

# Lecture 27 — Memory Profiling, Profiler Guided Optimization

Jeff Zarnett  
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering  
University of Waterloo

December 9, 2019

# Part I

## Memory Profiling

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:

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

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==
```



What happened?

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

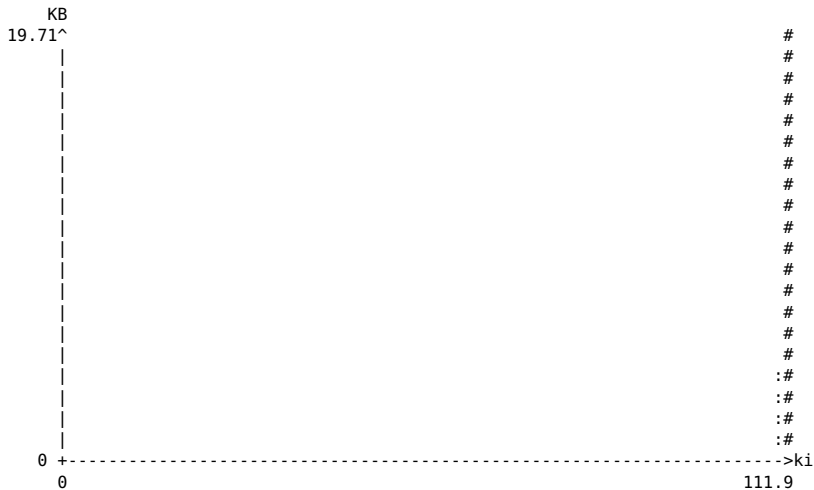
Weird. What we got instead was the file  
`massif.out.[PID]`.

`massif.out.[PID]:`  
plain text, sort of readable.

Better: `ms_print`.

Which has nothing whatsoever to do with  
Microsoft. Promise.

# Post-Processed Output



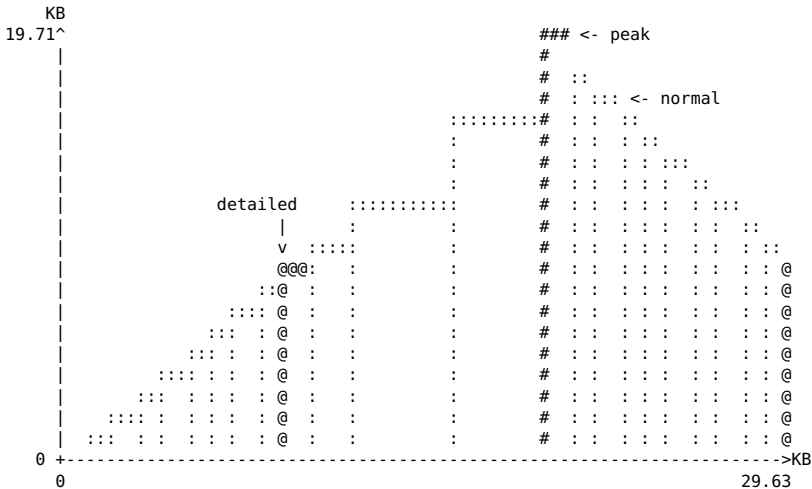
For a long time, nothing happens,  
then...kaboom!

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.

# ASCII Art (telnet towel.blinkenlights.nl)



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.

# “Is he dead?” “Terminated.”

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== 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)
```

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?

- 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

# Profiler Guided Optimization



Using static analysis,  
the compiler makes its best predictions  
about runtime behaviour.

Example: branch prediction.

---

```
void whichBranchIsTaken(int a, int b)
{
    if (a < b) {
        puts("a is less than b.");
    } else {
        puts("b is >= a.");
    }
}
```

---

---

```
void devirtualization(int count)
{
    for (int i = 0; i < count; i++)
    {
        (*p) (x, y);
    }
}
```

---

---

```
void switchCaseExpansion(int i)
{
    switch (i)
    {
        case 1:
            puts("I took case 1.");
            break;
        case 2:
            puts("I took case 2.");
            break;
    }
}
```

---

How can we know where we go?

- could provide hints...

Java HotSpot virtual machine:  
updates predictions on the fly.

So, just guess.  
If wrong, the Just-in-Time compiler adjusts &  
recompiles.

The compiler runs and it does its job and that's it;  
the program is never updated with newer  
predictions if more data becomes known.

C: usually no adaptive runtime system.

POGO:

- observe actual runs;
- predict the future.

So, we need multi-step compilation:

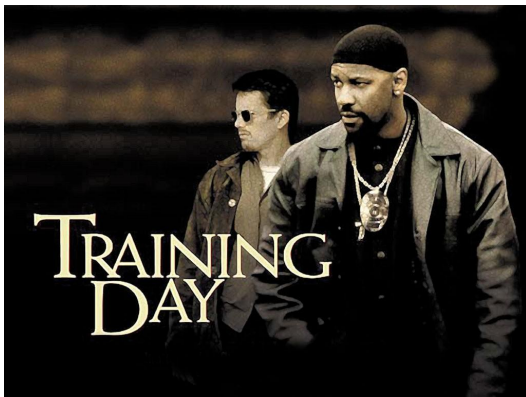
- compile with profiling;
- run to collect data;
- recompile with profiling data to optimize.

First, generate an executable with instrumentation.

The compiler inserts a bunch of probes into the generated code to record data.

- Function entry probes;
- Edge probes;
- Value probes.

Result: instrumented executable  
plus empty database file (for profiling data).





Second, run the instrumented executable.

Real-world scenarios are best.

Ideally, spend training time on perf-critical sections.

Use as many runs as you can stand.

Don't exercise every part of the program  
(ain't SE 465 here!)

That would be counterproductive.

Usage data must match real world scenarios,  
or compiler gets misfacts about what's important.

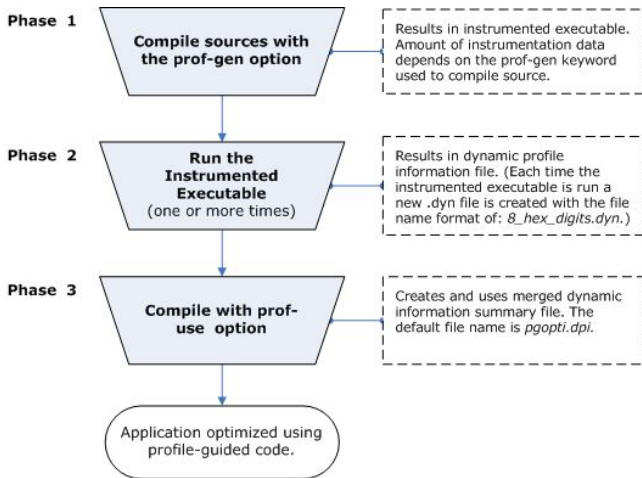
Or you might end up teaching it that almost nothing  
is important... (“everything's on the exam!”)

Finally, compile the program again.

Inputs: source plus training data.

Outputs: (you hope) a better output executable than from static analysis alone.

# Summary Graphic



Not necessary to do all three steps for every build.

Re-use training data while it's still valid.

Recommended dev workflow:

- dev A performs these steps,  
checks the training data into source control
- whole team can use profiling information for  
their compiles.

# Not fixing all the problems in the world

What does it mean for it to be better?

The algorithms will aim for speed in areas that are “hot”.

The algorithms will aim for minimal code size in areas that are “cold” .

Less than 5% of methods compiled for speed.

Can combine multiple training runs and manually give suggestions about important scenarios.

The more a scenario runs in the training data, the more important it will be, from POGO's point of view.

Can merge multiple runs with user-assigned weightings.

In the optimize phase, compiler uses the training data for:

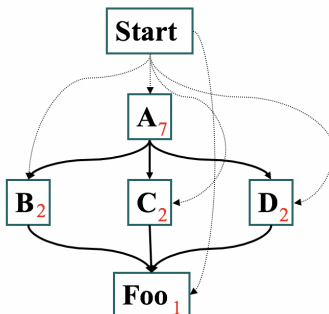
- 1 Full and partial inlining
- 2 Function layout
- 3 Speed and size decision
- 4 Basic block layout
- 5 Code separation
- 6 Virtual call speculation
- 7 Switch expansion
- 8 Data separation
- 9 Loop unrolling



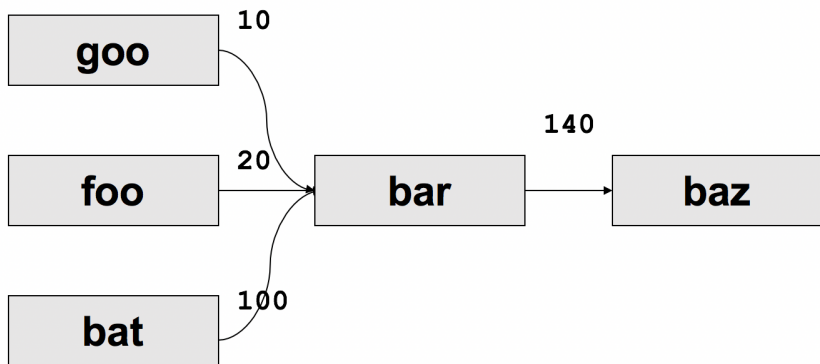
Most performance gains from inlining.

Decisions based on the call graph path profiling.

But: behaviour of function foo may be very different when called from B than when called from D.

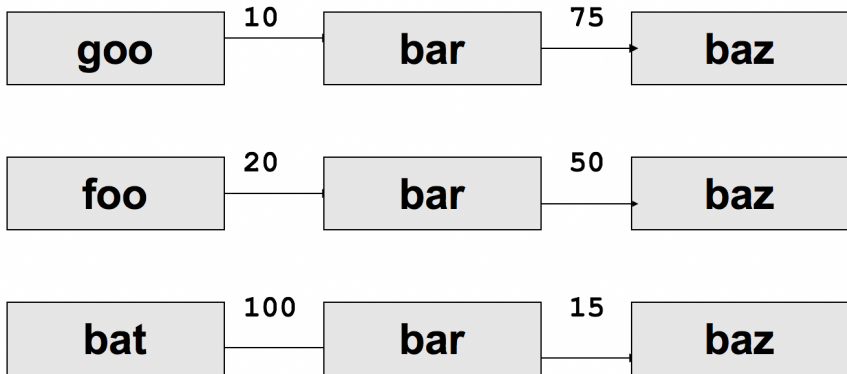


Example 2 of relationships between functions.  
Numbers on edges represent the number of invocations:

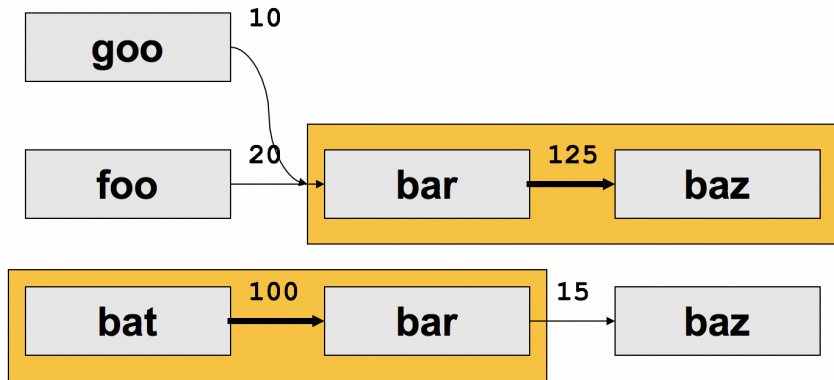


# The POGO View of the World

When considering what to do here, POGO takes the view like this:



# The POGO View of the World



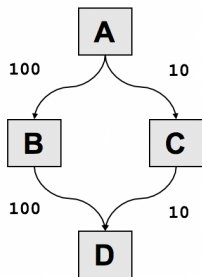
Call graph profiling data also good for packing blocks.

Put most common cases nearby.

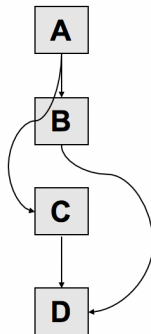
Put successors after their predecessors.

Packing related code = fewer page faults (cache misses).

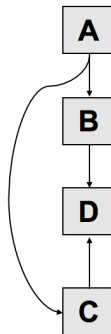
Calling a function in same page as caller = “page locality”.



**Default layout**



**Optimized layout**



According to the author, “dead” code goes in its own special block.

Probably not truly dead code (compile-time unreachable).

Instead: code that never gets invoked in training.

OK, how well does POGO work?

The application under test is a standard benchmark suite (Spec2K):

| <b>Spec2k:</b>            | <b>sjeng</b> | <b>gobmk</b> | <b>perl</b> | <b>povray</b> | <b>gcc</b> |
|---------------------------|--------------|--------------|-------------|---------------|------------|
| <b>App Size:</b>          | Small        | Medium       | Medium      | Medium        | Large      |
| <b>Inlined Edge Count</b> | 50%          | 53%          | 25%         | 79%           | 65%        |
| <b>Page Locality</b>      | 97%          | 75%          | 85%         | 98%           | 80%        |
| <b>Speed Gain</b>         | 8.5%         | 6.6%         | 14.9%       | 36.9%         | 7.9%       |

We can speculate about how well synthetic benchmarks results translate to real-world application performance...