



Humboldt University

Computer Science Department
Systems Architecture Group
<http://sar.informatik.hu-berlin.de>

Operating Systems Principles

Valgrind

Valgrind



What is Valgrind?

- a multipurpose code profiling and memory debugging tool
- monitors memory usage such as calls to malloc and free (or new and delete in C++)
- Valgrinds detects:
 - usage of uninitialized memory
 - write off the end of an array
 - if you forget to free a pointer
- Other tools
 - *Addrcheck*, similar to Memcheck but with much smaller CPU and memory overhead.
 - *Massif*, a heap profiler
 - *Helgrind* and *DRD*, detect race conditions in multithreaded code
 - *Cachegrind*, a cache profiler (GUI KCacheGrind)
 - *Callgrind*, a callgraph analyzer (GUI KCacheGrind)

Valgrind



Usage

- `valgrind --tool=valgrind_tool program_name`
- tools:
 - memcheck
 - callgrind
 - cachegrind
- Example

```
% valgrind --tool=memcheck program_name
...
==18515== malloc/free: in use at exit: 0 bytes in 0 blocks.
==18515== malloc/free: 1 allocs, 1 frees, 10 bytes allocated.
==18515== For a detailed leak analysis, rerun with: --leak-check=yes
```

Valgrind - Memcheck

- Code

```
#include <stdlib.h>

int main()
{
  char *x = malloc(100); /* or, in C++, "char *x = new char[100] */
  x = NULL;
  return 0;
}
```

- Run

```
% valgrind --tool=memcheck --leak-check=yes example1
```

- Output

```
==2330== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2330== at 0x1B900DD0: malloc (vg_replace_malloc.c:131)
==2330== by 0x804840F: main (example1.c:5)
```

- Result

- **Memory leak**
- forget to call *free()* for „x“

Valgrind - Memcheck

- Code

```
#include <stdlib.h>

int main()
{
    char *x = malloc(10);
    x[10] = 'a';
    return 0;
}
```

- Run

```
% valgrind --tool=memcheck --leak-check=yes example2
```

- Output

```
==9814== Invalid write of size 1
==9814== at 0x804841E: main (example2.c:6)
==9814== Address 0x1BA3607A is 0 bytes after a block of size 10 alloc'd
==9814== at 0x1B900DD0: malloc (vg_replace_malloc.c:131)
==9814== by 0x804840F: main (example2.c:5)
```

- Result

- **Invalid write**

Valgrind - Memcheck

- Code

```
#include <stdio.h>

int main()
{
    int x;
    if(x == 0) {
        printf("X is zero"); /* replace with cout and include iostream for C++ */
    }
    return 0;
}
```

- Run

```
% valgrind --tool=memcheck --leak-check=yes example3
```

- Output

```
==17943== Conditional jump or move depends on uninitialised value(s)
==17943== at 0x804840A: main (example3.c:6)
```

- Result

- Use Of Uninitialized Variables

Valgrind - Memcheck



- Code

```
#include <stdio.h>

int foo(int x) {
    if(x < 10) {
        printf("x is less than 10\n");
    }
}

int main() {
    int y;
    foo(y);
}
```

- Output

```
==4827== Conditional jump or move depends on uninitialised value(s)
==4827== at 0x8048366: foo (example4.c:5)
==4827== by 0x8048394: main (example4.c:14)
```

- Result

- Use Of Uninitialized Variables

Valgrind - Callgrind

- Code

```
#include <iostream>
#include <atomic>

std::atomic<int> cnt(0);

void f() {
    for (int n = 0; n < 10000; ++n)
        cnt.fetch_add(1, std::memory_order_relaxed);
}

void g() {
    for (int n = 0; n < 10000; ++n)
        cnt.fetch_add(1, std::memory_order_seq_cst);
}

void h() {
    for (int n = 0; n < 10000; ++n)
        cnt++;
}

int main()
{
    f(); g(); h();

    std::cout << "Counter: " << cnt << "\n";
}
```


Valgrind - Callgrind

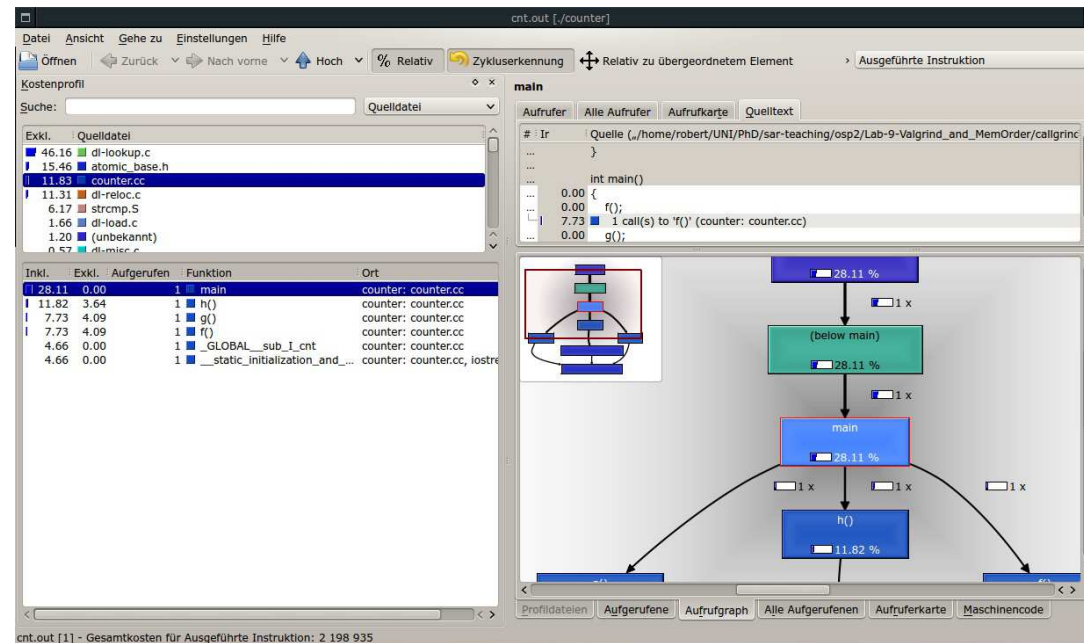


- Run

```
% valgrind --tool=callgrind --collect-jumps=yes --dump-instr=yes --callgrind-out-file=cnt.out ./counter
```

- Result
 - File *cnt.out*
 - can be used with *KCachegrind*

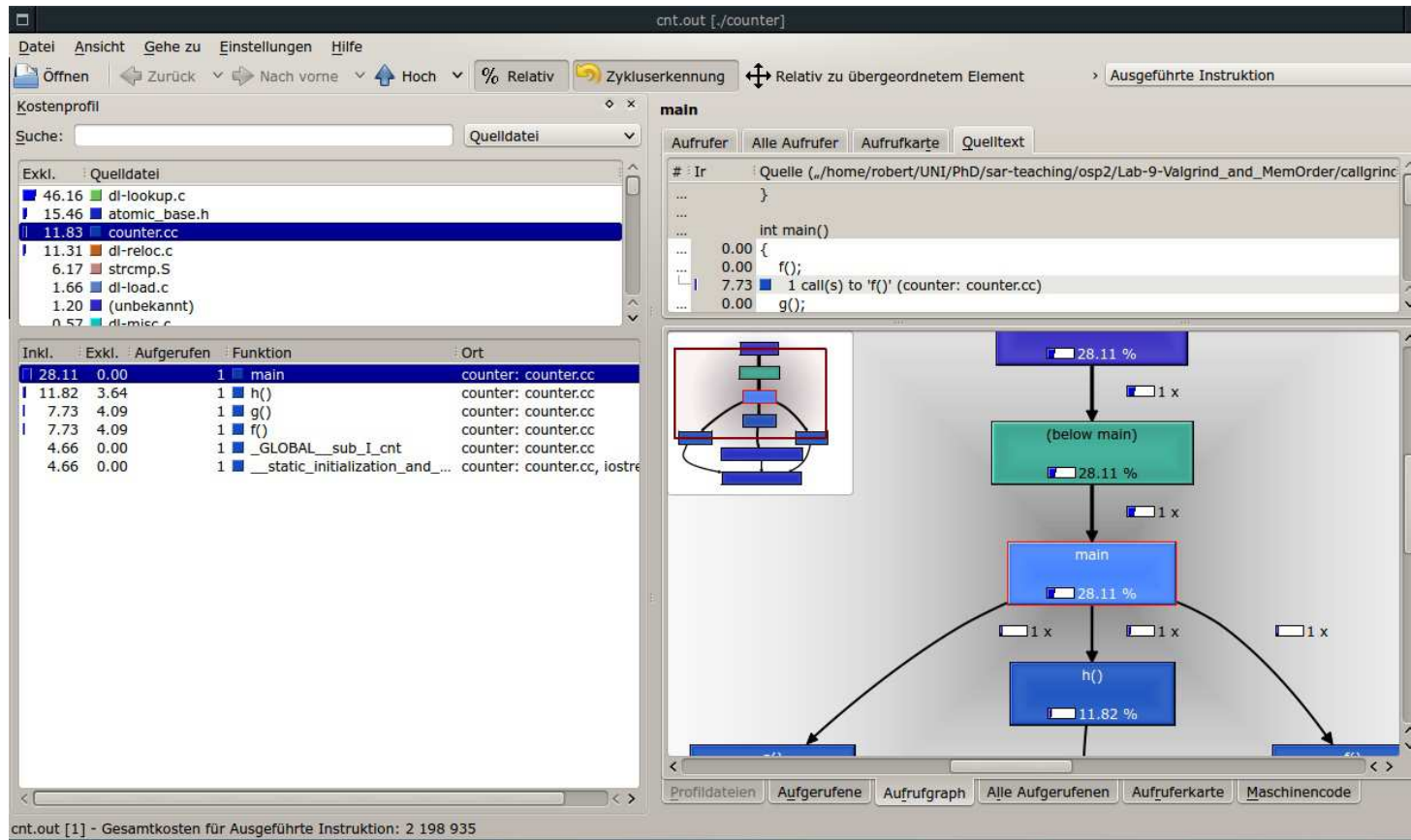
```
% kcachegrind cnt.out
```



KCachegrind



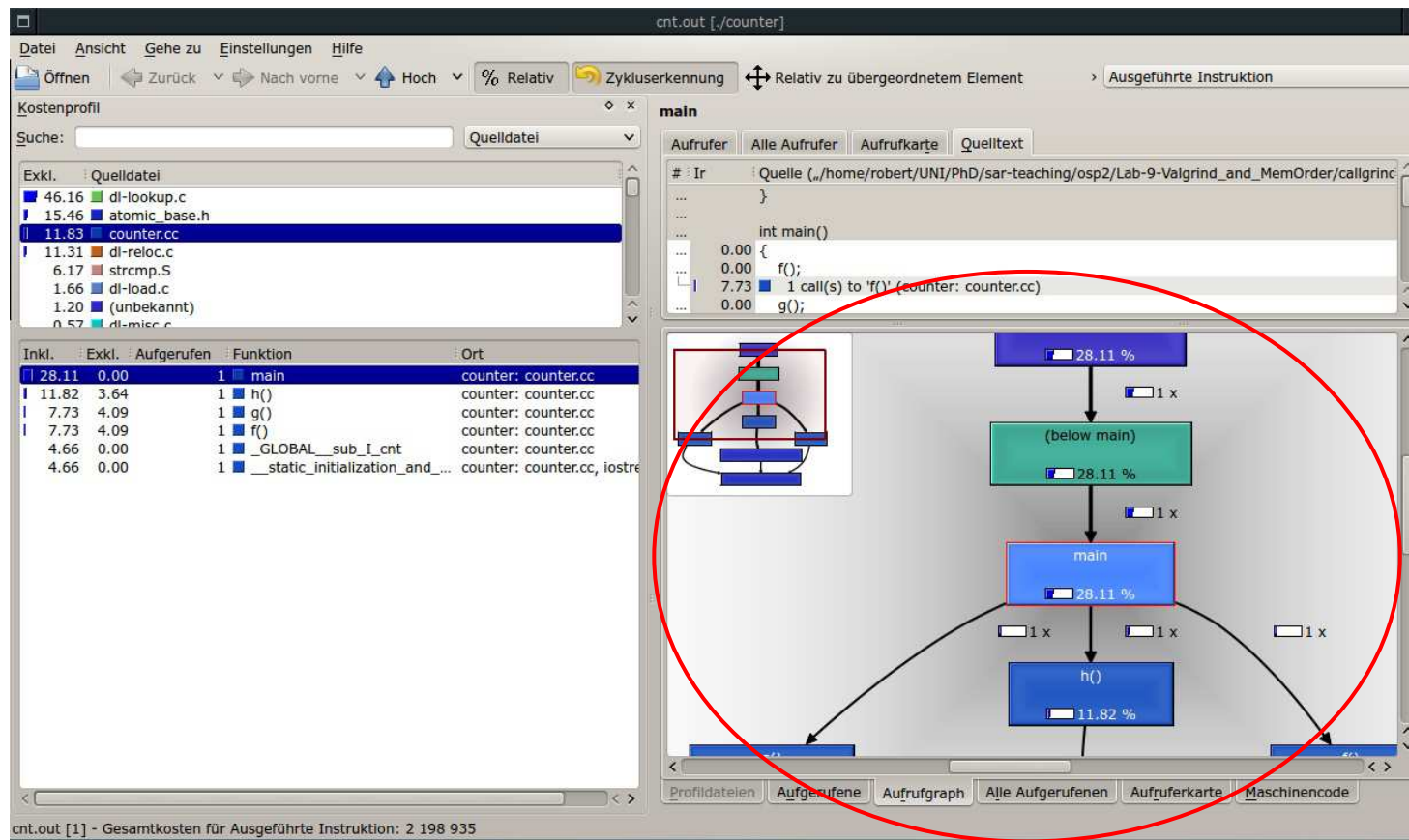
- A profile data visualization



KCachegrind

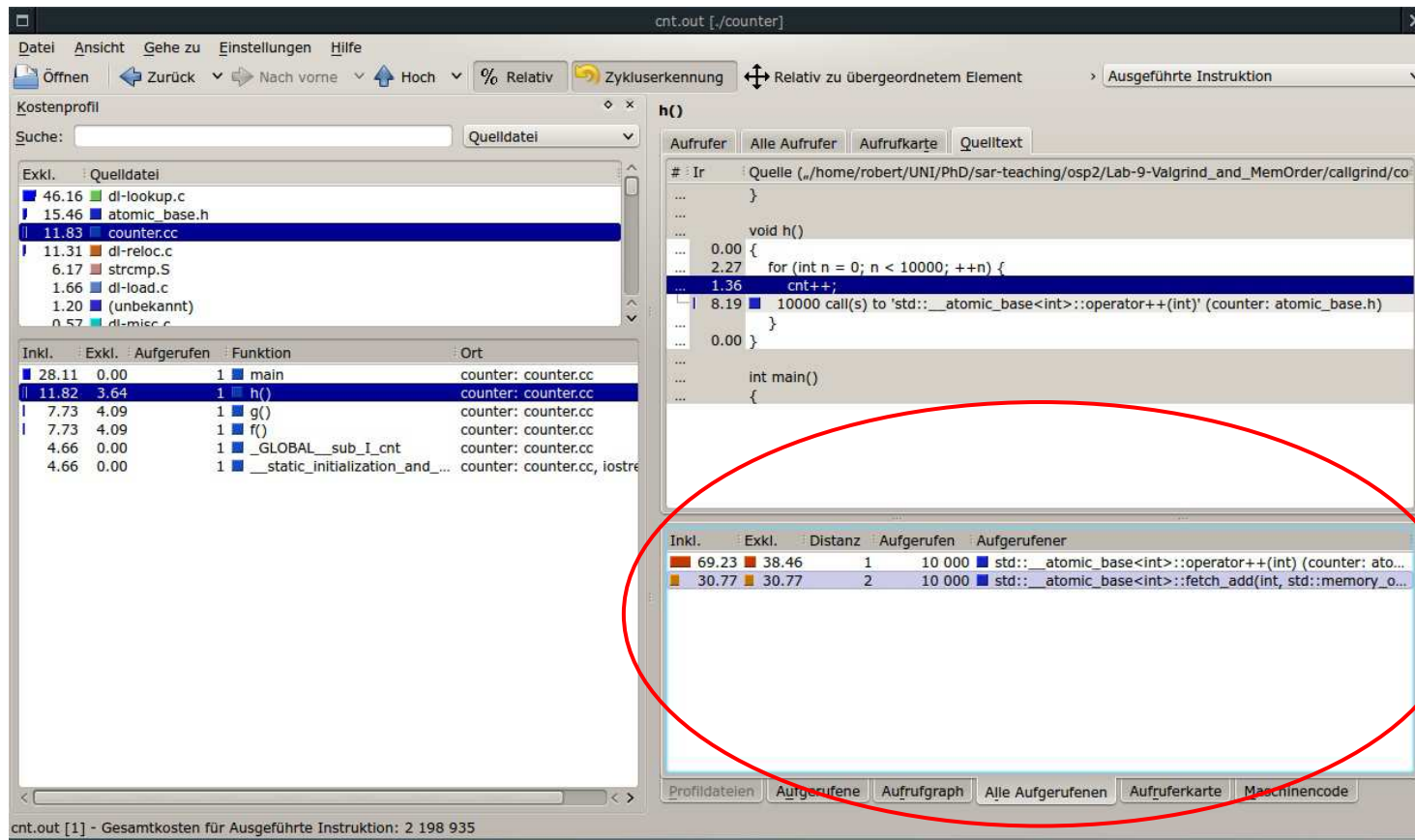


- Aufrufgraph
 - Welche Funktion ruft welche Funktion auf



KCachegrind

- Aufgerufene
 - Welche Funktion ruft welche Funktion auf

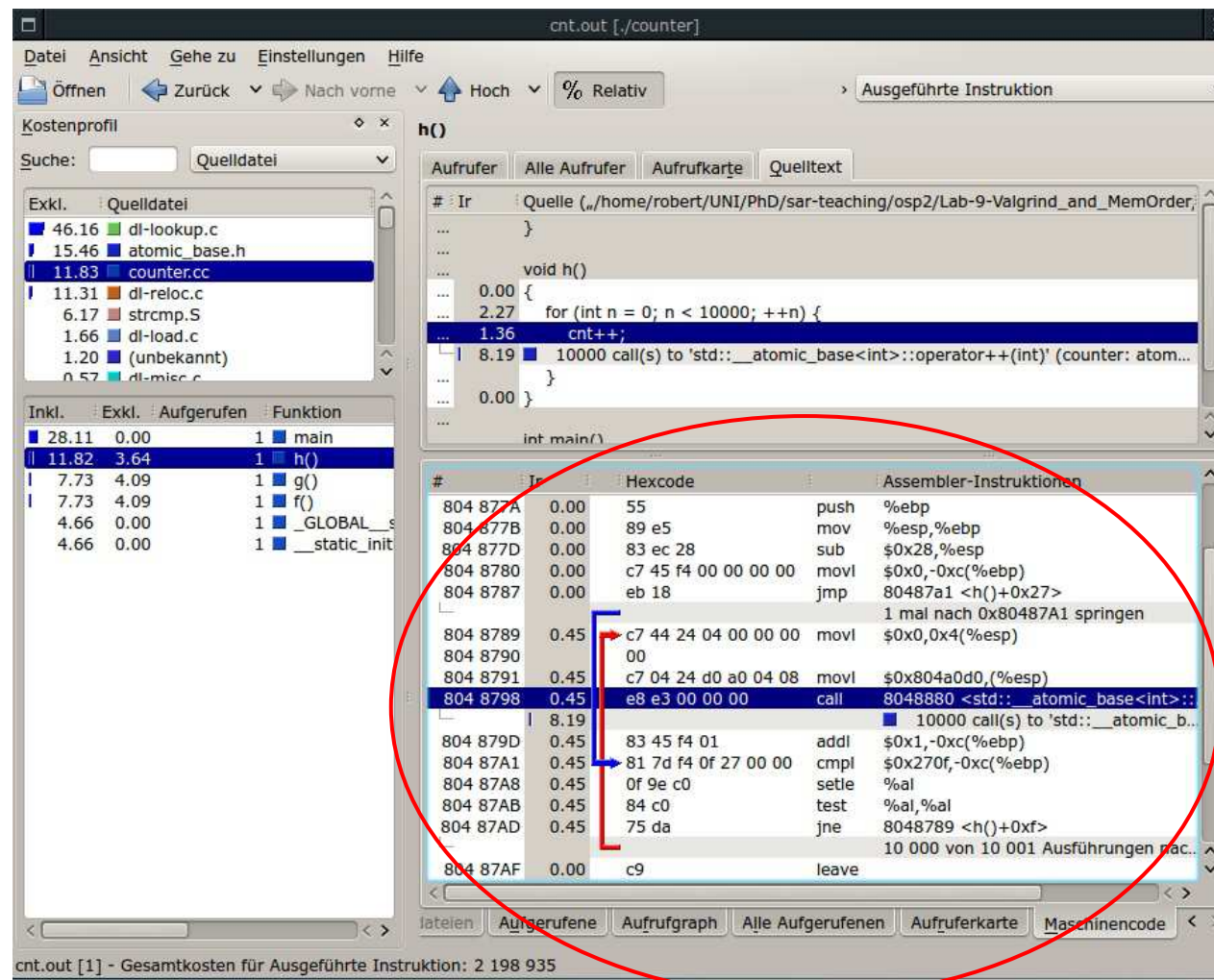


The screenshot shows the KCachegrind interface. The main window displays the source code for the function `h()` in `counter.cc`. The function contains a loop that calls `cnt++` 10,000 times. The 'Aufgerufene' table at the bottom of the window is circled in red and contains the following data:

Inkl.	Exkl.	Distanz	Aufgerufen	Aufgerufener
69.23	38.46	1	10 000	std::atomic_base<int>::operator++(int) (counter: ato...
30.77	30.77	2	10 000	std::atomic_base<int>::fetch_add(int, std::memory_o...

KCachegrind

- Maschinencode
 - Zeigt den Maschinencode des Programms



The screenshot shows the KCachegrind interface. The main window displays the assembly code for the `h()` function. A red circle highlights the assembly instructions for the call to `std::atomic_base::operator++(int)` at address `8048798`. The instructions are as follows:

#	Ir	Hexcode	Assembler-Instruktionen
804 877A	0.00	55	push %ebp
804 877B	0.00	89 e5	mov %esp,%ebp
804 877D	0.00	83 ec 28	sub \$0x28,%esp
804 8780	0.00	c7 45 f4 00 00 00 00	movl \$0x0,-0xc(%ebp)
804 8787	0.00	eb 18	jmp 80487a1 <h()+0x27>
1 mal nach 0x80487A1 springen			
804 8789	0.45	c7 44 24 04 00 00 00	movl \$0x0,0x4(%esp)
804 8790	0.00	00	
804 8791	0.45	c7 04 24 d0 a0 04 08	movl \$0x804a0d0,(%esp)
804 8798	0.45	e8 e3 00 00 00	call 8048880 <std::atomic_base<int>::operator++(int)::operator++(int)>
10000 call(s) to 'std::atomic_b..			
804 879D	0.45	83 45 f4 01	addl \$0x1,-0xc(%ebp)
804 87A1	0.45	81 7d f4 0f 27 00 00	cmpl \$0x270f,-0xc(%ebp)
804 87A8	0.45	0f 9e c0	setle %al
804 87AB	0.45	84 c0	test %al,%al
804 87AD	0.45	75 da	jne 8048789 <h()+0xf>
10 000 von 10 001 Ausführungen pac..			
804 87AF	0.00	c9	leave

Links



<http://valgrind.org/>

<http://valgrind.org/docs/manual/manual-core.html>

<http://www.cprogramming.com/debugging/valgrind.html>