Exercise 04

VU Performance-oriented Computing, Summer Semester 2024

Calvin Hoy

2024-04-13

(A) Memory profiling

Only snapshots are captured by `massif`, so not every heap allocation or deallocation is captured. There are two kinds of snapshots – normal snapshots and detailed snapshots. By default, only every tenth snapshot and/or the peak snapshot is a detailed snapshot, as can be seen in fig. 1.

All measurements were taken on a laptop with an i5-8250U and averaged over five runs. Wall, user and system time, as well as memory use were measured. Runtimes are specified in seconds, memory use in KiB.

The build flags used were `-O2 -g` (CMake profile "RelWithDebInfo").

`npb_bt`

`npb_bt` does not perform any explicit calls to `malloc` or `calloc`. some heap allocations are still observed however, as the code calls out to `fopen` twice, as well as `printf`, and both these functions use the heap. `printf` takes the larger chunk of memory at 1 KiB.

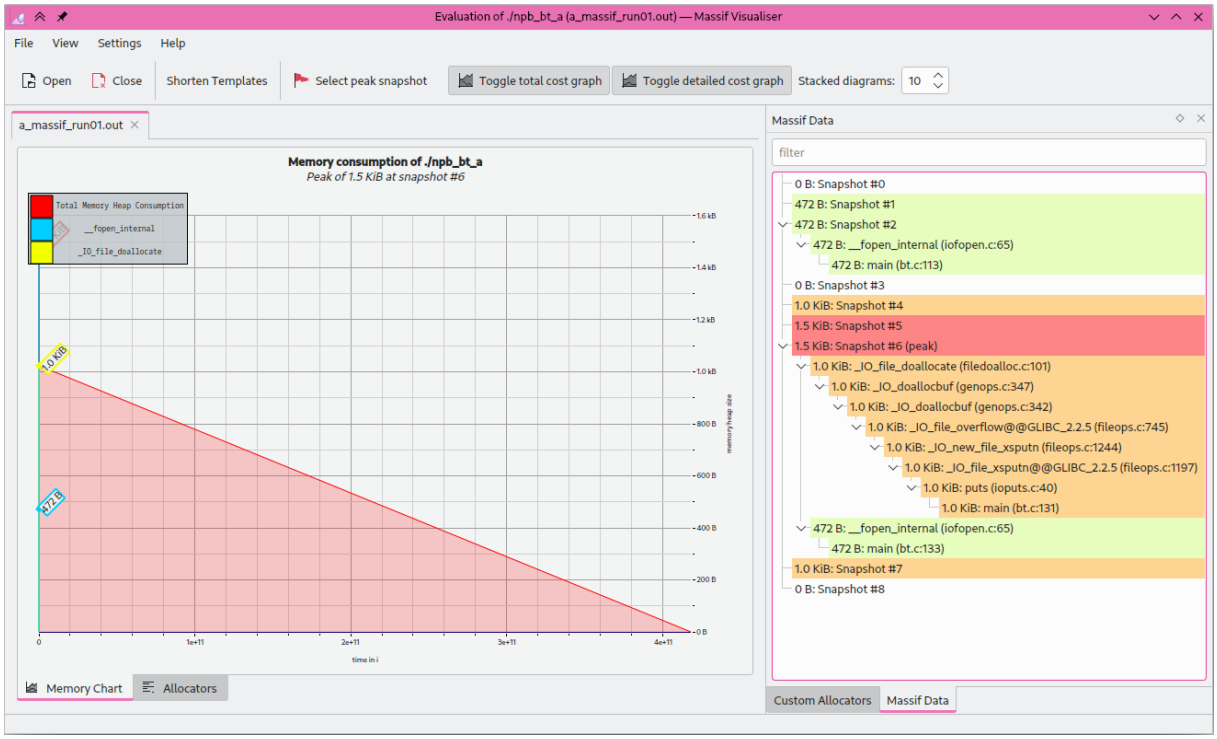

Figure 1: Massif Visualizer – `npb_bt` – Memory Chart

`ssca2 17`

`ssca2` performs heap allocations in the `computeGraph` function, as well as on initialization of the `sprng` library. The latter are performed via a function called `mymalloc` defined in `sprng2.0/memory.{h,c}` – but this is just a wrapper around `malloc`, so no custom allocator needs to be accounted for.

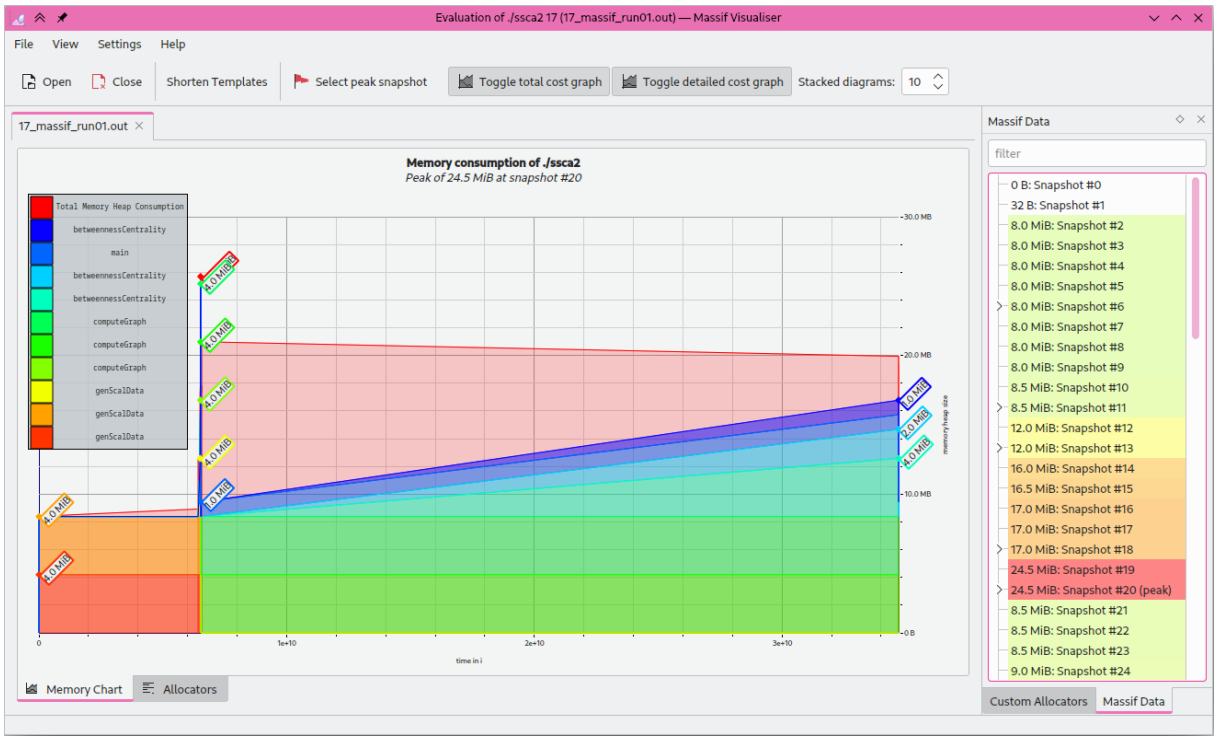

Figure 2: Massif Visualizer – `ssca2` – Memory Chart

The chart shows that memory usage peaks 6.5 seconds into the execution at 24.5 MB, nearly immediately drops down to 21 MiB, then slowly tapers off to 20 MiB before freeing everything. It is unclear to me why snapshot #21, immediately after the peak snapshot #20, only shows 8.5 MiB used.

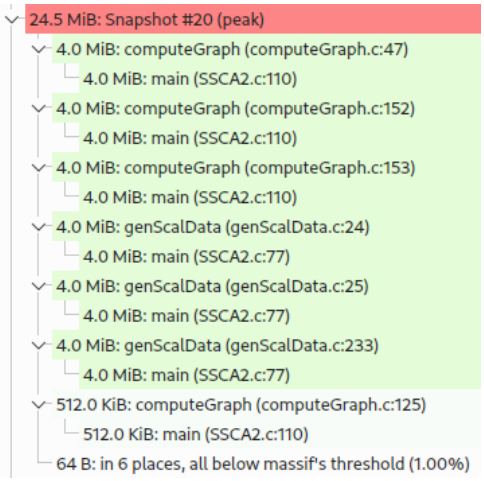

Figure 3: Massif Visualizer – `ssca2` – peak snapshot

Function	Peak	Location
genScalData	4.0 MiB	genScalData.c:24
genScalData	4.0 MiB	genScalData.c:25
genScalData	4.0 MiB	genScalData.c:233
computeGraph	4.0 MiB	computeGraph.c:47
computeGraph	4.0 MiB	computeGraph.c:152
computeGraph	4.0 MiB	computeGraph.c:153
betweennessCentrality	4.0 MiB	betweennessCentrality.c:168
betweennessCentrality	2.0 MiB	betweennessCentrality.c:144
main	1.0 MiB	SSCA2.c:159
betweennessCentrality	1.0 MiB	betweennessCentrality.c:196
betweennessCentrality	1.0 MiB	betweennessCentrality.c:198
findSubGraphs	512.0 KiB	findSubGraphs.c:46
computeGraph	512.0 KiB	computeGraph.c:125
genScalData	512.0 KiB	genScalData.c:146
findSubGraphs	512.0 KiB	findSubGraphs.c:50
computeGraph	512.0 KiB	computeGraph.c:49
betweennessCentrality	512.0 KiB	betweennessCentrality.c:63
betweennessCentrality	512.0 KiB	betweennessCentrality.c:195
betweennessCentrality	512.0 KiB	betweennessCentrality.c:197
below threshold	20.1 KiB	

Figure 4: Massif Visualizer – `ssca2` – Allocators tab

(B) Measuring CPU counters

There are eight hardware events mentioned by `perf list`:

- bus-cycles
- cpu-cycles
- instructions
- ref-cycles
- branch-instructions
- branch-misses
- cache-references
- cache-misses

These were passed in a comma-separated list to `perf stat -e` before the command itself.

When enabling all eight counters at the same time, `perf` is not able to capture all events, and the number reported is an estimate. Hence, only four counters each were enabled across two runs per workload.

npb_bt_w

```
     48,838,508 bus-cycles
  6,901,882,216 cpu-cycles
 19,196,489,407 instructions              #    2.78  insn per cycle
  3,662,887,875 ref-cycles
    354,754,182 branch-instructions
      1,365,383 branch-misses             #    0.38% of all branches
     99,216,534 cache-references
      2,637,581 cache-misses              #    2.66% of all cache refs
```

npb_bt_w

```
  1,329,918,572 bus-cycles
153,854,135,608 cpu-cycles
418,057,831,024 instructions              #    2.72  insn per cycle
 99,743,892,150 ref-cycles
  7,621,459,135 branch-instructions
     18,028,247 branch-misses             #    0.24% of all branches
  3,693,856,288 cache-references
  1,286,734,900 cache-misses              #   34.83% of all cache refs
```

`ssca2 17`

```
   558,453,580 bus-cycles
75,121,662,149 cpu-cycles
34,373,965,057 instructions           #    0.46  insn per cycle
41,884,017,975 ref-cycles
 5,722,756,148 branch-instructions
   631,545,309 branch-misses          #   11.04% of all branches
 9,241,032,529 cache-references
 2,626,042,706 cache-misses           #   28.42% of all cache refs
```

Notable observations:

- `npb_bt` has very good branch prediction behaviour, missing only 0.38%/0.24% of branches for workload sizes W and A, respectively. This is in stark contrast to `ssca2`, which misses 11.04% of all branches.
- `npb_bt` executes 2.78/2.72 instructions per cycle on average, while `ssca2` only manages 0.46 – likely due in part to the many branch misses.
- The cache behaviour of `npb_bt` is better for workload size W than for workload size A.

(A, B) Performance impact

		Mean				Variance			
Workload	profiler	wall	user	system	mem	wall	user	system	mem
`npb_bt_w`	none	2.084	2.082	0.000	4440.0	0.007	0.007	0.000	4000.0
	massif	27.960	27.884	0.032	42646.4	0.002	0.002	0.000	9508.8
	perf	2.064	2.050	0.002	15770.4	0.001	0.001	0.000	2676.8
`npb_bt_a`	none	49.596	49.524	0.002	46419.2	3.182	3.212	0.000	5579.2
	massif	595.982	595.054	0.194	83493.6	0.297	0.105	0.002	851796.8
	perf	48.054	47.970	0.006	46330.4	6.424	6.452	0.000	5532.8
`ssca2 17`	none	21.186	21.108	0.002	27560.0	1.049	1.044	0.000	7336.0
	massif	53.554	53.270	0.124	65593.6	0.155	0.166	0.001	5004.8
	perf	22.500	22.402	0.014	27508.0	0.141	0.139	0.000	3128.0

For `npb_bt_w`, wall time increases roughly 13.4x when using `massif`, and memory use roughly 9.6x. Roughly the same increase in execution time (~12x) was observed for the larger workload size `npb_bt_a`, while memory use showed a ~1.8x increase.

For `ssca2 17`, the increase is ~2.53x for wall time and ~2.38x for memory use. System time, though it remains insignificant, sees an increase of at least two orders of magnitude in all cases.

Meanwhile, `perf` using only hardware counters effectively has no effect on performance, deviating from unprofiled runs by no more than 5%.