Valgrind

David Gunter

High Performance Computing Division

Los Alamos National Laboratory

Valgrind is an instrumentation framework for building dynamic analysis tools.

- **Includes a set of production-quality tools**
  - Memcheck – memory error detector
  - Cachegrind – cache and branch-prediction profiler
  - Callgrind – call-graph generating extension to Cachegrind
  - Massif – heap profiler
  - Helgrind – thread error detector
- **You can also use Valgrind to build new tools.**
  - Most are in *experimental* state, others in limbo
  - http://valgrind.org/downloads/variants.html
  - You can also use Valgrind to build new tools.

Why you should use it

Dynamic memory allocation and errors associated with it are arguably the most frustrating issues to deal with. Valgrind can help:

- Automatically detect many memory management and threading bugs, saving hours of debugging time.
- Valgrind tools allow very detailed profiling to help find bottlenecks in your programs, often resulting in program speed-up.
- Ease of use: Valgrind uses dynamic binary instrumentation – no need to modify, recompile or relink your applications. Simply prefix your command line with `valgrind` and everything works.
- Valgrind works with programs written in any language.
- Valgrind works with MPI: Open-MPI and MVAPICH/MVAPICH2
- Valgrind is extensible.
- Valgrind is actively maintained and has a large user-base

http://valgrind.org/gallery/users.html

Common Errors

- Use of uninitialized memory
- Reading/writing memory after it has been freed
- Reading/writing off the end of allocated blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks – where pointers to allocated blocks become lost
- Mismatched use of malloc/new/new[] vs free/delete/delete[]

The catch?

Valgrind simulates the hardware of your target platform and runs your code inside this measurement-enhanced simulation

- **Large overhead**

  Programs run significantly more slowly under Valgrind. Depending on which tool you use, the slowdown factor can range from 5 – 100.

- **Measurements may not be absolutely accurate – but they are close!**

Memcheck: Memory Error Checker

- **Aimed primarily at Fortran, C and C++ programs.**
- **All reads and writes of memory are checked, and calls to malloc/new/free/delete are intercepted. Will report if:**
  - Accesses memory it shouldn't (not yet allocated, freed, past the end of heap blocks, inaccessible areas of the stack).
  - Uses uninitialized values in dangerous ways.
  - Leaks memory.
  - Does bad frees of heap blocks (double frees, mismatched frees).
  - Passes overlapping source and destination memory blocks to memcpy() and related functions.
- **Memcheck reports these errors as they occur, giving the source line number, and also a stack trace of the functions called to reach that line.**
- **Memcheck tracks addressability at the byte-level, and initialization of values at the bit-level. It can detect the use of single uninitialized bits, and does not report spurious errors on bitfield operations.**
- **Memcheck runs programs about 10–30× slower than normal.**

Cachegrind: Cache profiler

- **Performs detailed simulation of I, I1, L2, and D caches**
- **Can accurately pinpoint the sources of cache misses in your code. It identifies for each line of source code the number of:**
  - Cache misses
  - Memory references
  - Instructions executed
- **Provides per-function, per-module and whole-program summaries.**
- **Useful for programs written in any language.**
- **Performance hit is about a 20—100× slowdown.**

Callgrind: Callgraphs + Cachegrind Info

- Is an extension that provides all the info Cachegrind yields
- Provides callgraph information.
- Kcachegrind is a separately available tool for visualisation for both Callgrind and Cachegrind output data
- Created by Josef Weidendorfer (Weidendorfer@in.tum.de) but included with the basic Valgrind distribution.

Massif: Heap Profiler

- **Performs detailed profiling by taking regular snapshots of a program's heap.**
- **Produces a graph showing heap usage over time**
  - including information about which parts of the program are responsible for most memory allocations
  - The graph is supplemented by a text or HTML file that includes more information for determining where the most memory is being allocated.
- **Massif runs programs about 20× slower than normal.**

Helgrind: Thread Debugger

- **Finds data races in multithreaded programs.**
- **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.**
  - Indicative of missing synchronization between threads, and could cause hard-to-find timing-dependent problems.
- **It is useful for any program that uses pthreads.**
- **Experimental tool, developer welcomes feedback**

Valgrind Availability

Platform	Version	Usage	Documentation	POC
LANL/Lobo	3.5.0	module load friendly-testing module load valgrind-ompi/3.5.0 -or- module load valgrind-mvapich/3.5.0	Valgrind website man valgrind	David Gunter dog@lanl.gov
LANL/RR-Dev	3.5.0	module load friendly-testing valgrind/ 3.2.0	Valgrind website man valgrind	David Gunter dog@lanl.gov
LANL/YR	3.2.0	module load hpc-tool valgrind/3.2.0	Valgrind website man valgrind	David Gunter dog@lanl.gov
SNL	TBD	N/A	Valgrind website man valgrind	TBD
LLNL	TBD	TBD	TBD	TBD

Usage Case: Memcheck – Uninitialized Memory

```
1  #include <stdlib.h>
2  int main() {
3
4     int p, t;
5
6     if (p == 5) /* Error */
7        t = p + 1;
8     return 0;
9  }
```

`p` is uninitialized and may contain garbage, resulting in an error if used to determine branch-outcome or memory address (ex: a[p] = y)

```
$ gcc -g -o uninit_memory uninit_memory.c
$ uninit_memory
$ valgrind --tool=memcheck uninit_memory
==18385== Memcheck, a memory error detector
==18385== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==18385== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==18385== Command: uninit_memory
==18385==
==18385== Conditional jump or move depends on uninitialised value(s)
==18385==    at 0x400450: main (uninit_memory.c:6)
==18385==
==18385==
==18385== HEAP SUMMARY:
==18385==     in use at exit: 0 bytes in 0 blocks
==18385==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==18385==
==18385== All heap blocks were freed -- no leaks are possible
==18385==
==18385== For counts of detected and suppressed errors, rerun with: -v
==18385== Use --track-origins=yes to see where uninitialised values come
from
==18385== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 7 from 7)
```

Usage Case: Memcheck – Invalid Read/Write

```c
#include <stdlib.h>
int main() {

   int *p, i, a;

   p = malloc(10*sizeof
  (int));
   p[11] = 1; /* write */
   a = p[11]; /* read  */
   free(p);
    return 0;
 }
```

```
$ gcc -g -o invalid_read_write invalid_read_write.c
$ invalid_read_write
invalid_read_write
*** glibc detected *** invalid_read_write: free(): invalid next size
(fast): 0x000000001a2b8010 ***
======= Backtrace: =========
/lib64/libc.so.6[0x2b68eb106ce2]
/lib64/libc.so.6(cfree+0x8c)[0x2b68eb10a90c]
invalid_read_write[0x400512]
/lib64/libc.so.6(__libc_start_main+0xf4)[0x2b68eb0b2974]
invalid_read_write[0x400429]
======= Memory map: ========
00400000-00401000 r-xp 00000000 00:1d 5484383                    invalid_read_write
00600000-00601000 rw-p 00000000 00:1d 5484383                    invalid_read_write
1a2b8000-1a2d9000 rw-p 1a2b8000 00:00 0                          [heap]
2b68eae78000-2b68eae94000 r-xp 00000000 08:02 137003070          /lib64/ld-2.5.so
2b68eae94000-2b68eae95000 rw-p 2b68eae94000 00:00 0
2b68eaeab000-2b68eaeac000 rw-p 2b68eaeab000 00:00 0
2b68eb093000-2b68eb094000 r--p 0001b000 08:02 137003070          /lib64/ld-2.5.so
2b68eb094000-2b68eb095000 rw-p 0001c000 08:02 137003070          /lib64/ld-2.5.so
2b68eb095000-2b68eb1e1000 r-xp 00000000 08:02 137003018          /lib64/libc-2.5.so
2b68eb1e1000-2b68eb3e1000 ---p 0014c000 08:02 137003018          /lib64/libc-2.5.so
2b68eb3e1000-2b68eb3e5000 r--p 0014c000 08:02 137003018          /lib64/libc-2.5.so
2b68eb3e5000-2b68eb3e6000 rw-p 00150000 08:02 137003018          /lib64/libc-2.5.so
2b68eb3e6000-2b68eb3ec000 rw-p 2b68eb3e6000 00:00 0
2b68eb3ec000-2b68eb3f9000 r-xp 00000000 08:02 137003048          /lib64/libgcc_s-4.1.2-20080825.so.1
2b68eb3f9000-2b68eb5f9000 ---p 0000d000 08:02 137003048          /lib64/libgcc_s-4.1.2-20080825.so.1
2b68eb5f9000-2b68eb5fa000 rw-p 0000d000 08:02 137003048          /lib64/libgcc_s-4.1.2-20080825.so.1
2b68ec000000-2b68ec021000 rw-p 2b68ec000000 00:00 0
2b68ec021000-2b68f0000000 ---p 2b68ec021000 00:00 0
7fffdb5b5000-7fffdb5ca000 rw-p 7ffffffea000 00:00 0              [stack]
ffffffffff600000-ffffffffffe00000 ---p 00000000 00:00 0          [vdso]
Abort
```

Usage Case: Memcheck – Invalid Read/Write (cont’d)

```
1   #include <stdlib.h>
2   int main() {
3
4      int *p, i, a;
5
6      p = malloc(10*sizeof(int));
7      p[11] = 1; /* write */
8      a = p[11]; /* read  */
9      free(p);
10      return 0;
11  }
```

Attempting to read/write from address `(p+sizeof(int)*11)` which has not been allocated.

```
$ valgrind --tool=memcheck invalid_read_write
==19490== Memcheck, a memory error detector
==19490== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==19490== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==19490== Command: invalid_read_write
==19490==
==19490== Invalid write of size 4
==19490==    at 0x4004F6: main (invalid_read_write.c:7)
==19490==  Address 0x517b06c is 4 bytes after a block of size 40 alloc'd
==19490==    at 0x4C20E27: malloc (vg_replace_malloc.c:195)
==19490==    by 0x4004E9: main (invalid_read_write.c:6)
==19490==
==19490== Invalid read of size 4
==19490==    at 0x400504: main (invalid_read_write.c:8)
==19490==  Address 0x517b06c is 4 bytes after a block of size 40 alloc'd
==19490==    at 0x4C20E27: malloc (vg_replace_malloc.c:195)
==19490==    by 0x4004E9: main (invalid_read_write.c:6)
==19490==
==19490==
==19490== HEAP SUMMARY:
==19490==     in use at exit: 0 bytes in 0 blocks
==19490==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==19490==
==19490== All heap blocks were freed -- no leaks are possible
==19490==
==19490== For counts of detected and suppressed errors, rerun with: -v
==19490== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 7 from 7)
```

Usage Case: Memcheck – Invalid Free

```
1  #include <stdlib.h>
2
3  int main() {
4
5  int *p, i;
6  p = malloc(10*sizeof int));
7    for(i = 0;i < 10;i++)
8         p[i] = i;
9    free(p);
10   free(p);  /* Error */
11   return 0;
12 }
```

```
$ valgrind --tool=memcheck invalid_free
==26808== Memcheck, a memory error detector
==26808== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==26808== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright
info
==26808== Command: invalid_free
==26808==
==26808== Invalid free() / delete / delete[]
==26808==    at 0x4C20A3C: free (vg_replace_malloc.c:325)
==26808==    by 0x400527: main (invalid_free.c:10)
==26808==  Address 0x517b040 is 0 bytes inside a block of size 40 free'd
==26808==    at 0x4C20A3C: free (vg_replace_malloc.c:325)
==26808==    by 0x40051E: main (invalid_free.c:9)
==26808==
==26808==
==26808== HEAP SUMMARY:
==26808==     in use at exit: 0 bytes in 0 blocks
==26808==   total heap usage: 1 allocs, 2 frees, 40 bytes allocated
==26808==
==26808== All heap blocks were freed -- no leaks are possible
==26808==
==26808== For counts of detected and suppressed errors, rerun with: -v
==26808== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 7 from 7)
```

Valgrind checks the address passed to the `free()` call and sees that it has already been freed.

Usage Case: Memcheck – Invalid Call Parameter

```
#include <stdlib.h>
#include <unistd.h>

int main() {
  int *p;

  p = malloc(10);
  read(0, p, 100); /* err */
  free(p);
   return 0;
 }
```

`read()` tries to read 100 bytes from stdin and place the results in `p` but the bytes after the firs 10 are unaddressable.

```
$ valgrind --tool=memcheck invalid_call_param
==27095== Memcheck, a memory error detector
==27095== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==27095== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==27095== Command: invalid_call_param
==27095==
==27095== Syscall param read(buf) points to unaddressable byte(s)
==27095==    at 0x4EEA620: __read_nocancel (in /lib64/libc-2.5.so)
==27095==    by 0x400550: main (invalid_call_param.c:8)
==27095==  Address 0x517b04a is 0 bytes after a block of size 10 alloc'd
==27095==    at 0x4C20E27: malloc (vg_replace_malloc.c:195)
==27095==    by 0x400539: main (invalid_call_param.c:7)
==27095==
12345678901234567890
==27095==
==27095== HEAP SUMMARY:
==27095==     in use at exit: 0 bytes in 0 blocks
==27095==   total heap usage: 1 allocs, 1 frees, 10 bytes allocated
==27095==
==27095== All heap blocks were freed -- no leaks are possible
==27095==
==27095== For counts of detected and suppressed errors, rerun with: -v
==27095== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 7 from 7)
```

Usage Case: Memcheck – Leak Detection

```c
#include <stdlib.h>

int main() {
   int *p, i;
   p = malloc(5*sizeof(int));
   for(i = 0;i < 5;i++)
      p[i] = i;
   return 0;
}
```

```
$ valgrind --leak-check=yes --tool=memcheck memory_leak
==27664== Memcheck, a memory error detector
==27664== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==27664== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==27664== Command: memory_leak
==27664==
==27664==
==27664== HEAP SUMMARY:
==27664==     in use at exit: 20 bytes in 1 blocks
==27664==   total heap usage: 1 allocs, 0 frees, 20 bytes allocated
==27664==
==27664== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==27664==    at 0x4C20E27: malloc (vg_replace_malloc.c:195)
==27664==    by 0x4004A9: main (memory_leak.c:5)
==27664==
==27664== LEAK SUMMARY:
==27664==    definitely lost: 20 bytes in 1 blocks
==27664==    indirectly lost: 0 bytes in 0 blocks
==27664==      possibly lost: 0 bytes in 0 blocks
==27664==    still reachable: 0 bytes in 0 blocks
==27664==         suppressed: 0 bytes in 0 blocks
==27664==
==27664== For counts of detected and suppressed errors, rerun with: -v
==27664== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 7 from 7)
```

20 unfreed blocks at routine exit – memory leak.

Usage Case: Cachegrind

```c
#include <stdio.h>
#define N 1000

double array_sum(double a[][N]);

int main(int argc, char **argv) {

   double a[N][N];
   int i,j;

    for (i=0;i<N;i++) {
       for (j=0;j<N;j++) {
          a[i][j] = 0.01;
       }
    }

    printf("sum = %10.3f\n", array_sum(a) );

    return 0;
 }

double array_sum(double a[][N]) {

   int i,j;
   double s;

   s=0;
   for (i=0;i<N;i++)
      for (j=0;j<N;j++)
         s += a[i][j];

   return s;
}
```

Fill 2D Array

Read 2D Array

- Array size is 1,000 x 1000 x 8 bytes = 8Mb
- 64kB L1i and 64kB L1d
- 512kB L2

Usage Case: Cachegrind (cont'd)

```
$ gcc -O2 -g -o loops-fast loops-fast.c
$ valgrind --tool=cachegrind ./loops-fast
```

```
==7796== Cachegrind, a cache and branch-prediction profiler
==7796== Copyright (C) 2002-2009, and GNU GPL'd, by Nicholas Nethercote et al.
==7796== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==7796== Command: ./loops-fast
==7796==
sum =  10000.000
==7796==
==7796== I   refs:      10,151,445
==7796== I1  misses:           845
==7796== L2i misses:           842
==7796== I1  miss rate:       0.00%
==7796== L2i miss rate:       0.00%
==7796==
==7796== D   refs:       2,053,226  (1,038,866 rd   + 1,014,360 wr)
==7796== D1  misses:       251,804  (  126,329 rd   +   125,475 wr)
==7796== L2d misses:       251,679  (  126,213 rd   +   125,466 wr)
==7796== D1  miss rate:       12.2% (     12.1%     +      12.3%  )
==7796== L2d miss rate:       12.2% (     12.1%     +      12.3%  )
==7796==
==7796== L2 refs:          252,649  (  127,174 rd   +   125,475 wr)
==7796== L2 misses:        252,521  (  127,055 rd   +   125,466 wr)
==7796== L2 miss rate:         2.0% (      1.1%     +      12.3%  )
```

Usage Case: Callgrind (extension to cachegrind)

```
valgrind --tool=callgrind --simulate-cache=yes ./loops-fast
==29254== Callgrind, a call-graph generating cache profiler
==29254== Copyright (C) 2002-2009, and GNU GPL'd, by Josef Weidendorfer et al.
==29254== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==29254== Command: ./loops-fast
==29254==
==29254== For interactive control, run 'callgrind_control -h'.
sum =  10000.000
==29254==
==29254== Events    : Ir Dr Dw I1mr D1mr D1mw I2mr D2mr D2mw
==29254== Collected : 10151442 1038367 1014859 845 126321 125483 842 126206 125473
==29254==
==29254== I   refs:      10,151,442
==29254== I1  misses:           845
==29254== L2i misses:           842
==29254== I1  miss rate:        0.0%
==29254== L2i miss rate:        0.0%
==29254==
==29254== D   refs:       2,053,226  (1,038,367 rd + 1,014,859 wr)
==29254== D1  misses:       251,804  (  126,321 rd +   125,483 wr)
==29254== L2d misses:       251,679  (  126,206 rd +   125,473 wr)
==29254== D1  miss rate:       12.2% (     12.1%   +      12.3%  )
==29254== L2d miss rate:       12.2% (     12.1%   +      12.3%  )
==29254==
==29254== L2 refs:          252,649  (  127,166 rd +   125,483 wr)
==29254== L2 misses:        252,521  (  127,048 rd +   125,473 wr)
==29254== L2 miss rate:         2.0% (      1.1%   +      12.3%  )
```

Usage Case: Callgrind (cont'd)

```
==29254== Events    : Ir       Dr      Dw      I1mr D1mr   D1mw   I2mr D2mr   D2mw
==29254== Collected : 10151442 1038367 1014859 845  126321 125483 842  126206 125473
```

- `Ir` = number if instructions executed
- `Dr` = number of memory (data) reads
- `Dw` = number of memory (data) writes
- `I1mr` = I1 cache read misses
- `D1mr` = D1 cache read misses
- `D1mw` = D1 cache write misses
- `I2mr` = I2 cache read misses
- `D2mr` = D2 cache read misses
- `D2mw` = D2 cache write misses

Usage Case: Callgrind (cont'd)

- Cachegrind saves output to a file '`callgrind.out.<pid>` by default
- Use callgrind_annotate to parse this file for detailed information

```
$ callgrind_annotate callgrind.out.29254
--------------------------------------------------------------------------------
Profile data file 'callgrind.out.29254' (creator: callgrind-3.5.0)
--------------------------------------------------------------------------------
I1 cache: 65536 B, 64 B, 2-way associative
D1 cache: 65536 B, 64 B, 2-way associative
L2 cache: 524288 B, 64 B, 8-way associative
Timerange: Basic block 0 - 2028047
Trigger: Program termination
Profiled target:  ./loops-fast (PID 29254, part 1)
Events recorded:  Ir Dr Dw I1mr D1mr D1mw I2mr D2mr D2mw
Events shown:     Ir Dr Dw I1mr D1mr D1mw I2mr D2mr D2mw
Event sort order: Ir Dr Dw I1mr D1mr D1mw I2mr D2mr D2mw
Thresholds:       99 0 0 0 0 0 0 0 0
Include dirs:
User annotated:
Auto-annotation:  off
```

Usage Case: Callgrind (cont'd)

```
--------------------------------------------------------------------------------
        Ir        Dr        Dw I1mr    D1mr    D1mw I2mr    D2mr    D2mw
--------------------------------------------------------------------------------
10,151,445 1,038,367 1,014,859  845 126,321 125,483  842 126,206 125,473  PROGRAM TOTALS

--------------------------------------------------------------------------------
       Ir        Dr        Dw I1mr    D1mr    D1mw I2mr    D2mr    D2mw  file:function
--------------------------------------------------------------------------------
6,007,017         4 1,000,004    2       1 125,002    2       1 125,002  loops-fast.c:main
4,005,003 1,000,001         0    0 125,001       0    0 125,001       .  loops-fast.c:array_sum
   28,082     9,818     4,106   13     182       5   13     156       4  /…/glibc-2.5-20061008T1257/
     elf/do-lookup.h:do_lookup_x [/lib64/ld-2.5.so]
   19,764     3,860     2,472   13      96       9   13      91       8  /…/glibc-2.5-20061008T1257/
     elf/dl-lookup.c:_dl_lookup_symbol_x [/lib64/ld-2.5.so]
```

Callgrind can be used to find performance problems that are not related to CPU cache

- What lines eat up most instructions (CPU cycles, time)
- What system/math/lib functions are called and what is their cost?

Usage Case: Massif

```c
#include <stdlib.h>

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;
}
```

```
valgrind --tool=massif massif_demo
==11496== Massif, a heap profiler
==11496== Copyright (C) 2003-2009, and GNU GPL'd,
     by Nicholas Nethercote
==11496== Using Valgrind-3.5.0 and LibVEX; rerun
     with -h for copyright info
==11496== Command: massif_demo
==11496==
==11496==
$ ms_print massif.out.11496
```

- Massif outputs to file
- ms_print generates
  - graph showing mem consumption over time
  - Detailed info about allocation sites including peak allocation points

Usage Case: Massif (cont'd)

```
--------------------------------------------------------------------------------
Command:            massif_demo
Massif arguments:   (none)
ms_print arguments: massif.out.11496
--------------------------------------------------------------------------------


    KB
19.71^                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
   0 +----------------------------------------------------------------------->ki
     0                                                                   155.4

Number of snapshots: 25
 Detailed snapshots: [9, 14 (peak), 24]
```

Mostly empty – Massif uses "instructions executed" as unit of time. For this short run program most of these are the loading linking of the program. `main()`, `g()`, and `f()` are only run at the end.

Usage Case: Massif (cont'd)

```
$ valgrind --tool=massif --time-unit=B massif_demo
$ ms_print massif.out.29949
--------------------------------------------------------------------------------
Command:            massif_demo
Massif arguments:   --time-unit=B
ms_print arguments: massif.out.29949
--------------------------------------------------------------------------------


    KB
19.71^                                               ###
     |                                               #
     |                                               #  ::
     |                                               #  : :::
     |                                      :::::::::#  : :  ::
     |                                      :        #  : :  : ::
     |                                      :        #  : :  : : :::
     |                                      :        #  : :  : : :  ::
     |                            :::::::::::        #  : :  : : :  : :::
     |                            :         :        #  : :  : : :  : :  ::
     |                        :::::         :        #  : :  : : :  : :  : ::
     |                     @@@:   :         :        #  : :  : : :  : :  : : @
     |                   ::@  :   :         :        #  : :  : : :  : :  : : @
     |                ::::  @  :   :         :        #  : :  : : :  : :  : : @
     |              :::  : @  :   :         :        #  : :  : : :  : :  : : @
     |            ::: :  : @  :   :         :        #  : :  : : :  : :  : : @
     |         :::: : :  : @  :   :         :        #  : :  : : :  : :  : : @
     |       :::  : : :  : @  :   :         :        #  : :  : : :  : :  : : @
     |    :::: :  : : :  : @  :   :         :        #  : :  : : :  : :  : : @
     |  :::  : :  : : :  : @  :   :         :        #  : :  : : :  : :  : : @
   0 +----------------------------------------------------------------------->KB
     0                                                                   29.63

Number of snapshots: 25
 Detailed snapshots: [9, 14 (peak), 24]
```

`--time-unit=B` sets "time" units to be bytes alloc/dealloc.

Usage Case: Massif (cont'd)

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0                0             0            0
  1          1,016            1,016            1,000            16            0
  2          2,032            2,032            2,000            32            0
  3          3,048            3,048            3,000            48            0
  4          4,064            4,064            4,000            64            0
  5          5,080            5,080            5,000            80            0
  6          6,096            6,096            6,000            96            0
  7          7,112            7,112            7,000           112            0
  8          8,128            8,128            8,000           128            0
```

Detailed information by snapshot. Each shows

- Snapshot number
- "time" taken (bytes in this example)
- Total memory consumption
- Number of useful heap bytes allocated at that point
- Number of extra heap bytes allocated (admin bytes and bytes due to round-up/alignment)
- Size of the stack (stack profiling off by default for performance, thus the zeroes here)

Usage Case: Massif (cont'd)

```
 9          9,144           9,144           9,000          144           0
98.43% (9,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->98.43% (9,000B) 0x40051A: main (massif_demo.c:18)
```

Snapshot 9 contains further detail. It gives an allocation tree (read from top down)

- First line indicates all heap allocation functions (malloc/new/new[]) and the percentage of allocations using them
- Second line indicates where these allocations were called: At this point in the program execution all allocations have been done from line 18, `a[i] = malloc(1000);`

Usage Case: Massif (cont'd)

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 10         10,160           10,160           10,000           160            0
 11         12,168           12,168           12,000           168            0
 12         16,176           16,176           16,000           176            0
 13         20,184           20,184           20,000           184            0
 14         20,184           20,184           20,000           184            0
99.09% (20,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->49.54% (10,000B) 0x40051A: main (massif_demo.c:18)
|
->39.64% (8,000B) 0x4004E4: g (massif_demo.c:4)
| ->19.82% (4,000B) 0x4004F9: f (massif_demo.c:9)
| | ->19.82% (4,000B) 0x400534: main (massif_demo.c:21)
| |
| ->19.82% (4,000B) 0x400539: main (massif_demo.c:23)
|
->09.91% (2,000B) 0x4004F4: f (massif_demo.c:8)
  ->09.91% (2,000B) 0x400534: main (massif_demo.c:21)
```

Next detailed snapshot at 14, point of maximum allocation peak

- Allocations are occurring in 3 areas of the code with percentage numbers for each
- Of the 8,000B requested in line 4, half were due to calls in line 9 while the other half were due to the calls in line 21

Usage Case: Massif (cont'd)

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 15         21,200           19,168           19,000           168            0
 16         22,216           18,152           18,000           152            0
 17         23,232           17,136           17,000           136            0
 18         24,248           16,120           16,000           120            0
 19         25,264           15,104           15,000           104            0
 20         26,280           14,088           14,000            88            0
 21         27,296           13,072           13,000            72            0
 22         28,312           12,056           12,000            56            0
 23         29,328           11,040           11,000            40            0
 24         30,344           10,024           10,000            24            0
99.76% (10,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->79.81% (8,000B) 0x4004E4: g (massif_demo.c:4)
| ->39.90% (4,000B) 0x4004F9: f (massif_demo.c:9)
| | ->39.90% (4,000B) 0x400534: main (massif_demo.c:21)
| |
| ->39.90% (4,000B) 0x400539: main (massif_demo.c:23)
|
->19.95% (2,000B) 0x4004F4: f (massif_demo.c:8)
| ->19.95% (2,000B) 0x400534: main (massif_demo.c:21)
|
```

Final snapshot reveals how the heap looked at program termination.

Usage Case: Parallel run using Open-MPI and IMB

```
$ mpirun -n 4 valgrind ./IMB-MPI1
```

Output report for each MPI process

```
[long output of stuff deleted]
==31283== HEAP SUMMARY:
==31283==     in use at exit: 511,683 bytes in 2,434 blocks
==31283==   total heap usage: 686,989 allocs, 684,555 frees, 1,666,080,129 bytes allocated
==31283==
==31282==
==31282== HEAP SUMMARY:
==31282==     in use at exit: 511,683 bytes in 2,434 blocks
==31282==   total heap usage: 687,044 allocs, 684,610 frees, 1,679,493,193 bytes allocated
==31282==
==31280==
==31280== HEAP SUMMARY:
==31280==     in use at exit: 316,027 bytes in 2,488 blocks
==31280==   total heap usage: 937,234 allocs, 934,746 frees, 2,985,419,267 bytes allocated
==31280==
==31281==
==31281== HEAP SUMMARY:
==31281==     in use at exit: 511,683 bytes in 2,434 blocks
==31281==   total heap usage: 923,980 allocs, 921,546 frees, 2,997,587,345 bytes allocated
==31281==
```

Usage Case: Parallel run using Open-MPI and IMB

```
==31280== LEAK SUMMARY:
==31280==    definitely lost: 15,068 bytes in 102 blocks
==31280==    indirectly lost: 19,628 bytes in 77 blocks
==31280==      possibly lost: 0 bytes in 0 blocks
==31280==    still reachable: 281,331 bytes in 2,309 blocks
==31280==         suppressed: 0 bytes in 0 blocks
==31280== Rerun with --leak-check=full to see details of leaked memory
==31280==
==31280== For counts of detected and suppressed errors, rerun with: -v
==31280== Use --track-origins=yes to see where uninitialised values come from
==31280== ERROR SUMMARY: 165 errors from 58 contexts (suppressed: 13 from 10)
==31280==
==31283== LEAK SUMMARY:
==31283==    definitely lost: 12,764 bytes in 70 blocks
==31283==    indirectly lost: 18,684 bytes in 41 blocks
==31283==      possibly lost: 215,456 bytes in 14 blocks
==31283==    still reachable: 264,779 bytes in 2,309 blocks
==31283==         suppressed: 0 bytes in 0 blocks
==31283== Rerun with --leak-check=full to see details of leaked memory
==31283==
==31282== LEAK SUMMARY:
==31282==    definitely lost: 12,764 bytes in 70 blocks
==31282==    indirectly lost: 18,684 bytes in 41 blocks
==31282==      possibly lost: 198,888 bytes in 13 blocks
==31282==    still reachable: 281,347 bytes in 2,310 blocks
==31282==         suppressed: 0 bytes in 0 blocks
==31282== Rerun with --leak-check=full to see details of leaked memory
==31282==
==31282== For counts of detected and suppressed errors, rerun with: -v
==31282== Use --track-origins=yes to see where uninitialised values come from
==31282== ERROR SUMMARY: 189 errors from 57 contexts (suppressed: 13 from 10)
==31283== For counts of detected and suppressed errors, rerun with: -v
==31283== Use --track-origins=yes to see where uninitialised values come from
==31283== ERROR SUMMARY: 189 errors from 57 contexts (suppressed: 13 from 10)
==31281== LEAK SUMMARY:…
```

References

- **Valgrind is freely available from:**

  http://www.valgrind.org

- **Valgrind is maintained by a network of developers**
  - Julian Seward, original creator and lead developer
    julian@valgrind.org
  - http://www.valgrind.org/info/developers.html

- **There is a tri-lab contract in place to support development of features of interest to DOE.**