is non-negotiable. As it happens, for this program, with its dataset, the transformation is acceptable. But there is no way for the optimizer to know it.

†. to do all the computation on the FP side..

The point is not that the optimizer is imperfect; all optimizers are imperfect†. Since we usually see 2x and sometimes 5x speedup with the optimizer turned on, the authors of the compiler, for the most part, did the right thing.

The point is that tcov allows you to quickly home in on the trouble spots. With its assistance you can speed up your code (algorithm improvements, or coding to the system) and provide useful input to the compiler group.

Tcov can also be used for “white box” testing to ensure application correctness. Due to its high overhead, it may not be feasible for use in highly interactive applications.

3.0.3 Another Example of tcov at work

Again, someone’s code didn’t run fast enough, so it wandered into my office. Tcov allowed me to quickly identify (and w/o thinking!) the problem spot.

```
do i = 1, maxloop
ans1 = ans1+1
dummy(1,i) = ans1
ans2 = sin(dummy(1,i)) - cos(dummy(1,i))
dummy(2,i) = ans2
ans3 = sqrt(dummy(1,i)**2 + dummy(2,i)**2
dummy(3,i) = ans3
write(3,*) (dummyu(j,i),j=1,3)
if (ans3 .le. 0.00001d0 .and. ans3 .ge. -.00001d0) then
write(3,*) 'ans3 = 0.0"
elseif (ans2 .le. 0.00001d0 .and. ans2 .ge. -0.000001d0) then
write(3,*) "ans2 = 0.0"
else
ans2 = atan(ans2/ans3)
ans1 = dummy(2,i)* dummy(3,i)
```

It should be obvious that the write statement is always executed. It turns out this was a debug statement, which the author forgot to delete. Now that they were evaluating high speed computers, they were upset that the code didn’t see the speedup promised (disk drives only turn so fast …:>). Aside from removing the write statement, I removed all of the IF tests from the body of the loop. IEEE arithmetic allows division by zero, etc. This enables the critical loops to be computed at full speed, and the final result checked (if desired). Removing a test here, and a test there, several million instructions can be removed from the run time. This is not, of course, SPARC, nor even Sun specific. All vendors with IEEE hardware benefit from such tinkering.

The *Numerical Computation Guide* should prove handy in illuminating this area.

†. And in this case, the compiler is carefully preserving the program semantics; which is the right behavior!

btw: removing the print and bogus IF tests the time went from 4m down to 5s. Needless to say, the owner of the code was a happy camper. 48x speedup doesn't happen every day, but you really come off as a hero when you can pull it off!

4.0 Recap

We've touched upon the Holy Algorithm of code tuning, and the tools which you can (and should) employ: the compiler, gprof, and tcov.

Master these and you can set up shop as your local code tuner. Be the talk of the neighborhood.

There is, of course, the complete set of vile and unnatural tricks which may be helpful for a given system, and a given language. Use such techniques with caution, make sure they DO make the code run faster; and document what you've done. Chances are, that the next system will require a different technique. So eschew assembly language coding and other *satanic* rituals (unless absolutely necessary).

5.0 How to Tickle the Compiler

Compilers which don't do a lot of optimization are very simple to tickle; a simple -O or moral equivalent suffices. For heavy duty optimization more is required of users. This is unfortunate, but inevitable.

For the compilers released in 1990 (f77v1.3.1, SunC1.0, etc.), we have reduced the minimal incantations; but the serious code tuner should be prepared to depart from the minimal script as circumstances dictate.

```
f77 -fast foo.f -Bstatic
```

will do most of what you'd probably ask for if you read the manuals. For those expensive routines at the top of your profile:

```
f77 -fast -O4 foo.f -Bstatic
```

Note that -fast **assumes** that you want to generate code which is optimal for the machine you are compiling on. If you want to generate code that is optimal for other configurations, you should take a gander at the manual.

Additional note. This is somewhat anti-social, linking with -Bstatic will put everything your application needs into the a.out… rather than using the nifty shared libraries. On a system running a single task, for a single user, -Bstatic is a couple of percent faster (your mileage may vary); on a busy system it will probably slow everybody down. On an otherwise "dry" system, this will almost always result in less runtime variance between runs.

The few (and so far there is only one known example) cases where linking Bdynamic (which is the default) makes your program go much slower can be detected by trying it both ways on a quiet system. If your application goes much slower please report it to the Answer Center. In the one known case, the performance fix was to extract (with ar) the one intensively used dynamic module and we added it to the .o file list for the makefile.

For cc users change f77 to SunC and you too will be happy.

What -fast does will vary from release to release; so a detailed description will not appear here (yet another reason to read the manual :>). The motivation is to pick the most often used performance options and bundle them. Highlights are:

- -dalign enables generation of double word loads and stores. This forces double word alignment which can break some existing C and Fortran applications. You can't get the wrong answer from this switch, but it can provoke core dumps.
- -native instructs the compiler to generate code that is optimal for the floating point unit in the machine that the compilation is occurring on. Currently, the specific choices are -cg87 and -cg89. Code compiled -cg87 can run on all SPARCs. Code compiled -cg89 cannot be executed on early 4/110's and 4/260's (however, 95%++ of machines in the field can handle -cg89 code).
- -libmil instructs the compiler to employ the best inline library templates available.
- -fnonstd sets the "fast" mode of the floating point hw (if relevant) and causes fatal traps on divide by zero, and the like. On current hardware, this results in gradual underflow being replaced with underflow to zero. See the Numerical Computation Guide for an explanation and rationale for why one might not really want to do this.

6.0 New Features

- f77v1.4 now supports Quad Precision. Previously, variables declared explicitly as REAL*16, or implicitly via -r8 combined with DOUBLE PRECISION, got 16 bytes of storage, but were computationally only 64-bits. Now full 128-bit arithmetic is employed. Current hardware does not have 128-bit support, so this is emulated in software. This is, of course, quite slow.
- -O4 causes automatic inlining, that is modules which are defined in the same file will be considered as candidates for inline expansion. This applies to C1.1 and f771.4 only.
- -a no longer disables optimization.Mixing -O and -a may result in some modest measurement error.
- More optimizations have been implemented, floating point register allocation improved, as well as many bug fixes. The net improvement on the SPEC benchmark suite was 18%. It should be remembered that speedup is highly code dependent. Consider the PERFECT Club suite, %ch in runtime f77v1.3.1 to f77v1.4:

%ch: -38	%ch: -28	%: -17
%ch: -36	%ch: -27	%ch: -13
%ch: -33	%ch: -22	%ch: -12
%ch: -32	%ch: -21	%ch: -11
30% of the codes, 30+% speedup	60% of the codes, 20+% speedup	100% of the codes, 10+% speedup

- Of course, 100% of PERFECT Club times; not all codes that can be run. It is exceedingly hard to pick any one of these performance figures, and assert it as **the** performance enhancement**.**

6.1 When the Optimizer is a Pessimizer

Aside from bugs, there are times when one can reasonably expect the optimizer to cost you performance.

- Optimization analysis time… if you spend more time compiling than executing, don't turn the optimizer up beyond -O2 (at least not without profiling data). Use the -time compiler option to see what part of the compilation process is expensive. If it is the link phase, either create a shared library (thus making the amount of work for the linker smaller) or consider rolling your smaller modules into a common file. This goes against good software engineering practice, but use of #include can mitigate the cost. Fortran users should remember that .F files are automagically run through cpp for you by the compiler.
- Optimizers have to make assumptions about your code. When you write

```
for(i=1; i<=N; i++) {
do i = 1, N
```

  it doesn't know how large N is. So it will guess (this is a feature of all optimizers, not Sun specific). If N is really large, you are probably getting good code… but if N is always 2; the optimizer will probably make this loop go slower. Perhaps much slower. So if you have code like this, which you know is really executed only a couple of times (and always that many times) you may wish to write the code out manually (or use a macro).
- Many modern optimizers do the moral equivalent of "loop unrolling" (or actual loop unrolling). In some cases it is possible for the optimizer to make the worst possible choice (this is unlikely in current Sun systems). On the Cydra 5, with a preproduction compiler, on a particular (popular) code, the result was rather impressive (50x or so slowdown). In that application there were lots of loops as above, which always executed 3 times. The loop unrolling algorithm picked some other number, larger than 3. So the highly optimized code was NEVER executed, only the grungy cleanup code was.

Note that this is *unlikely to affect* Sun users at this time. But if you find yourself spending all your time in one pesky loop; break it out into a subroutine, check out the run lengths. Then you can use the -Qoption iropt -l<your guess>. Your guess should be a factor of the most common run length

- When you move loops into separate modules you will sometimes see performance of what would appear to be unaffilated source affected (faster or slower). This is due to the "global" (via. whole module) analysis employed by the compiler for optimization and register allocation.
- When autoinlining takes place, code size is increased. This can result in poorer use of the cache. There is an otherwise undocumented family of switches which may prove interesting. If you compile -O4, and one wants to see what is inlined, add

  -Qoption iropt -O4,inlining_report

  An example follows as Appendix A.

7.0 Some remarks on the relative efficiency of f77 vs. C

From time to time, folks suggest recoding f77 applications in C to "make them go faster"

Unfortunately, with highly optimizing compilers, the reverse is more likely to be effective! (not to worry, SunOS is not being recoded in f77). To understand why this is the case, consider how optimizers work. Three very basic (and not the only) problems exist:

- **aliasing**. It is not legal to move two arbitrary lines of c code...since nearly all lines of c code contain unconstrained pointer references. This inhibits 99% (well some huge number) of all code motion....which is key to high performance platforms from the CDC 6600 on. Further complicating matters, two arguments in a C calling sequence are allowed to have overlapping storage!

This is an open problem, and is considered very hard. The fact that no one has solved it despite a decade of effort should be noted.

- **for vs. do:** do loops are primitive, but map into all known hardware. for loops are more elegant, but since one is allowed to store into the index variable, many useful optimizations are harder (not impossible, as gcc proves).

A simple program which does a simple loop, on SunC and Sunf77 and computes anything, runs faster in f77.

With work simple special cases can be detected, but it is work that is unnecessary for f77.

- "cos" in Fortran is an OPERATOR. The compiler knows what it means. It may, for example, generate a 68881 *fcos* instruction immediately. To get something different you have to instruct the compiler that in this program "cos" does not mean cosine.

  "cos" in C is a FUNCTION. The compiler does NOT know what it means. The compiler must generate a function call to an external function which is found in a library by the linker. There is no way to instruct the compiler that "cos" means cosine. SVID and ANSI C aggravate this problem by requiring a lot of Erna setting in the elementary transcendental functions so a simple fox instruction isn't adequate.

This is a fundamental distinction between C and Fortran. On Suns, in order to get close to comparable performance between double precision C and double precision Fortran, you need to compile with the same optimization level (be explicit, -O is not the same for both) and use -libmil, which contain kludges around the problem. They also solve the SVID and ANSI C problem the way Cray does, by ignoring them. A part of the problem is solved by ANSI C (math.h can be known to the compiler) but it creates others (ERRNO).

To get comparable performance in single precision, you also have to compile the C program with -fsingle, to inhibit C's automatic promotion of all temporaries to double. This is not necessary for ANSI C, if one employs functional prototypes.

The bottom line is that C semantics make it harder to generate optimal code.

8.0 Typical problems and cures

- Very large system time (like 2x the user time!) is typically a symptom of floating point exceptions being handled poorly. f77 users get an automatic call to ieee_retrospective so they get some warning (ignore inexact, unless you are really hip to the IEEE spec). C programmers should add a call to ieee_retrospective to their exit code. What you don't know **can** hurt you!

  If the underflow exception has been raised (and the only other exception raised was inexact), you have a lot of computation taking place very close to zero. Non-IEEE machines typically just return zero when you get "close enough". This isn't the best modern technology can offer (IEEE's gradual underflow can enable you to get better (more accurate) answers in many cases), but if it is what you want add a call to nonstandard_arithmetic()[†] to your program. See the Numerical Computation Guide for details.

  To track down other problems, add a call to ieee_handler("set","common",SIGFPE_ABORT) details to be found in the docs mentioned above. You will probably want to compile with the -g option to get an exact line number (my

†. This used to be done via a call to abrupt_underflow.

approach is to use the optimized code, find the module, then recompile just it. This can produce misleading results in pathological cases, but for big jobs there may not be any other practical choice. except adb :>).

- Register windows are a good thing; but sometimes you can run out. When that happens, the OS emulates having more. This is fine, but slow. If you find yourself with large system times (and you've taken care of FP problems), and you have complex call graphs (see your gprof report) you may find it worth your while to inline some modules. To accomplish this, find the most expensive caller and locate the callee in the same file. If there are several modules which employ the same module(s), co-locate them all. Compile that file at -O4 (irrespective of what turns out to be interesting elsewhere).
- Code runs fine w/o optimization, core dumps with -fast. This is probably due to -fast doing a -dalign for you. Crack open your manual (or call up the man page) and specify the compiler options you want manually (or use adb, or dbx to track down what in your program is poorly aligned!). In Fortran this often comes about by having INTEGER and DOUBLE PRECISION variables in the same common block (which is, of course, legal). On a CDC, or a Cray the code probably had REAL and INTEGERs together, which the standard specifies should occupy the same space. If you converted the code by hand to 32-bit machines you may have introduced this alignment problem. Use -r8 instead of changing the code (f77v1.3 and beyond). C programmers (since you guys use typedefs more than f77 users employ structured variables) have lots more ways to introduce non-double word alignment. The speedup dalign offers (on current machines) tends to be 10%-30%.
- Code produces different answers at different optimization levels. Welcome to the wonderful world of computer arithmetic! As optimizers move your code around (in perfectly legal ways) you may get results other than you expect. Hopefully you restricted yourself to numerically stable algorithms, and your data is not ill-conditioned. Try to use some savvy, if the results only differ in the last couple digits, it's probably harmless. If it's in the most significant digits, use the optimizer selectively. When you locate the module which "breaks" under optimization take it to your local numerical guru, or start breaking it up into submodules. Continue until happy, or exhausted. You may wish to try the f77 -r8 option which promotes all floats to doubles (caveat: see above for caution about DOUBLE PRECISION variable treatment). If this makes your problem go away, you probably have a subtle coding flaw in your code or your algorithm.

  If you do take the time to track down your problem, and you find that there is some computation which only differs by a mite; but this error grows, your algorithm and/or your implementation is not numerically stable.
- Another common cause of getting different, wrong, or no answers when optimization is turned up, is using a variable before it is initialized. Since Unix (as most other operating systems) zeros memory at program load time, at low optimization levels one will find uninitialized variables are typically "preinitial-

ized” to zero. As the optimizer eliminates unnecessary memory references, it is quite possible that instead of zero, one gets the results of some arbitrary previous computation instead. Using a variable before it is set is illegal, so such transformations are in fact reasonable for the compiler to perform. In order to prevent such “accidents” one should always *lint* C code. One should always use functional prototypes in ANSI C, and one should procure a FORTRAN static analysis tools. Some which are available on Sun systems are:

- Fortran-lint IPT 415-494-7500
- FORWARN Quibus Enterprises 217 356 8876
- Flint[tm†] Programming Research, (44) 1 942 9242 and 817-589-0949
- FORCHECK Leiden University 31-71-276804

I have personally **never** run across a large code (100,000+ lines) which doesn’t have at least one bug which this sort of tool can uncover. Most codes turn out to have many unsuspected latent bugs. I have personally used the IPT and Quibus products. Both work, the Quibus product is cheaper, but does significantly less analysis. PRL’s flint sounds like the most complete product.

- Compiling with optimization takes too long. Use perfmeters to check for thrashing, if you don’t have enough RAM.

  Most often, the modules which cause optimizers the most grief get little benefit from optimization (e.g. Fortran main program with tons of COMMONs, EQUIVALENCES, and subroutine calls … with little compute actually performed. Using the profiler to guide your optimization usage clears much of this up. If you do have a module which does suck up time and is still a dog to compile, consider breaking it up into smaller chunks.

- Our compilers are wonderful, but not perfect. If you have exhausted all other avenues and can bear to part with your code, please file a bug report. Optimization shortfalls may be hard to track down, or require a while to solve; wrong answers will be dealt with more swiftly.

†. flint is a trademark of Programming Research Ltd.

9.0 Reading List

1. *Sun Fortran Users and Programmers Guides*, lookup f77v1.3 800-3418-10, 800-3417-10 and f77v1.4 800-5302-10 , 800-5303-10.
2. *Numerical Computation Guide,* 800-3555-10 (for C1.0, f77v1.3) and 800-5277-10 (for C1.1, f77v1.4). ***This replaces the old Floating Point Programmers Guide and its Addenda.***
3. *Dr. Science's Big Book of Science*, Dr. Science, Duck's Breath Mystery Theatre. ISBN 0-8092-4849-2. ***Won't help you get code running, but it will make you feel better about things in general.***
4. *Fortran Optimization*, Michael Metcalf, Academic Press 1982.
   Somewhat dated.
5. *Writing Efficient Programs, Programming Pearls, More Programming Tools*, Jon Louis Bentley. Prentice Hall, AW.
   Builds strong minds, six ways … highly recommended.
6. *A Guidebook to Fortran on Supercomputers*, Levesque&Williamson, Academic Press 1989.
   "*We dedicate this Book*
   To Charles Babbage who thought up the computer
   To Augusta Ada Byron who taught us all how to program,
   To Herman Hollerith without whom we wouldn't have (been) counted,
   To John von Neumann who taught us how to do one thing at a time,
   To Seymour Cray who showed us how to do everything at once."
7. *Optimizing Fortran Programs,* C.F. Schofield, Halsted Press (aka Wiley) 1989. ***Contains lots of good examples, and is pretty tutorial. Chapter 7 on numerics is not to my taste, and should be overlooked. Recommended for all new to code tuning, especially scientific codes.***
8. *Devil's DP Dictionary,* Stan Kelly-Bootle, McGraw Hill 1981. *Dictionary which defines things the way they really are. A satire in the spirit of Bierce's original Devil's Dictionary.*
9. *How To Lie With Statistics*, Darrell Huff. W.W. Norton & Company 1954 ***(frequently reprinted, my copy is from the 34th printing).***
10. *The Official handbook of Practical Jokes*, Peter Van Der Linden, Signet 1989. ***It is of interest to note, that Peter is a Project Leader in the SunLanguages organization.***
11. *Computer Architecture: A Quantitative Approach. David Patterson, John Hennessy.* ***Best single book on CA from several perspectives. A must read. ISBN 1-55880-069-8***

Appendix A - Autoinlining

What are the effects of inlining? Let us return to sdrwave. In this analysis, we consider a version where all the modules are in a single file. All runs with f77v1.4, on a 4/75, five runs whose times are averaged:

```
No inlining (f77 -fast -O3)
avg = 11.222  stdev =   0.517 #pts   5

some inlining (f77 -fast -O4)
avg = 11.012  stdev =   0.013 #pts   5

All inlined   (f77 -fast -O4 -Qoption <see below>)
avg = 11.64  stdev =   0.389  #pts   5
```

Why? Well, as it happens, little of the code matters!!

Compilation -a, running tcov of entire program reveals

```
 1137  Basic blocks in this file
  315  Basic blocks executed
27.70  Percent of the file executed

     10308136  Total basic block executions
      9066.08  Average executions per basic block
```

Thus we see that most of the code isn't even used ?!?!?! Much of the code inlined is never executed, but expanding it inline does impact memory system performance.

Let's consider gprof output†

No modules inlined edition

```
  %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 24.4      2.77      2.77     1984     1.40     1.40  _sdpsc_ [8]
 21.2      5.17      2.40     8248     0.29     0.29  _snrm2_ [9]
 17.4      7.14      1.97     1968     1.00     5.01  _sdstp_ [6]
 14.4      8.77      1.63     3992     0.41     0.70  _sdcor_ [7]
  8.2      9.70      0.93     3993     0.23     0.41  _utpde_ [11]
  3.4     10.08      0.38      100     3.80   108.84  _sdriv3_ [5]
  3.0     10.42      0.34     3993     0.09     0.09  _func_ [12]
...
```

†. Column misalignment in this document, if any, courtesy of Frame2.1X.

Clearly, the rest of the code has little impact on performance. One might compare the names in the executable (nm a.out) with those that occur here. One would then see how many entire modules are never invoked (e.g. saxpy).

We now turn our attention to the other part of a gprof report.

```
                                       called/total       parents
index  %time     self descendents  called+self    name    index
                                       called/total       children
-----------------------------------------------
                0.00        0.00        1/8248        _sdntl_ [54]
                0.57        0.00     1968/8248        _sdriv3_ [5]
                0.67        0.00     2287/8248        _sdstp_ [6]
                1.16        0.00     3992/8248        _sdcor_ [7]
[9]      21.8   2.40        0.00     8248         _snrm2_ [9]
                0.00        0.00     8248/121922      .mul [18]
-----------------------------------------------
```

From this entry, we see that snrm2 (the second most expensive routine) is called by only 4 others. This often can be taken as a hint that they might be worth co-locating in a single file (or at least arranging to providing their .o files to the linker together.

```
                1.97        7.88     1968/1968        _sdriv3_ [5]
[6]      89.5   1.97        7.88     1968         _sdstp_ [6]
                1.63        1.16     3992/3992        _sdcor_ [7]
                2.77        0.01     1984/1984        _sdpsc_ [8]
                0.01        1.62     3992/3993        _utmol_ [10]
                0.67        0.00     2287/8248        _snrm2_ [9]
                0.01        0.00       33/33          _sdscl_ [30]
                0.00        0.00        1/1           _sdntl_ [54]
                0.00        0.00      344/121922      .mul [18]
               0.00        0.00      610/610         _r_pow_ [264]
               0.00        0.00      610/610         _pow_rr [263]
-----------------------------------------------
                1.63        1.16     3992/3992        _sdstp_ [6]
[7]      25.4   1.63        1.16     3992         _sdcor_ [7]
                1.16        0.00     3992/8248        _snrm2_ [9]
```

Clearly we can work “the other way” and consider the expensive caller, co-locating it’s most expensive children.

Of course, in this case, all the code is in a single file. So the issue is moot.

Some inlined (f77 -fast -O4)

```
  %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 22.1      2.52      2.52     1984     1.27     1.27  _sdpsc_ [8]
 21.9      5.01      2.49     8248     0.30     0.30  _snrm2_ [9]
```

```
 17.3       6.98     1.97     1968      1.00      4.92  _sdstp_ [6]
 14.2       8.60     1.62     3993      0.41      0.44  _utpde_ [11]
 13.1      10.09     1.49     3992      0.37      0.68  _sdcor_ [7]
  4.7      10.63     0.54      100      5.40    109.09  _sdriv3_ [5]
...
-----------------------------------------------
                0.00        0.00        1/8248        _sdntl_ [36]
                0.59        0.00     1968/8248        _sdriv3_ [5]
                0.69        0.00     2287/8248        _sdstp_ [6]
                1.21        0.00     3992/8248        _sdcor_ [7]
[9]     22.5    2.49        0.00     8248         _snrm2_ [9]
                0.00        0.00     8248/121922      .mul [26]
-----------------------------------------------
                1.97        7.71     1968/1968        _sdriv3_ [5]
[6]     87.3    1.97        7.71     1968         _sdstp_ [6]
                1.49        1.21     3992/3992        _sdcor_ [7]
                2.52        0.01     1984/1984        _sdpsc_ [8]
                0.04        1.74     3992/3993        _utmol_ [10]
                0.69        0.00     2287/8248        _snrm2_ [9]
                0.02        0.00       33/33          _sdscl_ [25]
                0.00        0.00        1/1           _sdntl_ [36]
                0.00        0.00      344/121922      .mul [26]
               0.00        0.00      610/610         _r_pow_ [241]
               0.00        0.00      610/610         _pow_rr [240]
-----------------------------------------------

                1.49        1.21     3992/3992        _sdstp_ [6]
[7]     24.3    1.49        1.21     3992         _sdcor_ [7]
                1.21        0.00     3992/8248        _snrm2_ [9]
```

Nothing changed! As it happens, according to the heuristics employed by the compiler, none of these were good candidates for inlining. Below we shall explain how to get the compiler to explain what it has done, and why.

inlining all (via. overriding the compilers judgement)

```
  %  cumulative   self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 51.0       6.07     6.07     1968     3.08     5.31  _sdstp_ [5]
 22.0       8.69     2.62     3992     0.66     0.66  _sdcor_ [6]
 13.9      10.34     1.65     3993     0.41     0.43  _utpde_ [8]
  8.7      11.38     1.04      100    10.40   114.89  _sdriv3_ [4]
  2.1      11.63     0.25                             mcount (125)
  0.4      11.68     0.05   121922     0.00     0.00  .mul [10]
  0.4      11.73     0.05                             mul_8bit [9]
```

```
  0.2       11.75      0.02      4092      0.00      0.00  _pow_ri [17]
  0.2       11.77      0.02      4092      0.00      0.00  _r_exp_ [18]
  0.2       11.79      0.02      3993      0.01      0.43  _utmol_ [7]
```

Note that now SDSTP consumes the majority of the runtime, and that between STSTP, SDCOR, UTPDE and SDRIV3 consume 96% of all the cycles!

```
                1.04        10.45     100/100           _MAIN_ [3]
[4]     98.6    1.04        10.45     100           _sdriv3_ [4]
                6.07         4.38    1968/1968          _sdstp_ [5]
                0.00         0.00    2656/121922        .mul [10]
```

Note: nrm2 completely gone.

```
                6.07         4.38    1968/1968          _sdriv3_ [4]
[5]     89.7    6.07         4.38    1968           _sdstp_ [5]
                2.62         0.00    3992/3992          _sdcor_ [6]
                0.02         1.71    3993/3993          _utmol_ [7]
                0.02         0.00   59354/121922        .mul [10]
                0.00         0.00     610/610           _r_pow_ [250]
                0.00         0.00     610/610           _pow_rr [249]
```

So clearly inlining did change the code. But as we see, in this application it did not help. So what really did go on?

If we compile

```
f77 -fast -O4 -Qoption iropt -O4,inlining_report†
```

we get the following sort of output:

```
sdrwave.f:
 MAIN:
...sdstp:
sdriv3:
sdzro:
second:

Inlining Setup Information
    HEURISTIC_THREE in effect
    DEFAULT HEURISTIC in effect
    -Mi100  Maximum number of inlines within a routine
    -Mm10000  Maximum size increase due to a routine
    -Mr4096  Maximum size increase within a routine
```

†. Undocumented features, such as these, are “not ready for prime time”, they have not been completely tested, nor have their interfaces been completed, and one should not rely on their being included in any future product(s). Use at your own risk. But if you do exercise them, please tell us how it worked out. Send results/comments to khb@eng.sun.com | Keith Bierman, SMI MTV 12-33, 2550 Garcia, Mtv. View CA 94043.

```
-Ma120  Maximum automatic storage
-Mc20  Maximum number of calls within a routine
-Ml30  Maximum level of Inlining
-Ms0  Maximum inline recursion
-Mt160  Maximum number of IR triples
```

These values may be user overridden (but they seldom should be) via

```
Qoption iropt -Mx value
```

If you do find some set of values turn out to be quite handy, we would appreciate your sharing the values, and your code with us. We might be able to use the information to evolve an even better heuristic.

```
template being created for _second_

Entry Number: 2
External Name: _second_
File Offset: 1392988
Archive File Name: (null)
Number Of Triples: 17
Number Of Formals: 0
Number Of Calls Within: 1
Size Of Auto Storage: 4
Optimizable
Externally Visible
No User Requests
Number Of Requests 2
```

This indicates that a template has been created for subroutine SECOND, as it has met the tests imposed by our heuristics.

```
max on number of triples met for _sdrmol_  556-160 over cap of 160
```

This is an example of a module which failed the tests imposed by our heuristic

```
Inlining Report for _xerprt_
    line no. 2412 --  template of _xgetua_ being auto inlined
    line no. 2412 --  template of _j4save_ being auto inlined
    line no. 2412 --  template of _j4save_ being auto inlined
    line no. 2412 --  template of _i1mach_ being auto inlined
```

This is an example of how we are informed of the inlining sites.

```
Inlining Report for _i1mach_
    nothing inlined
```

Examination of the entire report, allows us to see that we only need override two parameters the code growth size, and the “max triplets”.

```
f77 -fast -O4 -Qoption iropt -Mr81920  -Qoption iropt -Mt800 \
-Qoption iropt -O4,inlining_report sdrwave.f
```

```
ldd[%fp-24],%o4
fstod%f30,%f18
sethi%hi(L28),%o0
add%o0,%lo(L28),%i0
st%f18,[%sp+LP18+4]
ld[%sp+LP18+4],%i1
st%f19,[%sp+LP18+4]
ld[%sp+LP18+4],%i2
call_printf,3
restore
LP18=96
.seg"data"! [internal]
.seg"data1"! [internal]
L28:
.ascii" distance = %f \n\0"
```

Now the O4 code

```
.seg"text"! [internal]
.proc7
.align4
.global_dist
_dist:
!#PROLOGUE# 0
!#PROLOGUE# 1
save%sp,-128,%sp
st%i0,[%fp+68]
ld[%fp+68],%f30
st%i1,[%fp+72]
ld[%fp+72],%f31
fmuld%f30,%f30,%f30
st%i2,[%fp+76]
ld[%fp+76],%f28
st%i3,[%fp+80]
ld[%fp+80],%f29
fmuld%f28,%f28,%f2
st%i4,[%fp+84]
st%i5,[%fp+88]
ld[%fp+88],%f27
ld[%fp+84],%f26
fmuld%f26,%f26,%f26
faddd%f30,%f2,%f2
faddd%f2,%f26,%f8
st%f8,[%sp+LP12+24]
ld[%sp+LP12+24],%o0
st%f9,[%sp+LP12+24]
ld[%sp+LP12+24],%o1
call_sqrt,2
spilld%f28,[%sp+LP12+8]
st%o0,[%sp+LP12+8]
ld[%sp+LP12+8],%f9
fitod%f9,%f0
ret
restore
LP12=96
.proc16
.align4
.global_main
_main:
!#PROLOGUE# 0
!#PROLOGUE# 1
save%sp,-144,%sp
ld[%fp-4],%f13
fstod%f13,%f28
ld[%fp-8],%f15
ld[%fp-12],%f11
fstod%f15,%f20
fmuld%f28,%f28,%f28
sethi%hi(0xf4240),%o0
add%o0,%lo(0xf4240),%i3
fstod%f11,%f30
mov1,%i5
fmuld%f20,%f20,%f20 (*below)
fmuld%f30,%f30,%f30
faddd%f28,%f20,%f20
faddd%f20,%f30,%f20
std%f20,[%fp-24]
L77017:
call_sqrt,2
ldd[%fp-24],%o0
st%o0,[%sp+LP18+8]
ld[%sp+LP18+8],%f1
fitod%f1,%f2
inc%i5
cmp%i5,%i3
bleL77017
fdtos%f2,%f30
fstod%f30,%f4
sethi%hi(L28),%o0
add%o0,%lo(L28),%i0
st%f4,[%sp+LP18+4]
ld[%sp+LP18+4],%i1
st%f5,[%sp+LP18+4]
ld[%sp+LP18+4],%i2
call_printf,3
restore
LP18=96
.seg"data"! [internal]
.seg"data1"! [internal]
L28:
.ascii" distance = %f \n\0"
```

Note that now that the distance computation has been moved into the loop body proper, the optimizer is able to eliminate a lot of overhead, **and** is then able to hoist the invariant code out of the loop. It is often the case that the inlining itself is not an important transformation; but providing more opportunities for the optimizer is **quite** useful.

If one takes the time to co-locate C callers and callees very substantial benefits may accrue, more so than in most Fortran programs.