ECE459: Programming for Performance Winter 2021

Lecture 28 — Memory Profiling, Cachegrind

Jeff Zarnett & Patrick Lam *2021-01-02*

Memory Profiling

Thus far we have focused on CPU profiling. Other kinds of profiling got some mention, but they're not the only kind of profiling we can do. Memory profiling is also a thing, and specifically we're going to focus on heap profiling.

During Rustification, we dropped a bunch of valgrind content. Valgrind is a memory error and leak detection toolset that is invaluable when programming in C++. (In brief: valgrind is a just-in-time engine that reinterprets machine code and instruments it to find sketchy things.) In particular, the memory leaks that valgrind's memcheck tool detects are, for the most part, not possible in Rust unless you try pretty hard[1]—the friendly compiler protects you against them by automatically dropping things.

We're going to look at two kinds of memory profiling here, both part of valgrind. They can both occur in Rust, but it looks like the more common kind is when a lot of memory is allocated and then freed. More completely: "places that cause excessive numbers of allocations; leaks; unused and under-used allocations; short-lived allocations; and allocations with inefficient data layouts". valgrind's DHAT tool[Dev20, Chapter 10] detects these.

The second kind of memory profiling addresses memory that memcheck reports as "Still Reachable". That is, things that remained allocated and still had pointers to them, but were not properly deallocated. Right, we care about them too, and for that we also want to do heap profiling. If we don't look after those things, we're just using more and more memory over time. That likely means more paging and the potential for running out of heap space altogether. Again, the memory isn't really lost, because we could free it.

DHAT

DHAT tracks allocation points and what happens to them over a program's execution. An allocation point is a point in the program which allocates memory; in C that would be `malloc`. In Rust there are a bunch of ways to allocate memory, one being `Box::new`. It aggregates allocation points according to the program stack. Let's first look at a leaf node from the documentation.

```
AP 1.1.1.1/2 {
  Total:     31,460,928 bytes (2.32%, 1,565.9/Minstr) in 262,171 blocks (4.41%, 13.05/Minstr),
     avg size 120 bytes, avg lifetime 986,406,885.05 instrs (4.91% of program duration)
  Max:       16,779,136 bytes in 65,543 blocks, avg size 256 bytes
  At t-gmax: 0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  At t-end:  0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  Reads:     5,964,704 bytes (0.11%, 296.88/Minstr), 0.19/byte
  Writes:    10,487,200 bytes (0.51%, 521.98/Minstr), 0.33/byte
  Allocated at {
    ^1: 0x95CACC9: alloc (alloc.rs:72)
      [omitted]
    ^7: 0x95CACC9: parse_token_trees_until_close_delim (tokentrees.rs:27)
    ^8: 0x95CACC9: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::
                          StringReader<'a>>::parse_token_tree (tokentrees.rs:81)
    ^9: 0x95CAC39: parse_token_trees_until_close_delim (tokentrees.rs:26)
```

[1] You can still create cycles in refcounted objects or call `std::mem::forget` or etc.

```
    ^10: 0x95CAC39: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::
                          StringReader<'a>>::parse_token_tree (tokentrees.rs:81)
    #11: 0x95CAC39: parse_token_trees_until_close_delim (tokentrees.rs:26)
    #12: 0x95CAC39: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::
                          StringReader<'a>>::parse_token_tree (tokentrees.rs:81)
  }
}
```

We can see the program point corresponding to this allocation: line 81 of `tokentrees.rs`. Furthermore, the most bytes alive at once from this site was 16,779,136: that was 65,543 instances of the memory allocated here, almost all 256-byte blocks. In total, this allocation site was executed 262,171 times, accounting for 31,460,928 bytes. This information can tell us how important this allocation site is compared to the entire execution we're inspecting. We can also see how long-lived the blocks are: on average, a block allocated here lives 4.91% of the program duration. If instead of 4.91% we had 0.004% and a large total, then the allocation point would be excessively transient and perhaps a good candidate for pooling allocations.

The Reads and Writes lines tell us how often the allocated memory is actually used. In this case, 19% of the bytes allocated here are ever read, and 33% are written to. That seems like not many, but if it's something like a vector that grows, you may not be able to do anything about that. This, however, doesn't look like a vector—you could check it by looking at the code.

You might also get a super-useful Accesses line if all of the blocks allocated at a site are the same size. That line tells you how many times each byte in the block was accessed. Zeros, written as dashes (-), are parts of the block that are never accessed, useful for finding data alignment holes (and hence potential inefficiencies).

Parents. Going up the tree, DHAT tells you about all of the allocation points that share a call-stack prefix. (The ^ before the number indicates that the line is copied from the parent; a # indicates that the line is unique to that node.)

A sibling of the example above would diverge at the call on line 10, not calling `parse_token_trees_until_close_delim` from line 81 of `tokentrees.rs`. At the parent, DHAT reports results for that node and all of its children.

~~Memory Profiling~~ Return to Asgard

JZ generally found a way to talk about Valgrind previously. But now, we're going to use its Massif tool. This is, obviously, a joke on "massive", combined with the name Sif, a Norse goddess associated with the earth (and in the Marvel movies, Shieldmaiden to Thor). While we're on the subject, Sif has an axe (shield?) to grind with Loki, because at some point he cut off her golden hair (and in the Marvel films, it grew back in dark). That Loki—what a trickster! Right, we're digressing…what do you mean the course isn't ECE 459: Norse Mythology?!

So what does Massif do? It will tell you about how much heap memory your program is using, and also how the situation got to be that way. So let's start with the example program from the documentation, badly translated by yours truly to non-idiomatic Rust [Dev16]:

```
fn g() {
    let a = Vec::<u8>::with_capacity(4000);
    std::mem::forget(a)
}

fn f() {
    let a = Vec::<u8>::with_capacity(2000);
    std::mem::forget(a);
    g()
}

fn main() {

    let mut a = Vec::with_capacity(10);
    for _i in 0..10 {
a.push(Box::new([0;1000]))
    }
    f();
    g();
}
```

After we compile, run the command:

```
plam@amqui ~/c/p/l/l/L/alloc> valgrind --tool=massif target/debug/alloc
==406569== Massif, a heap profiler
==406569== Copyright (C) 2003-2017, and GNU GPL'd, by Nicholas Nethercote
==406569== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==406569== Command: target/debug/alloc
==406569==
==406569==
```

Doesn't that look useful?! What happened? Your program executed slowly, as is always the case with any of the Valgrind toolset, but you don't get summary data on the console like we did with Valgrind or helgrind or cachegrind. Weird. What we got instead was the file `massif.out.406569` (matches the PID of whatever we ran). This file, which you can open up in your favourite text editor, is not especially human readable, but it's not incomprehensible like the output from cachegrind ("Aha, a 1 in column 4 of line 2857. That's what's killing our performance!"). There is an associated tool for summarizing and interpreting this data in a much nicer way: `ms_print`, which has nothing whatsoever to do with Microsoft. Promise.

If we look at the output there (hint: pipe the output to `less` or something, otherwise you get a huge amount of data thrown at the console), it looks much more user friendly.

```
    KB
78.57^                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                      :#
     |                                                                     @:#
     |                                                                    :@:#
     |                                                                    :@:#
     |                                                                   ::@:#
     |                                                                  :::@:#
     |                                                                  :::@:#
     |                                                                 ::::@:#
     |                                                                :::::@:#
   0 +----------------------------------------------------------------------->ki
     0                                                                   474.1
```

Now wait a minute. This bar graph might be user friendly but it's not exactly what I'd call…useful, is it? For a long time, nothing happens, then…kaboom! According to the docs, what actually happened here is, we gave in a trivial program where most of the CPU time was spent doing the setup and loading and everything, and the trivial program ran for only a short period of time, right at the end. So for a relatively short program we should tell Massif to care more about the bytes than the CPU cycles, with the `--time-unit=B` option. Let's try that.

```
    KB
78.57^                                               ##
     |                                               # :::
     |                                               # :  ::
     |                                               # :  : ::
     |                                      :::::::::# :  : : :::
     |                                      :        # :  : : :  ::
     |                                      :        # :  : : :  : ::
     |                                      :        # :  : : :  : : :::
```

```
  |                              ::::::::::        # :  : : :  : : :  ::
  |                              :        :        # :  : : :  : : :  : ::
  |                         :::::::        :        # :  : : :  : : :  : : @
  |                       :::    :        :        # :  : : :  : : :  : : @
  |                    @@@: :    :        :        # :  : : :  : : :  : : @
  |                  ::@  : :    :        :        # :  : : :  : : :  : : @
  |                ::: @  : :    :        :        # :  : : :  : : :  : : @
  |             :::: : @  : :    :        :        # :  : : :  : : :  : : @
  |           :::  : : @  : :    :        :        # :  : : :  : : :  : : @
  |         ::: :  : : @  : :    :        :        # :  : : :  : : :  : : @
  |      :::: : :  : : @  : :    :        :        # :  : : :  : : :  : : @
  |    :::  : : :  : : @  : :    :        :        # :  : : :  : : :  : : @
0 +----------------------------------------------------------------------->KB
  0                                                                   121.6
```

Neat. Now we're getting somewhere. We can see (from the text below the graph) that 48 snapshots were taken. It will take snapshots whenever there are appropriate allocation and deallocation statements, up to a configurable maximum, and for a long running program, toss some old data if necessary. Let's look in the documentation to see what the symbols mean (they're not just to look pretty). So, from the docs [Dev16]:

- Most snapshots are normal (they have just basic information) They use the ':' characters.
- Detailed snapshots are shown with '@' characters. By default, every 10th snapshot is detailed.
- There is at most one peak snapshot. The peak snapshot is a detailed snapshot, and records the point where memory consumption was greatest. The peak snapshot is represented in the graph by a bar consisting of '#' characters.

As a caveat, the peak can be a bit inaccurate. Peaks are only recorded when a deallocation happens. This just avoids wasting time recording a peak and then overwriting it; if you are allocating a bunch of blocks in succession (e.g. a bunch of structs that have a buffer) then you would constantly be overwriting the peak over and over again. Also, there's some loss of accuracy to speed things up. Well, okay.

So let's look at the snapshots. We'll start with the normal ones. Here are the first 5, numbers 0 through 4:

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0                0             0            0
  1            488              488              472            16            0
  2            624              624              592            32            0
  3          1,656            1,656            1,616            40            0
  4          1,768            1,768            1,736            32            0
```

The columns are pretty much self explanatory, with a couple exceptions. The time(B) column corresponds to time measured in allocations thanks to our choice of the time unit at the command line. The extra-heap(B) represents internal fragmentation[2] in the blocks we received. The stacks column shows as zero because by default, Massif doesn't look at the stack. It's a heap profiler, remember?

Number 5 is a "detailed" snapshot, so I've separated it out, and reproduced the headers there to make this a little easier to remember what they are.

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  5          1,768            1,768            1,736            32            0
```

[2] Remember from operating systems: if the user asked for some n bytes where n is not a nice multiple the returned block may be "rounded up". So a request for 1000 bytes is bumped up to 1016 bytes in this example. The extra space is "wasted" but it's nicer than having a whole bunch of little tiny useless fragments of the heap to be managed.

```
98.19% (1,736B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->57.92% (1,024B) 0x492509B: _IO_file_doallocate (filedoalloc.c:101)
... etc
```

So the additional information we got here is a reflection of where our heap allocations took place. Thus far, all the allocations are associated with the runtime library and `lang_start`, so not that useful to us.

Then let's look at the peak snapshot (again, trimmed significantly to call out the first thing we need to see here):

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 31         84,072           80,456           80,285          171            0
99.79% (80,285B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->99.53% (80,080B) 0x10F97B: alloc::alloc::alloc (alloc.rs:81)
| ->49.82% (40,080B) 0x10F771: <alloc::alloc::Global as core::alloc::Alloc>::alloc (alloc.rs:169)
| | ->49.72% (40,000B) 0x10D90D: alloc::raw_vec::RawVec<T,A>::allocate_in (raw_vec.rs:88)
| | | ->49.72% (40,000B) 0x10D744: alloc::raw_vec::RawVec<T>::with_capacity (raw_vec.rs:140)
| | |   ->49.72% (40,000B) 0x10C412: alloc::vec::Vec<T>::with_capacity (vec.rs:355)
| | |     ->39.77% (32,000B) 0x10FCF2: alloc::g (main.rs:2)
| | |     | ->19.89% (16,000B) 0x10FD5F: alloc::f (main.rs:9)
```

Massif has found all the allocations in this program and distilled them down to a tree structure that traces the path through which all of these various memory allocations occurred. So not just where the `malloc` call happened, but also how we got there. With lots of Rust helper functions as well.

If I ask valgrind what it thinks of this program, it says:

```
plam@amqui ~/c/p/l/l/L/alloc> valgrind target/debug/alloc
==406822== Memcheck, a memory error detector
==406822== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==406822== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==406822== Command: target/debug/alloc
==406822==
==406822==
==406822== HEAP SUMMARY:
==406822==     in use at exit: 40,000 bytes in 3 blocks
==406822==   total heap usage: 25 allocs, 22 frees, 82,177 bytes allocated
==406822==
==406822== LEAK SUMMARY:
==406822==    definitely lost: 40,000 bytes in 3 blocks
==406822==    indirectly lost: 0 bytes in 0 blocks
==406822==      possibly lost: 0 bytes in 0 blocks
==406822==    still reachable: 0 bytes in 0 blocks
==406822==         suppressed: 0 bytes in 0 blocks
==406822== Rerun with --leak-check=full to see details of leaked memory
==406822==
==406822== For lists of detected and suppressed errors, rerun with: -s
==406822== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

We used to say: "So probably a good idea to run valgrind first and make it happy before we go into figuring out where heap blocks are going with Massif." Now, well, just use Rust and then DHAT. But Massif is still another tool at your disposal. Okay, what to do with the information from Massif, anyway? It should be pretty easy to act upon this information. Start with the peak snapshot (worst case scenario) and see where that takes you (if anywhere). You can probably identify some cases where memory is hanging around unnecessarily.

Things to watch out for:

- memory usage climbing over a long period of time, perhaps slowly, but never really decreasing—memory is filling up somehow with some junk?

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

Other cool things we can do with Massif [Dev16]:

- Look into stack allocation (`--stacks=yes`) option. This slows stuff down a lot, and not really necessary since we want to look at heap.
- Look at the children of a process (anything split off with `fork`) if desired.
- Check low level stuff: if we're doing something other than `malloc`, `calloc`, `new`, etc. and doing low level stuff like `mmap` or `brk` that is usually missed, but we can do profiling at page level (`--pages-as-heap=yes`).

As is often the case, we have examined how the tool works on a trivial program. As a live demo, well, tune in and be surprised!

Cachegrind

Cachegrind is another tool in the package and this one is much more performance oriented than the other two tools. Yes, Valgrind's memcheck and Helgrind look for errors in your program that are likely to lead to slowdowns (memory leaks) or make it easier to parallelize (spawn threads) without introducing errors. Cachegrind, however, does a simulation of how your program interacts with cache and evaluates how your program does on branch prediction. 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 (perhaps, 10 cycles); a miss that requires going to memory requires about 200 cycles. A mispredicted branch costs somewhere between 10-30 cycles. All figures & estimates from the cachegrind manual [Dev15].

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 for normal Valgrind operation, you probably want to turn optimizations on (`-O2` or perhaps `-O3` in `gcc`). You still want debugging symbols, of course, but enabling optimizations will tell you more about what is going to happen in the released version of your program.

In the `live-coding/L28` directory you will find `search.c` from ECE 254. Using the `-branch-sim=yes` option because by default it won't show it:

```
jz@Loki:~/ece254$ valgrind --tool=cachegrind --branch-sim=yes ./search
==16559== Cachegrind, a cache and branch-prediction profiler
==16559== Copyright (C) 2002-2013, and GNU GPL'd, by Nicholas Nethercote et al.
==16559== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==16559== Command: ./search
==16559==
--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%   )
```

So we see a breakdown of the instruction accesses, data accesses, and how well the last level of cache (L3 here) does.

Why did I say enable optimization? Well, here's the output of the search program if I compile with the `-O2` option:

```
jz@Loki:~/ece254$ valgrind --tool=cachegrind --branch-sim=yes ./search
==16618== Cachegrind, a cache and branch-prediction profiler
==16618== Copyright (C) 2002-2013, and GNU GPL'd, by Nicholas Nethercote et al.
==16618== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==16618== Command: ./search
==16618==
--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.

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; 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. Also be sure that you're reasoning about a realistic workload.

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, and if we have the source code available, so much the better, because it will give you line by line information. That's way too much to show even in the notes, so it's the sort of thing I can show in class (or you can create for yourself) but here's a small excerpt from the `search.c` example:

```
--------------------------------------------------------------------------------
-- Auto-annotated source: /home/jz/ece254/search.c
--------------------------------------------------------------------------------
 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw Bc Bcm Bi Bim
127    1    1 96    3    0  4    0    0 23  11  0   0      for ( int i = arg->startIndex; i < arg->endIndex; ++i ) {
147    0    0 84    3    2  0    0    0 21   9  0   0          if ( array[i] == arg->searchValue ) {
  6    0    0  4    0    0  2    0    0  0   0  0   0              *result = i;
  2    0    0  0    0    0  0    0    0  0   0  0   0              break;
  .    .    .  .    .    .  .    .    .  .   .  .   .          }
  .    .    .  .    .    .  .    .    .  .   .  .   .      }
```

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). They have the highest penalty. You might also try to look at the Bcm and Bim (branch mispredictions) to see if you can give some better hints about what the likelihood of branch prediction is [Dev15]. Of course, to learn more about how Cachegrind actually does what it does and how it runs the simulation, 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...

References

[Dev15] Valgrind Developers. Cachegrind: a cache and branch-prediction profiler, 2015. Online; accessed 25-November-2015. URL: `http://valgrind.org/docs/manual/cg-manual.html`.

[Dev16] Valgrind Developers. Massif: a heap profiler, 2016. Online; accessed 23-January-2016. URL: `http://valgrind.org/docs/manual/ms-manual.html`.

[Dev20] Valgrind Developers. Valgrind documentation, 2020. Online; accessed 13-December-2020. URL: `https://www.valgrind.org/docs/manual/index.html`.