Profiling and Debugging Lab

Aaron Birkland[*]
Cornell Center for Advanced Computing

May 17, 2012

1 GDB debugging

This lab exercise serves as an introduction to debugging via GDB (The GNU Debugger). While one may normally wish to debug within an IDE using a comfortable GUI, GDB and its command-line interface is lightweight, powerful, installed virtually everywhere, and usable with little fuss. It is a useful "least common denominator" to know.

This lab will focus around a poorly program "scramble" containing several bugs. This program is supposed to accept a user-provided text string and print a scrambled representation of this string to `STDOUT`.

1.1 Setup

To begin, we will unpack the lab materials and compile the example program.

1. Unpack the lab materials into your home directory if you haven't done so already.

   ```
   % cd
   % tar xvf ~tg459572/LABS/profile_debug.tar
   % cd profile_debug
   ```

2. Compile the scramble program. We are intentionally starting off *without* specifying debug symbols.

   ```
   % cc scramble.c -o scramble
   ```

3. Run the scramble program with some text to scramble as an argument. It should crash with a segmentation fault.

   ```
   % ./scramble "scramble me"
   Segmentation fault
   ```

[*]Thanks to Carlos Rosales, Kent Milfield, Yaakoub Y. El Kharma, and others at TACC for materials and inspiration for the DDT and PerfExpert labs

1.2 Analyzing core dumps

When a program crashes unexpectedly, the OS can dump a copy of its current memory state into a core file. This file can be analyzed later with GDB. Typically, a user can set a size limit for core dumps. This is useful to prevent serious disk usage mishaps for programs that use large amounts of memory. On Ranger, the default is 0, i.e. it will not dump a core grater than zero bytes large unless you direct otherwise.

1. In the C shell (default on Ranger), type `limit` to see default values. (If you switched to the bash shell, use `ulimit -a` instead.)

   ```
   % limit
   cputime      unlimited
   filesize     unlimited
   datasize     unlimited
   stacksize    unlimited
   coredumpsize 0 kbytes
   memoryuse    unlimited
   vmemoryuse   8388608 kbytes
   descriptors  16384
   memorylocked unlimited
   maxproc      267264
   ```

   As you can see, the default `coredumpsize` is 0. (For bash shells, look for a line labeled `core file size`.)

2. Change the core dump size to unlimited. (On Bash, use `ulimit -c unlimited`.)

   ```
   % limit coredumpsize unlimited
   ```

3. Run the scramble program again and look for the dump file. Its name should be something like `core.PID` where `PID` is the process ID number. For example, `core.11781`.

   ```
   % ./scramble "scramble me"
     Segmentation fault (core dumped)
   ```

4. Run GDB using the executable and core file as arguments. This will tell the debugger to analyze the given memory image created by the given executable.

   ```
   % gdb scramble core.11781
   ```

   You will see some text flash by saying how the program was invoked and how it crashed (Segmentation fault), ending up at a gdb prompt `(gdb)`. Note the various "no debugging symbols found" messages.

```
(no debugging symbols found)
Using host libthread_db library "/lib64/tls/libthread_db.so.1".

Reading symbols from shared object read from target memory...
warning: no loadable sections found in added symbol-file shared object read
from target memory (no debugging symbols found)...done.
Loaded system supplied DSO at 0x7fffa53fd000
Core was generated by `./scramble scramble me'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib64/tls/libc.so.6...(no debugging symbols
found)...done.
Loaded symbols for /lib64/tls/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...(no debugging symbols
found)...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2

#0  0x000000000040055e in scramble ()
(gdb)
```

5. To figure out *where* the program crashed, print out a stack backtrace. At the `(gdb)` prompt, type in `bt` to print a stack backtrace.

   ```
   (gdb) bt
   #0  0x000000000040055e in scramble ()
   #1  0x00000000004005b6 in main ()
   ```

   As you can see, the output is somewhat helpful. We can see the memory addresses of our stack frames, as well as the name of the functions they represent. So we know that our program crashed somewhere in `scramble()`, but not much else. Look at the code. Intuitively, `strlen()` could be a problem (is the string null terminated?), as could array bounds or pointers. We don't have enough information to tell.

6. Try to print out a variable. Unfortunately, this does not work. Our problems stem from the fact that we forgot to compile with debugging symbols. We will correct this in the next exercise.

   ```
   (gdb) print i
   No symbol table is loaded.  Use the "file" command.
   ```

7. Exit GDB by typing in `q` at the prompt.

   ```
   (gdb) q
   ```

1.3 Debugging symbols

When we compile with debugging symbols enabled, the debugger becomes much more useful, as it can correlate our source code with functions and variables present in memory.

1. When we looked at the backtrace in the previous section, we could determine the function names associated with each stack frame. This is because the compiler includes some symbols in the executable by default. These can be removed with the `strip` utility. While this will result in a small executable, debugging will be even less useful. Strip the `scramble` executable and see. Load the core and do a backtrace.

   ```
   % strip scramble
   % gdb scramble.c core.11781
   (no debugging symbols found)...done.
   Loaded system supplied DSO at 0x7fffa53fd000
   Core was generated by `./scramble scramble me'.
   Program terminated with signal 11, Segmentation fault.
   Reading symbols from /lib64/tls/libc.so.6...(no debugging symbols
   found)...done.
   Loaded symbols for /lib64/tls/libc.so.6
   Reading symbols from /lib64/ld-linux-x86-64.so.2...(no debugging symbols
   found)...done.
   Loaded symbols for /lib64/ld-linux-x86-64.so.2

   #0  0x000000000040055e in ?? ()
   (gdb) bt
   #0  0x000000000040055e in ?? ()
   #1  0x00000000004005b6 in ?? ()
   #2  0x00002b061a9f34cb in __libc_start_main () from /lib64/tls/libc.so.6
   #3  0x000000000040048a in ?? ()
   #4  0x00007fffa53120a8 in ?? ()
   #5  0x000000000000001c in ?? ()
   #6  0x0000000000000002 in ?? ()
   #7  0x00007fffa5313304 in ?? ()
   #8  0x00007fffa531330f in ?? ()
   #9  0x0000000000000000 in ?? ()
   (gdb) q
   ```

   This is almost completely useless for debugging, as we have little more than a list of memory addresses and an occasional symbol from an included library. While you will likely never have reason to strip executables, you may encounter executables and libraries that are stripped and thus are nearly impossible to debug if they cause a crash.

2. Compile the program with debugging symbols and no optimization. Aggressive optimization will break the correlation between between the source code and the native machine instructions, complicating the debugging process.

```
% cc -g -O0 scramble.c -o scramble
```

3. Run the program, allow it to dump core when it segfaults, and load the *new* core file into gdb as before.

   ```
   % ./scramble "scramble me"
   Segmentation fault (core dumped)
   % gdb scramble core.28616
   Core was generated by `./scramble scramble me'.
   Program terminated with signal 11, Segmentation fault.
   Reading symbols from /lib64/tls/libc.so.6...done.
   Loaded symbols for /lib64/tls/libc.so.6
   Reading symbols from /lib64/ld-linux-x86-64.so.2...done.
   Loaded symbols for /lib64/ld-linux-x86-64.so.2
   #0  0x000000000040055e in scramble (message=0x7ffff111f304 "./scramble",
       buffer=0x4006c0 "") at scramble.c:9
   9           buffer[i] = ((message[i] + i) % 93) + 33;
   (gdb)
   ```

   Much better! Even without doing a back trace, we see exactly where the crash occurred, line 9 of `scramble.c`

4. Print out some variables to help us figure out what is going on at line 9. By inspecting `i`, `message`, and `buffer`, can you figure out a possible cause of the crash?

   ```
   (gdb) print i
   $1 = 0
   (gdb) print buffer
   $2 = 0x4006c0 ""
   (gdb) print message
   $3 = 0x7ffff111f304 "./scramble"
   ```

5. We see that `message` has a valid string, and `i` has not been incremented yet, so it is likely that something is wrong with writing to `buffer`. As it turns out, we mistakenly initiated it with an unmodifiable string literal on line 16. Change line 16 from `char *buffer = "";` to `char buffer[16];` (not an ideal fix, as it opens the door to different kinds of bugs, but it will fix our segfault). Compile and run.

   ```
   % cc -g -O0 scramble.c scramble
   % ./scramble "scramble me"
   OQ9*:*7-82@
   ```

1.4 Live debugging with breakpoints

In this section, we will use GDB to launch and debug a running program, fixing additional bugs in the process.

1. Try running our newly fixed executable with another text string to scramble. Unfortunately, the output doesn't seem to change from before. Clearly, there is another bug.

   ```
   % ./scramble "scramble another"
   OQ9*:*7-82@
   ```

2. We want to use GDB to start execution and stop at a specific point so we can inspect variables and verify that scramble() is scrambling what we think it ought to be scrambling (the user input string). Start by loading GDB with the executable.

   ```
   % gdb scramble
   ```

3. Set a breakpoint for line 9, which is where we'll stop execution and inspect variables.

   ```
   (gdb) break 9
    Breakpoint 1 at 0x400544: file scramble.c, line 9.
   ```

4. Now tell GDB to run the executable with the given text argument. It will stop at the requested breakpoint.

   ```
   (gdb) run "scramble me"
   Starting program: /share/home/01871/apb18/profile_debug/scramble "scramble
   me" Reading symbols from shared object read from target memory...warning: no
   loadable sections found in added symbol-file shared object read from target
   memory done.
   Loaded system supplied DSO at 0x7fff7e566000

   Breakpoint 1, scramble (
       message=0x7fff7e539236 "/share/home/01871/apb18/profile_debug/scramble",
       buffer=0x7fff7e538710 "P\006@") at scramble_fixed.c:9
   9           buffer[i] = ((message[i] + i) % 93) + 33;
   ```

5. Now let's see the content of `message` to see what `scramble()` will be processing.

   ```
   (gdb) print message
   $1 = 0x7fff7e539236 "/share/home/01871/apb18/profile_debug/scramble"
   ```

We see that this is not our input text; it is the name of the executable file! Looking back through the code, we remember that `argv[0]` always contains the name of the executable, `argv[1]` contains the first command line argument, etc. In line 23, we gave `scramble()` the wrong element of `argv`.

6. Quit out of gdb and fix the source so that line 23 reads `scramble(argv[1], buffer);` instead of `scramble(argv[0], buffer);`

1.5 On your own

Further exploration of our program will reveal that it still contains some serious bugs. In fact, we should be surprised that it runs at all without crashing (and on other platforms or other compilers, it may). If you have extra time, try different inputs and use the debugger to help track down the source of these additional bugs.

- Our fix to the initial segfault was very dangerous. We created a fixed-size buffer on the stack with `char buffer[16]`, yet allow writing past the boundaries of that array if the input text is longer than 16 characters long. Writing past the array boundaries clobbers any data in the stack, including the stack frame for the `scramble()` function. Try giving it an input longer than 16 characters on the command line, and compare to doing the same through GDB. Do you get different behaviours? Why?
- In C, strings are null-terminated. Does the code account for this correctly?

2 Distributed debugging with DDT

This lab exercise serves as an introduction to distributed debugging. DDT is a proprietary debugging application installed at TACC which can be used to easily debug distributed MPI jobs as well as OpenMP. In this lab, we will launch a series of communicating MPI processes that experience a deadlock and use DDT to help figure out why this is happening.

2.1 Setup

Since DDT is a graphical debugger, we will be running DDT on Ranger and viewing it on out local machine. Linux uses the X windowing system for graphical display, which is transparent over the network. In order to view the debug windows, the local machine needs an X server to display the images of the GUI. Because the lab machines do not have an X server installed, we will run Linux within a virtual machine and use its own built in X server for display. On Linux, the process is seamless.

1. Launch “Oracle VM Virtual Box” on the lab machine and start an Ubuntu virtual machine.
   (a) In Run dialog box of the start menu, start typing `virtual`. You will see `Oracle VM Virtual Box` appear as a program above. Select it from the start menu.
   (b) When VirtualBox starts, select “Ubuntu” from the left panel, and click on the green arrow named “start”. This will start a virtual machine that will go through its own boot process and eventually display a desktop window.

(c) Click on the terminal icon to launch a new terminal. From there, we will ssh into Ranger.

(d) In the terminal, type `ssh -X USER@ranger.tacc.utexas.edu`, where `USER` is your user ID. Log in this way.

2. Unpack the lab materials into your home directory if you haven't done so already.

```
 % cd
 % tar xvf ~tg459572/LABS/profile_debug.tar
 % cd profile_debug
```

3. Compile the MPI example code with debugging symbols, but no optimizations.

```
% mpicc -g -O0 -o deadlock deadlock.c
```

4. Load the DDT module

```
% module load ddt
```

5. Run DDT and verify that it displays correctly on screen

```
% ddt deadlock
```

2.2 Launching MPI jobs with DDT

DDT is able to launch distributed MPI jobs for the purpose of debugging. Given an MPI executable, this job can be launched with a variety of user-configurable options. DDT needs to manage the job submission itself, as this allows the debugger to insert the necessary trickery in order to track each job's execution and communication state. This section will take you through the job submission process.

1. With DDT open, select "Run and Debug a program" from the available selections. A window will pop up with various queue submission and job control options. This will determine how the debugger will submit the debug job.

2. On the `Options` line, click on the `Change...` button. This will allow you to change the MPI implementation and Debugger if needed. By default, Ranger uses mvapich 1. Verify that mvapich 1 is the MPI implementation, and click `OK`.

3. On the `Queue submission parameters` line, click the `Change...` button. If there is no value for "project", type in `TG-TRA120006`. Also verify that the queue is `development`, Cores per node is `16way`, and wall clock limit is 10 minutes. Click `OK` when done.

4. Make sure the `Number of processes` us set to 12. This will tell the debugger to launch 12 MPI processes.

5. Click on the `Advanced>>` button. Make sure the `Enable Memory Debugging` box is selected.

6. Finally, check `Submit` near the bottom. This will cause various job submission windows to appear, disappear, and update as the job is queued and finally started. DDT will connect to the running processes and show a debugging window when this is complete.

2.3 Distributed Debugging

In this section, we will explore a deadlocked MPI job with DDT in order to become familiar with its interface and controls

1. At this stage, DDT is loaded and is paused at MPI initialization. Click the green "play" arrow near the top to start execution.

2. Click on the `Input/Output` tab of the panel on the lower left. The program is supposed to print two messages: One before an MPI data exchange, and one after. We never see the second message, the processes appear to be stuck. Clock the "pause" button near the top to start investigating. When paused, you can investigate the stack and variables of each process.

3. At the top of the screen, there are various numbers in boxes. These are the rank numbers of MPI processes. Clicking on a number will focus the debugger on that particular process. Look at a few processes and note where they are stopped.

4. Double click on variable names to show their value on in the upper right panel.

5. In the `View` menu, select "Message Queues." This will bring up a window graphically representing the state of messages. It looks like all but one process is trying to send something to 0.

6. Close the queue window and investigate the process that was not stuck in a send. Why is it not stuck?

7. Recall that an `MPI_Send` operation requires a matching receive. Go to process 0, and notice that it is in the middle of a send. Inspect the value of `i` by double clicking on it in order to determine who it is trying to send to. By inspecting both processes, the deadlock is apparent.

3 On Your Own

- Fix the code by adjusting the order of sends and receives. Use DDT to launch the job and confirm whether this fix worked.

- Try running through DDT again, but before pressing "play" to start execution, place a break point in somewhere in one of the processes. Double click on the line to set a breakpoint. A red dot will appear. Start execution and see what happens. What state(s) are the other processes in? What happens if you step one instruction forward? How many processes are affected when you do that?

4 PerfExpert profiling

This lab exercise serves as an introduction to profiling an application using PerfExpert. PerfExpert was developed by TACC to provide an easy to use tool for doing performance analysis of scientific applications and recommending possible performance enhancements. In this lab, we will profile a simple application, interpret the performance results, implement some of the suggested modifications, and compare.

For more information on PerfExpert, see `http://www.tacc.utexas.edu/perfexpert` .

4.1 Setup

To begin, we will unpack the test files and compile our example program.

1. Unpack the lab materials into your home directory if you haven't done so already.

```
% cd
% tar xvf ~tg459572/LABS/profile_debug.tar
% cd profile_debug
```

2. Load the PerfExpert module.

```
% module load papi java perfexpert
```

3. Compile the `perf` program with debugging symbols and full optimization.

```
% cc -g -O3 -o perf ./perf.c
```

4. In the `profile_debug` directory, there should be a shell script `perf_job.sh`. We will submit this to the job queue. It will use PerfExpert to profile our application (achieved by the `perfexpert_run_exp` line). Run it via:

```
% qsub perf_job.sh
```

   If successful, you will see in the end a line like

```
Your job 2569911("PerfExpert") has been submitted
```

5. Verify that it worked. Once your job starts, it will create a file with a name like `PerfExpert.o2569911` in your current directory. This contains the status of the PerfExpert job. You may notice that it runs PerfExpert several times on your executable. When the job has completed, you will see an xml file appear in the directory named something like `experiment-PerfExpert.o2569911.xml`. This is the profiling data that has been collected and that we will analyze.

4.2 Viewing profiling data and recommendations

The PerfExpert application provides a text user interface to interpret the profiling results previously collected. In this section, we will examine our `perf` application.

1. `perfexpert` expects at least two arguments: a threshold value and a data file. The threshold directs perfexpert only to display routines that consume more than the given threshold worth of execution time. In this case, we shall examine our results at a 10% threshold. Run PerfExpert against the previously generated xml file. You should see output like:

```
  % perfexpert 0.1 experiment-PerfExpert.o2569911.xml


Input file: "experiment-PerfExpert.o2569911.xml"
Total running time for "experiment-PerfExpert.o2569911.xml" is 2.368 sec

Loop in function main() (100% of the total runtime)
===============================================================================
ratio to total instrns        %  0.........25...........50.........75........100
   - floating point      :   25 ************
   - data accesses       :   25 ************
* GFLOPS (% max)         :    2 *
-------------------------------------------------------------------------------
performance assessment     LCPI good......okay......fair......poor......bad....
* overall                :  3.2 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
upper bound estimates
* data accesses          :  3.3 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - L1d hits            :  0.3 >>>>>
   - L2d hits            :  0.4 >>>>>>>>>
   - L2d misses          :  0.4 >>>>>>>>
   - L3d misses          :  2.2 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
* instruction accesses   :  0.3 >>>>>
   - L1i hits            :  0.3 >>>>>
   - L2i hits            :  0.0 >
   - L2i misses          :  0.0 >
* data TLB               :  1.9 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
* instruction TLB        :  0.0 >
* branch instructions    :  0.4 >>>>>>>
   - correctly predicted :  0.4 >>>>>>>
   - mispredicted        :  0.0 >
* floating-point instr   :  0.7 >>>>>>>>>>>>>>>
   - fast FP instr       :  0.7 >>>>>>>>>>>>>>>
   - slow FP instr       :  0.0 >
```

2. PerfExpert will print a report on screen with runtime measurements. From this, we see that floating point and data access instructions are roughly even, but PerfExpert

thinks our data access patterns (particularly L3 Cache misses) and TLB misses are "poor" to "bad". These are measured in terms of "LCPI", which stands for "Local cycles per instruction" and are an indicator of the *relative* number of CPU cycles spent in each category compared to the total execution of that segment. It looks like most of the CPU cycles have been wasted for cache misses. Look at the code and see if there are any patterns that lead to poor cache utilization.

3. Now that we have an idea of the basic execution characteristics of our program, we can ask PerfExpert to provide some suggested changes. This is achieved by providing the `-r` flag. PerfExpert will select potentially applicable advice from a database. The examples given will not be taken from your code.

```
  % perfexpert -r 0.1 experiment-PerfExpert.o2569911.xml
```

   You will see lots of text fly by. It will be easiest to scroll through it by using `less`. Use up/down arrows to scroll through, and type `q` when done.

```
  % perfexpert -r 0.1 experiment-PerfExpert.o2569911.xml | less
```

4. Implement the first recommended suggestion. In case it displays suggestions in a different order for some people, the first suggestion for the purpose of this lab is:

```
change the order of loops
This optimization may improve the memory access pattern and make it more cache
and TLB friendly.

loop i {
  loop j {...}
}
 =====>
loop j {
  loop i {...}
}
```

   It looks like this change is applicable to the `compute()` routine. Swap the `j` and `k` `for` loops to make the code look like:

```
void compute()
{
  register int i, j, k;
  for (i = 0; i < n; i++) {
    for (k = 0; k < n; k++) {
      for (j = 0; j < n; j++) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```

4.3 Comparing Profiles

PerfExpert has a comparison mode that allows profiling results from different runs to be directly compared, highlighting the differences. In this section, we will run our modified (and hopefully better) code through the profiler and see where we may have made some improvements.

1. Compile the newly modified `perf.c` and submit it for profiling. After this is done, you should end up with two xml files in the directory: The old one you generated before, and the new one.

   ```
   % cc -g  -O3 -o perf ./perf.c
   % qsub perf_job.sh
   ```

2. Use PerfExpert to examine this new profile. The runtime and cache/TLB misses are much better!

   ```
   % perfexpert 0.1 experiment-PerfExpert.o2570094.xml
   ```

3. Use PerfExpert to compare the two xml files. This will highlight the difference (improvement or regression) between PerfExpert runs.

   ```
   % perfexpert -a 0.1 experiment-PerfExpert.o2569911.xml \
     experiment-PerfExpert.o2570094.xml
   ```

   You will see lines that look like:

   ```
   * data accesses          :      >>>>>>>>>>>11111111111111111111111111111111111+
      - L1d hits            :      >>>>>
      - L2d hits            :      >>>>>111
      - L2d misses          :      >111111
      - L3d misses          :      >1111111111111111111111111111111111111111111
   ```

   A “1” is present wherever the line was longer in file the first file, and a “2” is present wherever a line was longer in the second file. From these results, we see that L3 data cache misses were significantly higher in file 1 than file 2.

4.4 On your Own

- Try compiling the code using the Intel compiler (remember to `module swap pgi intel` and use `icc` to compile). What happens when you look at the results? Remember, PerfExpert uses statistical sampling to measure performance.
- Make the matrix dimensions larger, compile, and submit again. Compare with the results from gcc (`cc`). If the intel results are so much faster, why do some performance metrics look worse?
- Try implemented additional recommended changes. Do any of them improve execution time? While PerfExpert can identify possible approaches that are applicable to your code, it cannot determine with certainty that implementing the changes will result in an improvement. Thus, your judgment plays an important role in interpreting the results.