Lecture 27 — Memory Profiling, Cachegrind

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 27, 2019

Part I

Memory Profiling

~~Memory Profiling~~ Return to Asgard

So far: CPU profiling.

Memory profiling is also a thing;
specifically heap profiling.

"Still Reachable": not freed & still have pointers,
but should have been freed?

As with queueing theory:

$$\text{allocs} > \text{frees} \implies \text{usage} \to \infty$$

At least more paging, maybe total out-of-memory.

But! Memory isn't really lost: we could free it.

Our tool for this comes from the Valgrind tool suite.

Shieldmaiden to Thor

What does Massif do?

- How much heap memory is your program using?
- How did this happen?

Next up: example from Massif docs.

Example Allocation Program

```
#include <stdlib.h>

void g ( void ) {
    malloc( 4000 );
}

void f ( void ) {
    malloc( 2000 );
    g();
}

int main ( void ) {
    int i;
    int* a[10];

    for ( i = 0; i < 10; i++ ) {
        a[i] = malloc( 1000 );
    }
    f();
    g();

    for ( i = 0; i < 10; i++ ) {
        free( a[i] );
    }
    return 0;
}
```

After we compile (remember -g for debug symbols), run the command:

```
jz@Loki:~/ece459$ valgrind --tool=massif ./massif
==25187== Massif, a heap profiler
==25187== Copyright (C) 2003-2013, and GNU GPL'd, by Nicholas Nethercote
==25187== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==25187== Command: ./massif
==25187==
==25187==
```

That Was Useful!!!

What happened?

1. The program ran slowly (because Valgrind!)
2. No summary data on the console
   (like memcheck or helgrind or cachegrind.)

Weird. What we got instead was the file `massif.out.[PID]`.

`massif.out.[PID]`:

plain text, sort of readable.

Better: `ms_print`.

Which has nothing whatsoever to do with Microsoft. Promise.

```
    KB
19.71^                                                                        #
     |                                                                        #
     |                                                                        #
     |                                                                        #
     |                                                                        #
     |                                                                        #
     |                                                                        #
     |                                                                        #
     |                                                                        #
     |                                                                        #
     |                                                                        #
     |                                                                        #
     |                                                                        #
     |                                                                        #
     |                                                                        #
     |                                                                        #
     |                                                                       :#
     |                                                                       :#
     |                                                                       :#
     |                                                                       :#
   0 +------------------------------------------------------------------------>ki
     0                                                                    111.9
```

User Friendly, But Not Useful

For a long time, nothing happens, then...kaboom!

Why? We gave it a trivial program.

We should tell Massif to care more about bytes than CPU cycles, with `--time-unit=B`.

Let's try that.

```
         KB
     19.71^                                               ### <- peak
          |                                               #
          |                                               #  ::
          |                                               #  : ::: <- normal
          |                                      :::::::::#  : :  ::
          |                                      :        #  : :  : ::
          |                                      :        #  : :  : : :::
          |                                      :        #  : :  : : :  ::
          |               detailed     :::::::::::        #  : :  : : :  : :::
          |                     |      :         :        #  : :  : : :  : :  ::
          |                     v  :::::         :        #  : :  : : :  : :  : ::
          |                     @@@:   :         :        #  : :  : : :  : :  : : @
          |                   ::@  :   :         :        #  : :  : : :  : :  : : @
          |                :::: @  :   :         :        #  : :  : : :  : :  : : @
          |              :::  : @  :   :         :        #  : :  : : :  : :  : : @
          |            ::: :  : @  :   :         :        #  : :  : : :  : :  : : @
          |         :::: : :  : @  :   :         :        #  : :  : : :  : :  : : @
          |       :::  : : :  : @  :   :         :        #  : :  : : :  : :  : : @
          |    :::: :  : : :  : @  :   :         :        #  : :  : : :  : :  : : @
          |  :::  : :  : : :  : @  :   :         :        #  : :  : : :  : :  : : @
        0 +----------------------------------------------------------------------->KB
          0                                                                   29.63
```

OK! Massif took 25 snapshots.

- whenever there are appropriate allocation and deallocation statements, up to a configurable maximum.

Long running program:
will toss some old data if necessary.

n	time(B)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
0	0	0	0	0	0
1	1,016	1,016	1,000	16	0
2	2,032	2,032	2,000	32	0
3	3,048	3,048	3,000	48	0
4	4,064	4,064	4,000	64	0
5	5,080	5,080	5,000	80	0
6	6,096	6,096	6,000	96	0
7	7,112	7,112	7,000	112	0
8	8,128	8,128	8,000	128	0

time(B) column = time measured in allocations (our choice of time unit on cmdline).

extra-heap(B) = internal fragmentation.

(Why are stacks all shown as 0?)

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  9          9,144            9,144            9,000           144            0
98.43% (9,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->98.43% (9,000B) 0x4005BB: main (massif.c:17)
```

Now: where did heap allocations take place?

So far, all the allocations took place on line 17, which was `a[i] = malloc( 1000 );` inside that for loop.

Peak Snapshot (Trimmed)

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 14         20,184           20,184           20,000           184            0
99.09% (20,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->49.54% (10,000B) 0x4005BB: main (massif.c:17)
|
->39.64% (8,000B) 0x400589: g (massif.c:4)
| ->19.82% (4,000B) 0x40059E: f (massif.c:9)
| | ->19.82% (4,000B) 0x4005D7: main (massif.c:20)
| |
| ->19.82% (4,000B) 0x4005DC: main (massif.c:22)
|
->09.91% (2,000B) 0x400599: f (massif.c:8)
  ->09.91% (2,000B) 0x4005D7: main (massif.c:20)
```

Massif found all allocations and distilled them to a tree structure.

We see not just where the `malloc` call happened, but also how we got there.

Termination gives a final output of what blocks remains allocated and where they come from.

These point to memory leaks, incidentally, and Memcheck would not be amused.

```
 24         30,344            10,024            10,000             24            0
99.76% (10,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->79.81% (8,000B) 0x400589: g (massif.c:4)
| ->39.90% (4,000B) 0x40059E: f (massif.c:9)
| | ->39.90% (4,000B) 0x4005D7: main (massif.c:20)
| |
| ->39.90% (4,000B) 0x4005DC: main (massif.c:22)
|
->19.95% (2,000B) 0x400599: f (massif.c:8)
| ->19.95% (2,000B) 0x4005D7: main (massif.c:20)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)
```

Here's what Memcheck thinks:

```
jz@Loki:~/ece459$ valgrind ./massif
==25775== Memcheck, a memory error detector
==25775== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==25775== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==25775== Command: ./massif
==25775==
==25775==
==25775== HEAP SUMMARY:
==25775==     in use at exit: 10,000 bytes in 3 blocks
==25775==   total heap usage: 13 allocs, 10 frees, 20,000 bytes allocated
==25775==
==25775== LEAK SUMMARY:
==25775==    definitely lost: 10,000 bytes in 3 blocks
==25775==    indirectly lost: 0 bytes in 0 blocks
==25775==      possibly lost: 0 bytes in 0 blocks
==25775==    still reachable: 0 bytes in 0 blocks
==25775==         suppressed: 0 bytes in 0 blocks
==25775== Rerun with --leak-check=full to see details of leaked memory
==25775==
==25775== For counts of detected and suppressed errors, rerun with: -v
==25775== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Valgrind (Memcheck) First

Run valgrind (Memcheck) first and make it happy before we go into figuring out where heap blocks are going with Massif.

Okay, what to do with the information from Massif, anyway?

Easy!

- Start with peak (worst case scenario) and see where that takes you (if anywhere).
- You can probably identify some cases where memory is hanging around unnecessarily.

Memory usage climbing over a long period of time, perhaps slowly, but never decreasing—memory filling with junk?

Large spikes in the graph—why so much allocation and deallocation in a short period?

Other Massif-ly Useful Things

- stack allocation (`--stacks=yes`).
- children of a process
  (anything split off with `fork`) if desired.
- low level stuff: if going beyond `malloc`, `calloc`, `new`, etc. and using `mmap` or `brk` that is usually missed, can do profiling at page level (`--pages-as-heap=yes`).

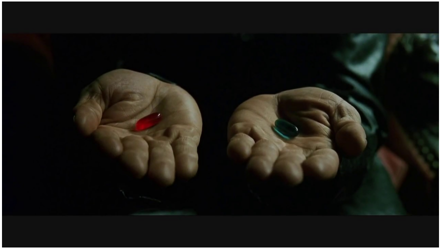
As is often the case,
we have examined the tool on a trivial program.

Let's see if we can do some
live demos of Massif at work.

Part II

Cachegrind

How Would You Know the Difference...



This is much more performance oriented than the other two tools.

It runs a simulation of how your program interacts with cache and evaluates how your program does on branch prediction.

Cachegrind

As we discussed earlier, cache misses and branch mispredicts have a huge impact on performance.

Recall that a miss from the fastest cache results in a small penalty (10 cycles).

A miss that requires going to memory requires about 200 cycles.

A mispredicted branch costs somewhere between 10-30 cycles.

Cachegrind Reporting

Cachegrind reports data about:

- The First Level Instruction Cache (I1) [L1 Instruction Cache]
- The First Level Data Cache (D1) [L1 Data Cache]
- The Last Level Cache (LL) [L3 Cache].

Unlike normal Valgrind operation, you probably want to turn optimizations on.

Sim-u-laaaaate!

```
jz@Loki:~/ece254$ valgrind --tool=cachegrind --branch-sim=yes ./search

--16559-- warning: L3 cache found, using its data for the LL simulation.
Found at 11 by thread 1
Found at 22 by thread 3
==16559==
==16559== I   refs:      310,670
==16559== I1  misses:      1,700
==16559== LLi misses:      1,292
==16559== I1  miss rate:    0.54%
==16559== LLi miss rate:    0.41%
==16559==
==16559== D   refs:      114,078  (77,789 rd   + 36,289 wr)
==16559== D1  misses:      4,398  ( 3,360 rd   +  1,038 wr)
==16559== LLd misses:      3,252  ( 2,337 rd   +    915 wr)
==16559== D1  miss rate:     3.8% (   4.3%     +    2.8%  )
==16559== LLd miss rate:     2.8% (   3.0%     +    2.5%  )
==16559==
==16559== LL refs:         6,098  ( 5,060 rd   +  1,038 wr)
==16559== LL misses:       4,544  ( 3,629 rd   +    915 wr)
==16559== LL miss rate:      1.0% (   0.9%     +    2.5%  )
==16559==
==16559== Branches:       66,622  (65,097 cond +  1,525 ind)
==16559== Mispredicts:     7,202  ( 6,699 cond +    503 ind)
==16559== Mispred rate:     10.8% (  10.2%     +   32.9%   )
```

Optimizations: Enabled!

```
jz@Loki:~/ece254$ valgrind --tool=cachegrind --branch-sim=yes ./search

--16618-- warning: L3 cache found, using its data for the LL simulation.
Found at 11 by thread 1
Found at 22 by thread 3
==16618==
==16618== I   refs:      306,169
==16618== I1  misses:      1,652
==16618== LLi misses:      1,286
==16618== I1  miss rate:    0.53%
==16618== LLi miss rate:    0.42%
==16618==
==16618== D   refs:      112,015  (76,522 rd   + 35,493 wr)
==16618== D1  misses:      4,328  ( 3,353 rd   +    975 wr)
==16618== LLd misses:      3,201  ( 2,337 rd   +    864 wr)
==16618== D1  miss rate:     3.8% (   4.3%     +    2.7%  )
==16618== LLd miss rate:     2.8% (   3.0%     +    2.4%  )
==16618==
==16618== LL refs:         5,980  ( 5,005 rd   +    975 wr)
==16618== LL misses:       4,487  ( 3,623 rd   +    864 wr)
==16618== LL miss rate:      1.0% (   0.9%     +    2.4%  )
==16618==
==16618== Branches:       65,827  (64,352 cond +  1,475 ind)
==16618== Mispredicts:     7,109  ( 6,596 cond +    513 ind)
==16618== Mispred rate:     10.7% (  10.2%     +   34.7%   )
```

Interesting results: our data and instruction miss rates went down marginally but the branch mispredict rates went up!

Well sort of - there were fewer branches and thus fewer we got wrong as well as fewer we got right.

So the total cycles lost to mispredicts went down.

Is this an overall win for the code? Yes.

Do the Math

In some cases it's not so clear cut, and we could do a small calculation.

If we just take a look at the LL misses (4 544 vs 4 487) and assume they take 200 cycles, and the branch miss penalty is 200 cycles, it went from 908 800 wasted cycles to 897 400. This is a decrease of 11 400 cycles.

Repeat for each of the measures and sum them up to determine if things got better overall and by how much.

Cachegrind Detailed Output

Cachegrind also produces a more detailed output file, titled cachegrind.out.<pid> (the PID in the example is 16618).

This file is not especially human-readable, but we can ask the associated tool `cg_annotate` to break it down for us.

The output is way too big for slides.

Advanced Cachegrind

Cachegrind is very... verbose... and it can be very hard to come up with useful changes based on what you see...

Assuming your eyes don't glaze over when you see the numbers.

Probably the biggest performance impact is last level cache misses (those appear as DLmr or DLmw).

You might also try to look at the Bcm and Bim (branch mispredictions) to see if you can give some better hints.

Of course, to learn more about how Cachegrind, the manual is worth reading.

Not that anybody reads manuals anymore...

Just give it a shot, when you get stuck, google the problem, click the first stack overflow link result...