SAP Research Talk

Valgrind

Linux x86 memory debugger and performance profiler

Aidan Shribman
SAP Labs Israel
November 2013

Agenda

Introduction

Detecting memory errors

Heap profiling and leak detection

Data concurrency race conditions

Profiling applications

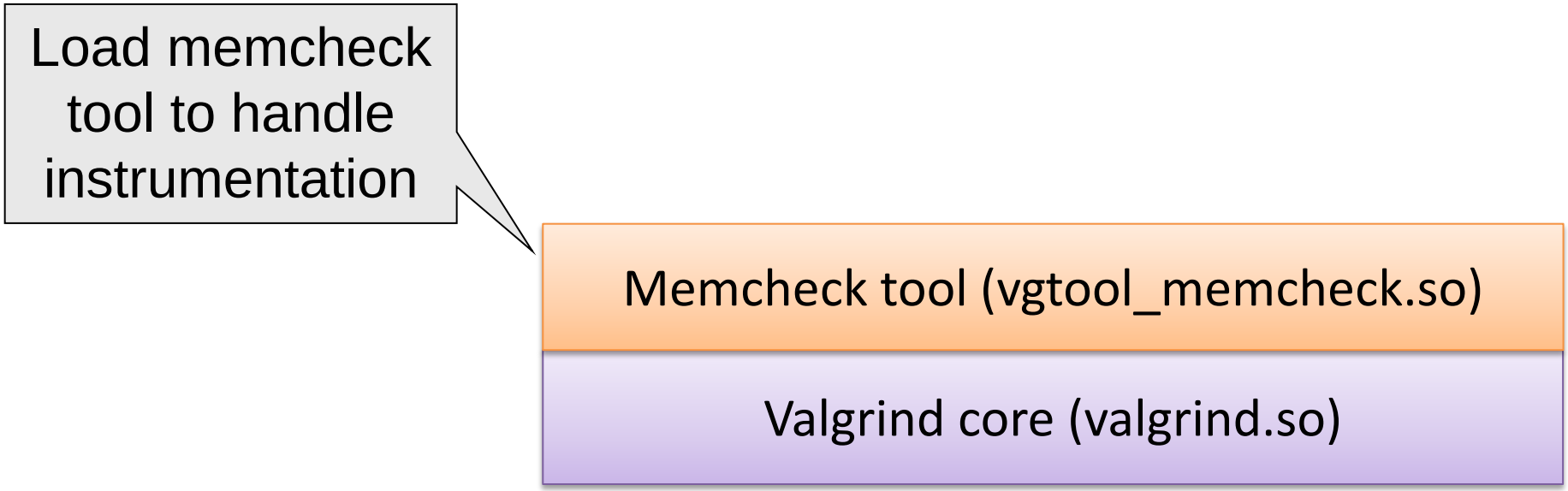
Summary

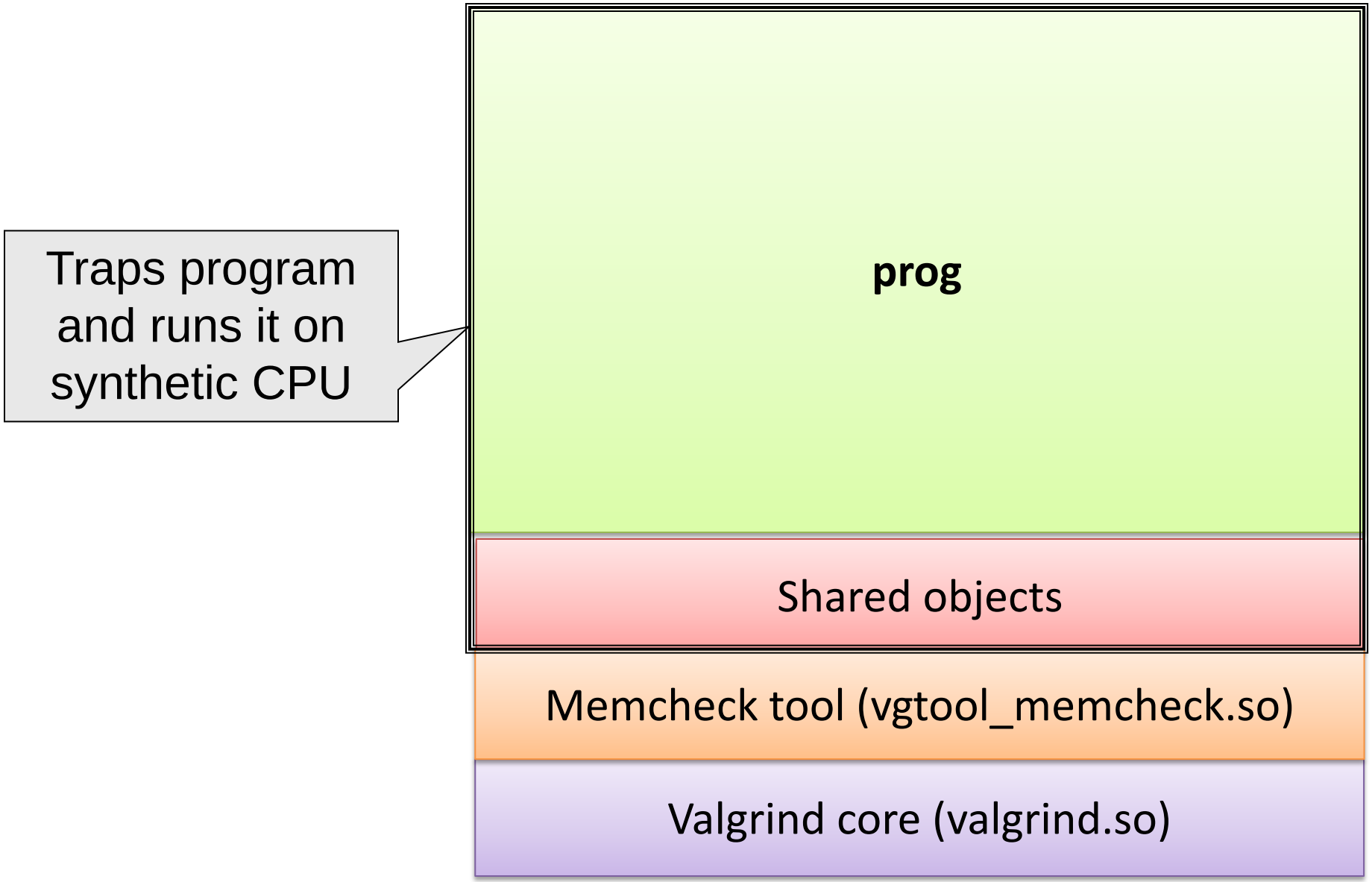
What is Valgrind?

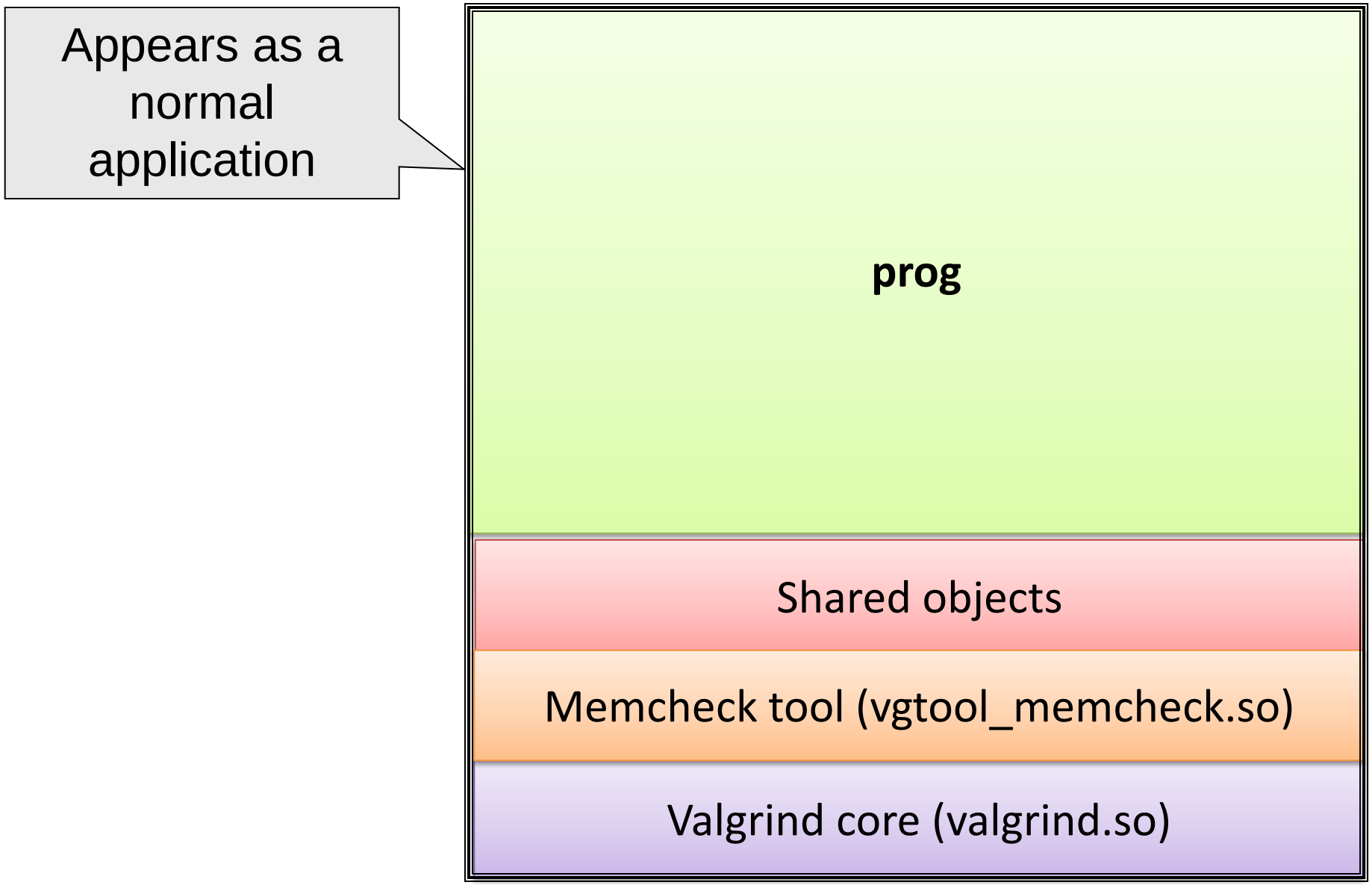
- **An open source system memory debugger:** Initiated by Julian Sward on 2000; released for Linux x86 on 2002 and winner of Open Source Award of 2004.
- **Simple and easy to use:** does not require re-compilation or re-linking: performs dynamic runtime instrumentation running on a synthetic CPU.
- **Used to validate many large Projects:** KDE, Emacs, Mozilla, OpenOffice and ... SAP HANA.
- **Programing language agnostic:** used for C/C++, Python, Java, JavaScript development.

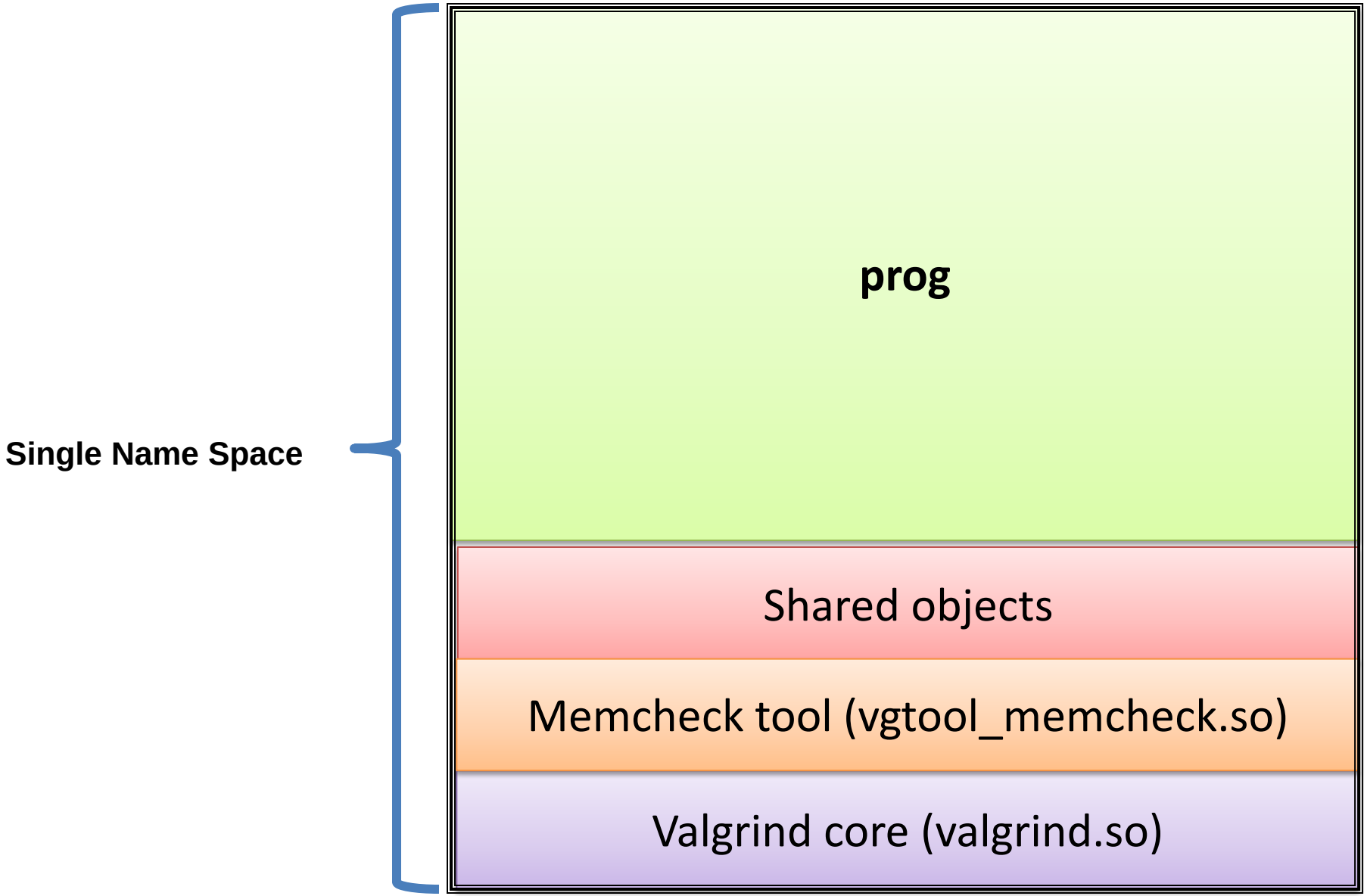
```
$ valgrind --tool=memcheck \
prog <args>
```

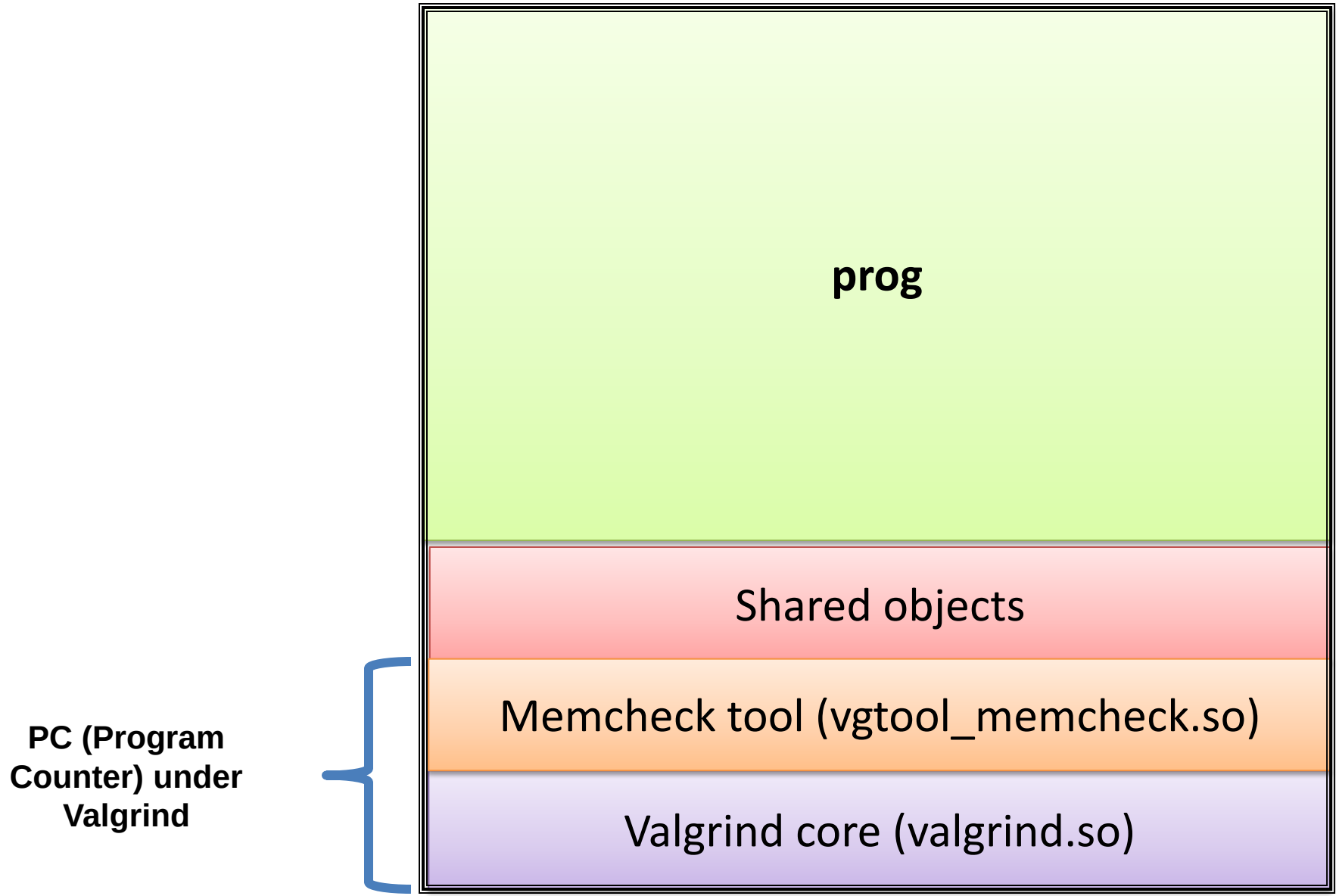

Uses
LD_PRELOAD
to load first
Valgrind core (valgrind.so)

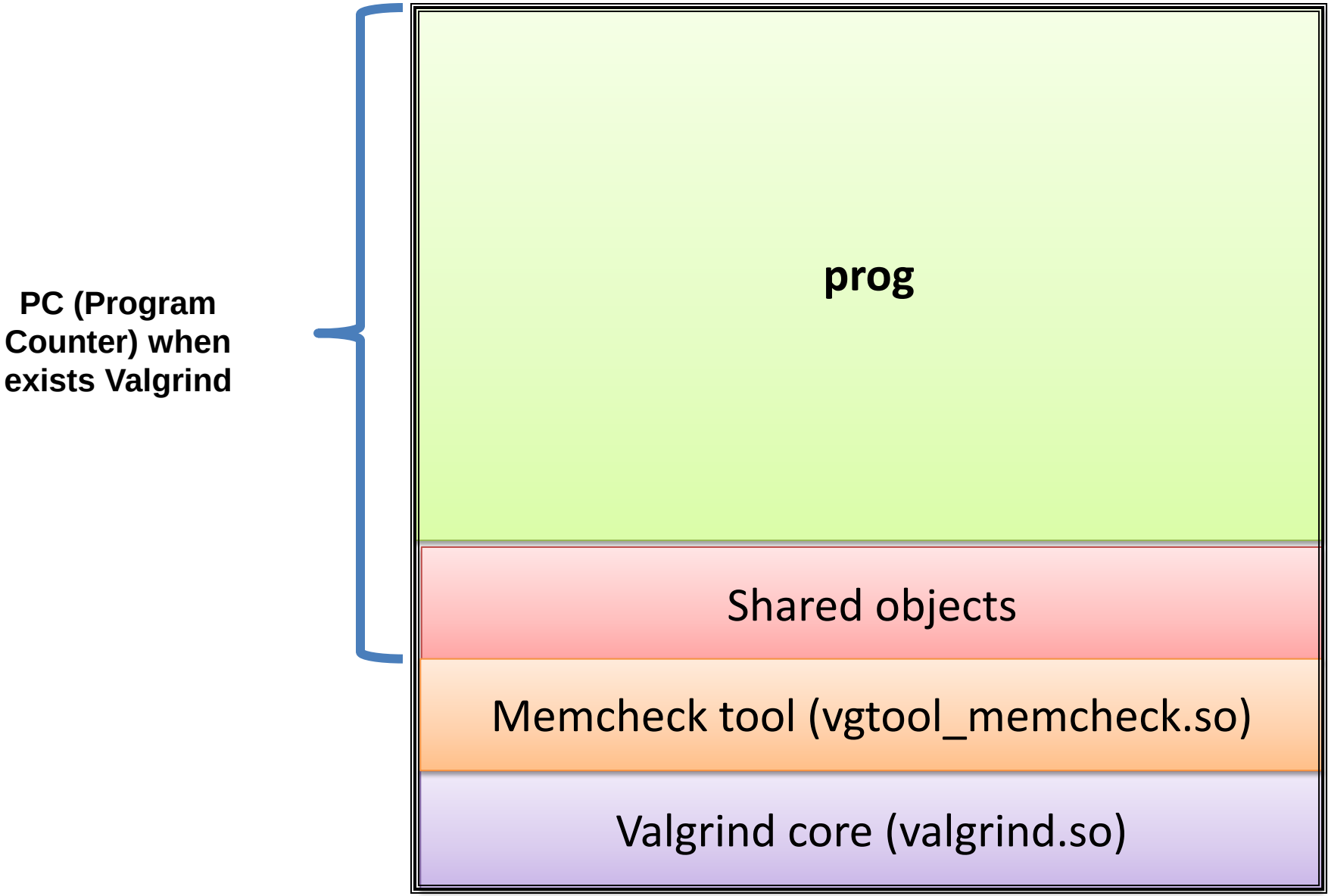

Load memcheck tool to handle instrumentation
Memcheck tool (vgtool_memcheck.so)
Valgrind core (valgrind.so)


Traps program and runs it on synthetic CPU
prog
Shared objects
Memcheck tool (vgtool_memcheck.so)
Valgrind core (valgrind.so)


Appears as a normal application
prog
Shared objects
Memcheck tool (vgtool_memcheck.so)
Valgrind core (valgrind.so)


Single Name Space
prog
Shared objects
Memcheck tool (vgtool_memcheck.so)
Valgrind core (valgrind.so)


prog
Shared objects
Memcheck tool (vgtool_memcheck.so)
Valgrind core (valgrind.so)
PC (Program Counter) under Valgrind


PC (Program Counter) when exists Valgrind
prog
Shared objects
Memcheck tool (vgtool_memcheck.so)
Valgrind core (valgrind.so)

Dynamic Binary Instrumentation

"UCode lies at the heart of the x86-to-x86 JITter. The basic premise is that dealing the the x86 instruction set head-on is just too darn complicated, so we do the traditional compiler-writer's trick and translate it into a simpler, easier-to-deal-with form."

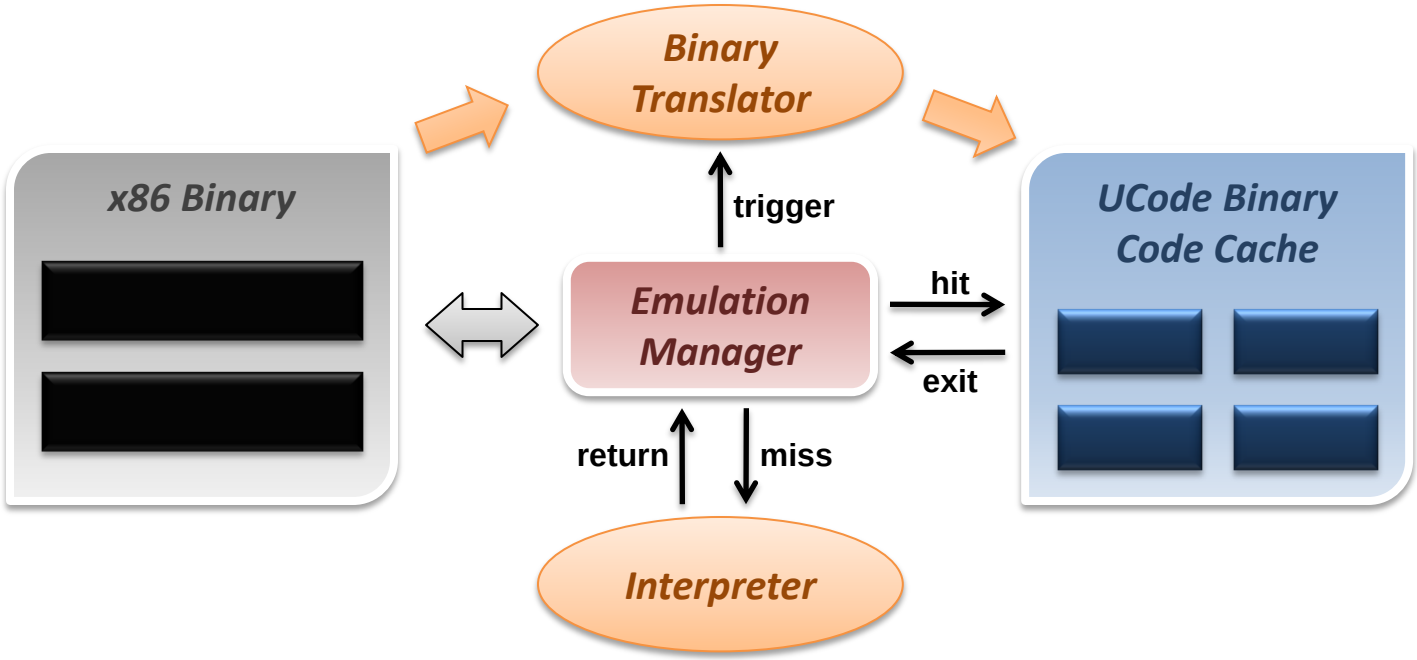
Julian Sward

The Instrumenting JITer

- Parse of an x86 basic block into a sequence of UCode instructions.
- UCode optimization with the aim of caching simulated registers in real registers.
- UCode instrumentation which adds value and address checking code.
- Post-instrumentation cleanup, removing redundant value-check computations.
- Register allocation done on UCode.
- Emission of final instrumented x86 code.

Dynamic Binary Translator

1. First time execution, no translated code in code cache.
2. Miss code cache matching, then directly interpret the guest instruction.
3. As a code block discovered, trigger the binary translation module.
4. Translate guest code block to host binary, and place it in the code cache.
5. Next time execution, run the translated code block from the code cache.



Installing Valgrind

```
$ zypper install valgrind
$ yum install valgrind
$ apt-get install valgrind
```

```
$ zypper install -y linux-kernel-headers
$ wget http://www.valgrind.org/downloads/valgrind-3.9.0.tar.bz2
$ tar xvfj valgrind-3.9.0.tar.bz2
$ cd valgrind-3.9.0
$ ./configure
$ make
$ make install
```

Memcheck

Memcheck

- Use of uninitialized memory
- Reading/writing memory after it has been freed
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks -- where pointers to malloc'd blocks are lost forever
- Passing of uninitialized or unaddressible memory to system calls
- Mismatched use of malloc/new/new[] vs free/delete/delete[]
- Some misuses of the POSIX pthreads API

- **A-bits:** every memory byte is shadowed by a single A (Addressability) bit indicating if the application can legitimately access the byte.
- **V-bits:** every memory byte is shadowed by eight V (Validity) bits indicating if the values of each bit are defined.
- **Heap blocks:** records heap blocks in auxiliary hash, enabling to track double-free and other miss-use.

```
$ valgrind --tool=memcheck \
      --num-callers=20 \
      --log-file=vg.log \
      --trace-children=yes \
      prog <args>
```

```
int v, *p = malloc(10 * sizeof(int));
v = p[10]; /* invalid read (heap) */
```

```
int a[10];
a[10] = 1; /* invalid write (stack) */
```

Not Detected!

```
int a, b;
b = a; /* pass uninitialized */
if (!b) /* use uninitialized */
        printf(“non deterministic\n”);
```

```
char *p = malloc(100); /* uninitialized */
write(1, p, 100); /* to syscall */
```

```
unsigned char c;
c |= 0x1; /* only 1st bit is initialized */
if (c & 0x1) /* bit-1 initialized */
        printf("always be printed\n");
if (c & 0x2) /* bit-2 uninitialized */
        printf("non deterministic\n");
```

Addrcheck

Addrcheck

- ~~Use of uninitialized memory~~
- Reading/writing memory after it has been freed
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks -- where pointers to malloc'd blocks are lost forever
- Passing of ~~uninitialized~~ or unaddressible memory to system calls
- Mismatched use of malloc/new/new[] vs free/delete/delete[]
- Some misuses of the POSIX pthreads API

```
$ valgrind --tool=addrcheck  \
        prog <args>
```

Performance slowdown

Program	Time (s)	Nulgrind	Memcheck	Addrcheck	Cachegrind
bzip2	10.7	2.4	13.6	9.1	31.0
crafty	3.5	7.2	44.6	26.5	107.4
gap	0.9	5.4	28.7	14.4	46.6
gcc	1.5	8.5	36.2	23.6	73.2
gzip	1.8	4.4	20.8	14.5	50.3
mcf	0.3	2.1	11.6	5.9	18.5
parser	3.3	3.7	17.4	12.5	34.8
twolf	0.2	5.2	29.2	18.5	53.3
vortex	6.5	7.5	47.9	32.7	88.4
ammp	18.9	1.8	24.8	21.1	47.1
art	26.1	5.9	14.1	11.5	19.4
equake	2.1	5.5	32.7	28.0	49.9
mesa	2.7	4.7	41.9	31.6	64.5
median		5.2	28.7	18.5	49.9

Memory increase

Program	Size (KB)	Nulgrind	Memcheck	Addrcheck	Cachegrind
bzip2	34	5.2	12.1	6.8	9.1
crafty	156	4.5	10.9	5.9	8.2
gap	140	5.6	12.7	7.3	9.7
gcc	564	5.9	13.1	7.6	9.9
gzip	30	5.5	12.6	7.2	9.4
mcf	30	5.7	13.5	7.7	9.9
parser	97	6.0	13.6	7.8	10.1
twolf	114	5.2	12.2	7.0	9.3
vortex	234	5.8	13.2	8.1	10.1
ammp	68	4.7	11.7	7.1	9.5
art	24	5.5	13.0	7.5	9.8
equake	44	5.0	12.2	7.1	9.2
mesa	69	4.8	11.2	6.7	8.9
median		5.5	12.6	7.2	9.5

Getting Accurate Backtraces

gcc main.c && strip a.out

```
==23580== Invalid write of size 1
==23580==    at 0x8048622: (within /root/dev/demo/test3)
==23580==    by 0x8048703: (within /root/dev/demo/test3)
==23580==    by 0x42015573: __libc_start_main (in /lib/tls/libc-2.3.2.so)
==23580==    by 0x80484A0: (within /root/dev/demo/test3)
==23580==  Address 0x3C03502E is 0 bytes after a block of size 10 alloc'd
==23580==    at 0x3C01E250: malloc (vg_replace_malloc.c:105)
==23580==    by 0x8048615: (within /root/dev/demo/test3)
==23580==    by 0x8048703: (within /root/dev/demo/test3)
==23580==    by 0x42015573: __libc_start_main (in /lib/tls/libc-2.3.2.so)
```

gcc main.c

```
==23587== Invalid write of size 1
==23587==    at 0x8048622: invalid_write (in /root/dev/demo/test3)
==23587==    by 0x8048703: main (in /root/dev/demo/test3)
==23587==  Address 0x3C03502E is 0 bytes after a block of size 10 alloc'd
==23587==    at 0x3C01E250: malloc (vg_replace_malloc.c:105)
==23587==    by 0x8048615: invalid_write (in /root/dev/demo/test3)
==23587==    by 0x8048703: main (in /root/dev/demo/test3)
```

gcc –g main.c

```
==23594== Invalid write of size 1
==23594==    at 0x8048622: invalid_write (demo_memcheck.c:34)
==23594==    by 0x8048703: main (demo_memcheck.c:68)
==23594==  Address 0x3C03502E is 0 bytes after a block of size 10 alloc'd
==23594==    at 0x3C01E250: malloc (vg_replace_malloc.c:105)
==23594==    by 0x8048615: invalid_write (demo_memcheck.c:31)
==23594==    by 0x8048703: main (demo_memcheck.c:68)
```

Error Suppression

```
$ valgrind --tool=memcheck  \
      --gen-suppressions=yes \
      --logfile=my.supp \
      prog <args>
```

```
{
   <gdk_set_locale>
   Memcheck:Cond
   ...
   fun:gdk_set_locale
}
```

```
{
   <libpango>
   Memcheck:Leak
   ...
   obj:/usr/*lib*/libpango*
}
```

```
$ valgrind --tool=memcheck  \
      --suppressions=my.supp.* \
      prog <args>
```

Attaching a Debugger

```
#include <malloc.h>

int main()
{
        int i;
        char  *a = malloc(3);
        for (i=0; i<4; i++)
                a[i] = 0;
}
```

```
$ valgrind --tool=memcheck \
      --db-attach=yes \
      ./prog <args>
```

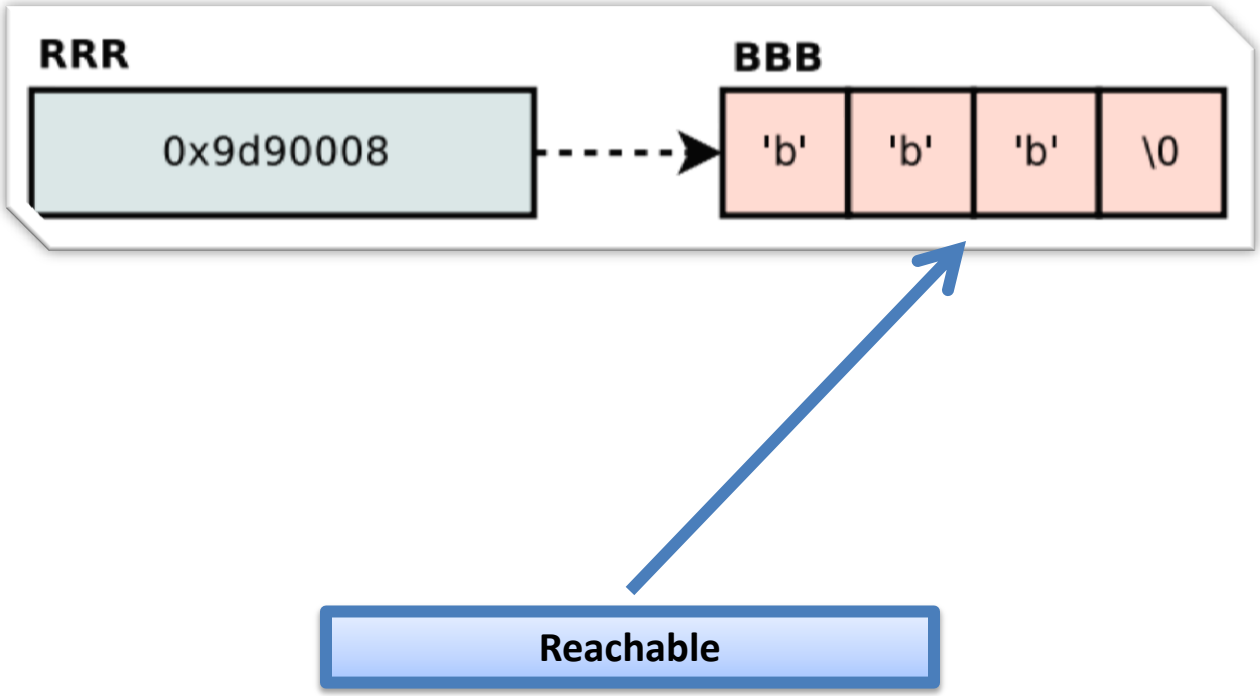
```
==3513== Memcheck, a memory error detector
==3513== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==3513== Using Valgrind-3.6.0.SVN-Debian and LibVEX;
==3513== Command: ./a.out
==3513==
==3513== Invalid write of size 1
==3513== at 0x40051C: main (in /home/ortyl/a.out)
==3513== Address 0x51b1043 is 0 bytes after a block of size 3 alloc'd
==3513== at 0x4C2815C: malloc (vg_replace_malloc.c:236)
==3513== by 0x400505: main (in /home/ortyl/a.out)
==3513==
==3513==
==3513== ---- Attach to debugger ? --- [Return/N/n/Y/y/C/c] ---- y
==3513== starting debugger with cmd: /usr/bin/gdb -nw /proc/3516/fd/1014 3516
(gdb)
```

Leak Check

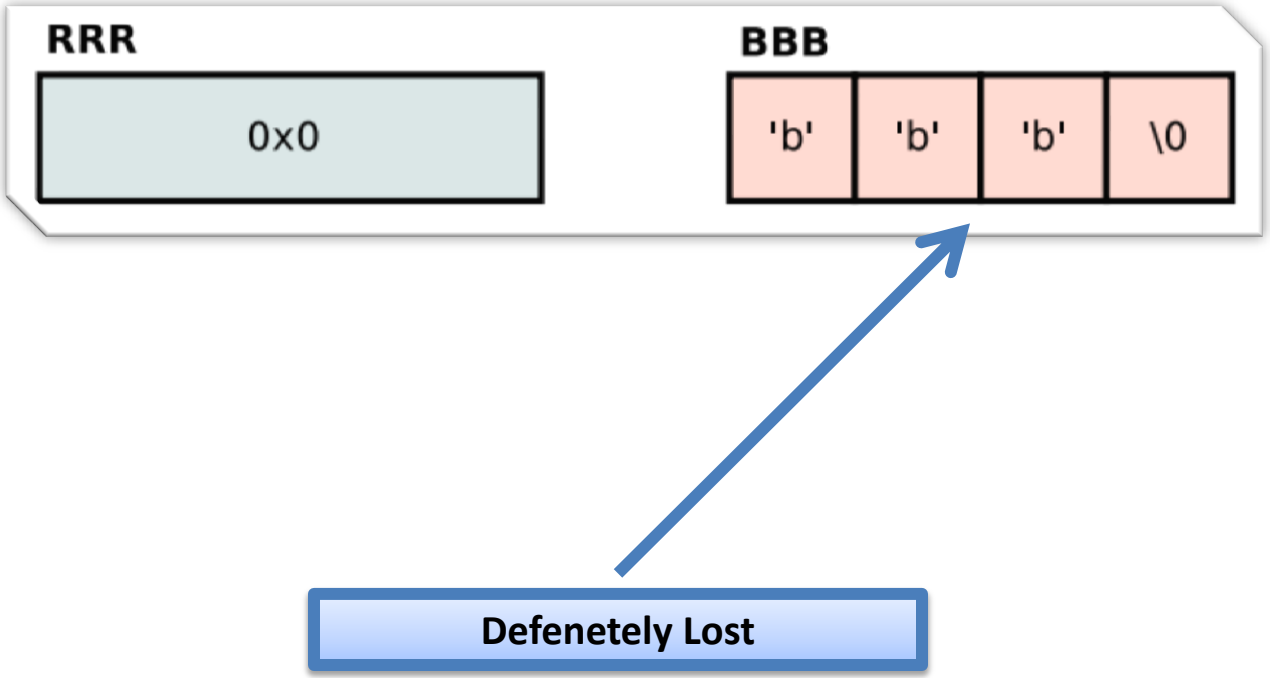
```
$ valgrind --tool=memcheck \
      --leak-check=yes \
      --show-reachable=yes \
      prog <args>
```

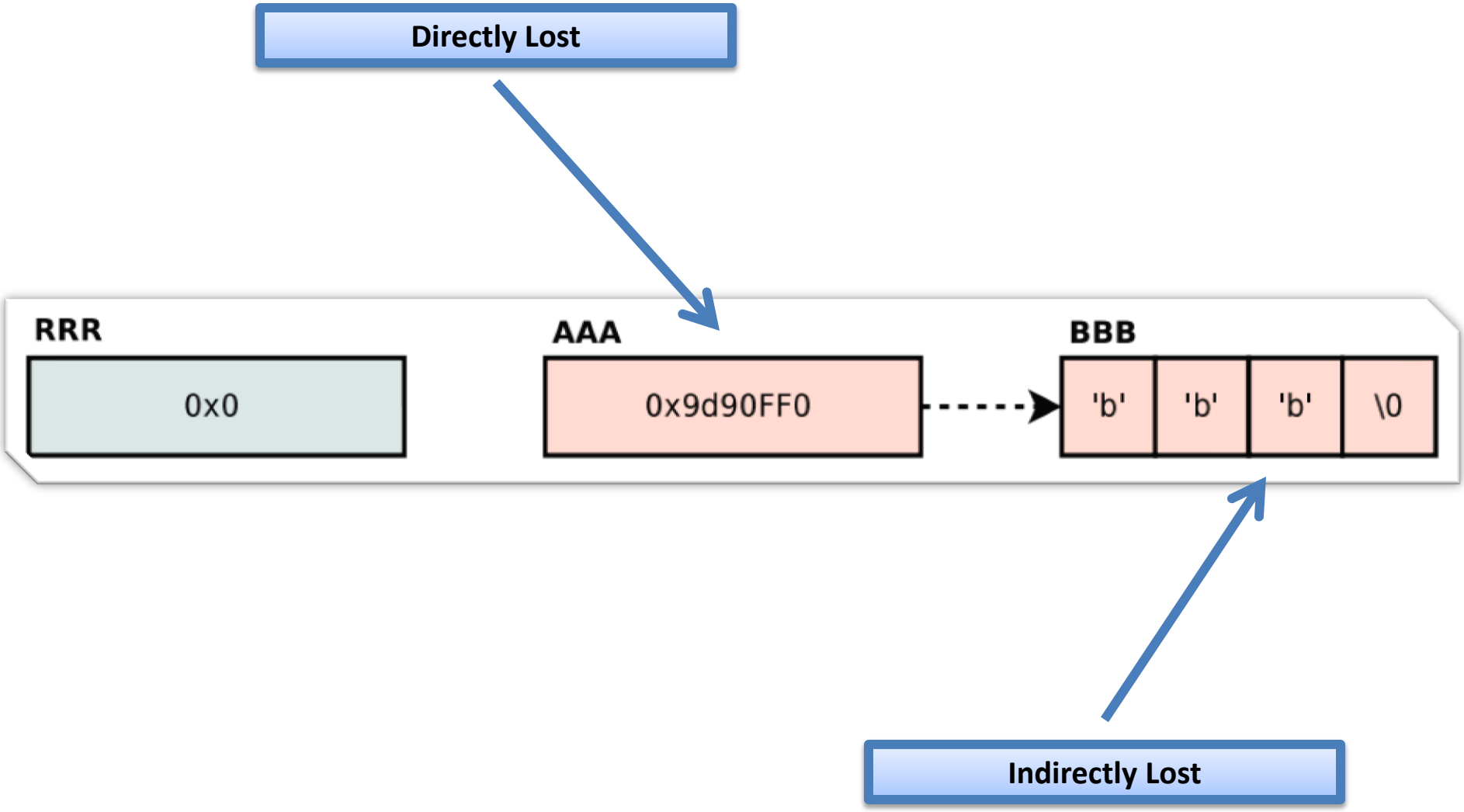
```
HEAP SUMMARY:
    in use at exit: 4 bytes in 1 blocks
  total heap usage: 1 allocs, 0 frees, 4 bytes allocated

4 bytes in 1 blocks are still reachable in loss record 1 of 1
   at 0x4024C1C: malloc (vg_replace_malloc.c:195)
   by 0x40B0CDF: strdup (strdup.c:43)
   by 0x804879B: main (in /home/aleksander/valgrind-memcheck)

LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 4 bytes in 1 blocks
        suppressed: 0 bytes in 0 blocks
```

```
void *rrr;

int main(void)
{
        rrr = strdup(”bbb”) ;
        return 0 ;
}
```

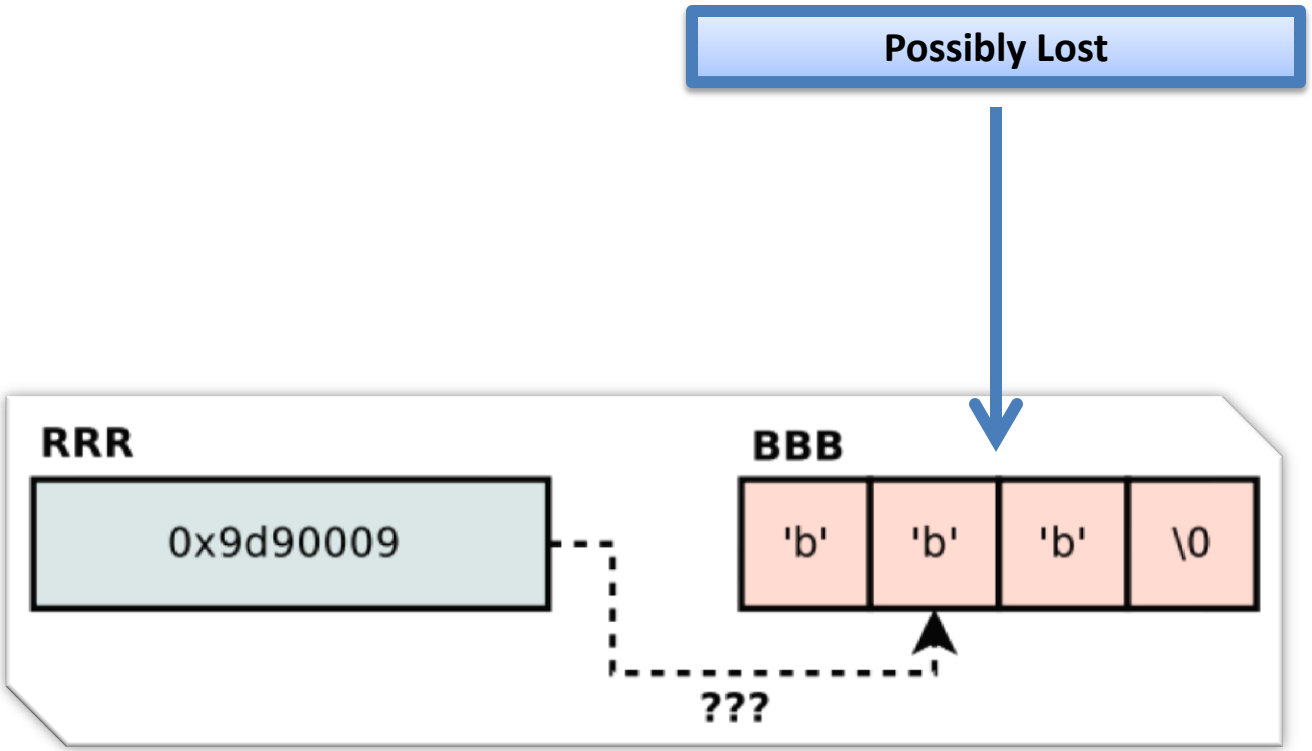

RRR
0x9d90008
BBB
'b'
'b'
'b'
\0
Reachable

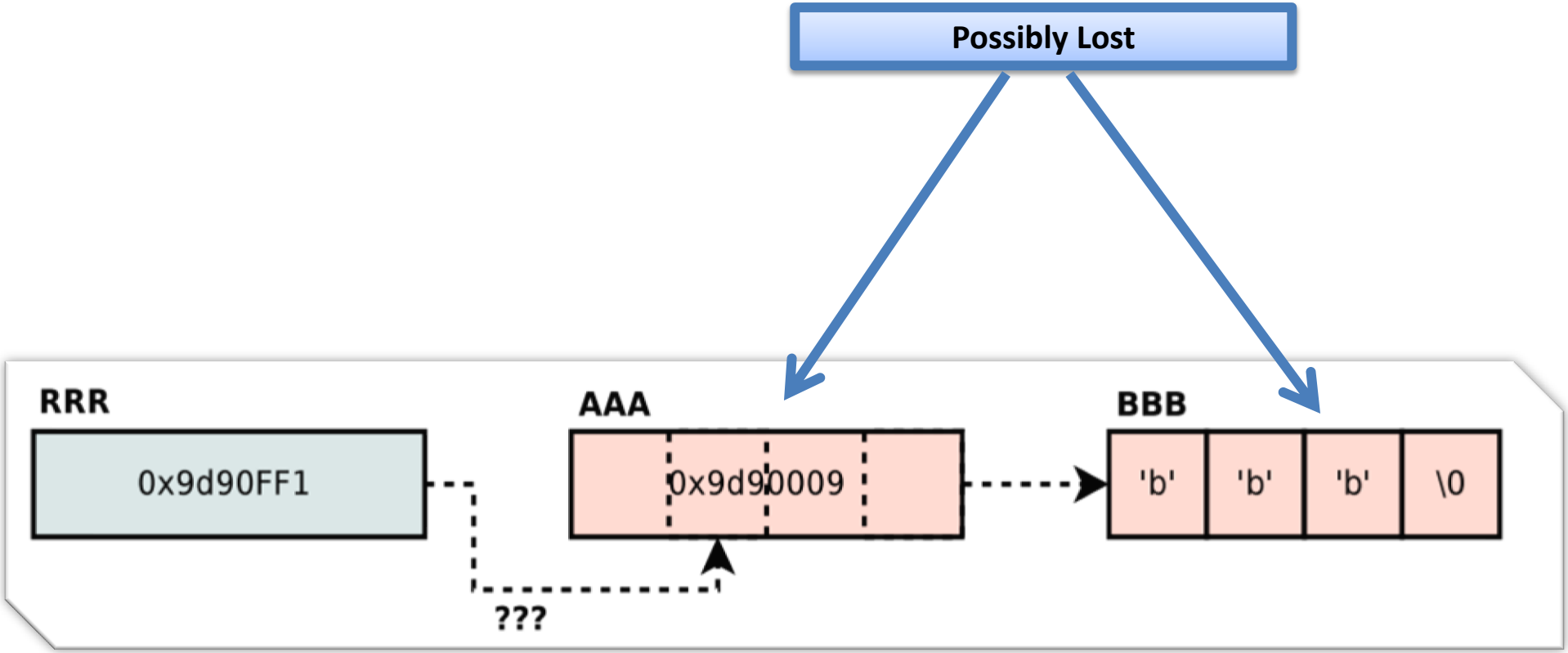
```
void *rrr;

int main(void)
{
        rrr = strdup(”bbb”) ;
        rrr = NULL;
        return 0 ;
}
```


RRR
0x0
BBB
'b'
'b'
'b'
\0
Defenetely Lost


Directly Lost
RRR
0x0
AAA
0x9d90FF0
BBB
'b'
'b'
'b'
\0
Indirectly Lost

```
void *rrr ;

int main(void)
{
        rrr = strdup(”bbb”) ;
        rrr = ((char *) rrr) + 1;
        return 0 ;
}
```


Possibly Lost
RRR
0x9d90009
BBB
'b'
'b'
'b'
\0
???

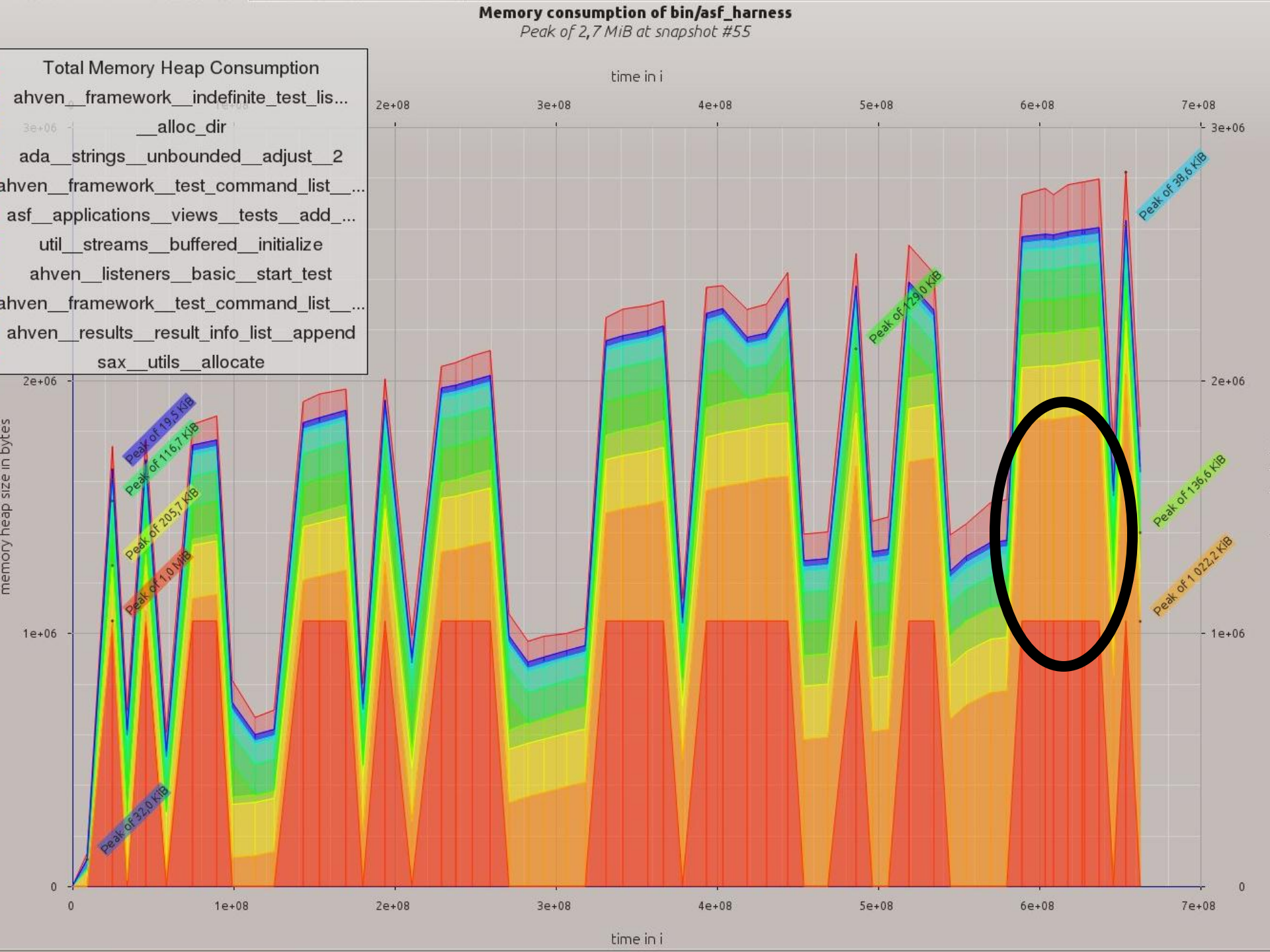

Possibly Lost
RRR
0x9d90FF1
AAA
0x9d90009
BBB
'b'
'b'
'b'
\0
???

```
$ valgrind --tool=memcheck \
      --vgdb-error=0 --vgdb=yes \
      prog <args>
```

```
(gdb) target remote | vgdb
(gdb) monitor leak_check full reachable any
==2418== 100 bytes in 1 blocks are still reachable in loss record 1 of 1
 ==2418== at 0x4006E9E: malloc (vg_replace_malloc.c:236)
==2418== by 0x804884F: main (prog.c:88)
==2418==
==2418== LEAK SUMMARY:
==2418== definitely lost: 0 bytes in 0 blocks
==2418== indirectly lost: 0 bytes in 0 blocks
==2418== possibly lost: 0 bytes in 0 blocks
==2418== still reachable: 100 bytes in 1 blocks
==2418== suppressed: 0 bytes in 0 blocks
==2418==
(gdb)
```

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.


Memory consumption of bin/asf_harness
Peak of 2,7 MiB at snapshot #55
Total Memory Heap Consumption
ahven__framework__indefinite_test_lis...
__alloc_dir
ada__strings__unbounded__adjust__2
ahven__framework__test_command_list__...
asf__applications__views__tests__add_...
util__streams__buffered__initialize
ahven__listeners__basic__start_test
ahven__framework__test_command_list__...
ahven__results__result_info_list__append
sax__utils__allocate
time in i
memory heap size in bytes
Peak of 19,5 KiB
Peak of 116,7 KiB
Peak of 205,7 KiB
Peak of 1,0 MiB
Peak of 32,0 KiB
Peak of 129,0 KiB
Peak of 38,6 KiB
Peak of 136,6 KiB
Peak of 1 022,2 KiB

Helgrind

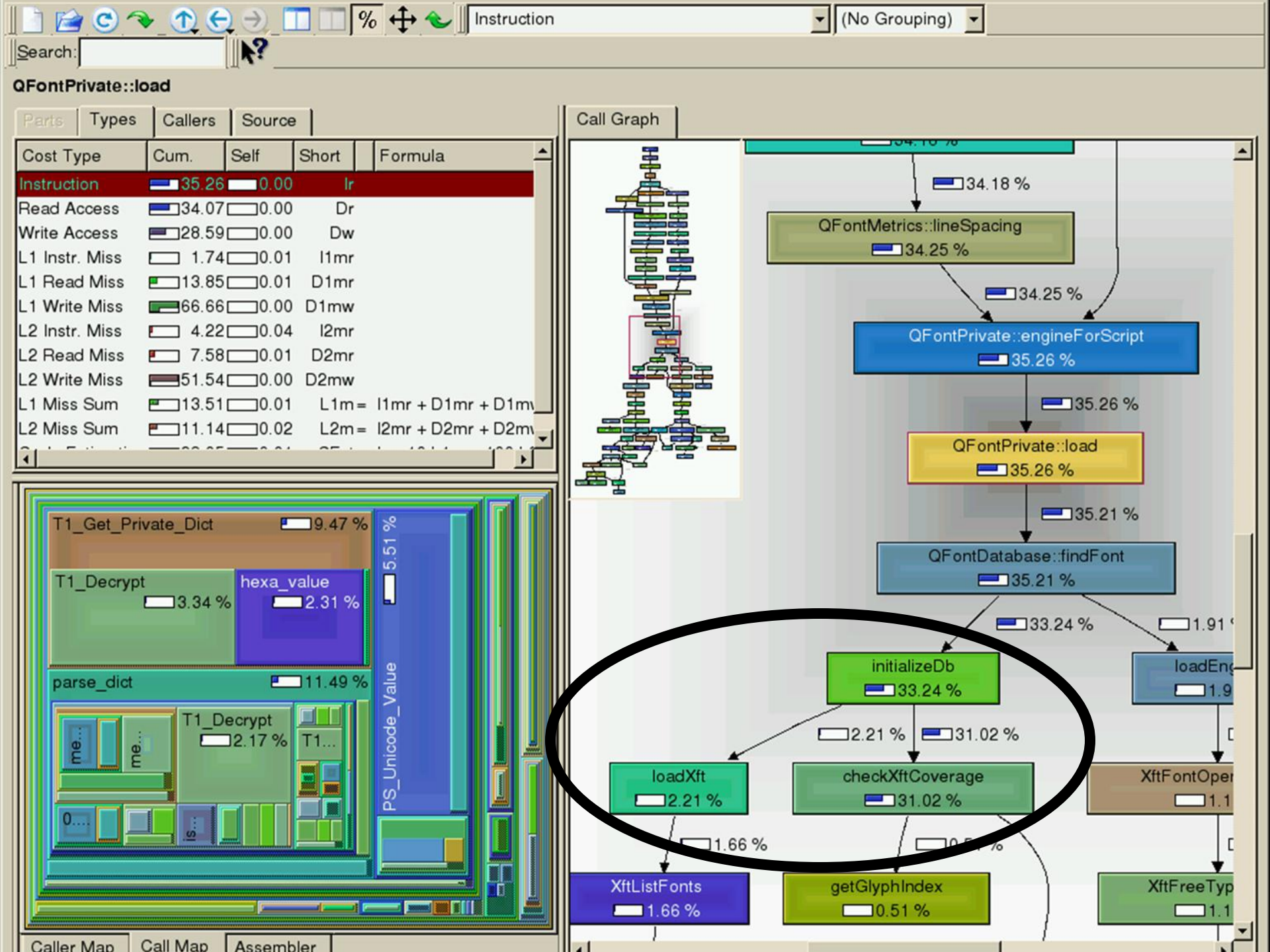
Helgrind is a thread debugger: finds data races in multithreaded programs. It looks for memory locations which are accessed by more than one (POSIX p-) thread, but for which no consistently used (pthread_mutex_) lock can be found.

```
static void *inc_shared (void *v) {
        shared++; /* un-protected access to shared */
        return 0;
}

pthread_t a, b;
pthread_create(&a, NULL, inc_shared, NULL);
pthread_create(&b, NULL, inc_shared NULL);
```

Callgrind

Callgrind is a profiling tool: it records the call history among functions in a program's run as a call-graph. Collecting number of instructions executed, their relationship to source lines, the caller/callee relationship between functions, and the numbers of such calls.

```
$ valgrind --tool=callgrind  prog <args>
$ callgrind_control --zero     # zero counters
... the application runs ...
$ callgrind_control --dump  # dump counters
$ kcachegrind cachegrind.out.<pid>
```

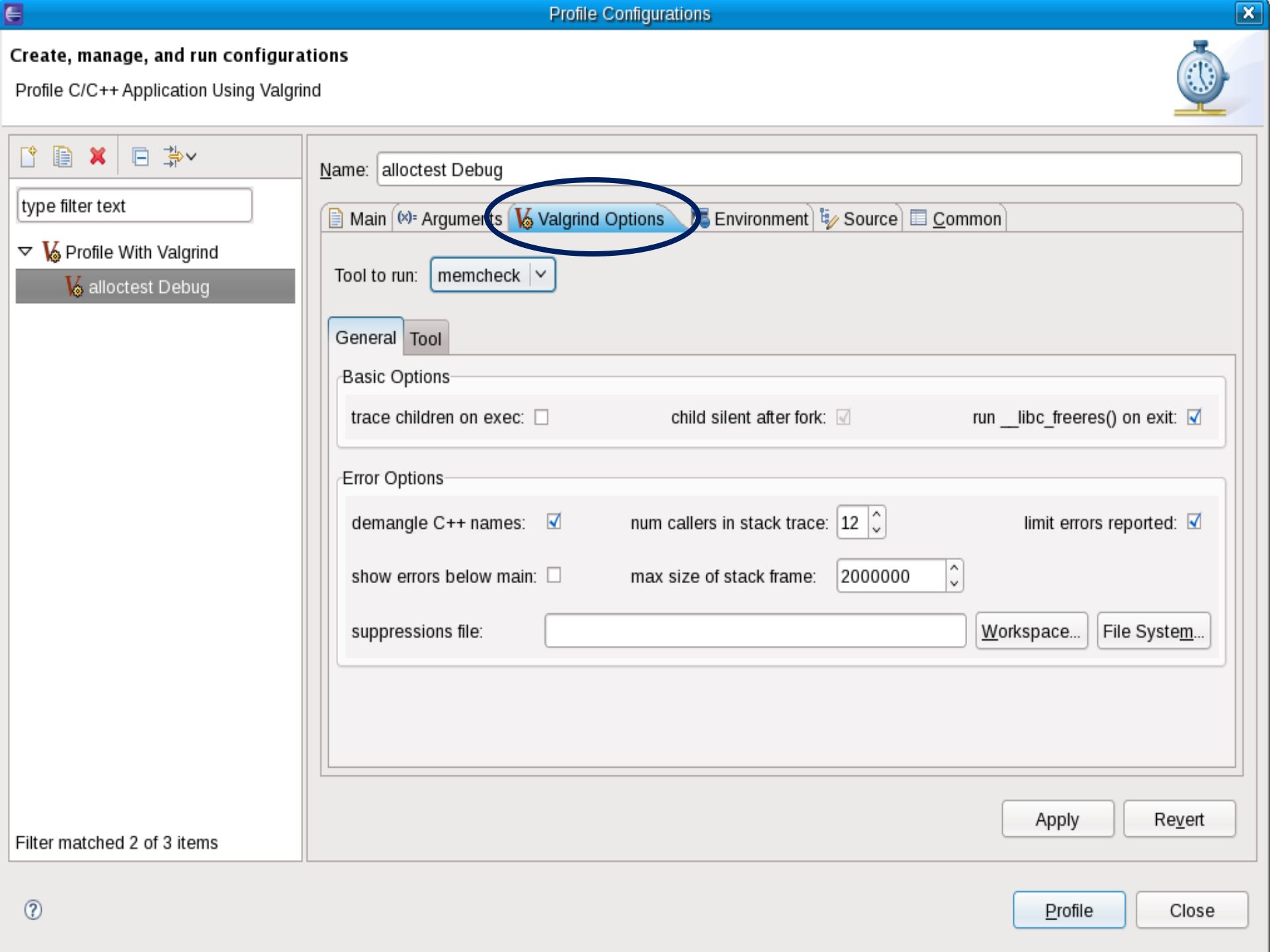

Instruction
(No Grouping)
Search:
QFontPrivate::load
Parts
Types
Callers
Source
Cost Type
Cum.
Self
Short
Formula
Instruction 35.26 0.00 Ir
Read Access 34.07 0.00 Dr
Write Access 28.59 0.00 Dw
L1 Instr. Miss 1.74 0.01 I1mr
L1 Read Miss 13.85 0.01 D1mr
L1 Write Miss 66.66 0.00 D1mw
L2 Instr. Miss 4.22 0.04 I2mr
L2 Read Miss 7.58 0.01 D2mr
L2 Write Miss 51.54 0.00 D2mw
L1 Miss Sum 13.51 0.01 L1m = I1mr + D1mr + D1mw
L2 Miss Sum 11.14 0.02 L2m = I2mr + D2mr + D2mw
T1_Get_Private_Dict 9.47 %
T1_Decrypt 3.34 %
hexa_value 2.31 %
5.51 %
parse_dict 11.49 %
T1_Decrypt 2.17 %
PS_Unicode_Value
Caller Map
Call Map
Assembler
Call Graph
34.18 %
QFontMetrics::lineSpacing 34.25 %
34.25 %
QFontPrivate::engineForScript 35.26 %
35.26 %
QFontPrivate::load 35.26 %
35.21 %
QFontDatabase::findFont 35.21 %
33.24 %
1.91
initializeDb 33.24 %
2.21 %
31.02 %
loadXft 2.21 %
checkXftCoverage 31.02 %
1.66 %
XftListFonts 1.66 %
getGlyphIndex 0.51 %

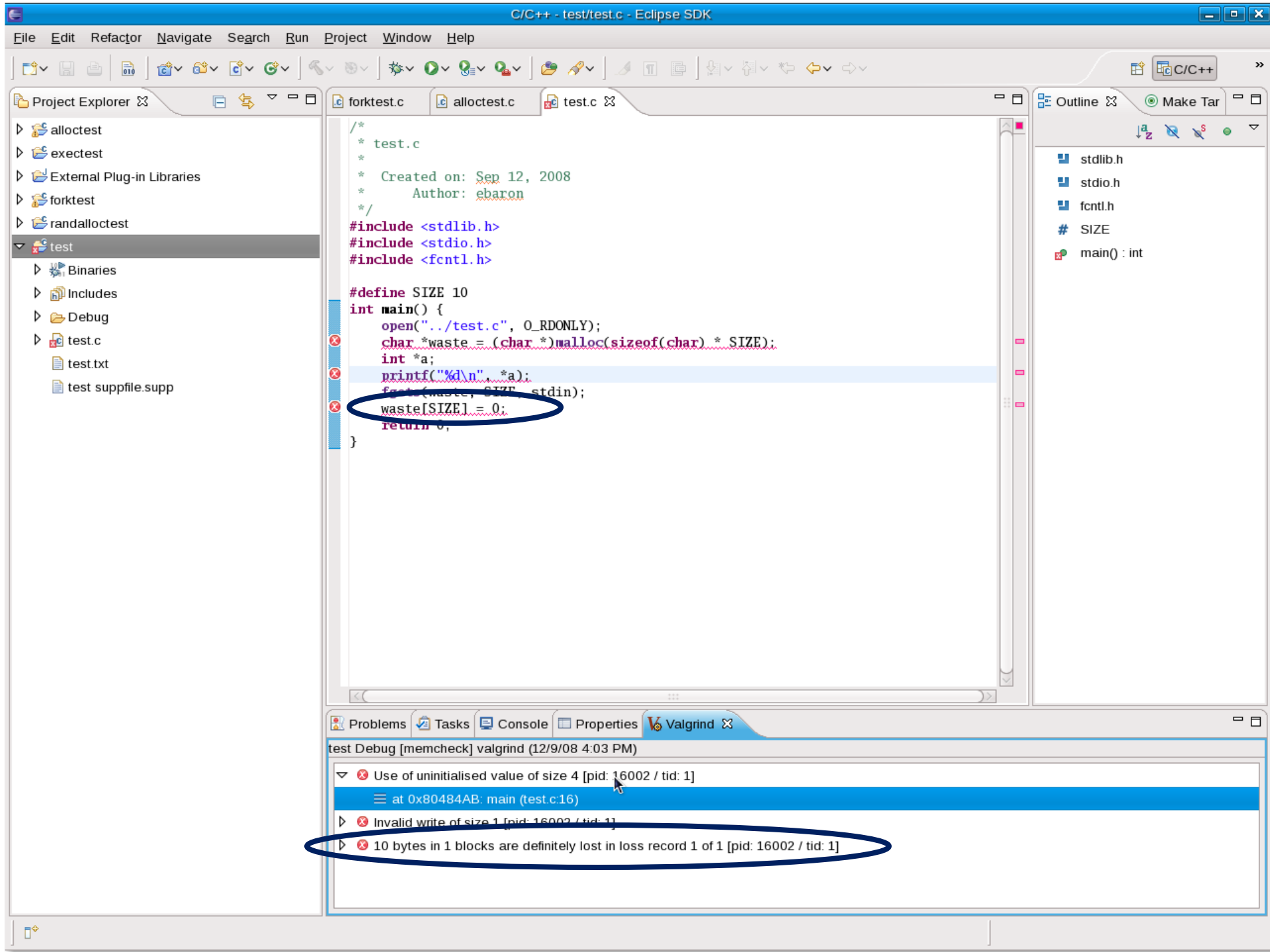
Profiling golden rules

- A human has very seldom a clue at all where in the code time is spent, therefore you must use a profiler.
- You should never optimize code unless the application feels slow as optimization always leads to code that is harder to maintain.
- Make sure to profile your application in a realistic context.

Eclipse

Ran Bittmann <ran.bittmann@sap.com>


Profile Configurations
Create, manage, and run configurations
Profile C/C++ Application Using Valgrind
type filter text
Profile With Valgrind
alloctest Debug
Filter matched 2 of 3 items
Name: alloctest Debug
Main
Arguments
Valgrind Options
Environment
Source
Common
Tool to run: memcheck
General
Tool
Basic Options
trace children on exec:
child silent after fork:
run __libc_freeres() on exit:
Error Options
demangle C++ names:
num callers in stack trace: 12
limit errors reported:
show errors below main:
max size of stack frame: 2000000
suppressions file:
Workspace...
File System...
Apply
Revert
Profile
Close


C/C++ - test/test.c - Eclipse SDK
File Edit Refactor Navigate Search Run Project Window Help
Project Explorer
alloctest
exectest
External Plug-in Libraries
forktest
randalloctest
test
Binaries
Includes
Debug
test.c
test.txt
test suppfile.supp
forktest.c
alloctest.c
test.c
/*
* test.c
*
* Created on: Sep 12, 2008
* Author: ebaron
*/
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#define SIZE 10
int main() {
open("../test.c", O_RDONLY);
char *waste = (char *)malloc(sizeof(char) * SIZE);
int *a;
printf("%d\n", *a);
waste[SIZE] = 0;
return 0;
}
Outline
Make Tar
stdlib.h
stdio.h
fcntl.h
SIZE
main() : int
Problems
Tasks
Console
Properties
Valgrind
test Debug [memcheck] valgrind (12/9/08 4:03 PM)
Use of uninitialised value of size 4 [pid: 16002 / tid: 1]
at 0x80484AB: main (test.c:16)
Invalid write of size 1 [pid: 16002 / tid: 1]
10 bytes in 1 blocks are definitely lost in loss record 1 of 1 [pid: 16002 / tid: 1]

C/C++ - Heap allocation chart for gnome-terminal - Eclipse SDK

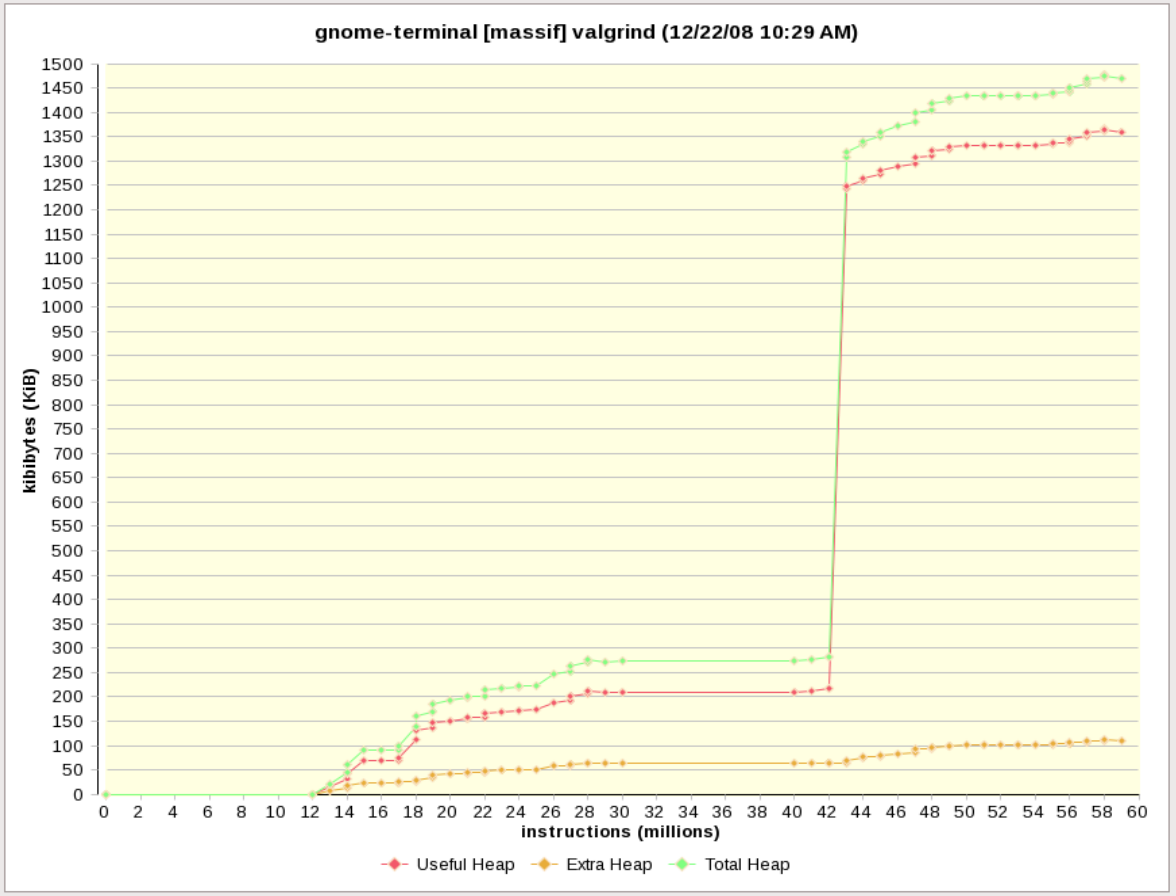
File Edit Refactor Navigate Search Run Project Window Help

C/C++ Java

Project Explorer

- alloctest
- exectest
- External Plug-in Libraries
- forktest
- gnome-terminal
  - Binaries
    - gnome-terminal - [x86/le]
  - Archives
  - Includes
  - build
  - src
- randalloctest
- test

Heap Chart - gnome-terminal



Outline Make Tar

An outline is not available.

Problems Tasks Console Properties Valgrind

gnome-terminal [massif] valgrind (12/22/08 10:29 AM)

- Snapshot 59 - 92.75% (1373487B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
- Snapshot 62 - 92.59% (1387525B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
- Snapshot 65 - 92.39% (1399350B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
  - 69.23% (1048576B) 0x4698AF8: google_breakpad::MinidumpGenerator::AllocateStack() (in /usr/lib/gtk-2.0/modules/libgnomebreakpad.so)
    - 69.23% (1048576B) 0x4698BAA: google_breakpad::MinidumpGenerator::MinidumpGenerator() (in /usr/lib/gtk-2.0/modules/libgnomebreakpad.so)
      - 69.23% (1048576B) 0x469760E: google_breakpad::ExceptionHandler::ExceptionHandler(std::string const&, bool (*)(void*), bool (*)(char const*, char const*, void*, bool)
        - 69.23% (1048576B) 0x469684D: gtk_module_init (in /usr/lib/gtk-2.0/modules/libgnomebreakpad.so)

Language Support

Language Survey

- **C: 56**
- **C+: 52**
- **Fortran: 6**
- **Java: 3**
- **asm: 3**
- **Python: 2**
- **TCL/TK: 1**
- **Objective C: 1**
- **Others: 3**

Python

```
$ cd python/dist/src

$ valgrind --tool=memcheck \
    --suppressions=Misc/valgrind-python.supp \
    ./python -E -tt ./Lib/test/regrtest.py –u \
    bsddb,network
```

Node.js

```
$ valgrind --leak-check=yes \
      node foo.js
```

Java JNI

```
$ valgrind --trace-children=yes \
        --leak-check=full \
        java -Djava.library.path=$(PWD) Foo
```

one slide to go ... ☺

- Valgrind is a robust x86-Linux memory debugger.
- Using it regularly enhances product's quality.
- When to use it?
  - All the time.
  - In automatic testing.
  - After big changes.
  - When a bug occurs.
  - When a bug is suspected.
  - Before a release.

THANK YOU!

Valgrind

www.valgrind.org

Aidan Shribman;

SAP Labs Israel

aidan.shribman@sap.com

References

- Aleksander Morgado. *Understanding Valgrind memory leak reports*. February 4, 2010
- Nicholas Nethercote and Julian Seward. *How to Shadow Every Byte of Memory Used by a Program*. 2007.
- Nicholas Nethercote and Julian Seward. *Valgrind: A Program Supervision Framework.* 2003.