Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Valgrind

Jean-Roland Schuler
jeanrola.schuler.home.hefr.ch
Jean-roland.schuler@hefr.ch

References

[1]: http://valgrind.org/
[2]: http://en.wikipedia.org/wiki/Valgrind

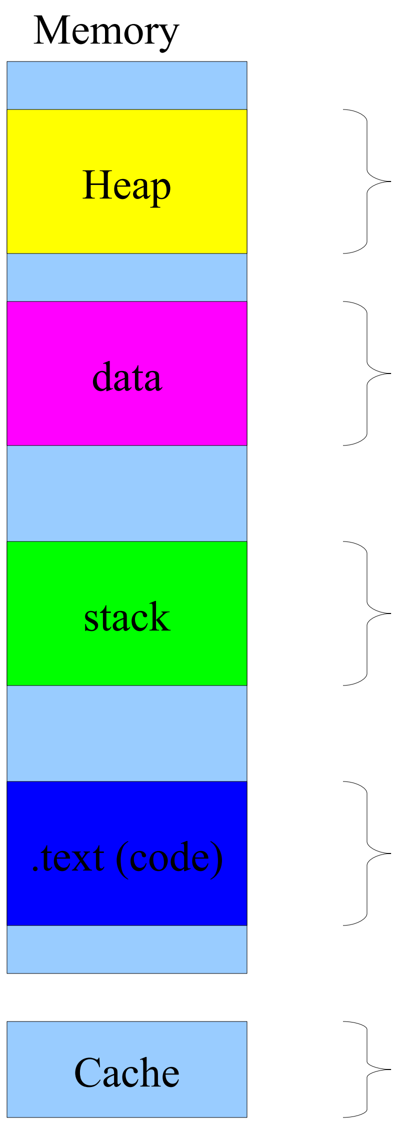
Valgrind [1], [2]

- Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.
- Valgrind is in essence a virtual machine using just-in-time (JIT) compilation techniques, including dynamic recompilation. Nothing from the original program ever gets run directly on the host processor. Instead, Valgrind first translates the program into a temporary, simpler form called Intermediate Representation (IR), which is a processor-neutral. After the conversion, a tool (next page) is free to do whatever transformations it would like on the IR, before Valgrind translates the IR back into machine code and lets the host processor run it.
- It runs on the following platforms: X86/Linux, AMD64/Linux, ARM/Linux, PPC32/Linux, PPC64/Linux, S390X/Linux, MIPS/Linux, ARM/Android (2.3.x and later), X86/Android (4.0 and later), X86/Darwin and AMD64/Darwin, Mac OS X 10.12.

Valgrind: tools

1. **Memcheck** is a memory error detector. It helps you make your programs, particularly those written in C and C++, more correct.
2. **Cachegrind** is a cache and branch-prediction profiler. It helps you make your programs run faster.
3. **Callgrind** is a call-graph generating cache profiler. It has some overlap with Cachegrind, but also gathers some information that Cachegrind does not.
4. **Helgrind** is a thread error detector. It helps you make your multi-threaded programs more correct.
5. **DRD** is also a thread error detector. It is similar to Helgrind but uses different analysis techniques and so may find different problems.
6. **Massif** is a heap and stack profiler. It helps you make your programs use less memory.
7. **DHAT** is a tool for examining how programs use their heap allocations. It tracks the allocated blocks, and inspects every memory access to find which block, if any, it is to.

Valgrind summary



Memcheck: heap overflow
: memory leak
Massif : heap profiler

Sgcheck : global array overflow

Memcheck : uninitialized memory
Sgcheck : memory overflow
Massif : stack profiling

Callgrind : call graph

Cachegrind

Memcheck

Memcheck is a memory error detector. It can detect the following problems that are common in C and C++ programs.

- Accessing memory you shouldn't, e.g. overrunning and under running heap blocks, and accessing memory after it has been freed.
- Using undefined values, i.e. values that have not been initialized, or that have been derived from other undefined values.
- Incorrect freeing of heap memory, such as double-freeing heap blocks, or mismatched use of *malloc-new* versus free *free/delete*
- Overlapping *src* and *dst* pointers in *memcpy* and related functions.
- Memory leaks.

Theses problems can be difficult to find, often remaining undetected for long periods, then causing occasional, difficult-to-diagnose crashes.

Memcheck: heap overflow

```
int mallocOverflow1(void)
{
int i;
int *p = malloc(sizeof(int) * 10);
    for (i = 0; i < 11; i++){
        p[i] = i;
    }
    free(p);
    return 0;
}
```

Valgrind: Heap overflow

```
#gcc -Wall -g -o file file.c

#valgrind ./file          or
#valgrind --tool=memcheck ./file

==6170== Command: ./code
==6170== Invalid write of size 4
==6170==    at 0x804858A: mallocOverflow1 (code.c:72)
==6170==    by 0x80484FA: main (code.c:26)
==6170==  Address 0x4027050 is 0 bytes after a block of size 40
alloc'd
==6170==    at 0x4007E08: malloc (vg_replace_malloc.c:270)
==6170==    by 0x804856B: mallocOverflow1 (code.c:70)
==6170==    by 0x80484FA: main (file.c:26)
==6170==
==6170== All heap blocks were freed -- no leaks are possible
==6170==
==6170== For counts of detected and suppressed errors, rerun with: -v
==6170== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from
0)
```

Memcheck: uninitialized memory

```
static int uninitializedMemory(void)
{
int i;
int a[10];
    for (i = 0; i < 9; i++)
        a[i] = i;


    for (i = 0; i < 10; i++)
    {
        printf("%d ", a[i]);
    }
    printf("\n");
    return 0;
}
```

Valgrind: uninitialized memory

```
#gcc -Wall -g -o file file.c

#valgrind ./file            or
#valgrind -v ./file         or
#valgrind --track-origins=yes -v ./file
#valgrind -s -v ./file        // = --track-origins=yes

==3668== Command: ./file
==3668==
==3668== Use of uninitialised value of size 4
==3668==    at 0x4DD4471B: _itoa_word (in /usr/lib/libc-2.15.so)
==3668==    by 0x4DD491D8: vfprintf (in /usr/lib/libc-2.15.so)
==3668==    by 0x4DD4EA6E: printf (in /usr/lib/libc-2.15.so)
==3668==    by 0x80484FA: main (file.c:32)
...
==3668== ERROR SUMMARY: 25 errors from 7 contexts (suppressed: 0 from 0)
```

Memcheck: memory leak

```
int memoryLeak(void)
{
int i;
int *a;

    for (i=0; i < 10; i++)
    {
       a = malloc(sizeof(int) * 100);
    }
    free(a);
    return 0;
}
```

Memcheck: memory leak

```
#gcc -Wall -g -o file file.c
#valgrind ./file          or
#valgrind -v ./file       or
#valgrind --leak-check=full ./file

==3698== HEAP SUMMARY:
==3698==     in use at exit: 3,600 bytes in 9 blocks
==3698==   total heap usage: 10 allocs, 1 frees, 4,000 bytes
allocated
==3698==
==3698== LEAK SUMMARY:
==3698==    definitely lost: 3,600 bytes in 9 blocks
==3698==    indirectly lost: 0 bytes in 0 blocks
==3698==      possibly lost: 0 bytes in 0 blocks
==3698==    still reachable: 0 bytes in 0 blocks
==3698==         suppressed: 0 bytes in 0 blocks
==3698== Rerun with --leak-check=full to see details of leaked memory
=
```

Massif

- Massif is a heap profiler. It measures how much heap memory your program uses. This includes both the useful space, and the extra bytes allocated for book-keeping and alignment purposes. It can also measure the size of your program’s stack(s), although it does not do so by default.

Massif: heap profiling

```
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;
}
```

Massif: heap profiling

```
#gcc -Wall -g -o file file.c
#valgrind --tool=massif --time-unit=B ./file    (file massif.out.pid
                                                 is generated)
#ms_print massif.out.PID
```

-time-unit: <i|ms|B> The time unit used for the profiling. i: Instructions executed (i), which is good for most cases; ms: Milliseconds, which is sometimes useful, B: Bytes allocated/deallocated, which is useful for very short-run programs

- Normal snapshots are represented in the graph by :
- Detailed snapshots are represented in the graph by @. There is a detailed snapshot every `-detailed_freq` option (default: 10)
- The peak snapshot is represented in the graph by: #

```
    KB
19.63^                                               ###
     |                                               #
     |                                               #  ::
     |                                               #  : :::
     |                                      :::::::::#  : :  ::
     |                                      :        #  : :  : ::
     |                                      :        #  : :  : : :::
     |                                      :        #  : :  : : :  ::
     |                            :::::::::::        #  : :  : : :  : :::
     |                            :         :        #  : :  : : :  : :  ::
     |                        :::::         :        #  : :  : : :  : :  : ::
     |                     @@@:   :         :        #  : :  : : :  : :  : : @
     |                   ::@  :   :         :        #  : :  : : :  : :  : : @
     |                :::: @  :   :         :        #  : :  : : :  : :  : : @
     |              :::  : @  :   :         :        #  : :  : : :  : :  : : @
     |            ::: :  : @  :   :         :        #  : :  : : :  : :  : : @
     |         :::: : :  : @  :   :         :        #  : :  : : :  : :  : : @
     |       :::  : : :  : @  :   :         :        #  : :  : : :  : :  : : @
     |    :::: :  : : :  : @  :   :         :        #  : :  : : :  : :  : : @
     |  :::  : :  : : :  : @  :   :         :        #  : :  : : :  : :  : : @
   0 +----------------------------------------------------------------------->KB
     0                                                                   29.48
```

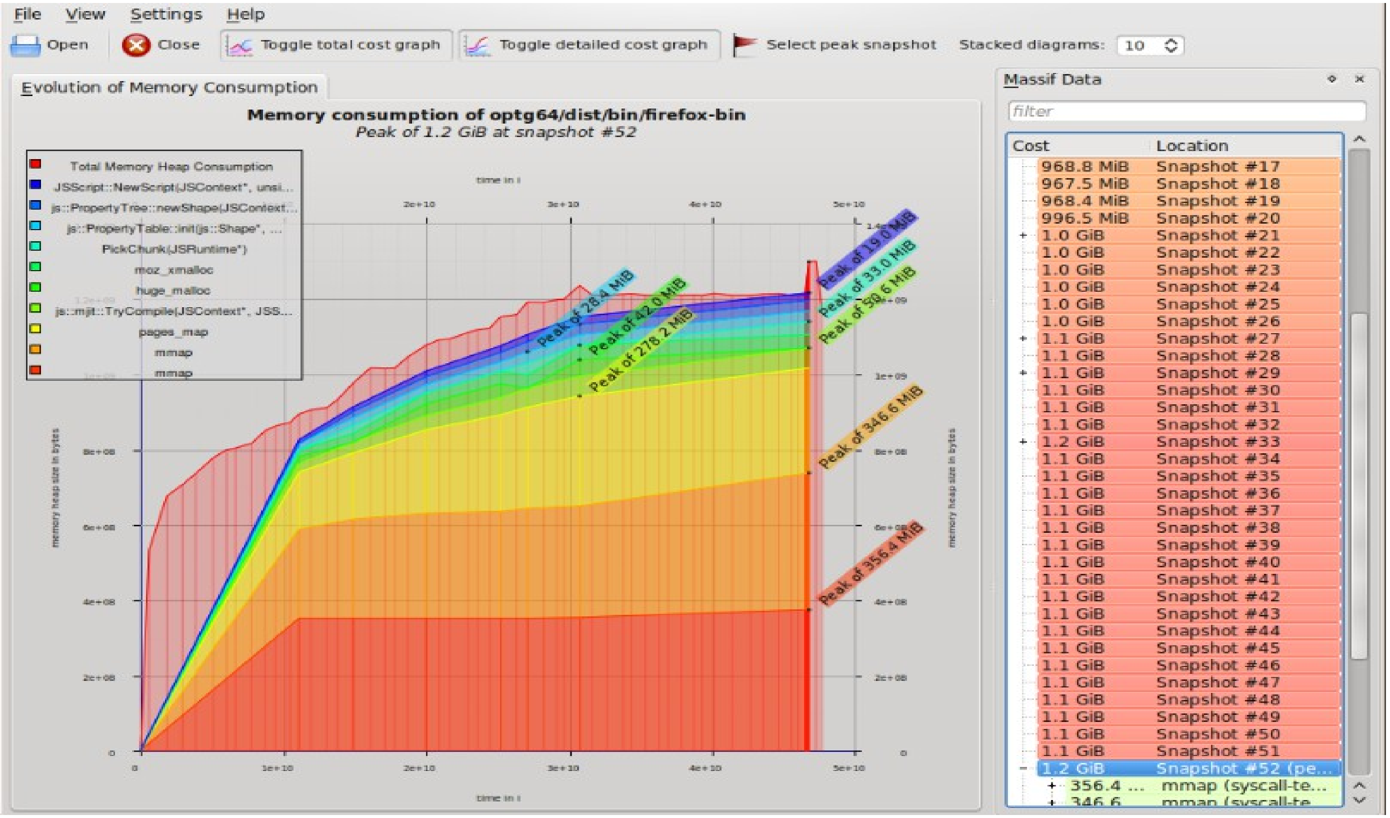
Massif: heap profiling

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0                0             0            0
  1          1,008            1,008            1,000             8            0
  2          2,016            2,016            2,000            16            0
  3          3,024            3,024            3,000            24            0
  4          4,032            4,032            4,000            32            0
```

n:	Snapshot number
time(B):	Time unit in byte
Total (B):	Total memory allocated
Useful-heap (B):	Memory used by the program
Extra-heap (B):	Memory not used by the program (but used by the OS)
Stack (B):	Memory stack profiling (by default off)

Massif: Analyze firefox

- http://blog.mozilla.org/nnethercote/2010/12/09/memory-profiling-firefox-with-massif/
- `valgrind --smc-check=all --trace-children=yes --tool=massif --pages-as-heap=yes --detailed-freq=1000000 optg64/dist/bin/firefox -P cad20 -no-remote`
- **Massif-visualizer** (kde-apps.org/content/show.php?content=122409)



Example: Memory lost

```
#gcc -Wall -g -o file file.c                    (program p 18)
#valgrind  ./file

==4164== HEAP SUMMARY:
==4164==     in use at exit: 10,000 bytes in 3 blocks
==4164==   total heap usage: 13 allocs, 10 frees, 20,000 bytes allocated
==4164==
==4164== LEAK SUMMARY:
==4164==    definitely lost: 10,000 bytes in 3 blocks
==4164==    indirectly lost: 0 bytes in 0 blocks
==4164==      possibly lost: 0 bytes in 0 blocks
==4164==    still reachable: 0 bytes in 0 blocks
==4164==         suppressed: 0 bytes in 0 blocks
==4164== Rerun with --leak-check=full to see details of leaked memory
```

Remark: With this command we know that memories are lost

Example: Memory lost

`valgrind --leak-check=full ./file`

```
==4198== HEAP SUMMARY:
==4198==     in use at exit: 10,000 bytes in 3 blocks
==4198==   total heap usage: 13 allocs, 10 frees, 20,000 bytes allocated
==4198==
==4198== 2,000 bytes in 1 blocks are definitely lost in loss record 1 of 3
==4198==    at 0x4007E08: malloc (vg_replace_malloc.c:270)
==4198==    by 0x8048455: f (code1.c:9)
==4198==    by 0x8048496: main (code1.c:21)
==4198==
==4198== 4,000 bytes in 1 blocks are definitely lost in loss record 2 of 3
==4198==    at 0x4007E08: malloc (vg_replace_malloc.c:270)
==4198==    by 0x8048441: g (code1.c:4)
==4198==    by 0x804845A: f (code1.c:10)
==4198==    by 0x8048496: main (code1.c:21)
==4198==
==4198== 4,000 bytes in 1 blocks are definitely lost in loss record 3 of 3
==4198==    at 0x4007E08: malloc (vg_replace_malloc.c:270)
==4198==    by 0x8048441: g (code1.c:4)
==4198==    by 0x804849B: main (code1.c:22)
==4198==
==4198== LEAK SUMMARY:
==4198==    definitely lost: 10,000 bytes in 3 blocks
==4198==    indirectly lost: 0 bytes in 0 blocks
==4198==      possibly lost: 0 bytes in 0 blocks
==4198==    still reachable: 0 bytes in 0 blocks
==4198==         suppressed: 0 bytes in 0 blocks
```

Remark: This command shows where the memories are lost

Example:Memory lost

```
#valgrind --tool=massif --time-unit=B ./file
#ms_prinf massif.out.pid
```

Remark: this graph tells the evolution of the heap memory

```
    KB
19.63^                                               ###
     |                                               #
     |                                               #  ::
     |                                               #  : :::
     |                                      :::::::::#  : :  ::
     |                                      :        #  : :  : ::
     |                                      :        #  : :  : : :::
     |                                      :        #  : :  : : :  ::
     |                            :::::::::::        #  : :  : : :  : :::
     |                            :         :        #  : :  : : :  : :  ::
     |                        :::::         :        #  : :  : : :  : :  : ::
     |                     @@@:   :         :        #  : :  : : :  : :  : : @
     |                   ::@  :   :         :        #  : :  : : :  : :  : : @
     |                :::: @  :   :         :        #  : :  : : :  : :  : : @
     |              :::  : @  :   :         :        #  : :  : : :  : :  : : @
     |            ::: :  : @  :   :         :        #  : :  : : :  : :  : : @
     |         :::: : :  : @  :   :         :        #  : :  : : :  : :  : : @
     |       :::  : : :  : @  :   :         :        #  : :  : : :  : :  : : @
     |    :::: :  : : :  : @  :   :         :        #  : :  : : :  : :  : : @
     |  :::  : :  : : :  : @  :   :         :        #  : :  : : :  : :  : : @
   0 +----------------------------------------------------------------------->KB
     0                                                                   29.48
```

Example: Memory lost

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0                0             0            0
  1          1,008            1,008            1,000             8            0
  2          2,016            2,016            2,000            16            0
  3          3,024            3,024            3,000            24            0
  4          4,032            4,032            4,000            32            0
  5          5,040            5,040            5,000            40            0
  6          6,048            6,048            6,000            48            0
  7          7,056            7,056            7,000            56            0
  8          8,064            8,064            8,000            64            0
  9          9,072            9,072            9,000            72            0
99.21% (9,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->99.21% (9,000B) 0x804847A: main (code1.c:19)
```

1st detailed snapshot, indicate where the memory is allocated

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 10         10,080           10,080           10,000            80            0
 11         12,088           12,088           12,000            88            0
 12         16,096           16,096           16,000            96            0
 13         20,104           20,104           20,000           104            0
 14         20,104           20,104           20,000           104            0
99.48% (20,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->49.74% (10,000B) 0x804847A: main (code1.c:19)
|
->39.79% (8,000B) 0x8048440: g (code1.c:4)
| ->19.90% (4,000B) 0x8048459: f (code1.c:10)
| | ->19.90% (4,000B) 0x8048495: main (code1.c:21)
| |
| ->19.90% (4,000B) 0x804849A: main (code1.c:22)
|
->09.95% (2,000B) 0x8048454: f (code1.c:9)
  ->09.95% (2,000B) 0x8048495: main (code1.c:21)
```

Peak snapshot, with the detail

Example: Memory lost

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 15         21,112           19,096           19,000            96            0
 16         22,120           18,088           18,000            88            0
 17         23,128           17,080           17,000            80            0
 18         24,136           16,072           16,000            72            0
 19         25,144           15,064           15,000            64            0
 20         26,152           14,056           14,000            56            0
 21         27,160           13,048           13,000            48            0
 22         28,168           12,040           12,000            40            0
 23         29,176           11,032           11,000            32            0
 24         30,184           10,024           10,000            24            0
99.76% (10,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->79.81% (8,000B) 0x8048440: g (code1.c:4)
| ->39.90% (4,000B) 0x8048459: f (code1.c:10)
| | ->39.90% (4,000B) 0x8048495: main (code1.c:21)
| |
| ->39.90% (4,000B) 0x804849A: main (code1.c:22)
|
->19.95% (2,000B) 0x8048454: f (code1.c:9)
| ->19.95% (2,000B) 0x8048495: main (code1.c:21)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)
```

Lost memories, and where the memory is allocated

Massif: example, stack profiling

```
void f(void)
{
char buffer [1000];

}

int main(void)
{
char buffer [100];

    f();
    return 0;
}
```

Massif: example, stack profiling

```
#gcc -Wall -g -o file file.c

#valgrind --tool=massif --time-unit=B --stacks=yes ./file
#ms_print massif.out.PID
```

```
    KB
2.352^       #
     |       #
     |       #
     |    :  #
     |    :  #
     |    :  #
     |  : :  #
     |  : :  #
     |  : :  #                                              :
     |  : :  #                             ::        @      : ::
     |  : :  #    ::       :::           : :         @      : ::
     |  : :@@#:   : :@:@@::: ::::::::::: ::: ::::::::@::::  : ::::
     |:::@:@ #:   : :@:@ ::: :::: ::: :: ::: ::::::: @:::   : ::::
     |: :@:@ #:   : :@:@ ::: :::: ::: :::::: ::::::: @::: ::::::::
     |: :@:@ #: ::: :@:@ ::: :::: ::: :::::: ::::::: @::: : ::::::
     |: :@:@ #::::: :@:@ ::: :::: ::: :::::: ::::::: @::: : ::::::
     |: :@:@ #::::: :@:@ ::: :::: ::: :::::: ::::::: @::: : ::::::    :
     |: :@:@ #::::: :@:@ ::: :::: ::: :::::: ::::::: @::: : :::::: @: :: :@: :
     |: :@:@ #::::: :@:@ ::: :::: ::: :::::: ::::::: @::: : :::::::@::::::@:::
     |: :@:@ #::::: :@:@ ::: :::: ::: :::::: ::::::: @::: : :::::::@::::::@:::
   0 +----------------------------------------------------------------------->KB
     0                                                                   183.2
```

Callgrind

- Callgrind: a call-graph generating cache and branch prediction profiler

- Callgrind is a profiling tool that records the call history among functions in a program's run as a call-graph.
- By default, the collected data consists of:
  - the number of instructions executed,
  - their relationship to source lines,
  - the caller/callee relationship between functions,
  - and the numbers of such calls.
- Optionally, cache simulation and/or branch prediction (similar to Cachegrind) can produce further information about the runtime behavior of an application.

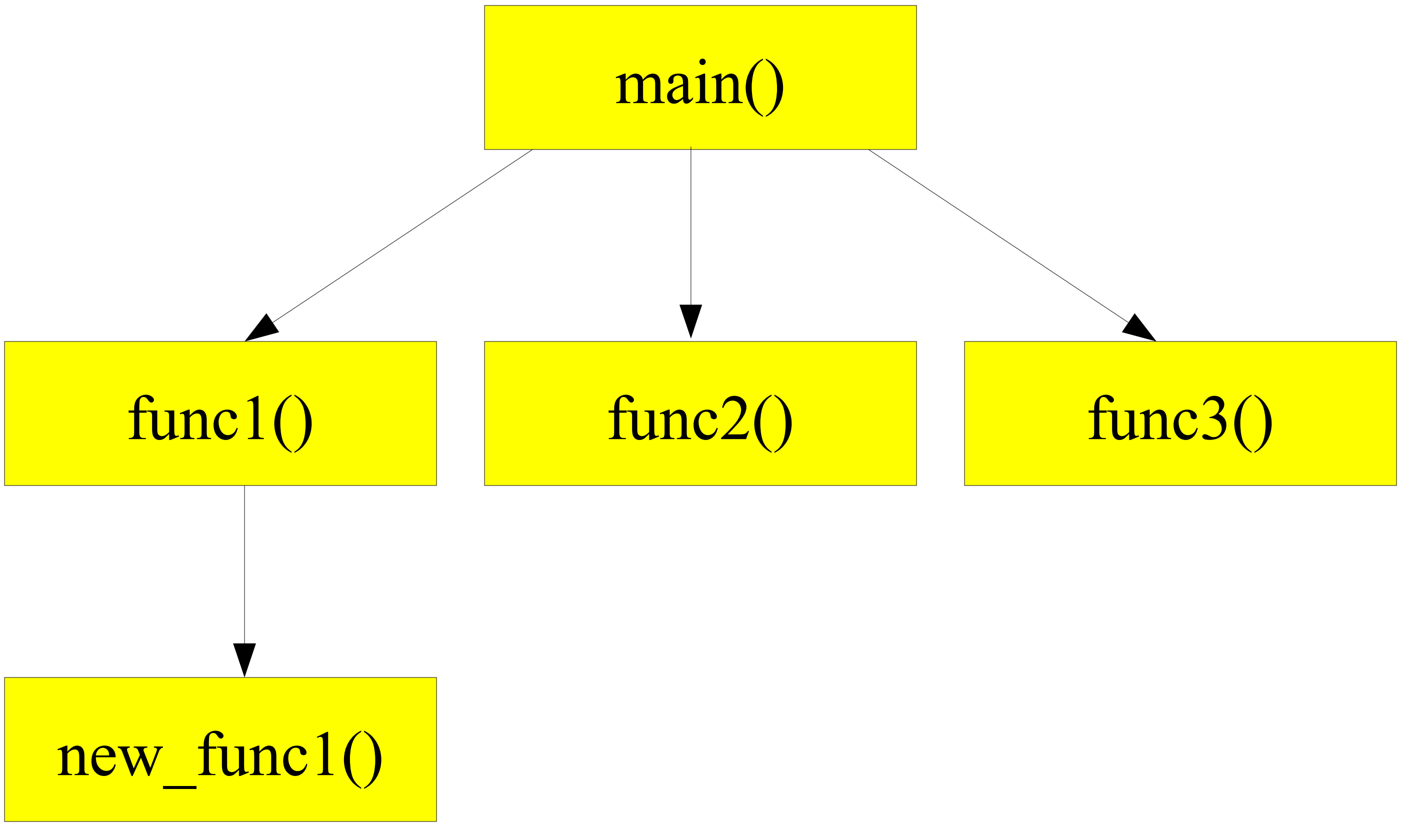
Callgrind: example

```c
// test.c
//gcc -Wall -o test test.c test_new.c


#include<stdio.h>

void new_func1(void);

void func1(void)
{
    printf("\n Inside func1 \n");
    int i = 0;

    for(;i<0xffff;i++);
    new_func1();
    return;
}

static void func2(void)
{
    printf("\n Inside func2 \n");
    int i = 0;

    for(;i<0xffaa;i++);
    return;
}
```

```c
static void func3(void)
{
    printf("\n Inside func2 \n");
    return;
}

int main(void)
{
    printf("\n Inside main()\n");
    int i = 0;

    for(;i<0xfff;i++);
    func1();
    func2();
    func3();

    return 0;
}
```

```c
//test_new.c
void new_func1(void) {
    printf("\n Inside new_func1()\n");
    int i = 0;

    for(;i<0xffee;i++);

    return;
}
```

Callgrind: example



Callgrind: example

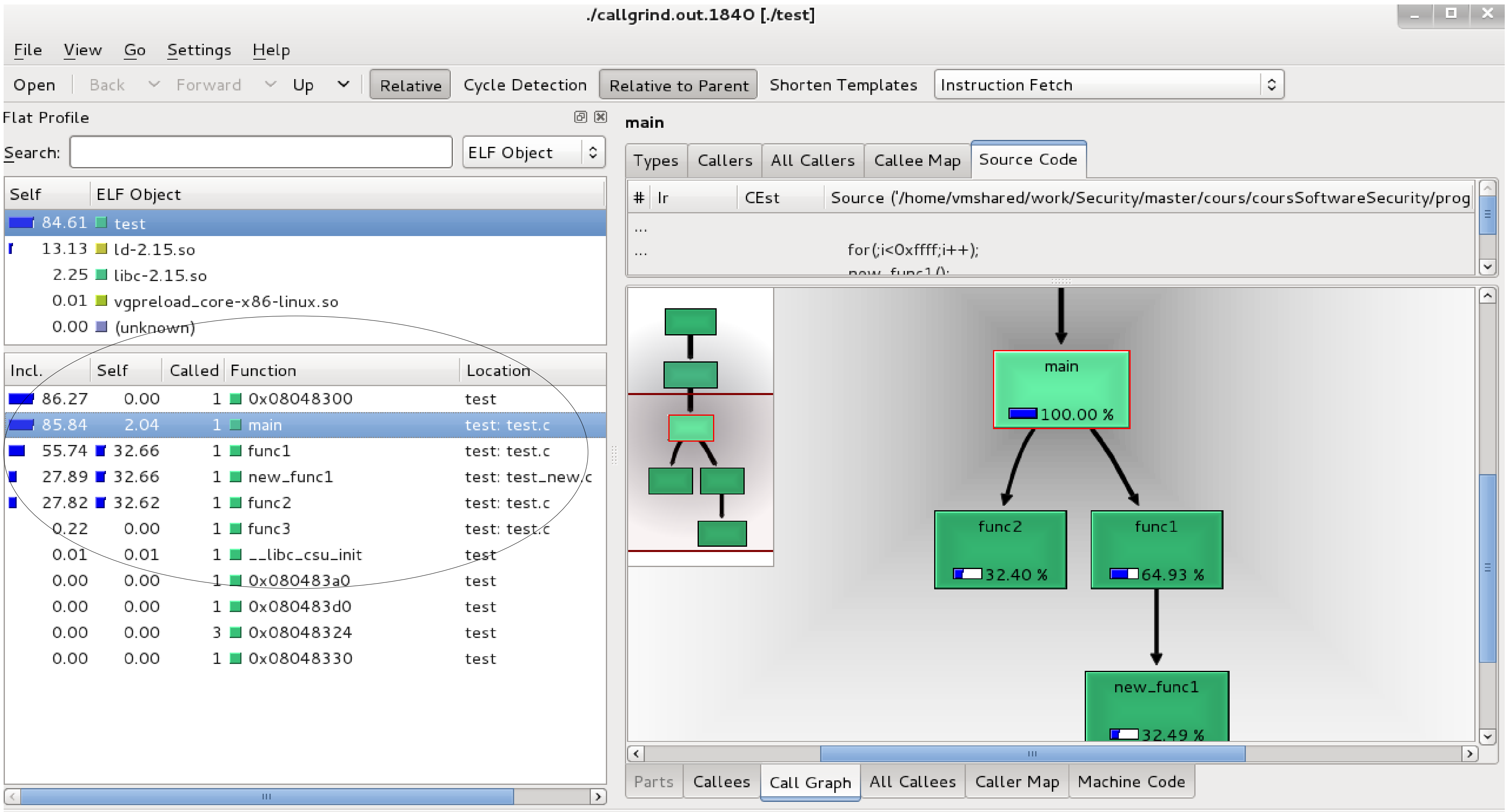
```
#gcc -Wall -g -o test test.c test_new.c
```

```
#valgrind --tool=callgrind ./test
```

- This command generates a file: *callgrind_out.PID*

```
#kcachegrind
```

Callgrind: example



Cachegrind

- Cachegrind: a cache and branch-prediction profiler

- Cachegrind simulates how your program interacts with a machine's cache hierarchy and (optionally) branch predictor.
- It simulates a machine with independent first-level instruction and data caches (I1 and D1), backed by a unified second-level cache (L2). This exactly matches the configuration of many modern machines.
- However, some modern machines have three levels of cache. For these machines (in the cases where Cachegrind can auto-detect the cache configuration) Cachegrind simulates the first-level and third-level caches. The reason for this choice is that the L3 cache has the most influence on runtime, as it masks accesses to main memory.
- Therefore, Cachegrind always refers to the I1, D1 and LL (last-level) caches.

Cachegrind

- Cachegrind gathers the following statistics (abbreviations used for each statistic is given in parentheses):

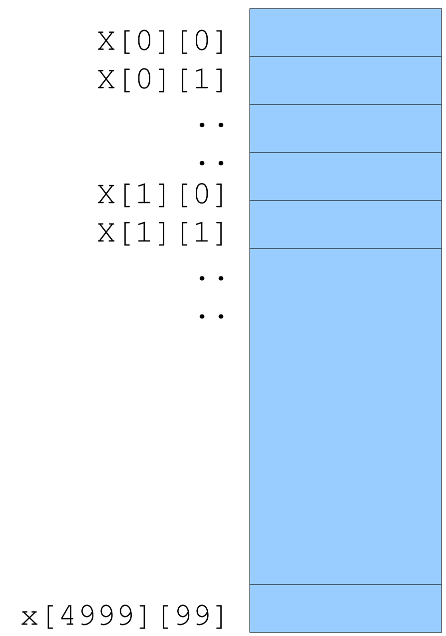
- **I cache reads** (**Ir**, which equals the number of instructions executed), I1 cache read misses (**I1mr**) and LL cache instruction read misses (**ILmr**).
- **D cache reads** (**Dr**, which equals the number of memory reads), D1 cache read misses (**D1mr**), and LL cache data read misses (**DImr**).
- **D cache writes** (**Dw**, which equals the number of memory writes), D1 cache write misses (**D1mw**), and LL cache data write misses **(DLmw**).
- Conditional branches executed (**Bc**) and conditional branches mispredicted **(Bcm**).
- Indirect branches executed (**Bi**) and indirect branches mispredicted (**Bim**).

- Note that **D1** total accesses is given by **D1mr + D1mw**, and that **LL** total accesses is given by **ILmr + DLmr + DLmw**.

Cachegrind: example

```
//gcc -Wall -o test test.c test_new.c

#include<stdio.h>

#define VAL1     5000
#define VAL2      100

static int x[VAL1][VAL2];

static int badFunction(void) {
int       i, j;

    for (j=0; j<VAL2; j++) {
        for (i=0; i<VAL1; i++)
            x[i][j] = x[i][j]*2;
    }
    return 0;
}
static int goodFunction(void) {
int       i, j;

    for (i=0; i<VAL1; i++)
    {
        for (j=0; j<VAL2; j++)
            x[i][j] = x[i][j]*2;
    }
    return 0;
}
```

```
int main (void)
{

    badFunction();
    goodFunction();
    return (0);
}
```



Cachegrind: example

```
#gcc -Wall -g -o test test.c
#valgrind --tool=cachegrind ./test
```

- This command generates a file: *cachegrind_out.PID*

```
#cg_annotate –auto=yes cachegrind_out.PID
```

Cachegrind: example

```
--------------------------------------------------------------------------------
I1 cache:         32768 B, 64 B, 8-way associative
D1 cache:         32768 B, 64 B, 8-way associative
LL cache:         2097152 B, 64 B, 8-way associative
Command:          ./1_test
Data file:        cachegrind.out.3489
Events recorded:  Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Events shown:     Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Event sort order: Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Thresholds:       0.1 100 100 100 100 100 100 100 100
Include dirs:
User annotated:
Auto-annotation:  on
```

Cachegrind: example

```
--------------------------------------------------------------------------------
       Ir I1mr ILmr         Dr      D1mr    DLmr        Dw D1mw DLmw
        .    .    .          .         .       .         .    .    .
        .    .    .          .         .       .         .    .    .  static int badFunction(void)
        3    0    0          0         0       0         1    0    0  {
        .    .    .          .         .       .         .    .    .  int       i, j;
        .    .    .          .         .       .         .    .    .
        .    .    .          .         .       .         .    .    .
      304    1    1        201         0       0         1    0    0      for (j=0; j<VAL2; j++)
        .    .    .          .         .       .         .    .    .      {
1,500,400    0    0  1,000,100         0       0       100    0    0          for (i=0; i<VAL1; i++)
5,500,000    0    0  2,500,000   500,000  31,251   500,000    0    0              x[i][j] = x[i][j]*2;
        .    .    .          .         .       .         .    .    .      }
        1    0    0          0         0       0         0    0    0      return 0;
        2    0    0          2         0       0         0    0    0  }
        .    .    .          .         .       .         .    .    .
        .    .    .          .         .       .         .    .    .
        .    .    .          .         .       .         .    .    .  static int goodFunction(void)
        3    0    0          0         0       0         1    0    0  {
        .    .    .          .         .       .         .    .    .  int       i, j;
        .    .    .          .         .       .         .    .    .
        .    .    .          .         .       .         .    .    .
   15,004    1    1     10,001         0       0         1    0    0      for (i=0; i<VAL1; i++)
        .    .    .          .         .       .         .    .    .      {
1,520,000    0    0  1,005,000         0       0     5,000    0    0          for (j=0; j<VAL2; j++)
5,500,000    0    0  2,500,000    31,251       0   500,000    0    0              x[i][j] = x[i][j]*2;
        .    .    .          .         .       .         .    .    .      }
        1    0    0          0         0       0         0    0    0      return 0;
        2    0    0          2         0       0         0    0    0  }
        .    .    .          .         .       .         .    .    .
```