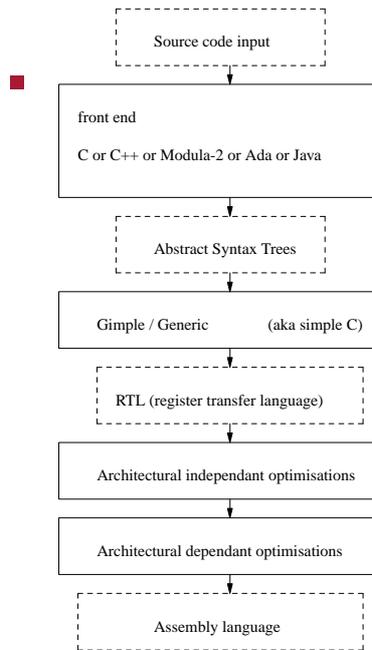


Optimisers: GCC and friends

- the GNU Compiler Collection consists of a number of compilers which can be built to natively support a variety of targets
 - the compilers could also be configured to cross compile - for non native targets
 - 71 architectures (x86_64, arm, risc-v, sparc, avr) etc
 - multiple platforms (GNU/Linux, OSX, Windows, Solaris, AIX) etc
- variety of languages: C, C++, Ada, Java, Modula-2, Fortran, D, Go
- the compiler has a vast number of optimiser options
 - some architecture specific, some generic
- compiler also has debugging and profiling options
 - highly useful to detect hot spots in your code

Structure of GCC



How to profile your code

- the gcc compilers all have the `-pg` option which enables profile generation of code
 - beware that it generates extra instructions to achieve instrumentation
 - beware of the Heisenberg principle
- nevertheless it is easy to use and very effective for profiling static programs
- however `pgc` is built as a dynamic shared library so a number of these easy to use options are off limits

GCC and tips

- GNU Compiler Collection consists of many language front ends to the gnu compiler
- here we will look at some of the common options to `gcc` and `g++`
- these slides are simply a taster and huge simplification of how GCC might be used

GCC debugging

- all front ends (in our case: gcc, g++ and gm2) accept `-g -O0` which tell the compiler not to optimize and emit debugging information for gdb

GCC debugging

- turn on all warnings by: `-Wall`
- so our command line to compile `hello.c` is:
- ```
$ gcc -g -O0 -Wall -c hello.c
```
- notice that this compiles `hello.c` but does not link it
- to link this we can:

- ```
$ gcc -g hello.o
```

GCC debugging

- we could combine the last two steps by:

- ```
$ gcc -g -O0 -Wall hello.c
```

## Debugging your code

- ```
$ gdb a.out
(gdb) break exit
(gdb) run
(gdb) quit
```

- set break points, single step code, finish functions, invoke functions as necessary

- ```
(gdb) print t
(gdb) break pf
(gdb) print pf(t)
(gdb) next
(gdb) step
(gdb) finish
```

## Valgrind

- no excuse for not using this program (for static programs)!
- it requires no effort to run your executable in valgrind
- `$ valgrind ./a.out`
- valgrind is a memory mismanagement detector, it can detect using memory which has not been allocated or has been freed

## What is wrong with this code?

```

#include <stdlib.h>

void myfunc (int n)
{
 int* a = malloc(n * sizeof(int));

 a[n] = 0;
}

int main ()
{
 myfunc (3);
 return 0;
}

```

## Valgrind gives you a huge hint

```

$ valgrind ./a.out
==30984== Command: ./a.out
==30984==
==30984== Invalid write of size 4
==30984== at 0x400511: myfunc (bad.c:7)
==30984== by 0x400526: main (bad.c:12)
==30984== Address 0x518b04c is 0 bytes after a block of

```

## Making your program go faster

- firstly profile your static code to check if there are any obvious inefficiencies
- `$ gcc -g -O0 -pg -c foo.c`  
`$ gcc -g -pg foo.o`

## Making your program go faster

- again we could combine these two commands with

- `$ gcc -g -O0 -pg foo.c`

- most large projects will involve a discrete compile and link step

## Making your program go faster

- run your program as before

- `$ ./a.out`

- now invoke the profiler

- `$ gprof a.out`

## Making your program go faster

- ```
Each sample counts as 0.01 seconds.
% cumulative self self total
time seconds seconds calls s/call s/call name
35.38 20.64 20.64 4771989280 0.00 0.00 II
25.59 35.56 14.93 1049864543 0.00 0.00 s
15.22 44.44 8.88 345772285 0.00 0.00 ma
8.52 49.41 4.97 868833748 0.00 0.00 IN
7.34 53.69 4.28 10274416 0.00 0.00 eva
```

- we could choose to rewrite the functions IN, scan or makemove

Making your code go even faster

- use options: -O1 or -O2 or -O3 on the command line to gcc

- these optimizations may vary according to architecture

Making your code go even faster

- for detail as to which optimizations they turn on use:

- ```
$ gcc -c -Q -O3 --help=optimizers | grep enabled
```

- to see the difference between -O2 and -O3 use:

- ```
$ gcc -c -Q -O3 --help=optimizers > /tmp/O3-opts
$ gcc -c -Q -O2 --help=optimizers > /tmp/O2-opts
$ diff /tmp/O2-opts /tmp/O3-opts | grep enabled
```

Making your code go even faster

- if you don't need full compliant math code, you could use the `-ffast-math` option (which will inline `sin`, `cos`, `tan` etc)

- ```
$ gcc -O3 -ffast-math -c foo.c
$ gcc -O3 -ffast-math foo.o
```

## Tutorial/coursework hint

- try using `-ffast-math` in

- ```
git-pge/c/Makefile.am

.c.lo:
    $(LIBTOOL) --tag=CC $(AM_LIBTOOLFLAGS) $(LIBTOOLFLA
    --mode=compile gcc -c $(CFLAGS_FOR_TARGET) \
    $(LIBCFLAGS) $(libgm2_la_M2FLAGS) \
    -ffast-math $< -o $@
```

- and rebuild pge
 - try experimenting with other optimisation flags
 - -O0, -O1, -O2, -O3

Size of code generated

- you can always check the size of your code via:

- ```
$ size a.out
```

- also optimize for space via the option `-Os`

- ```
$ gcc -Os -c foo.c
$ gcc -Os foo.o
```

How to debug a shared library (PGE)

- sadly we cannot use `gcc -pg` to profile as pge is compiled into a shared library
- however we can use the tool `google-pprof` which will profile shared libraries

Add a profiling library during the shared library link stage (-lprofiler)

- ```
libpgeif.la: $(MY_DEPS)
...
$(LIBTOOL) --tag=CC --mode=link gcc -g _m2_pgeif.lo $(
pgeif_wrap.lo \
-L$(GM2LIBDIR)/lib64 \
-rpath `pwd` -liso -lgcc -lstdc++ -lpth -lc -lm -lp
-o libpgeif.la
cp .libs/libpgeif.so ../_pgeif.so
cp pgeif.py ../pgeif.py
```

## Add a profiling library during the shared library link stage (-lprofiler)

- ```
$ CUPROFILE_FREQUENCY=10000 LD_PRELOAD=/usr/lib/libprofiler.so
CUPROFILE=dump.txt ./localrun.sh ../git-pge/examples/sj
$ ls -l
$ google-pprof --text localrun.sh dump.txt_7987 | less
```

Add a profiling library during the shared library link stage (-lprofiler)

- ```
Total: 1346 samples
248 18.4% 18.4% 248 18.4% SDL_SetTimer
172 12.8% 31.2% 172 12.8% __dubsin
109 8.1% 39.3% 109 8.1% __nanosleep_nocanc
97 7.2% 46.5% 177 13.2% initEntity
52 3.9% 50.4% 81 6.0% Indexing_GetIndice
41 3.0% 53.4% 41 3.0% findChildAndParent
29 2.2% 55.6% 29 2.2% Indexing_InBounds
11 0.8% 64.9% 183 13.6% slow
10 0.7% 65.7% 10 0.7% coord_initCoord
10 0.7% 66.4% 10 0.7% do_cos
10 0.7% 67.2% 229 17.0% roots_findAllRoots
```

## A graphical performance call tree

- `$ google-pprof --gv localrun.sh dump.txt_8708`
- which gives an indication of the call tree and performance implications

## Profiling

- you will need to experiment with the optimization flags and also profile the shared library after each change
- a good idea to create a table of optimizations and also track the performance hot spots

## How to debug your code

- generally you should turn off optimization when debugging your code
  - use `-O0 -g`
- this will produce precise code to line number and variable access
  - the code will go slower, obviously, which might in extreme cases change the bug behaviour compared to `-O3`
  - you will need to experiment and become comfortable with these tools
- your experience will enable you to tradeoff these issues with your own bugs

## Taking a different language approach

- pge is written in Modula-2 and has been translated into C
- the Modula-2 compiler has additional debugging options which provide very exact source/code correlation
  - if requested it will insert extra redundant `nop` instructions so that end of statements can be single stepped

## Taking a different language approach

- consider the C fragment

```
if (foo)
{
 int x = 1;
 y = x + 1;
}
```

- if this code were compiled with debugging `-g` and `-O0` and single stepped
  - a user would probably not be able to single step every line
  - the `{` line and the `}` do not correspond with any assembly code
- most languages have these syntax sugar statements which do not map onto real assembly instructions

## Whole program optimisation in gm2

- pge showed a 40% improvement (frames per second) on the armv7l with `-fm2-whole-program`

## Taking a different language approach

- the Modula-2 compiler gm2 has the `-fm2-g` option which will generate redundant `nop` instructions for the explicit purpose of providing an exact single step user experience
- oddly enough, on an AMD and Intel 64 bit machine, the performance penalty is almost unnoticeable

## Adding Bungees into PGE

- so far springs, polygons and circle objects have been introduced into PGE
- recall that the spring has an at rest length  $l_0$  and the two objects are currently  $l_1$  distance apart
- a bungee is a modification of the spring object
  - it only pull objects together if  $l_1 > l_0$

## Bungee API

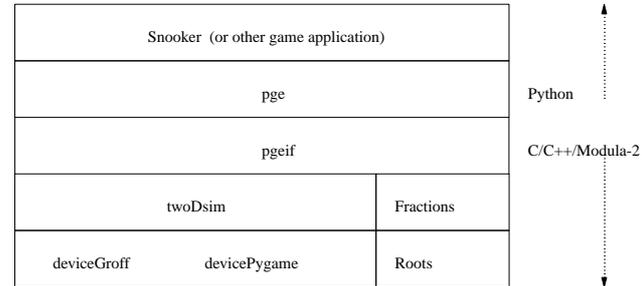
- in Python we could introduce the bungee method which is defined as:

Sandpit/pge/python/pge.py

```
#
bungee - Pre-condition: a spring object. toBungee is
Post-condition: if toBungee is True convert
to a bungee
else convert the bungee back
regular spring.
#
def bungee (self, toBungee):
 # finish this method
 # it should check self is a spring
```

## PGE Layers and associated files

- there are a number of layers in PGE



## Layers and source files to be altered

- pge/python/pge.py
  - the user level python API file
  - this is the only PGE visible file to the user
- pge/i/pgeif.i
  - the swig interface (python calling C/C++ definition)
  - remember to edit both sections (C/C++ section and the Python section)
  - hint look for %{ and }% delimiters

## Layers and source files to be altered

- pge/c/Gpgeif.h
  - header file for pgeif.c
  - contains the external functions implemented inside pgeif.c
- pge/c/pgeif.c
  - its purpose is to allow, colours, polygons, circles, springs, to be given a unique integer
  - thereafter all references to objects will be achieved via the objects, id.
  - notice that inside twoDsim.c colours and circles are different

## Layers and source files to be altered

- `pge/c/twoDsim.c`
  - the actual game engine, which implements polygons, circles, springs
- `pge/c/GtwoDsim.h`
  - the header file for `pge/c/twoDsim.c` which defines all external functions

## Layers and source files to be altered

- all these files will need bungee references added to them
- start at the top `pge/python/pge.py` and work downwards
- follow per object gravity as a guide
- you will need to actually implement bungee's inside `twoDsim.c` (alter the behaviour of a spring)

## Layers and source files to be altered

- hint
  - add an extra field to spring `isBungee` and set it to `FALSE` by default in `twoDsim_spring`

## Tutorial

- use these slides to add bungees into your version of `pge`
- write some simple test code in Python to create a bungee spring
  - ensure that it also has a fps counter on screen
  - write down the fps
- now see if you can rebuild `pge` using some of the optimisation techniques discussed in the slides
  - does the fps change?