SunPro

A Sun Microsystems, Inc. Business

Keith Bierman

You and Your Compiler

or

How to Tune Code for Fun and Profit

Criteria to Compare Compilers

- Robustness
    - Legal programs compile and generate correct results
    - Illegal programs fail to compile/execute in an intuitive way
    - Invariant with respect to optimization
- Quality of Code generated (highly/aggressively optimizing)
- Ease of use
    - Compilation speed
    - Debugging support
    - Secret handshakes required? How many?

Robustness

- A metric is the number of applications
  - 2,000+ ISV SPARC alone (Summer 1990)
  - Large end user applications (CERN, Fermi, GM, etc.)
- Another metric is the number of users
- Size of applications
  - Huge number of tiny projects
  - Many multiK line projects
  - Several million+ line projects

Code Quality

- Usually measured by benchmarks…but Benchmarks can only measure the total system. time to execute =

function (ISA, implementation, ABI, compiler, libraries, OS)

Types of Benchmarks

- Engineering Benchmarks: Designed to break the compiler, stress the hardware, etc.
- Marketing Benchmarks: Designed to flatter one system and to break all others
- Silly Public Benchmarks: Dhrystones, LINPACK, Savage, Whetstone, etc.
- Real benchmarks aka customer codes, real applications, next best thing - SPEC, PERFECT Club

What we can expect from optimizing compilers:

- We should expect to use them in conjunction with profiling
- We can expect them to reveal bugs in our code, including numerical instabilities in our algorithms, dangling pointers and uninitalized variables.
- We can expect them to require considerable machine resources to compile. This can be managed by using the optimizer on "full" only when necessary.
- We can expect, perhaps, a 4x speedup over unoptimized code on RISC and other modern architectures.

Most applications benefit best from clever algorithms.

- Concentrate on designing a good application.
- 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.

Go to step #2 until happiness is maximized.

Simple Example

SDRWAVE, a NIST benchmark. See SIAM Newsletter Volume 21, Nov 6 1988.

4600 lines of Fortran source in 47 modules.

We pretend that it is a large, potentially buggy application. Thus we are loathe to simply turn on the optimizer.

Our initial run time is 200 seconds on a Sun 4/260/FPU2 running f77v1.2

Flat Profile

```
%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
- .mul is the library routine integer multiply *pseudo op*. Suggests not enough optimization was employed
- User modules (written in Fortran) have the form: _mycode_in Solaris 1.x

gprof

```
_sdrmol_ [44]
_icvt [81]
_sdriv3_ [9]
_sdstp_ [2]
_snrm2_ [3]
_bcpde_ [14]
_utpde_ [10]
_sdpsc_ [1]
_sdscl_ [16]
_sdntp_ [12]
_func_ [4]
_pde_ [6]
.mul [8]
```

- 12 user routines call .mul. The most expensive (w.r.t .mul) is pde. We could recompile with optimization added for just one routine at a time.

  Being lazy we take these first 11 all at once.

Recompiling:

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

By turning on optimization for 30% of the program (counting by module), we obtain a 5x speedup.

% time	cumulative seconds	self 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)

Post Mortem

1. Compile and link with profiling enabled.
2. Execute for some typical dataset.
3. Generate profile report.
4. Turn up optimization on the most expensive routines.
5. Go to step #2 until happiness is maximized.

- One time through this “loop” was sufficient. More optimization slowed down development by a lot and was, in fact, slightly worse for execution.
- Only a dozen routines required optimization.
- This is not unusual.

A somewhat more entertaining example

- In the example above, everything went well. Now we consider a trickier case, where we track down a couple of optimizer bugs...

  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 recompiling 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_
```

Recompiling forcen (and only forcen) with the -a option. Link with -a also. Note: with the advent of C1.1, f77v1.4 -a no longer disables 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
```

Hack #1

- **3 to x*x*x. 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[1].
- 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 sometimes accomplish.
- Can we do better ?

1. This optimization has been restored to the compiler.

```
                do 200 j = jbeg,108
2316978 ->          xdiff = xi - x0(j)
                    xdiff = xdiff - 2*int(xdiff)
```

- float = float - 2*int
  - Optimal code would be fstoi, fadd, fsub
  - Generated code was fstoi, sto, ld, sll, st,ld, fitos
  - Hack fix is to write 2.0*int(xdiff)
- Problem is when 2*int overflows, transformation is wrong.
  - It is a feature of SPARC ISA that communication from FP to INT regs is via memory. Fortunately it is a relatively rare event.
  - Note that 2*x gets optimized as it should, as does 2.0*int(x).

Recap

- The tuning algorithm is application, language and system independent.
- Master it and you can set up shop as your local code tuner.

  Be the talk of the neighborhood!
- Locating the bottlenecks is science. What to do about them is somewhat arty.

  The color of the art is Black.

  Eschew assembly language coding and other satanic rituals, except when necessary.

A Sun Microsystems, Inc. Business

“Blind” Test

- time a.out   real 4m19s user 4m 1s

%time	cumsecs	#call	ms/call	name
24.0	58.32			mcount
10.2	83.10	4999995	0.05	_fp_rightshift
7.8	102.15	100000	0.19	_s_wsle
7.3	119.96			.div
.				
.				
1.3	213.95			_sincos

- User thought they were measuring cost of trig computations.
- What does this suggest ?

Write statements in innermost loop!

- Was a bug in the code. The user did not intend to write every data point. The output file was simply deleted by the shell script anyway.

- If we exploit ieee math, we can do away with all of the IF tests. Leaving the critical loop as:

  ```
  ans2 = atan(ans2/ans3)
  ans1 = dummy(2,i)* dummy(3,i)
  ```

- Now our run time is 9.6s real time. _sincos is now consumes 62% of the runtime. W/o profiling, the runtime is a mere 5s real time.

C example

Analysis of a C benchmark indicated that this bit of code was the critical path. What might you do differently ?

```
time_t t;
int b, c;
int i,j,k;

t = time(NULL);
for (k = 1; k <= 10;  k++) {
for (i = 1; i <= MAX_LOOP;  i++)
   for (j = 1; j <= MAX_LOOP; j++)
      a[i][j] = i*j + i*i + j*j;

for (i = 1; i <= MAX_LOOP;  i++)
   for (j = 1; j <= MAX_LOOP; j++) {
      b = a[j][i];
      c = a[i][j];
      a[j][i] = c/b;
       a[i][i] = b/c;
   }
    }
```

Changing b, c to float perhaps ?

With the all integer computation, the runtime was 43s.
With the float computation, the runtime was 33s.

25% speedup

Note that with only one / (via. I commented out the computation of a[i][i]) the runtimes are identical.

Rule of thumb: If you can do a bunch of work on the FP side; do it. Converting for a single operation is at best a draw.
Note: strictly speaking this conversion is illegal (a float cannot safely contain an int); but analysis of the input constraints allowed us to make this transformation. A compiler would be forced to use double.

How to Tickle the Compiler

1. In f77v1.3/C1.0 and f77v1.4/C1.1, “typical” best performance is obtained via:

   f77 -fast foo.f -Bstatic -lm

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

   The former is the best compromise between code quality and compile speed. The latter is for the computationally expensive kernels.

   -fast is a collection of our favorite options, including:

- -O2
- -dalign
- -fnonstd
- -libmil
- -native

Strictly speaking these are not all optimizations. But these correspond to the most commonly used features for a large number of users.
It is assumed that you will be executing the code on the same model as the compile is taking place on.

SPARCworks/SPARCompilers 2.0.1

- New code generation option cg92
- mixing of -O -g permitted
- “lazy” stabs evaluation
- New Performance Analyzer and Collector
- New code browsers
- All documentation on-line
- Bug fixes
- -bsdmalloc (4.x only)
- -nolib (get no default libraries)
- -xF (Solaris 2.x only)

SunC 2.0.1

- ANSI C front end (not called cc in 4.x!)
- -libieee (use IEEE rather than SVID, etc. error returns)
- new driver for Solaris 2.x (getopt compliance)

New Features in f77 2.0.1

- Cross reference listing
- f77 internal names have been changed (namespace pollution reduction)
- dbx support for Array slices
- general dbx improvement for f77 features
- better support of VAX structures

How to Tickle the Compiler

1. In f772.0.1/C2.0.1 (Solaris 1.x), "typical" best performance is obtained via:

    As before + some extra options:

    -bsdmalloc ! use a less memory efficient but quicker malloc

    -cg92 enable SuperSPARC specific code generation

2. For Solaris 2.x

    As before + -dn -xF

Strategy for Large Applications

- Get it right at -cg89 and low optimization (say -O2)
- Profile
- Isolate most expensive logic
- Tweak, put callers/callees together.
- Try compiling just these at -O4, -cg92, etc. measure effects
- Make shared library variants for different platforms

When the Optimizer is a Pessimizer

Aside from mis-features or 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.

When the Optimizer is a Pessimizer

- Optimizers have to make assumptions about your code. When you write

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

do i = 1, N
```

- The compiler can’t generally know what N is. It will assume that N is relatively large. If N is always 1 or 2, optimization will probably be pessimization.
- When loop unrolling is performed (high levels of optimization) this effect is non-trivial.
- Variables must be defined before used. This is not diagnosed by the compiler, and it is a common error.

Old application-level performance tools

- time a.out
  - produces real (wall) user (cpu) and system (I/O and such) report. csh version reports %cpu utilized
- cc $options -pg foo.c -o a.out;a.out; gprof a.out > profile
  - Produces both the "flat" and "call graph" reports discussed in the sdrwave example. Note that if you compile -pg you must link -pg
- (f77 1.3+) call io_stats() combined with -pg, provides an I/O usage report
- tcov
  - As illustrated in the sdrwave example, tcov provides statement counts; useful for very fine-grained analysis.

SunPro
A Sun Microsystems, Inc. Business

New application-level performance tools (Solaris 2.x only)

- dbx collector
- Analyzer (can run on 4.x with 2.x experiment files)
- More accurate
- Analyze Reactive programs

Miscellaneous

- IEEE arithmetic makes it possible to avoid many branches, consider
  - sum = 0. do i = 1, n;sum = arctan(x/y(i)) + arcsin(r/z(i)) + sum;end do
  - On a non-IEEE compliant machine, we would need to test to see if y, were too close to zero each time through the loop. With IEEE arithmetic one can divide freely and test after the loop (saving 2n tests in this trivial example). See the Numerical Computation Guide for details of how to get at the various IEEE features on Sun systems
- Many C programmers have gotten into the habit of using fixed-point (integer) arithmetic for applications which are naturally floating-point, the reason being that on some machines the integer unit is/was considerably faster than the FPU. This is seldom true on Sun systems, and FP should be used whenever it makes sense.

Wrap Up

- Use profiling tools to locate interesting parts of your application.
- For large applications, learn the ins and outs of the local system; exploit IEEE arithmetic, or implementation-specific features in your tightest bottlenecks.
- The optimizer is your friend!

Additional C example

- From comp.benchmarks

```
/*  Message-ID: <1902@ryn.esg.dec.com>
*/
/*   Date:  25 Jun 90 14:10:59
GMT           */

/*  1000 digits of PI    */
/*  'Spigot' algorithm originally
     due to Stanly Rabinowitz */

#include <stdio.h>
#include <math.h>

main()
{
  long d = 4, r = 10000, n = 251
  long m = 3.322*n*d;
  long i, j, k, q;
  static long a[3340];

  for (i = 0; i <= m; i++) a[i] = 2;
  a[m] = 4;

  for (i = 1; i <= n; i++) {
    q = 0;
    for (k = m; k > 0; k--) {
      a[k] = a[k]*r+q;
      q = a[k]/(2*k+1);
      a[k] -= (2*k+1)*q;
      q *= k;
    }
    a[0] = a[0]*r+q;
    q = a[0]/r;
    a[0] -= q*r;
    printf("%04d%s", q, i & 7 ? "  " :
"\n");
  }
}
```

- The original runtime, on a 4/75:

```
     4.700000 ; 4.710000 ; 4.710000
     4.700000 ; 4.710000
avg = 4.706  stdev =   0.0041833 #pts   5
```

- Changing the declarations of d,r,n and a to double. And 2*k to 2.0*k:

```
     3.090000 ; 3.120000 ; 3.080000
     3.090000 ; 3.080000
avg = 3.092  stdev =   0.0125499 #pts   5
```

- An easy 30% speedup.