Debugging and Profiling

Dr. Axel Kohlmeyer

Senior Scientific Computing Expert

Information and Telecommunication Section
The Abdus Salam International Centre
for Theoretical Physics

http://sites.google.com/site/akohlmey/

akohlmey@ictp.it

What is Debugging?

- Identifying the cause of an error and correcting it
- Once you have identified defects, you need to:
  - find and understand the cause
  - remove the defect from your code
- Statistics show about 60% of bug fixes are wrong:
  -> they remove the symptom, but not the cause
- Improve productivity by getting it right the first time
- A lot of programmers don't know how to debug!
- Debugging needs practice and experience:
  -> understand the science and the tools

More About Debugging

- Debugging is a last resort:
  - Doesn't add functionality
  - Doesn't improve the science
- The best debugging is to avoid bugs:
  - Good program design
  - Follow good programming practices
  - Always consider maintainability and readability of code over getting results a bit faster
  - Maximize modularity and code reuse

Errors are Opportunities

- Learn from the program you're working on:
  - Errors mean you didn't understand the program.
    If you knew it better, it wouldn't have an error.
    You would have fixed it already
- Learn about the kind of mistakes you make:
  - If you wrote the program, you inserted the error
  - Once you find a mistake, ask yourself:
    - Why did you make it?
    - How could you have found it more quickly?
    - How could you have prevented it?
    - Are there other similar mistakes in the code?

How to NOT do Debugging

- Find the error by guessing
- Change things randomly until it works (again)
- Don't keep track of what you changed
- Don't make a backup of the original
- Fix the error with the most obvious fix
- If wrong code gives the correct result,
  and changing it doesn't work, don't correct it.
- If the error is gone, the problem is solved.
  Trying to understand the problem, is a waste of time

The Physics of Strange Bugs

- Heisenbug: bug disappears when debugging a problem (compiling with -g or adding prints)
- Schroedingbug: bug only shows up after you found out that the code could not have worked at all in the first place
- Mandelbug: bug whose causes are too complex to be reliably reproduced; it thus defies repair
- In contrast a “regular”, straightforward to solve bug would be referred to as a “Bohr bug”.

Debugging Tools

- Source code comparison and management tools: diff, vimdiff, emacs/ediff, cvs/svn/git
  - Help you to find differences, origins of changes
- Source code analysis tools: compiler warnings, ftnchek, lint
  - Help you to find problematic code
    -> Always enable warnings when programming
    -> Always take warnings seriously (but not all)
    -> Always compile/test on multiple platforms
  - Bounds checking allows checking of (static) memory allocation violations (no malloc)

More Debugging Tools

- Debuggers and debugger frontends:
  gdb (GNU compilers), idb (Intel compilers),
  ddd (GUI), eclipse (IDE), gdb-mode (emacs)
- gprof (profiler) as it can generate call graphs
- Valgrind, an instrumentation framework
  - Memcheck: detects memory management problems
  - Cachegrind: cache profiler, detects cache misses
  - Callgrind: call graph creation tool
  - Helgrind: thread debugger

Purpose of a Debugger

- More information than print statements
- Allows to stop/start/single step execution
- Look at data and modify it
- *'Post mortem'* analysis from core dumps
- Prove / disprove hypotheses
- No substitute for good thinking
- But, sometimes good thinking is not a substitute for effectively using a debugger!
- Easier to use with modular code

Using a Debugger

- When compiling use -g option to include debug info in object (.o) and executable
- 1:1 mapping of execution and source code only when optimization is turned off
  -> problem when optimization uncovers bug
- GNU compilers allow -g with optimization
  -> not always correct line numbers
  -> variables/code can be 'optimized away'
- **`strip`** command removes debug info

Using **gdb** as a Debugger

- **gdb ex01-c** launches debugger, loads binary, stops with **(gdb)** prompt waiting for input:
- **run** starts executable, arguments are passed
- Running program can be interrupted (ctrl-c)
- **gdb -p <pid>** attaches gdb to an already running process with given process id (PID)
- **continue** continues stopped program
- **finish** continues until the end of a subroutine
- **step** single steps through program line by line
- **next** single steps but doesn't step into subroutines

More Basic `gdb` Commands

- **`print`** displays contents of a known data object
- **`display`** is like print but shows updates every step
- **`where`** shows stack trace (of function calls)
- **`up down`** allows to move up/down on the stack
- **`break`** sets break point (unconditional stop), location indicated by file name+line no. or function
- **`watch`** sets a conditional break point (breaks when an expression changes, e.g. a variable)
- **`delete`** removes display or break points

Post Mortem Analysis

- Enable core dumps: **`ulimit -c unlimited`**
- Run executable until it crashes; will generate a file **`core`** or **`core.<pid>`** with memory image
- Load executable and core dump into debugger **`gdb myexe core.<pid>`**
- Inspect location of crash through commands: **`where, up, down, list`**
- Use **`directory`** to point to location of sources

Debugging Parallel Programs

- Thread level debugging is built into gdb

  => use the command **`thread`** to switch between threads and display current thread id
- Thread ids are counted starting from 1
- Debugging MPI programs in parallel requires a parallel debugger that can forward debugger commands to all copies of the program
- The poor man's parallel debugger:

```
mpirun -np 2 xterm -e gdb -x script ./a.out
```

Using **`valgrind`**

- Run **`valgrind ./exe`** to instrument and run
- **`memcheck`** is default tool and most common
- Output will list individual errors and summary
- With debug info present can resolve problems to line of code, otherwise to name of function
- Also monitors memory allocation / deallocation to flag memory leaks (“forgotten” allocations)
- Instrumentation slows down execution
- Can produce “false positives” (flag non-errors)

How to Report a Bug(?) to Others

- Research whether bug is known/fixed
  -> web search, mailing list archive, bugzilla
- Provide description on how to reproduce the problem. Find a minimal input to show bug.
- Always state hardware/software you are using (distribution, compilers, code version)
- Demonstrate, that you have invested effort
- Make it easy for others to help you!

Profiling

- Profiling usually means:
  - Instrumentation of code (e.g. during compilation)
  - Automated collection of timing data during execution
  - Analysis of collected data, breakdown by function
- Example: gcc -o some_exe.x -pg some_code.c

  ./some_exe.x

  gprof some_exe.x gmon.out
- Profiling is often incompatible with code optimization or can be misleading (inlining)

PERF – Hardware Assisted Profiling

- Modern x86 CPUs contain performance monitor tools included in their hardware
- Linux kernel versions support this feature which allows for very low overhead profiling without instrumentation of binaries
- **`perf stat ./a.out`** -> profile summary
- **`perf record ./a.out; perf report`** gprof like function level profiling (with coverage report and disassembly, if debug info present)

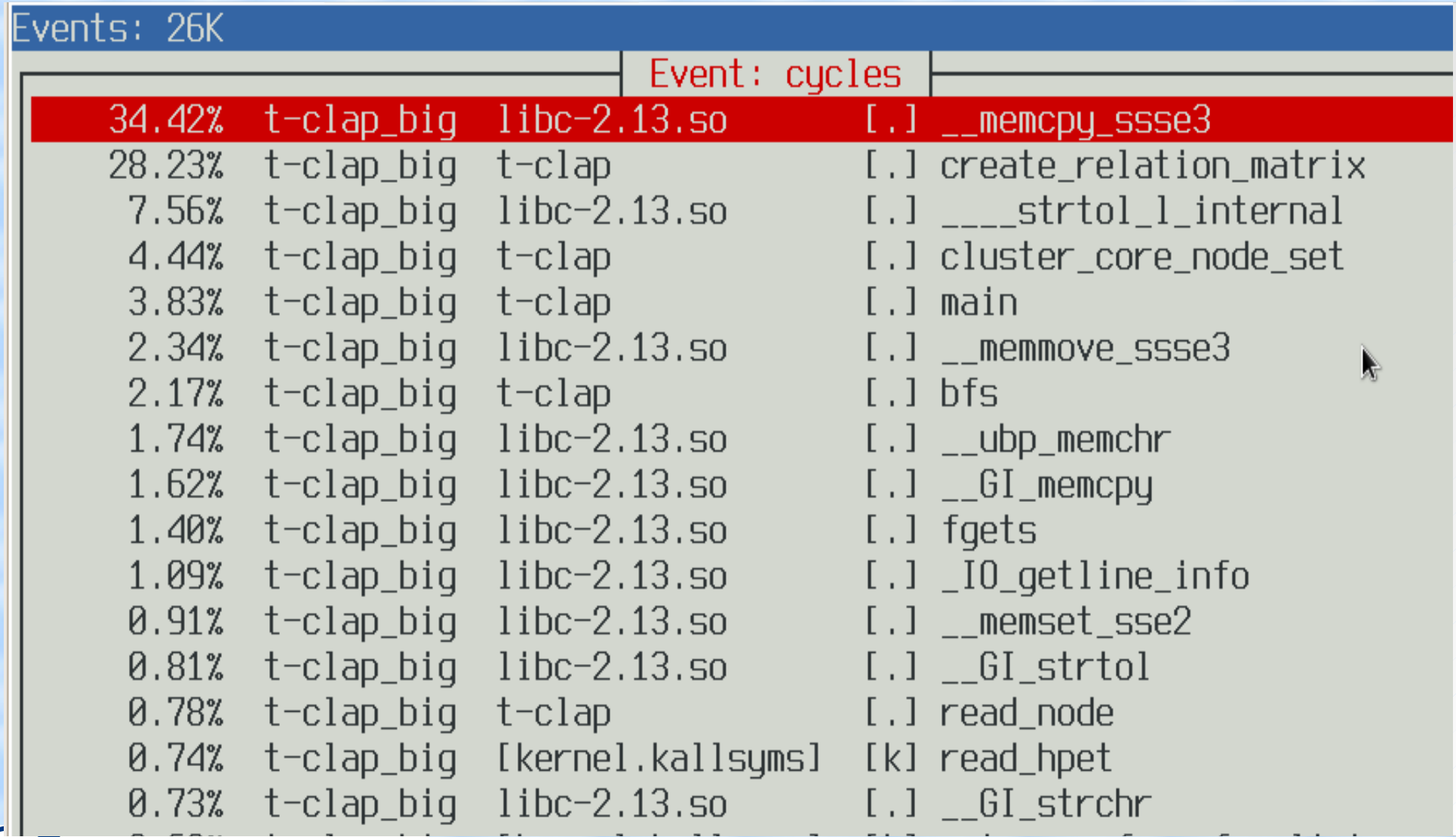
Profiling Examples

```
# gfortran -pg prog1.f ; ./a.out ; gprof --flat-profile ./a.out gmon.out

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.69      4.30     4.30    10000     0.00     0.00  xaver_
  0.00      4.30     0.00        1     0.00     4.30  MAIN__

# make CFLAGS=-pg mountain ; ./mountain ; gprof -p mountain gmon.out

  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 98.30      5.80     5.80     3206     1.81     1.81  test
  1.87      5.91     0.11        1   110.19   110.19  init_data
  0.00      5.91     0.00     2920     0.00     0.00  access_counter
  0.00      5.91     0.00     1460     0.00     0.00  get_counter
  0.00      5.91     0.00     1460     0.00     0.00  start_counter
  0.00      5.91     0.00     1459     0.00     0.00  add_sample
  0.00      5.91     0.00     1459     0.00     0.00  has_converged
  0.00      5.91     0.00      288     0.00    18.33  fcyc2
  0.00      5.91     0.00      288     0.00    18.33  fcyc2_full
[...]
```

Profiling with `perf stat`

```
Performance counter stats for './t-clap_big'
      26768.141153  task-clock-msecs
       44415055252  instructions   (~1.18 IPC)
       12836799487  branches
          71893989  branch-misses  (~0.56%)
         749245773  cache-references
         222548146  cache-misses   (~29%)
      26.976495019  seconds time elapsed

Performance counter stats for './t-clap_small'
      16657.158128  task-clock-msecs
       42539302044  instructions   (~1.84 IPC)
       12722925205  branches
          72503705  branch-misses  (~0.57%)
         168421526  cache-references
          24221380  cache-misses   (~14%)
      16.757377494  seconds time elapsed
```

Profiling with `perf record`



Profiling with `perf record`