Guide to callgrind

A CS107 joint effort (Julie & Nate)

CS107 will make you a big fan of Valgrind for its help with memory errors and leaks (cs107 guide to valgrind memcheck (/class/cs107/guide/valgrind.html)), but you may not realize that the `memcheck` tool is not the only utility that Valgrind provides. The Valgrind framework is a platform that supports a variety of runtime analysis tools. Its standard tool suite includes `memcheck` (detects memory errors/leaks), `massif` (reports on heap usage), `helgrind` (detects multithreaded race conditions), and `callgrind/cachegrind` (profiles CPU/cache performance). This tool guide will introduce you to the Valgrind `callgrind` profiling tool.

Code profiling

A **code profiler** is a tool to analyze a program and report on its resource usage (where "resource" is memory, CPU cycles, network bandwidth, and so on). The first step in program optimization is to gather factual, quantitative data from representative program execution using a profiler. The profiling data will give insights into the patterns and peaks of resource consumption so you can determine if there is a concern at all, and if so, where the problem is concentrated, allowing you to focus your efforts on the passages that most need attention. You can also measure and re-measure with the profiler to verify your efforts are bearing fruit.

Most code profilers operate via a *dynamic analysis*--- which is to say that they observe an executing program and take live measurements---as opposed to *static analysis* which examines the source and predicts behavior. Dynamic profiler operate in a variety of ways: some by interjecting counting code into the program, other take samples of its activity at a high-frequency, and others run the program in a simulated environment with built-in monitoring.

The standard C/unix tool for profiling is `gprof` , the gnu profiler. This no-frills tool is a statistical sampler that tracks time spent at the function-level. It take regular snapshots of a running program and traces the function call stack (just like you did in crash reporter!) to observe activity. You can check out the online gprof manual (http://sourceware.org/binutils/docs-2.16/gprof/) if you are curious about its features and use.

The Valgrind profiling tools are cachegrind and callgrind. The cachegrind tool simulates the L1/L2 caches and counts cache misses/hits. The callgrind tool counts function calls and the CPU instructions executed within each call and builds a function callgraph. The callgrind tool includes a cache simulation feature adopted from cachegrind, so you can actually use callgrind for both CPU and cache profiling. The callgrind tool works by instrumenting your program with extra instructions that record activity and keep counters.

Running callgrind and callgrind_annotate

To profile, you run your program under Valgrind and explicitly request the callgrind tool (if unspecified, the tool defaults to memcheck).

```
valgrind --tool=callgrind program-to-run program-arguments
```

The above command starts up valgrind and runs the program inside of it. The program will run normally, albeit a bit more slowly, due to Valgrind's instrumentation. When finished, it briefly summarizes the total number of collected events:

```
==22417== Events    : Ir
==22417== Collected : 7247606
==22417==
==22417== I   refs:      7,247,606
```

Valgrind will have written the information about the above 7 million collected events to an output file named `callgrind.out.id`. (pid is the process id. In the above example, the process id was 22417). This is an ordinary text file, but not intended for you to read directly. Instead, you run the annotator `callgrind_annotate` on this output file to display the information in an useful way:

```
callgrind_annotate --auto=yes callgrind.out.pid
```

The output from the annotator will be given in `Ir` counts which are "instruction read" events. The output shows total number of events occurring within each function (i.e. number of instructions executed) and displays the list of functions sorted in order of decreasing count. Your high-traffic functions will be listed at the top. The option --auto=yes further breaks down the results by reporting counts for each C statement (without the auto option, you get counts summarized at the function level, which is often too coarse to be useful).

In order to additionally monitor cache hits/misses, invoke valgrind callgrind with the simulate-cache option like this:

```
valgrind --tool=callgrind --simulate-cache=yes program-to-run program-arguments
```

The brief summary printed at the end will now include events collected about access to the L1 and L2 caches, as shown below:

```
==16409== Events    : Ir Dr Dw I1mr D1mr D1mw I2mr D2mr D2mw
==16409== Collected : 7163066 4062243 537262 591 610 182 16 103 94
==16409==
==16409== I   refs:      7,163,066
==16409== I1  misses:          591
==16409== L2i misses:           16
==16409== I1  miss rate:       0.0%
==16409== L2i miss rate:       0.0%
==16409==
==16409== D   refs:      4,599,505  (4,062,243 rd + 537,262 wr)
==16409== D1  misses:          792  (      610 rd +     182 wr)
==16409== L2d misses:          197  (      103 rd +      94 wr)
==16409== D1  miss rate:       0.0% (      0.0%   +     0.0%  )
==16409== L2d miss rate:       0.0% (      0.0%   +     0.0%  )
==16409==
==16409== L2 refs:           1,383  (    1,201 rd +     182 wr)
==16409== L2 misses:           213  (      119 rd +      94 wr)
==16409== L2 miss rate:        0.0% (      0.0%   +     0.0%  )
```

When you run callgrind_annotate on this output file, the annotations will now include the cache activity as well as instruction counts.

Interpreting the results

Understanding the Ir counts. The Ir counts are basically the count of assembly instructions executed. A single C statement can translate to 1, 2, or several assembly instructions. Consider the passage below that has been annotated by callgrind_annotate. The profiled program sorted a 1000-member array using selection sort. A single call to the swap function requires 15 instructions: 3 for the prologue, 3 for assign to tmp, 4 for copy from * b to * a, 3 for assign from tmp and 2 more for the epilogue. (Note that the cost of the prologue and epilogue is annotated on the opening and closing braces.) There were 1,000 calls to swap, accounting for a total of 15,000 instructions.

```
       . void swap(int *a, int *b)
       3,000  {
       3,000      int tmp = *a;
       4,000      *a = *b;
       3,000      *b = tmp;
       2,000  }
           .
           . int find_min(int arr[], int start, int stop)
       3,000  {
       2,000      int min = start;
   2,005,000      for(int i = start+1; i <= stop; i++)
   4,995,000          if (arr[i] < arr[min])
       6,178              min = i;
       1,000      return min;
       2,000  }
           . void selection_sort(int arr[], int n)
           3  {
       4,005      for (int i = 0; i < n; i++) {
       9,000          int min = find_min(arr, i, n-1);
   7,014,178  => sorts.c:find_min (1000x)
      10,000          swap(&arr[i], &arr[min]);
      15,000  => sorts.c:swap (1000x)
           .    }
           2  }
           .
```

The callgrind_annotate includes a function call summary, sorted in order of decreasing count, as shown below:

```
7,014,178  sorts.c:find_min [sorts]
   25,059  ???:do_lookup_x [/lib/ld-2.5.so]
   23,010  sorts.c:selection_sort [sorts]
   20,984  ???:_dl_lookup_symbol_x [/lib/ld-2.5.so]
   15,000  sorts.c:swap [sorts]
```

By default, the counts are *exclusive*--- the counts for a function include only the time spent in that function and not in the functions that it calls. For example, the 23,010 instructions counted for the selection_sort function includes the 9,000 instructions to set up and make call to find_min, but not the 7 million instructions that executed in find_min itself. The alternate means of counting is *inclusive* (use the option --inclusive=yes to callgrind_annotate if you prefer this way of accounting) does include the cost of sub-calls in the upper-level totals. In general, using exclusive counts is great way to highlight your bottlenecks -- the functions/statements where the most time is being spent are the ones you want to reduce the number of calls or streamline what happens within a call. The most heavily trafficked passages are easily spotted by looking for the highest counts. In the code above, the work is concentrated in the loop to find the min value -- techniques such as caching the min array element rather than re-fetching and unrolling the loop might be useful here.

Understanding the cache statistics. The cache simulator models a machine with a split L1 cache (separate instruction I1 and data D1), backed by a unified second-level cache (L2). This matches the general cache design of most modern machines, including our myths. The events recorded by the cache simulator are:

- `Ir` : I cache reads (instructions executed)
- `I1mr` : I1 cache read misses (instruction wasn't in I1 cache but was in L2)
- `I2mr` : L2 cache instruction read misses (instruction wasn't in I1 or L2 cache, had to be fetched from memory)
- `Dr` : D cache reads (memory reads)
- `D1mr` : D1 cache read misses (data location not in D1 cache, but in L2)
- `D2mr` : L2 cache data read misses (location not in D1 or L2)

- `Dw` : D cache writes (memory writes)
- `D1mw` : D1 cache write misses (location not in D1 cache, but in L2)
- `D2mw` : L2 cache data write misses (location not in D1 or L2)

Again, the strategy is to use callgrind_annotate to report the hit/miss counts per statement and look for those statements that are contributing a large total number of read/writes or a disproportionate number of cache misses. Even a small number of misses can be quite important, as a L1 miss will typically cost around 5-10 cycles, an L2 miss can cost as much as 100-200 cycles, so shaving even a few of those can be a big boost.

Looking at the annotated result from the previous selection sort program shows the code to be quite cache-friendly -- just a few misses as its traverses along the array, and otherwise, a tremendous number of I1 and D1 hits.

```
--------------------------------------------------------------------------------
-- Auto-annotated source: sorts.c
--------------------------------------------------------------------------------
       Ir         Dr       Dw I1mr D1mr D1mw I2mr D2mr D2mw
        .          .        .    .    .    .    .    .    . void swap(int *a, int *b)
    3,000          0    1,000    1    0    0    1    .    . {
    3,000      2,000    1,000    .    .    .    .    .    .     int tmp = *a;
    4,000      3,000    1,000    .    .    .    .    .    .     *a = *b;
    3,000      2,000    1,000    .    .    .    .    .    .     *b = tmp;
    2,000      2,000        .    .    .    .    .    .    . }
        .          .        .    .    .    .    .    .    .
        .          .        .    .    .    .    .    .    . int find_min(int arr[], int start, int st
op)
    3,000          0    1,000    1    0    0    1    .    . {
    2,000      1,000    1,000    0    0    1    0    0    1      int min = start;
    2,005,000 1,002,000 500,500    .    .    .    .    .    .    for(int i = start+1; i <= st
op; i++)
4,995,000 2,997,000        0    0   32    0    0   19    .        if (arr[i] < arr[m
in])
    6,144      3,072    3,072    .    .    .    .    .    .            min = i;
    1,000      1,000        .    .    .    .    .    .    .    return min;
    2,000      2,000        .    .    .    .    .    .    . }
        .          .        .    .    .    .    .    .    . void selection_sort(int arr[], int n)
        3          0        1    1    0    0    1    .    . {
    4,005      2,002    1,001    .    .    .    .    .    .    for (int i = 0; i < n; i++) {
    9,000      3,000    5,000    .    .    .    .    .    .        int min = find_min(arr, i, n
-1);
    7,014,144 4,006,072 505,572    1   32    1    1   19    1  => sorts.c:find_min
 (1000x)
   10,000      4,000    3,000    .    .    .    .    .    .        swap(&arr[i], &arr[min]);
   15,000      9,000    4,000    1    0    0    1    .    . => sorts.c:swap (1000x)
        .          .        .    .    .    .    .    .    .    }
        2          2        .    .    .    .    .    .    . }
```

Tips and tricks

A few tips about using callgrind effectively:

- Ordinarily, we recommend that you debug/test on code compiled without optimization (i.e. -O0), but measuring performance is a bit different. You want to be executing the optimized code to find out what bottlenecks exist even with the compiler's optimization help.
- Callgrind measures only that code which is executed, so be sure you are making diverse and representative runs that exercise all appropriate code paths.
- You can compare the results from multiple runs to understand the performance variation on different inputs.

- Callgrind records the count of instructions, not the actual time spent in a function. If you have a program where the bottleneck is file I/O, the costs associated with reading and writing files won't show up in the profile, as those are not CPU-intensive tasks.
- If the compiler inlines a function, it will not appear as a separate entry. Instead, the cost of the inlined function will be included in the cost of the caller(s). Also remember that inline keyword is merely advisory; the compiler may inline or not as its discretion.
- The per-function annotation is often too coarse to be useful, the line-by-line counts are key for getting more helpful detail. You can even drop down to observe event counts at the assembly level. Build the code with assembly-level debug information by editing your Makefile to include compiler flags `-Wa,--gstabs -save-temps` . Then when running callgrind, use the option `--dump-instr=yes` which requests counts per assembly instruction. When annotating this output, callgrind_annotate will now match events to assembly statements. Cool!
- L2 misses are much more expensive than L1 misses, so pay attention to passages with high `D2mr` or `D2mw` counts. You can use callgrind_annotate show/sort options to focus on key events, e.g. `callgrind_annotate --show=D2mr --sort=D2mr` will highlight `D2mr` counts.
- There are additional callgrind features that allow you to fine-tune the simulation parameters and control which/when events are collected, as well as many options for the annotator to configure how the report is presented. Refer to the online Callgrind manual (http://valgrind.org/docs/manual/cl-manual.html) for more details.