Debugging and Tuning

*Linux for EDA*

Fabio Somenzi

Fabio@Colorado.EDU

University of Colorado at Boulder
Outline

- Compiling
  - gcc
  - icc/ecc
- Debugging
  - valgrind
  - purify
  - ddd
- Profiling
  - gcov, gprof
  - quantify
  - vtl
  - valgrind
Compiling

- Compiler options related to
  - static checks
  - debugging
  - optimization
- Profiling-driven optimization
Compiling with GCC

- gcc -Wall -O3 -g
  - reports most uses of potentially uninitialized variables
  - -O3 (or -O6) necessary to trigger dataflow analysis
  - can be fooled by
    ```
    if (cond) x = VALUE;
    ...
    if (cond) y = x;
    ```
  - Uninitialized variables not considered for register allocation may escape
- Achieving -Wall-clean code is not too painful and highly desirable
- Compiling C code with g++ is more painful, but has its rewards
Compiling with GCC

- gcc -mcpu=pentium4 -malign-double
  - -mcpu=pentium4 optimizes for the Pentium 4, but produces code that runs on any x86
  - -march=pentium4 uses Pentium 4-specific instructions
  - -malign-double forces alignment of double’s to double-word boundary
  - Use either for all files or for none

- gcc -mfpmath=sse
  - Controls the use of SSE instructions for floating point
  - For complete listing, check gcc’s info page under Invoking gcc → Submodel Options
Compiling with ICC

- ICC is the Intel compiler for IA-32 systems.
- icc -O3 -g -ansi -w2 -Wall
  - Aggressive optimization
  - Retain debugging info
  - Strict ANSI conformance
  - Display remarks, warnings, and errors
  - Enable all warnings
- Remarks tend to be a bit overwhelming
- Fine grain control over diagnostic: see man page
Compiling with ICC

- `icc -tpp7`
  - Optimize instruction scheduling for Pentium 4
  - Also `icc -mcpu=pentium4`
- `icc -ipo`
  - Multi-file interprocedural optimizations
- `icc -axW`
  - Generate both Pentium 4 and generic instructions
- `icc -xW`
  - Generate code specific for the Pentium 4
  - Also `icc -march=pentium4`
- `icc -align`
  - Analyze and reorder memory layout
GCC: Profiler-Driven Optimization

gcc -fprofile-arcs test.c
  Instrumented compilation

./test input
  Instrumented execution
  Produces .da files
  Can be repeated with different inputs

gcc -fbranch-probabilities test.c
  Feedback compilation
ICC: Profiler-Driven Optimization

- `icc -prof_gen test.c`
  - Instrumented compilation
  - `/test input`
  - Instrumented execution
  - Produces `.dyn` and `.dpi` files
  - Can be repeated with different inputs
- `icc -prof_use test.c`
  - Feedback compilation
Debugging

- Dynamic analysis tools
  - valgrind, purify

- Classical debuggers
  - gdb, idb and their graphical front-ends, especially...
  - ddd
Valgrind

- Tool for debugging and profiling Linux-x86 executables
- Valgrind consists of:
  - **core**: synthetic CPU
  - **skins**: perform analyses
- Available skins
  - memcheck and addcheck: memory debugging
  - cachegrind: cache profiling
  - helgrind: races in multithreaded programs
Valgrind: Memory Debugging

- Use of uninitialized memory
- Reading/writing memory after it has been free’d
- Reading/writing off the end of malloc’d blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks – where pointers to malloc’d blocks are lost forever
- Passing of uninitialized and/or unaddressable memory to system calls
- Mismatched use of malloc/new/new [] vs. free/delete/delete []
- Some misuses of the POSIX pthreads API
1: #include <stdlib.h>
2: main()
3: {
4:   char *x, *d = "foo";
5:
6:   x = malloc(922);
7:   x = malloc(123);
8:   x = malloc(-9);
9:
10:  free(d);
11:  free(x);
12:  free(x);
13: }
Valgrind: Memory Debugging

valgrind -leak-check=yes -show-reachable=yes mtest

- Warning: silly arg (-9) to malloc()
- Invalid free() / delete / delete[]
  - in main (mtest.c:10)
- 123 bytes in 1 blocks are definitely lost
  - in main (mtest.c:7)
- 922 bytes in 1 blocks are definitely lost
  - in main (mtest.c:6)

Why isn’t the double free(x) reported?
Valgrind: Memory Debugging

- Valgrind tracks each byte with nine status bits
  - one tracks addressibility of that byte
  - the other eight track the validity of the byte
- Valgrind can be used to debug dynamically-linked ELF x86 executables, without modification, or recompilation
  - `valgrind ls -ls`
- Valgrind can attach GDB to the running program at the point(s) where errors are detected
- Valgrind works on large applications
  - Mozilla
  - OpenOffice
  - emacs-21.2
  - Gcc
  - AbiWord
  - KDE3
Valgrind

- http://developer.kde.org/~sewardj/
  - Last stable version 20031012
- Only on x86-Linux
- Works on many distributions, but not all
  - Yes: RH 7.2 7.3 8 9
  - No: RH 7.1
- kcachegrind GUI only available under KDE
- memcheck slows down execution by 25-50 times
- addrcheck is lighter weight, but does not track read-before-write’s
- the -gen-suppressions=yes option tells Valgrind to print out a suppression for each error that appears
IBM Rational PurifyPlus

- http://www.rational.com/
- Runtime analysis
  - Memory corruption detection
  - Memory leakage detection
- Requires instrumentation
  - `purify gcc -g mtest.c`
- Languages: C, C++
Purify: Bad Function Parameter
Purify: Memory Leaks

Program exited with status code 134832472.
int main(int argc, char *argv[]) {
    int *a, i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);
    shell_sort(a, argc);
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");
    free(a);
    return 0;
}
Purify: Out-of-Bounds Read

```
Finished sample (4 errors, 0 leaked bytes)
Purify instrumented ./sample (pid 28494 at Sun Nov 2 18:06:06 2003)
Command-line: ./sample 2 6 4 8 5
ABR: Array bounds read (2 times)
This is occurring while in:
  shell_sort [sample.c:16]
  do {
    h /= 3;
    for (i = h; i < size; i++) {
      int v = a[i];
      for (j = i; j >= h && a[j - h] > v; j -= h)
        a[j] = a[j - h];
      if (i == j)
        .......
  }
main [sample.c:34]
__libc_start_main [libc-start.c:129]
_start [crt1.o]
Reading 4 bytes from 0x80b5434 in the heap.
Address 0x80b5434 is 1 byte past end of a malloc'd block at 0x80b5420 of 20 bytes.
This block was allocated from:
  malloc [rtlib.o]
  main [sample.c:30]
  __libc_start_main [libc-start.c:129]
  _start [crt1.o]
```
Purify: Suppressions

Message to suppress: MAF: Memory allocation failed

Where to suppress: In call-chain

Call chain:
- malloc
- main
- __libc_start_main
- start

suppress maf malloc; main

Make permanent in file ../../../purify

Apply Dismiss Help
Purify: Library Functions

Library functions allow developer to customize data collected for a given application.

Memory usage profiling:

```c
#ifdef PURIFY
...
  purify_all_inuse();
...
#endif
```

Used in VIS together with a couple of scripts to profile memory usage on a per-package basis.

Link to `libpurify_stubs.a`
The Cost of Instrumentation

- One data point
  - no instrumentation: 64 s
  - `valgrind -skin=addrcheck`: 860 s
  - `valgrind -skin=memcheck`: 1287 s
  - `purify`: 1725 s

- The `addrcheck` skin checks the validity of addresses but not of data.

- Only `purify` detects this uninitialized memory read.

```c
int main() {
    int a;
    return a;
}
```
GDB and IDB

- Better used through a graphical front-end
  - Ddd
  - emacs’s GUD
  - UPS (http://ups.sourceforge.net/main.html)
  - Insight (http://sources.redhat.com/insight/)

- GDB and IDB largely compatible
  - `idb -gdb` is similar to `gdb`
  - otherwise, it is similar to `dbx`
  - Both can be used with the “other” compiler

- There are other debuggers as well
  - TotalView
  - Idebug (Java)
The Data Display Debugger

- Front-end for
  - C/C++ (gdb, idb)
  - Other languages supported by gcc (e.g., Fortran)
  - Perl
  - Python
  - Java

- Available also for other operating systems
  - Works with other inferior debuggers too (e.g., dbx)
  - Requires X server

- http://www.gnu.org/software/ddd
The Data Display Debugger

The DDD Layout using Stacked Windows
DDD: Displaying Data

- (gdb) graph display array[0] @ nelem
  - Shows **array slice** in the data window
  - Optionally use **rotate** button for more compact display

- (gdb) graph plot array[0] @ nelem
  - Runs **gnuplot** on array slice and displays result in new window
  - Plot is updated when data changes
  - Plot can be customized and saved
  - Animations are possible
DDD: Plotting

Plotting 1-D and 2-D Arrays

A 1-D Array
Change Style
A 2-D Array
Rotate View

Plotting 1-D and 2-D Arrays
# DDD: Machine-Level Debugging

## Displaying Register Values

- The register name is copied to ()
- Select register

---

Displaying Register Values
Profiling

- Gcov
- Gprof
- VTune
- Valgrind
Optimization Tips

- Static branch prediction in the Pentium 4
  - Forward branches are not taken
  - Backward branches are taken
- Use `const`; avoid `register`
- Fit data structures to cache lines
- Profiling tools help identify
  - hotspots
  - inefficient memory layout
  - insufficiently tested code
- Remember: Only optimize what is critical
Sampling vs. Counting

- Sampling: the program counter is periodically examined
- Basic block counting: the executable is instrumented so that the frequencies of execution of all basic blocks are recorded
  - Only reliable mechanism for
    - coverage measurement
    - fine tuning
  - Does not account for memory hierarchy
Gcov: Coverage Analysis

- gcc -fprofile-arcs -ftest-coverage -o lfsr lfsr.c
- ./lfsr
- gcov lfsr.c

100.00% of 10 source lines executed in file lfsr.c
Creating lfsr.c.gcov.
int main()
{
    unsigned int r = 1;
    int i;
    for (i = 0; i < 1000000000; i++) {
        unsigned int b = r & 1;
        r >>= 1;
        if (b)
            r ^= 0x8805;
    }
    printf("%u\n", r);
    exit(0);
}
Gprof

gcc -o lfsr -g -pg -fprofile-arcs -O3 \ 
   -mcpu=pentium4 lfsr.c
./lfsr
gprof --line --flat-profile lfsr

Flat profile:
Each sample counts as 0.01 seconds.

% cumulative  self time  seconds  seconds  name
   cumulative  self  time  seconds  seconds  name

              %    %      %    %      %    %
              61.59 8.11   8.11 8.11 main  (lfsr.c:13)
              28.91 11.91  3.81 3.81 main  (lfsr.c:17)
               3.19 12.33  0.42 0.42 main  (lfsr.c:14)
               3.19 12.75  0.42 0.42 main  (lfsr.c:15)
               2.89 13.13  0.38 0.38 main  (lfsr.c:16)
               0.23 13.16  0.03 0.03 main  (lfsr.c:14)
IBM Rational Quantify and Purecov

- Basic-block counting profiling
- Call graph analysis
- Source annotation
Intel VTune for Linux

vtl: command line version of the performance analyzer for Linux

Sampling: non-intrusive, system-wide profiling
  - relies on the CPU performance monitoring registers

Call graph: low overhead analysis of program flow
  - requires instrumentation

http://www.intel.com/software/products/vtune/vlin/
  - Current release is 1.1
  - Several Red Hat and SUSE releases supported
VTune: Sampling

- `vtl activity -c sampling run`
  - Runs the sampling collector for all processes
  - Automatically calibrates collection parameters
  - Collects data on clock ticks and retired instructions

- `vtl show`
  - Displays activities that have been run for a project

- `vtl view a1::r1 -processes`
  - Presents the results of activity a1::r1 organized by process

- `vtl -help -c sampling`
  - Shows what events can be sampled
VTune: Call Graph

- vtl activity -c callgraph -app ./mypgm
  -moi ./mypgm run
  
  Runs the callgraph collector for mypgm
  
  Performs instrumentation (including library functions)
  
  Collects function call data

- vtl show
  
  Displays activities that have been run for a project

- vtl view al::rl -functions
  
  Shows timing information for each function
  
  Use -calls for call-graph edge data

- vtl view al::rl -critical-path
  
  Shows the critical path
Valgrind: Cache Profiling

- Valgrind contains built-in support for cache profiling
  - `valgrind -skin=cachegrind my-program`
  - detailed simulation of L1-D, L1-I, unified L2

- `vg_annotate` annotates source code

- Cache configuration auto-detected using the CPUID instruction
  - can be overridden
Valgrind: Cache Profiling

<table>
<thead>
<tr>
<th></th>
<th>I refs:</th>
<th>73,173,467</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I1 misses:</td>
<td>70,260</td>
</tr>
<tr>
<td></td>
<td>L2i misses:</td>
<td>1,734</td>
</tr>
<tr>
<td></td>
<td>I1 miss rate</td>
<td>0.9%</td>
</tr>
<tr>
<td></td>
<td>L2i miss rate</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>D refs:</th>
<th>39,315,546 (28,535,016 rd + 10,780,530 wr)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D1 misses:</td>
<td>456,530 (344,528 rd + 112,002 wr)</td>
</tr>
<tr>
<td></td>
<td>L2d misses:</td>
<td>249,456 (162,814 rd + 86,642 wr)</td>
</tr>
<tr>
<td></td>
<td>D1 miss rate</td>
<td>1.1% (1.2% + 1.0%)</td>
</tr>
<tr>
<td></td>
<td>L2d miss rate</td>
<td>0.6% (0.5% + 0.8%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>L2 refs:</th>
<th>526,790 (414,788 rd + 112,002 wr)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L2 misses:</td>
<td>251,190 (164,548 rd + 86,642 wr)</td>
</tr>
<tr>
<td></td>
<td>L2 miss rate</td>
<td>0.2% (0.1% + 0.8%)</td>
</tr>
</tbody>
</table>
The End