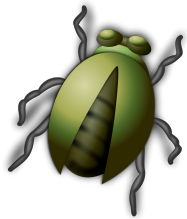
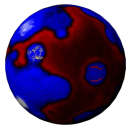


Detecting bugs and bottlenecks in C/C++ sources



Introducing *valgrind* and *gprof*

Michael Bauer
Complex Systems and Nonlinear Dynamics
TU Chemnitz



SFG-Meeting for Doctoral Researchers, St.-Afra-Klosterhof Meißen



CHEMNITZ UNIVERSITY OF
TECHNOLOGY

michael.bauer@physik.tu-chemnitz.de

2012-06-21

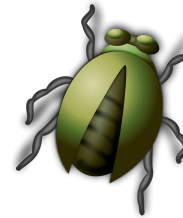
Motivation

“Certain input data crashes my program with a segmentation fault.”

“My program eats all the RAM. I need a machine with more.”

“My program is doing weird things.”






“Did you check it with *valgrind*?”



- every code has bugs
- there is room for optimization

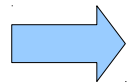
“Huh, what?”

General concepts

1. find/remove bugs 
2. find/reduce bottlenecks 
3. find/remove new bugs 
4. restart with 2. until:
 - no further speed improvement and 
 - no bugs are found 


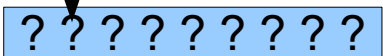

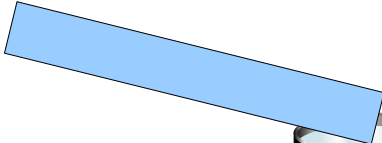
Comparing different runs

- use identical parameter values
- use **same input data** (files)
- generate **same random sequence**:
seed random number generator with identical
initial value (e.g.: `srand48(1)`)

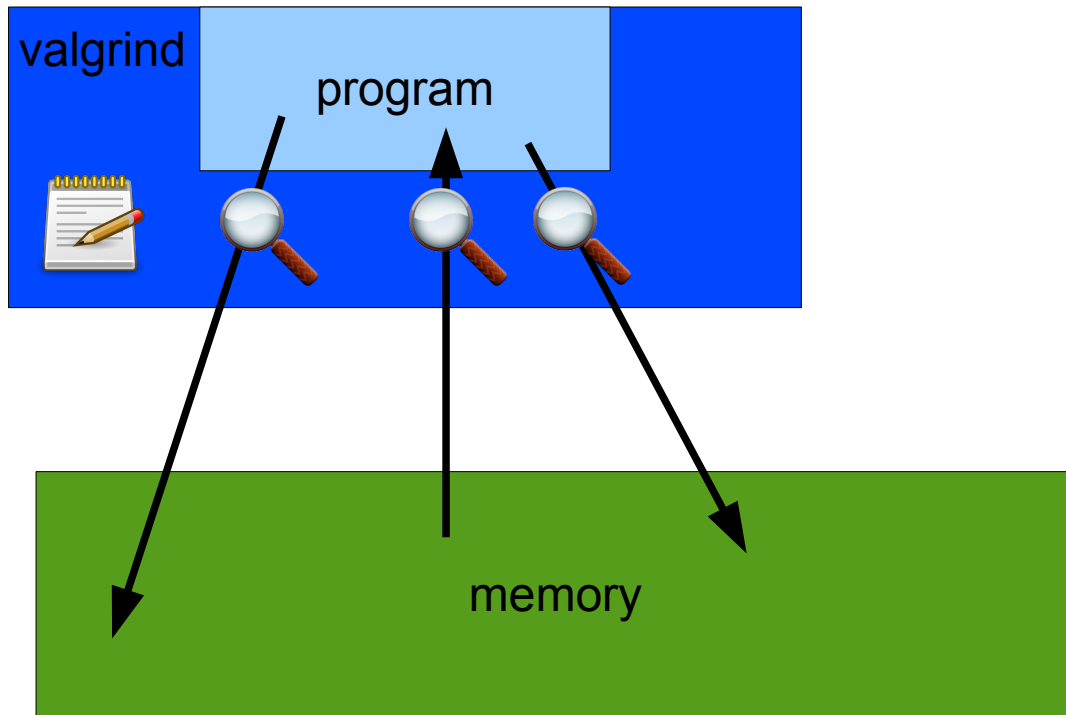


results and output data
of different runs should be identical

Memory problems

- buffer overruns:  access outside boundaries, memory corruption
- uninitialized memory:  unpredictable results
- incorrect memory management:  access of unallocated/freed memory, delete vs. delete[]
- memory leaks:  missing statements, out-of-memory errors, slowdown due to swapping




Mechanisms behind *valgrind*



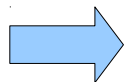
- advantages: no source code modification
- disadvantages: slowdown

- independent of source language
- even works without recompilation
- executes program in **virtual machine**
- logs every memory access
- monitors allocation and deallocation

Source code with memory bugs

```
1  #include <iostream>
2
3  int main(void) {
4      const int size = 100;
5      int i = 0;
6      double sum = 0.0;
7
8      double * vec = new double[size];
9
10     for (i = size; i > 0; i--) { 
11         vec[i] = static_cast<double>(i);
12     }
13
14     for (i = 0; i < size; i++) { 
15         sum += vec[i];
16     }
17
18     std::cout << "sum=" << sum << std::endl;
19     return 1; 
20 }
```

- compile:
g++ -g -W -Wall -pedantic -o main main.cpp



run:

valgrind --tool=memcheck --leak-check=yes --track-origins=yes ./main

Output of memcheck: buffer overrun

```
==5027== Memcheck, a memory error detector
==5027== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==5027== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==5027== Command: ./main
==5027==
==5027== Invalid write of size 8
==5027==    at 0x804871C: main (main.cpp:11)
==5027==   Address 0x4318348 is 0 bytes after a block of size 800 alloc'd
==5027==    at 0x4026E3F: operator new[](unsigned int) (vg_replace_malloc.c:299)
==5027==   by 0x80486FE: main (main.cpp:8)
...
```

```
8     double * vec = new double[size];
9
10    for (i = size; i > 0; i--) {
11        vec[i] = static_cast<double>(i);
12    }
```

Where is the problem in line 11?

Debugging the code would be helpful!

Attach debugger to memcheck

```
valgrind --tool=memcheck --leak-check=yes --track-origins=yes --db-attach=yes ./main
```

```
==10849== Memcheck, a memory error detector
==10849== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==10849== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==10849== Command: ./main
==10849==
==10849== Invalid write of size 8
==10849==    at 0x804871C: main (main.cpp:11)
==10849==   Address 0x4318348 is 0 bytes after a block of size 800 alloc'd
==10849==    at 0x4026E3F: operator new[](unsigned int) (vg_replace_malloc.c:299)
==10849==   by 0x80486FE: main (main.cpp:8)
==10849==
==10849== ---- Attach to debugger ? --- [Return/N/n/Y/y/C/c] ---- Y
==10849== starting debugger with cmd: /usr/bin/gdb -nw /proc/10850/fd/1014 10850
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-50.el6)
...
Reading symbols from /lib/ld-linux.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib/ld-linux.so.2
0x0804871c in main () at main.cpp:11
11         vec[i] = static_cast<double>(i);
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.47.el6_2.12.i686
libgcc-4.4.6-3.el6.i686 libstdc++-4.4.6-3.el6.i686
(gdb) print i
$1 = 100
```

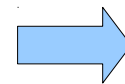
 index error: vec[] only has elements 0 to 99

memcheck: uninitialised memory

```
...
==5027== Conditional jump or move depends on uninitialised value(s)
==5027==    at 0x41CA8A5: __printf_fp (in /lib/libc-2.12.so)
==5027==    by 0x41C6539: vfprintf (in /lib/libc-2.12.so)
==5027==    by 0x41ED82F: vsnprintf (in /lib/libc-2.12.so)
==5027==    by 0x40C0989: ??? (in /usr/lib/libstdc++.so.6.0.13)
==5027==    by 0x40C330D: std::ostreambuf_iterator<char, std::char_traits<char> > ...
==5027==    by 0x40C3665: std::num_put<char, std::ostreambuf_iterator<char, ...
==5027==    by 0x40D5D4C: std::ostream& std::ostream::_M_insert<double>(double) ...
==5027==    by 0x40D5F14: std::ostream::operator<<(double) (in /usr/lib/libstdc+ ...
==5027==    by 0x8048784: main (main.cpp:18)
==5027== Uninitialised value was created by a heap allocation
==5027==    at 0x4026E3F: operator new[](unsigned int) (vg_replace_malloc.c:299)
==5027==    by 0x80486FE: main (main.cpp:8)
...
```

... and tons of similar errors more:
all referring to `main.cpp:18` and `main.cpp:8`

```
8     double * vec = new double[size];
|
14    for (i = 0; i < size; i++) {
15        sum += vec[i];
16    }
17
18    std::cout << "sum=" << sum << std::endl;
```



vec[0] was not
initialized

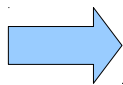
memcheck: memory leak

```
...
==5027==
==5027== HEAP SUMMARY:
==5027==   in use at exit: 800 bytes in 1 blocks
==5027== total heap usage: 1 allocs, 0 frees, 800 bytes allocated
==5027==
==5027== 800 bytes in 1 blocks are definitely lost in loss record 1 of 1
==5027==   at 0x4026E3F: operator new[](unsigned int) (vg_replace_malloc.c:299)
==5027==   by 0x80486FE: main (main.cpp:8)
==5027==
==5027== LEAK SUMMARY:
==5027==   definitely lost: 800 bytes in 1 blocks
==5027==   indirectly lost: 0 bytes in 0 blocks
==5027==   possibly lost: 0 bytes in 0 blocks
==5027==   still reachable: 0 bytes in 0 blocks
==5027==   suppressed: 0 bytes in 0 blocks
==5027==
...
```

not cleaned up properly

consumes all RAM
if allocation is
inside a loop

```
8     double * vec = new double[size];
```



memory was allocated with **new**
but never freed again

insert after line 16:

```
delete[] vec;
```

Suppressed errors

```
...  
==5027== For counts of detected and suppressed errors, rerun with: -v  
==5027== ERROR SUMMARY: 146 errors from 80 contexts (suppressed: 21 from 10)
```

not all errors were displayed



- every library may contain memory bugs
- some errors are “by design”
- *valgrind* suppresses known external errors
- own suppression rules can be defined

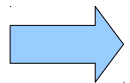
Program without bugs and leaks

```
==19177== Memcheck, a memory error detector
==19177== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==19177== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==19177== Command: ./main
==19177==
sum=4950
==19177==
==19177== HEAP SUMMARY:
==19177==   in use at exit: 0 bytes in 0 blocks
==19177== total heap usage: 1 allocs, 1 frees, 800 bytes allocated
==19177==
==19177== All heap blocks were freed -- no leaks are possible
==19177==
==19177== For counts of detected and suppressed errors, rerun with: -v
==19177== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 20 from 9)
```

output of program

everything cleaned up

no errors



could be some work
until the output of *valgrind* for
your program looks like above



Bottlenecks

- cache access for large data structures
- slow/complex functions
- expensive algorithms inside loops



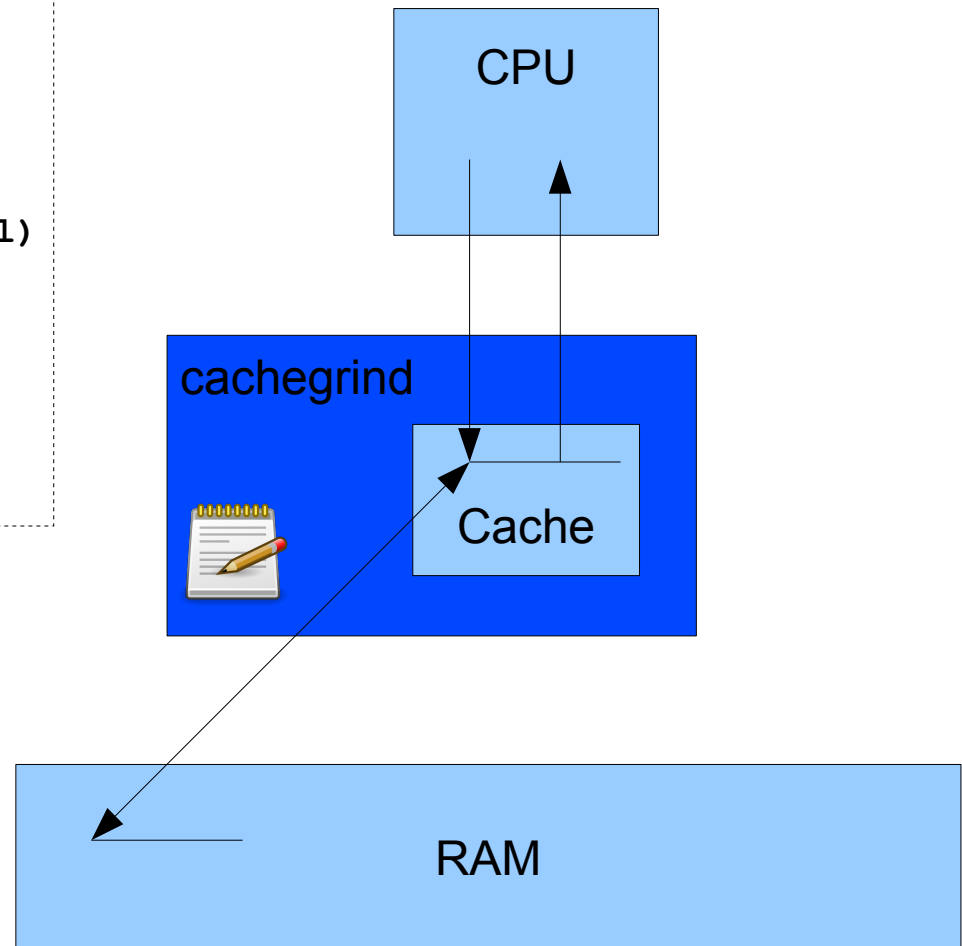
Optimizing cache access

```
const unsigned long _numRows = 100000;  
const unsigned long _numCols = 100;  
  
double ** _data;  
  
...  
  
unsigned long row, col;  
for (col = 0; col < _numCols; ++col)  
{  
for (row = 0; row < _numRows; ++row)  
{  
    _data[row][col] = drand48();  
}  
}
```



runtime:
0m10.370s

valgrind --tool=cachegrind ./main



Profiling the bad code

```
==22619== Cachegrind, a cache and branch-prediction profiler
==22619== Copyright (C) 2002-2010, and GNU GPL'd, by Nicholas Nethercote et al.
==22619== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==22619== Command: ./main-unoptimized-for
==22619==
==22619==
==22619== I   refs:      14,227,732,318
==22619== I1  misses:      1,282
==22619== LLi misses:      1,234
==22619== I1  miss rate:    0.00%
==22619== LLi miss rate:  0.00%
==22619==
==22619== D   refs:      9,090,532,395 (5,690,976,115 rd + 3,399,556,280 wr)
==22619== D1  misses:      108,562,631 ( 7,489,387 rd + 101,073,244 wr)
==22619== LLd misses:      108,386,395 ( 7,321,226 rd + 101,065,169 wr)
==22619== D1  miss rate:    1.1% ( 0.1% + 2.9% )
==22619== LLd miss rate:  1.1% ( 0.1% + 2.9% )
==22619==
==22619== LL refs:      108,563,913 ( 7,490,669 rd + 101,073,244 wr)
==22619== LL misses:      108,387,629 ( 7,322,460 rd + 101,065,169 wr)
==22619== LL miss rate:    0.4% ( 0.0% + 2.9% )
```

only few instruction misses

some data misses



runtime in valgrind:
5m54.599s
slowdown: 35x



analyze in detail:
kcachegrind cachegrind.out.22619

Analyze output with KCachegrind

The screenshot displays the KCachegrind application interface. The top menu bar includes File, View, Go, Settings, and Help. Below the menu is a toolbar with icons for Open, Back, Forward, Up, Relative, Cycle Detection, and Relative to Parent. A dropdown menu shows 'L1 Data Read Miss'.

The main window is divided into several panels:

- Flat Profile:** Shows a list of source files with their respective costs. 'main.cpp' is highlighted with a cost of 6 317 917.
- Function Call Graph:** Shows the call stack for the selected function. 'Array::fill()' is selected, showing it was called from 'main.cpp' at address 6 253 348.
- Event Table:** A table showing various events and their costs. The 'L1 Data Read Miss' event is highlighted, with a cost of 6 253 348. The formula for this event is $L1m = I1mr + D1mr + D1mw$.
- Event Details:** A table showing the breakdown of the selected event. The 'L1 Data Read Miss' event is highlighted, with a cost of 6 253 348. The formula for this event is $L1m = I1mr + D1mr + D1mw$.

At the bottom of the window, the status bar displays: 'cachegrind.out.22619 [1] - Total L1 Data Read Miss Cost: 7 489 387'.

Optimized cache access

```
==22281== Cachegrind, a cache and branch-prediction profiler
==22281== Copyright (C) 2002-2010, and GNU GPL'd, by Nicholas Nethercote et al.
==22281== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==22281== Command: ./main-optimized-for
==22281==
==22281==
==22281== I   refs:          14,242,730,818
==22281== I1  misses:          1,282
==22281== LLi misses:          1,234
==22281== I1  miss rate:         0.00%
==22281== LLi miss rate:         0.00%
==22281==
==22281== D   refs:          9,098,531,595 (5,697,975,415 rd + 3,400,556,180 wr)
==22281== D1  misses:          15,002,398 ( 1,301,904 rd + 13,700,494 wr)
==22281== LLd misses:          14,821,711 ( 1,131,926 rd + 13,689,785 wr)
==22281== D1  miss rate:         0.1% ( 0.0% + 0.4% )
==22281== LLd miss rate:         0.1% ( 0.0% + 0.4% )
==22281==
==22281== LL refs:          15,003,680 ( 1,303,186 rd + 13,700,494 wr)
==22281== LL misses:          14,822,945 ( 1,133,160 rd + 13,689,785 wr)
==22281== LL miss rate:         0.0% ( 0.0% + 0.4% )
```

data misses were reduced



runtime in valgrind:
5m29.112s
slowdown: 60x



runtime after optimization:
0m4.796s
speedup: 2x

Success measured by cachegrind

Self	Source File	Event Type	Incl.	Self	Short	Formula
6 317 917	main.cpp	Instruction Fetch	1 800 001 516	1 800 001 516	Ir	
1 171 470	(unknown)	L1 Instr. Fetch Miss	1	1	l1mr	
0	iostream	ILmr	1	1	ILmr	
		Data Read Access	900 000 706	900 000 706	Dr	
		L1 Data Read Miss	6 253 348	6 253 348	D1mr	
		DLmr	6 250 405	6 250 405	DLmr	

Search: Source File

Self	Source File
1 171 431	(unknown)
130 473	main.cpp
0	iostream

Self	Function	Location
65 924	Array::fill()	main.cpp
64 534	Array::~~Array()	main.cpp
15	Array::create()	main.cpp
0	main	main.cpp
0	global constructors keyed...	main.cpp
0	__static_initialization_and...	main.cpp, iostream
0	Array::Array(unsigned lon...	main.cpp

Types Callers All Callers Callee Map Source Code

Event Type	Incl.	Self	Short	Formula
Instruction Fetch	1 815 000 016	1 815 000 016	Ir	
L1 Instr. Fetch Miss	1	1	l1mr	
ILmr	1	1	ILmr	
Data Read Access	907 000 006	907 000 006	Dr	
L1 Data Read Miss	65 924	65 924	D1mr	
DLmr	62 548	62 548	DLmr	
Data Write Access	201 000 003	201 000 003	Dw	
L1 Data Write Miss	12 627 954	12 627 954	D1mw	
DLmw	12 625 287	12 625 287	DLmw	
L1 Miss Sum	12 693 879	12 693 879	L1m = l1mr + D1mr + D1mw	

D1mr L1m Count Callee

Parts Callees Call Graph All Callees Caller Map Machine Code

cachegrind.out.22281 [1] - Total L1 Data Read Miss Cost: 1 301 904

Analyzing function calls

- determine which function calls what other function
- detecting dependencies, loops and recursive calls

```
class CoinTossing
{
public:
    CoinTossing();

    void run();

    void tossIt();
    void setHeads();
    void setTails();

    void calculation(long);
};
```



runtime:
0m10.415s

```
g++ -g -W -Wall -pedantic -o main main.cpp
```

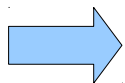
Callgrind

```
valgrind --tool=callgrind ./main
```

```
==14756== Callgrind, a call-graph generating cache profiler
==14756== Copyright (C) 2002-2010, and GNU GPL'd, by Josef Weidendorfer et al.
==14756== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==14756== Command: ./main
==14756==
==14756== For interactive control, run 'callgrind_control -h'.
Please enter the probability for heads: 0.1
sum: -796
==14756==
==14756== Events      : Ir
==14756== Collected : 22897841697
==14756==
==14756== I   refs:      22,897,841,697
```



runtime in valgrind:
6m14.553s
slowdown: 36x



analyze in detail:
callgrind-annotate --auto=yes

```
kcachegrind callgrind.out.14756
```

Analyze output of callgrind

The screenshot displays the Callgrind tool interface. On the left, the 'Flat Profile' shows the following data:

Source File	Count
main.cpp	22 896 109 694
(unknown)	1 732 003
iostream	0

Below the flat profile is a table of inclusions:

Incl.	Self	Called	Function	Location
22 896 353 160	40	1	main	main: m...
22 896 228 877	10 013	1	CoinTossing::run()	main: m...
22 896 218 864	19 102	1 000	CoinTossing::tossIt()	main: m...
22 896 066 504	6 000 019 000	1 000	CoinTossing::calculation(u...	main: m...
16 896 047 504	16 896 047 504	2 816	CoinTossing::calculation(u...	main: m...
16 164 058 370	12 572	898	CoinTossing::setTails()	main: m...
6 732 022 134	1 428	102	CoinTossing::setHeads()	main: m...
102 644	8	1	global constructors keyed ...	main: m...
102 636	16	1	_static_initialization_and...	main: m...
11	11	1	CoinTossing::CoinTossing()	main: m...

The main window shows the 'CoinTossing::tossIt()' call graph. The graph starts with 'main' (22 896) calling 'CoinTossing::run()' (22 896 218 864) 1 x. 'CoinTossing::run()' calls 'CoinTossing::tossIt()' (22 896 218 864) 1 000 x. 'CoinTossing::tossIt()' branches into 'CoinTossing::setTails()' (16 164 058 370) 898 x and 'CoinTossing::setHeads()' (6 732 022 134) 102 x. Both 'setTails()' and 'setHeads()' call 'CoinTossing::calculation(unsigned long)' (22 896 066 504) 898 x and 102 x respectively. Finally, 'CoinTossing::calculation(unsigned long)' calls 'CoinTossing::calculation(unsigned long)' (16 896 047 504) 1 000 x, which then calls 'CoinTossing::calculation(unsigned long)' (16 896 047 504) 1 816 x.

At the bottom, the status bar reads: callgrind.out.14756 [1] - Total Instruction Fetch Cost: 22 897 841 697

Profiling with GCC

- profiling tools embedded in GNU Compiler Collection (GCC)
- compilation adds profiling instrumentation via option (-pg)
- sampling-based approach

➡ runtime of relevant parts must be much larger than sampling interval to avoid measurement errors

Compile and profile

- **compile**: `g++ -pg -g -W -Wall -pedantic -o main main.cpp`
- **run**: `./main`
- produces binary output **gmon.out**, which contains profiling data



runtime:
0m10.633s

- no slowdown in our example
- slowdown in real-world projects: around 5x
➡ much faster than *callgrind* (30x)

gprof: flat profile

`gprof --flat-profile main`

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
100.00	10.56	10.56	1000	0.01	0.01	CoinTossing::calculation(long)
0.00	10.56	0.00	1000	0.00	0.01	CoinTossing::tossIt()
0.00	10.56	0.00	898	0.00	0.01	CoinTossing::setTails()
0.00	10.56	0.00	102	0.00	0.01	CoinTossing::setHeads()
0.00	10.56	0.00	1	0.00	0.00	global constructors keyed to main
0.00	10.56	0.00	1	0.00	0.00	__static_initialization_a...
0.00	10.56	0.00	1	0.00	10.56	CoinTossing::run()
0.00	10.56	0.00	1	0.00	0.00	CoinTossing::CoinTossing()

- gather data from several runs:
move `gmon.out` to `gmon.out.{id}`

```
gprof --sum main gmon.out.*  
gprof --flat-profile main gmon.sum
```

gprof: call graph

gprof --graph main

Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 0.09% of 10.56 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.00	10.56		main [1]
		0.00	10.56	1/1	CoinTossing::run() [4]
		0.00	0.00	1/1	CoinTossing::CoinTossing() [12]

				2816	CoinTossing::calculation(long) [2]
		1.08	0.00	102/1000	CoinTossing::setHeads() [6]
		9.48	0.00	898/1000	CoinTossing::setTails() [5]
[2]	100.0	10.56	0.00	1000+2816	CoinTossing::calculation(long) [2]
				2816	CoinTossing::calculation(long) [2]

		0.00	10.56	1000/1000	CoinTossing::run() [4]
[3]	100.0	0.00	10.56	1000	CoinTossing::tossIt() [3]
		0.00	9.48	898/898	CoinTossing::setTails() [5]
		0.00	1.08	102/102	CoinTossing::setHeads() [6]

		0.00	10.56	1/1	main [1]
[4]	100.0	0.00	10.56	1	CoinTossing::run() [4]
		0.00	10.56	1000/1000	CoinTossing::tossIt() [3]

Summary

- check for bugs
valgrind --tool=memcheck
- look for cache misses
valgrind --tool=cachegrind
- look for functions with bottlenecks
valgrind --tool=callgrind
- analyze output with KCachegrind
- significant slowdown of execution
- *gprof* provides information similar to *callgrind*
but much faster

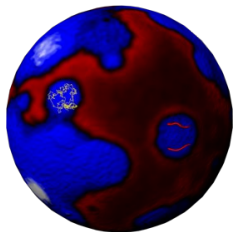
Alternative tools

- memory debuggers:
Purify, Insure++
- profiling tools:
Quantify
- thread checking:
valgrind --tool=helgrind

References

- valgrind.org
sourceware.org/binutils/docs/gprof/
- T. Grötzer, U. Holtmann, H. Keding and M. Wloka: *The Developer's Guide to Debugging*, Springer (2008).
- W. von Hagen: *The Definitive Guide to GCC*, Apress (2006).

Acknowledgements



Sächsische Forschergruppe FOR877
“From local constraints to macroscopic transport”