

10/24/12

# Valgrind

Eric Heim



**GERMAN  
CANCER RESEARCH CENTER  
IN THE HELMHOLTZ ASSOCIATION**

## ***What is Valgrind?***

- A collection of tools for memory missmanagment detection and profiling on linux and mac

## ***What can Valgrind do for you?***

- Detect memory leaks
- Detect reads/writes inappropriate areas of memory
- Profile function execution
- Profile cache hit/miss
- And many more

## Memcheck:

`valgrind --tool=memcheck --leak-check=yes program`

- Program runs several times slower
- Code need to be compiled in debug
- Output is shown on the terminal

```
3 void foo()
4 {
5     float* f = new float[10];
6 }
7 int main()
8 {
9     foo();
10    return 0;
11 }
```

```
==1632== HEAP SUMMARY:
==1632==    in use at exit: 40 bytes in 1 blocks
==1632== total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==1632==
==1632== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1632==    at 0x4C2B307: operator new[](unsigned long) (in /usr/lib/valgrind/valgrind-core-3.10.0/libvalgrind-core-3.10.0.so)
==1632==    by 0x4006FD: foo() (main.cpp:5)
==1632==    by 0x40070C: main (main.cpp:9)
==1632==
==1632== LEAK SUMMARY:
==1632==    definitely lost: 40 bytes in 1 blocks
==1632==    indirectly lost: 0 bytes in 0 blocks
==1632==    possibly lost: 0 bytes in 0 blocks
==1632==    still reachable: 0 bytes in 0 blocks
==1632==    suppressed: 0 bytes in 0 blocks
==1632==
```

## **Definitely lost:**

- Memory is lost. Error need to be fixed !

## **Indirectly lost:**

- Memory is indirectly lost through a pointer. When the root of a tree is definitely lost, all childs are indirectly lost.

## **Possibly lost:**

- Pointer to the buffer start is lost, but another pointer references part of the buffer

## **Still reachable:**

- Memory is still reachable that could have been deleted

## *What memcheck can also detect:*

1. Usage of uninitialized variables
2. Mismatch of new/[ ]/malloc and delete/[ ]/free
3. Access violations in heap memory

## *What memcheck can't find:*

4. Out of bound checks in arrays allocated on the stack

```
6 // 2. mismatch new / free
7 float* f = new float[10];
8 free(f);
9
10 // 3. access violations
11 char* c = new char[10];
12 c[10] = 'a';
13
14 // 4. out of bound check
15 int i[10];
16 i[10]=2; // won't be detected
17
```

## Callgrind:

- The callgrind tool records the callhistory and instructions issued per function
- Results are stored in a file
- Caches can be simulated
- Available for different Architectures

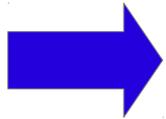
## Usage:

```
valgrind --tool=callgrind --dump-instr=yes --simulate-cache=yes program
```

Hint: (compile program in release with debuginfo)

## Problems:

- Program runs 4 – 20 times slower
- Everything is simulated and captured, also parts of the program that are not interesting



*Only the code of interest need to be profiled*

Macros for profiling parts of the code are provided by  
***callgrind.h***

Disable simulation and capture via commandline arguments:

- `--instr-atstart=<yes|no>` [default: yes] (disable simulation)
- `--collect-atstart=<yes|no>` [default: yes] (disable capture)

Macros:

- **Start simulation:**  
`CALLGRIND_START_INSTRUMENTATION`
- **Capture data:**  
`CALLGRIND_TOGGLE_COLLECT`
- **Stop simulation:**  
`CALLGRIND_STOP_INSTRUMENTATION`

And finally we got our output:

```
20 ob=(7) /usr/lib/libstdc++.so.6.0.17
21 fl=(7) ???
22 fn=(298) std::future_category()
23 0xb2580 0 1 1 0 1 1 0 1 1
24 +7 0 1
25 jcnd=1/1 +17 0
26 * 0
27 +17 0 1
28 +7 0 1
29 +4 0 1 0 1
30 cfn=(304) __cxa_guard_acquire
31 calls=1 0x5f2f0 0
32 * 0 24 9 5 4 1 0 4
33 cob=(1) /usr/lib/ld-2.16.so
34 cfi=(1) ???
35 cfn=(166) _dl_runtime_resolve
36 calls=1 0x140e0 0
37 * 0 758 216 87 0 6 0 0 1
38 * 0 5 3 2 1 0 0 1
39 +5 0 1
40 +2 0 1
41 +2 0 1
42 +7 0 1 0 1
43 cfn=(310) std::error_category::error_category()
44 calls=1 0xb2a90 0
45 * 0 4 2 1 1 1 0 1
46 cob=(1)
47 cfi=(1)
48 cfn=(166)
49 calls=1 0x140e0 0
50 * 0 810 227 87 0 7
```

~17.000 lines for the simple program from the previous slides !!!

# Valgrind – kcachegrind

Use kcachegrind to visualize and analyze the data.

The screenshot displays the kcachegrind application interface. At the top, the window title is `/home/heim/Programming/dkz/bin/callgrind.out.2349 [home/heim/Programming/dkz/bin/MBI/MBI-build/bin/MitkAnisotropicRegistrationTestDriver mitkAICPR...`. The menu bar includes `File View Go Settings Help`. Below the menu, there are navigation buttons: `Open`, `Back`, `Forward`, `Up`, `% Relative`, `Cycle Detection`, and `Relative to Parent`. A dropdown menu shows `Instruction Fetch`.

The main interface is divided into two main sections. On the left is the **Flat\_Profile** window, which contains a table with columns `Incl.`, `Self`, `Called`, and `Fu`. The table lists various functions and their associated costs. On the right is the **Call Graph** window, which visualizes the execution flow between different functions. The graph shows a central node `mitk::AnisotropicCorrespondingPointRegistration::DoRegister()` (38.02%) which calls several other functions, including `mitk::AnisotropicCorrespondingPointRegistration::WeightedPointRegisterInvNe...` (38.02%), `mitk::RegistrationEvaluation::ComputeWeightedFRE(itk::SmartPointer<mitk::...` (11.30%), `mitk::AnisotropicCorrespondingPointRegistration::CalculateConfigChange(itk::...` (4.79%), and `mitk::AnisotropicCorrespondingPointRegistration::C_marker(itk::SmartPointer...` (7.32%).

Incl.	Self	Called	Fu
100.00	0.00	(0)	1
100.00	0.00	1	6
59.95	0.00	6	6
59.95	0.02	6	6
57.04	0.26	6 096	6
55.42	0.66	280 072	6
38.02	0.00	6	6
34.05	1.25	291 270	6
25.55	0.34	730 231	6
24.02	0.05	291 270	6
23.98	10.85	291 270	6
15.19	0.63	2 190 693	6
12.85	2.85	2 190 693	6
11.30	0.01	18	6
11.09	0.12	730 231	6
10.86	0.16	730 231	6
10.77	0.23	1 460 462	6
8.13	0.06	326 825	6
7.89	4.41	25 851 051	6
7.71	0.04	326 831	6
7.68	1.38	326 831	6
7.32	0.06	11	6
6.06	2.52	24 402 191	6
5.95	0.00	6	6
5.65	0.16	730 231	6
5.46	0.08	730 231	6
5.36	0.23	730 231	6
5.13	0.07	730 231	6
4.79	0.02	11	6
4.46	0.02	24	6
4.42	0.52	6 336 446	6
4.38	0.01	6	6
4.32	0.55	7 044 571	6
4.03	1.15	730 231	6
3.94	0.13	6 372 630	6
3.88	3.88	3 832 298	6
3.77	1.60	7 044 583	6
3.65	0.00	18	6
3.64	0.70	7 281 349	6

Any Questions?