Valgrind testing suite

Rafi wiener

About

- Valgrind – pronounced val (short for value) grind (pronounced with a short 'i' -- ie. "grinned" (rhymes with "tinned")
- Valgrind is an instrumentation framework for building dynamic analysis tools. It comes with a set of tools that debugs, profiles, and more to improve C\C++ applications
- It was developed by Julian Seward at/around Cambridge University, UK
- open source
  - works on x86, AMD64, PPC code
- easy to run
- Overhead is the problem
  - 5-10x slowdown without any instrumentation

How does valgrind work

- Valgrind runs the application on a synthetic CPU. As new code is executed for the first time, Valgrind's core hands the code to the selected tool (memcheck, race…). Your code is modified by these tools by adding specific instructions necessary for the tool to perform its job. These tools intercept system calls and record information before and after those calls execute. Afterwards, the tool hands the result back to the core which coordinates the continued execution of this instrumented code.
- Valgrind simulates every single instruction your program executes. Because of this, the active tool checks, or profiles, not only the code in your application but also in all supporting dynamically-linked libraries, including the C library, graphical libraries, and so on.

Goal

- C\C++ are well known to be difficult languages for writing code. Most issues are due to the memory management. Contrary to Java\Python and other high level languages with a garbage collector, in C\C++ it's the programmer's work to verify memory is valid and accessed correctly.
- C\C++ is famous for producing undefined behaviors due to invalid instructions coded by the developer.
- Bugs can happen because an uninitialized value, and debugging is frustrating and can take hours.
- Reading a non-allocated memory address, or reading memory after having freed it will not always crash your application, and such errors are difficult to find.
- Besides memory, there are other resources that must be closed after they are no longer required, or an application can run out of resources, such as file descriptors, for example.
- But Valgrind is more then a debugger. It can help you find potential deadlocks by showing you lock acquisition, pointing you to where data was changed, with no lock protecting it.
- Valgrind can also show you memory usage of each part of the application and track memory usage over time,

Limitations

- Valgrind is a runtime analysis tool, unlike cppcheck\parasoft here you must run the buggy code in order to detect issues.
- If you are using other libraries, Valgrind will warn you about issues found there, errors you might not able to, or don’t want to fix.
- Valgrind can’t detect out-of-range reads or writes to stack\global arrays.
- When HW is involved, Valgrind can report false positive issues.
  - If HW maps memory, Valgrind is not aware about the HW.
- Valgrind is slow and can cause your application to choose different paths.
  - E.g. application that after few µs consider something as timeout

Basic run - preparation

- Compile your application with debug symbols
- The manual says that it is better to compile without optimizations, although I always compiled with and didn't notice any issue.
- That's it!!!
- For a basic run, just run the program as follows:
- *Valgrind [your application] [your applications args]*
- By default, the memory analysis tool will run.

Useful flags

- --log-file – prints output to file instead of the screen
- --trace-children – tracks a new process started with the exec system call
- --track-fds – tracks non-closed file descriptors
- --suppressions – suppression file to use
- --read-var-info – prints more precise error messages
- --track-origin – tracks the origin of uninitialized value, by default Valgrind will warn only when data is used
  - Very expensive and very useful
- --show-mismatched-frees – reports allocating with new, freeing with free, and similar undefined behaviors.
- --show-leak-kinds – (to be explained in slides that follow)

Leak kinds

There are several types of memory loss, and some are considered OK.

- Definitely loss – This means that no pointer to the block can be found. The block is classified as "lost" because the programmer could not possibly have freed it at program exit since no pointer to it exists. This is likely a symptom of having lost the pointer at some earlier point in the program. Such cases should be fixed by the programmer.
- Indirectly lost – This means that your program is leaking memory in a pointer-based structure. (E.g., if the root node of a binary tree is "definitely lost", all the children will be "indirectly lost".) If you fix the "definitely lost" leaks, the "indirectly lost" leaks should go away.
- Possibly lost – This means that your program is leaking memory, unless you're doing unusual things with pointers, such as causing them to point to the middle of an allocated block.
- Still reachable memory - A pointer or chain of start-pointers to the block has been found. Since the block is still being pointed at, the programmer could, at least in principle, have freed it before program exit. "Still reachable" blocks are very common and are arguably not a problem. So, by default, Memcheck won’t report such blocks individually.

Memory leak example

```
int run() {
     int *arr = new int[10];
}

int main() {
    run();
    return 0;
}
```

```
==639== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==639==    at 0x4A065BA: operator new[](unsigned long) (vg_replace_malloc.c:264)
==639==    by 0x4005A9: run() (in /tmp/a.out)
==639==    by 0x4005B8: main (in /tmp/a.out)
```

PID

Invalid write

- Invalid write happens when writing data to places your not supposed to

```
int run() {
    int *arr = new int[10];
    arr[10] = 1;
}

int main() {
    run();
    return 0;
}
```

```
==1752== Invalid write of size 4
==1752==    at 0x4005B6: run() (in /tmp/a.out)
==1752==    by 0x4005C6: main (in /tmp/a.out)
==1752==  Address 0x4c27068 is 0 bytes after a block of size 40 alloc'd
==1752==    at 0x4A065BA: operator new[](unsigned long) (vg_replace_malloc.c:264)
==1752==    by 0x4005A9: run() (in /tmp/a.out)
==1752==    by 0x4005C6: main (in /tmp/a.out)
```

Invalid read

- Invalid read error happens when application reads from a places it’s not supposed to e.g. deleted memory. Also reading from any address you didn’t allocated can cause this error

```
int main() {
    int *a = new int[10];
    a[0] = 10;
    delete[] a;
    printf("%d\n", a[0]);
    return 0;
}
```

```
==8097== Invalid read of size 4
==8097==    at 0x4006E4: main (in /tmp/vma/vma/a.out)
==8097==  Address 0x5a15040 is 0 bytes inside a block of size 40 free'd
==8097==    at 0x4C2963D: operator delete[](void*) (vg_replace_malloc.c:621)
==8097==    by 0x4006DF: main (in /tmp/vma/vma/a.out)
==8097==  Block was alloc'd at
==8097==    at 0x4C288A8: operator new[](unsigned long) (vg_replace_malloc.c:423)
==8097==    by 0x4006BE: main (in /tmp/vma/vma/a.out)
```

Syscall param points to uninitialized byte(s)

- Sending uninitialized memory to system call

```
epoll_event ev;
ev.events = EPOLLIN | EPOLLPRI;
ev.data.fd = fd;
int ret = epoll_ctl(m_epfd, operation, fd, &ev);
```

```
epoll_event ev = { 0 };
```

```
==31283== Syscall param epoll_ctl(event) points to uninitialised byte(s)
==31283==    at 0x5EC0CBA: epoll_ctl (in /usr/lib64/libc-2.17.so)
==31283==    by 0x4F16835: wakeup_pipe::do_wakeup() (wakeup_pipe.cpp:98)
==31283==    by 0x4EB04EA: event_handler_manager::register_timer_event(int,
timer_handler*, timer_req_type_t, void*, timers_group*) (event_handler_manager.cpp:106)
==31283==    by 0x4F282B9: stats_data_reader::register_to_timer()
(stats_publisher.cpp:137)
==31283==    by 0x4F23BE2: do_global_ctors_helper (main.cpp:793)
```

Conditional jump or move depends on uninitialized value

```
int main() {
    int x;
    printf ("x = %d\n", x);
}
```

```
Conditional jump or move depends on uninitialised value(s)
at 0x402DFA94: _IO_vfprintf (_itoa.h:49)
by 0x402E8476: _IO_printf (printf.c:36)
by 0x8048472: main (tests/manuel1.c:8)
```

- It's important to understand that your program can copy around junk (uninitialized) data as much as it likes. Memcheck observes this, keeping track of the data, without complaining. A complaint is issued only when your program attempts to make use of the uninitialized data in a way that might affect your program's externally-visible behavior. In this example, x is uninitialized. Memcheck observes the value being passed to _IO_printf and then to _IO_vfprintf, but makes no comment. However, _IO_vfprintf has to examine the value of x so it can turn it into the corresponding ASCII string, and it is at this point that Memcheck complains.

Invalid free

- Invalid free can happen when
  - You double free the same address
  - Deallocate with free, memory allocated with new
  - Use delete instead of delete[] or vice versa

```
Mismatched free() / delete / delete []
at 0x40043249: free (vg_clientfuncs.c:171)
by 0x4102BB4E: QGArray::~QGArray(void) (tools/qgarray.cpp:149)
by 0x4C261C41: PptDoc::~PptDoc(void) (include/qmemarray.h:60)
by 0x4C261F0E: PptXml::~PptXml(void) (pptxml.cc:44)
Address 0x4BB292A8 is 0 bytes inside a block of size 64 alloc'd
at 0x4004318C: operator new[](unsigned int) (vg_clientfuncs.c:152)
by 0x4C21BC15: KLaola::readSBStream(int) const (klaola.cc:314)
by 0x4C21C155: KLaola::stream(KLaola::OLENode const *) (klaola.cc:416)
by 0x4C21788F: OLEFilter::convert(QCString const &) (olefilter.cc:272)
```

- Remember In c++
  - If allocated with malloc, calloc, realloc, valloc or memalign, you must deallocate with free.
  - If allocated with new, you must deallocate with delete.
  - If allocated with new[], you must deallocate with delete[].

MISC.

- Overlap between source and destination in memcpy, strcpy, strcat and more

```
==27492== Source and destination overlap in memcpy(0xbffff294, 0xbffff280, 21)
==27492== at 0x40026CDC: memcpy (mc_replace_strmem.c:71)
==27492== by 0x804865A: main (overlap.c:40)
```

- Fishy memory size to allocate

```
==32233== Argument ’size’ of function malloc has a fishy (possibly negative) value: -3
==32233== at 0x4C2CFA7: malloc (vg_replace_malloc.c:298)
==32233== by 0x400555: foo (fishy.c:15)
==32233== by 0x400583: main (fishy.c:23)
```

Suppression file

- Although Valgrind is pretty accurate, sometimes it reports false positive. In order to filter out those well known issues, you can run Valgrind with a suppression file.
- Suppression files are also good if you are using third party libraries that you don't want to, or can't, fix.
- The best way to generate suppressions is to run Valgrind with *--gen-suppressions=all*

```
{
  [title] \\ will be used in valgrind summery report
   Memcheck:Leak \\ type
   match-leak-kinds: definite \\ sub type
   ... \\ wildcard skips object function…
   fun:_ZN20net_device_table_mgr16create_new_entryEj \\ mangled name
   fun:_ZN15cache_table_mgrI10ip_addressP14net_device_valE17register
}
{
   rdmacm Value8 rdma_get_devices
   Memcheck:Value8
   ...
   fun:rdma_get_devices
}
```

Suppressing in the code

- In your application you can inform Valgrind about initialized memory to prevent false positives.
- VALGRIND_MAKE_MEM_UNDEFINED(ptr, size)
- VALGRIND_MAKE_MEM_DEFINED(ptr, size)

```
struct ibv_qp_attr qp_attr;
struct ibv_qp_init_attr qp_init_attr;
if (ibv_query_qp(qp, &qp_attr, IBV_QP_STATE, &qp_init_attr)) return -1;
VALGRIND_MAKE_MEM_DEFINED(&qp_attr, sizeof(qp_attr));
return qp_attr.qp_state;
```

- Other macros are available to invoke Valgrind in the middle of your application run.
- Those macro should only be used in a debug mode due do performance and not to add valgrind dependency to the project.

Sgcheck

- Run with *valgrind* --tool=exp-sgcheck
- SGCheck is an experimental tool for finding overruns of stack and global arrays.
- SGCheck and Memcheck are complementary: their capabilities do not overlap. Memcheck performs bounds checks and use-after-free checks for heap arrays. It also finds uses of uninitialized values created by heap or stack allocations. But it does not perform bounds checking for stack or global arrays.
- SGCheck, on the other hand, does bounds-checking for stack or global arrays, but it doesn't do anything else.
- Experimental tool is still under development

Helgrind

- Run with *valgrind –tool=helgrind*
- Helgrind is a Valgrind tool for detecting synchronization errors in C, C++ (and Fortran) programs that use the POSIX pthreads threading primitives
- Helgrind detects:
  - Misuses of the POSIX pthreads API.
  - Potential deadlocks arising from lock ordering problems.
  - Data races - accessing memory without adequate locking or synchronization.
- If you want to test C++11 threads you need to add macros
- It doesn't recognize atomic and futex.
- It is very slow - up to 100x slower.

Helgrined – api misuse

- Misuses of the POSIX pthreads API.
  - unlocking an invalid mutex,
  - unlocking a not-locked mutex,
  - destroying an invalid or a locked mutex
  - recursively locking of a non-recursive mutex
  - waiting on an uninitialized pthread barrier
  - …

```
Thread #1 unlocked a not-locked lock at 0x7FEFFFA90
at 0x4C2408D: pthread_mutex_unlock (hg_intercepts.c:492)
by 0x40073A: nearly_main (tc09_bad_unlock.c:27)
by 0x40079B: main (tc09_bad_unlock.c:50)
Lock at 0x7FEFFFA90 was first observed
at 0x4C25D01: pthread_mutex_init (hg_intercepts.c:326)
by 0x40071F: nearly_main (tc09_bad_unlock.c:23)
by 0x40079B: main (tc09_bad_unlock.c:50)
```

Helgrind - deadlock

- Helgrind monitors the order in which threads acquire locks. This allows it to detect potential deadlocks which could arise from the formation of cycles of locks. Detecting such inconsistencies is useful because, while deadlocks are usually obvious, potential deadlocks may never be discovered during testing and could later lead to hard-to-diagnose in-service failures.

    - Imagine some shared resource R, which, for whatever reason, is guarded by two locks, L1 and L2, which must both be held when R is accessed.
    - Suppose a thread acquires L1, then L2, and proceeds to access R. The implication of this is that all threads in the program must acquire the two locks in the order first L1 then L2. Not doing so risks deadlock.

```
Thread #1: lock order "0x7FF0006D0 before 0x7FF0006A0" violated
Observed (incorrect) order is: acquisition of lock at 0x7FF0006A0
    at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
    by 0x400825: main (tc13_laog1.c:23)
followed by a later acquisition of lock at 0x7FF0006D0
    at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
    by 0x400853: main (tc13_laog1.c:24)
Required order was established by acquisition of lock at 0x7FF0006D0
    at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
    by 0x40076D: main (tc13_laog1.c:17)
followed by a later acquisition of lock at 0x7FF0006A0
   at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
   by 0x40079B: main (tc13_laog1.c:18)
```

Helgrind - deadlock

- Helgrind builds a directed graph indicating the order in which locks have been acquired in the past. When a thread acquires a new lock, the graph is updated, and then it is checked to see if it contains a cycle. The presence of a cycle indicates that a potential deadlock involving the locks exists in the cycle. In general, Helgrind will choose two locks involved in the cycle and show you how their acquisition ordering has become inconsistent. It does this by showing the program point that first defined the ordering, and then the program point which later violated it. Here is a simple example involving just two locks:

```
Thread #1: lock order "0x7FF0006D0 before 0x7FF0006A0" violated
Observed (incorrect) order is: acquisition of lock at 0x7FF0006A0
    at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
    by 0x400825: main (tc13_laog1.c:23)
followed by a later acquisition of lock at 0x7FF0006D0
    at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
    by 0x400853: main (tc13_laog1.c:24)
Required order was established by acquisition of lock at 0x7FF0006D0
    at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
    by 0x40076D: main (tc13_laog1.c:17)
followed by a later acquisition of lock at 0x7FF0006A0
   at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
   by 0x40079B: main (tc13_laog1.c:18)
```

Data race example

```
int var = 0;
void* child_fn ( void* arg ) {
    var++; /* Unprotected relative to parent */ /* line 6 */
    return NULL;
}
int main ( void ) {
    pthread_t child;
    pthread_create(&child, NULL, child_fn, NULL);
    var++; /* Unprotected relative to child */ /* line 13 */
    pthread_join(child, NULL);
    return 0;
}
```

```
Possible data race during read of size 4 at 0x601038 by thread #1
Locks held: none
at 0x400606: main (simple_race.c:13)
This conflicts with a previous write of size 4 by thread #2
Locks held: none
at 0x4005DC: child_fn (simple_race.c:6)
by 0x4C29AFF: mythread_wrapper (hg_intercepts.c:194)
by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
by 0x511C0CC: clone (in /lib64/libc-2.8.so)
Location 0x601038 is 0 bytes inside global var "var"
declared at simple_race.c:3
```

DRD

- DRD is an experimental thread analyzer. It does everything that helgrind does, but it also comes with some extra analysis tools.
- Run it with *valgrind –tool=drd*
- First you should clean all memory issues in your code
- It has better support for Boost\C++ threads
- http://valgrind.org/docs/manual/drd-manual.html#drd-manual.C++11

Lock contention

- Basically locking is not expensive. What’s expensive is waiting to acquire the lock, AKA lock contention.
- A lock that has been “fought” over can cause serious degradation in time critical application.
- Sometimes this can be solved rewriting those critical sections, either with a single thread, or with several tasks, or by locking and unlocking the section several times.
- Drd can detect locks with high contention with the arguments
  - --exclusive-threshold=<n> - write lock held more then n milis
  - --shared-threshold=<n> - read lock held more then n milis

```
==10668== Acquired at:
==10668==    at 0x4C267C8: pthread_mutex_lock (drd_pthread_intercepts.c:395)
==10668==    by 0x400D92: main (hold_lock.c:51)
==10668== Lock on mutex 0x7fefffd50 was held during 503 ms (threshold: 10 ms).
==10668==    at 0x4C26ADA: pthread_mutex_unlock (drd_pthread_intercepts.c:441)
==10668==    by 0x400DB5: main (hold_lock.c:55)
```

Massif – heap profiler

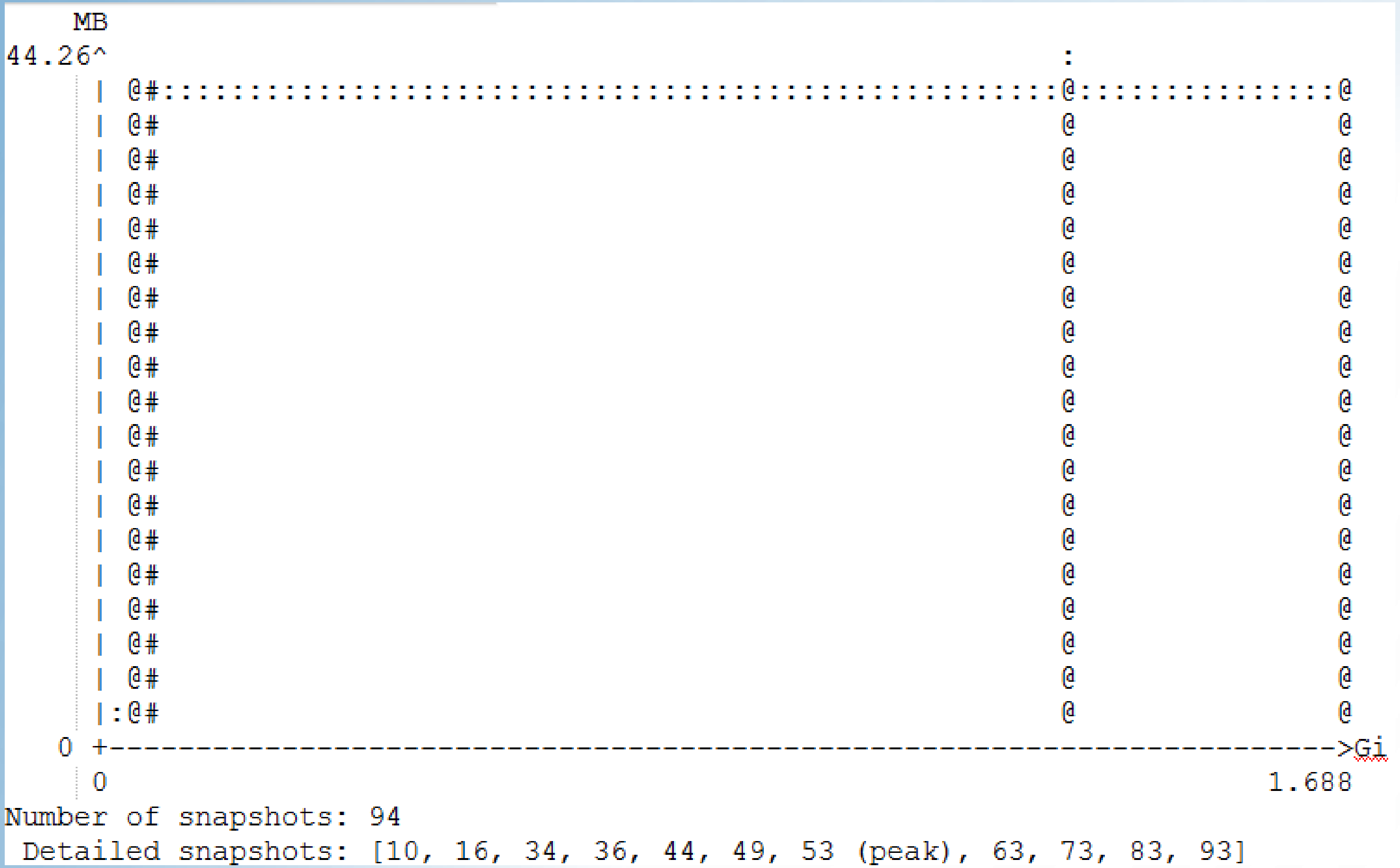
- This tool measures how much heap memory your program uses. This includes both the used space, and the extra bytes allocated for book-keeping and alignment purposes. It can also measure the size of your program's stack(s)
- It's good for testing long running applications to verify they don't "forget" to free memory after it's no longer used.
  - Think about an stl map that always grow…
- It is important to note that Massif not only tells you how much heap memory your program is using, but it also gives you very detailed information to show you which parts of your program are responsible for allocating the heap memory.

Massif

- Run with *valgrind –tool=massif* …
- Data is written either to your chosen file (with --massif-out-file) or massif.out.[pid]
- To see the results run *ms_print [file]*

Massif -graph

- : standard snapshot
- @ detailed snapshot
- # peek consumption



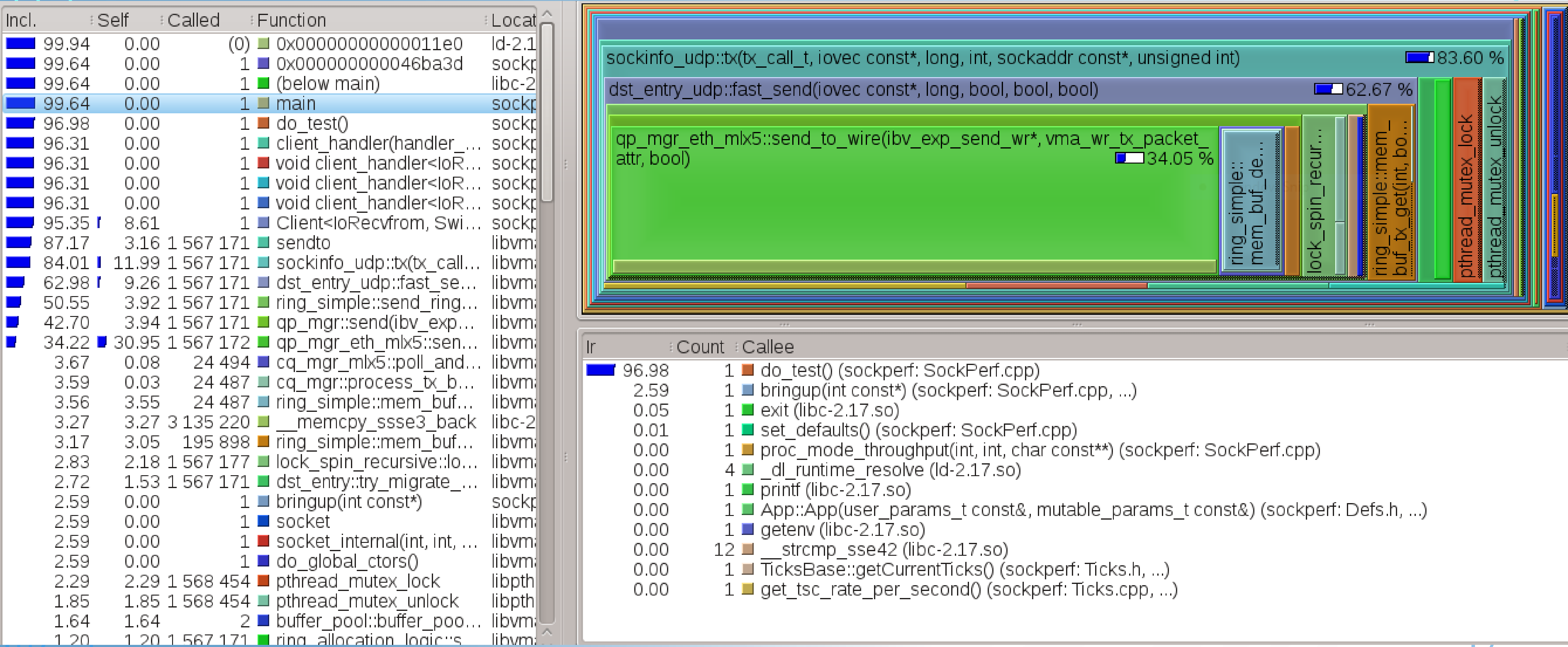
```
    MB
44.26^                                                       :
     | @#::::::::::::::::::::::::::::::::::::::::::::::::::::@:::::::::::::::@
     | @#                                                    @               @
     | @#                                                    @               @
     | @#                                                    @               @
     | @#                                                    @               @
     | @#                                                    @               @
     | @#                                                    @               @
     | @#                                                    @               @
     | @#                                                    @               @
     | @#                                                    @               @
     | @#                                                    @               @
     | @#                                                    @               @
     | @#                                                    @               @
     | @#                                                    @               @
     | @#                                                    @               @
     | @#                                                    @               @
     | @#                                                    @               @
     | @#                                                    @               @
     |:@#                                                    @               @
   0 +----------------------------------------------------------------------->Gi
     0                                                                   1.688
Number of snapshots: 94
 Detailed snapshots: [10, 16, 34, 36, 44, 49, 53 (peak), 63, 73, 83, 93]
```

Massif raw data

```
--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 23    302,595,069          528,600          528,478           122            0
 24    302,595,113          463,072          462,965           107            0
 25    302,595,146          463,032          462,933            99            0
 26    302,595,185          397,504          397,420            84            0
 27    302,595,218          397,464          397,388            76            0
 28    302,596,530          393,360          393,292            68            0
 29    302,596,567          262,328          262,280            48            0
 30    302,596,629          262,224          262,192            32            0
 31    302,596,668          262,200          262,176            24            0
 32    302,596,701              144              128            16                  0
88.89% (128B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->66.67% (96B) 0x46B18F: _GLOBAL__sub_I__Z9print_logPKcP8fds_data (new_allocator.h:104)
| ->66.67% (96B) 0x6FA21B: __libc_csu_init (in /usr/local/bin/sockperf)
|   ->66.67% (96B) 0x5A8CAC3: (below main) (in /usr/lib64/libc-2.17.so)
|
->22.22% (32B) 0x4E2C61E: _dlerror_run (in /usr/lib64/libdl-2.17.so)
| ->22.22% (32B) 0x4E2C126: dlsym (in /usr/lib64/libdl-2.17.so)
|   ->22.22% (32B) 0x6F9BEF: vma_set_func_pointers_internal(void*) (vma-redirect.cpp:122)
|     ->22.22% (32B) 0x6F6EFB: set_defaults() (SockPerf.cpp:2214)
|       ->22.22% (32B) 0x46B27A: main (SockPerf.cpp:3295)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)
```

Callgrind

- Run with *valgrind --tool=callgrind*
- 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.


sockinfo_udp::tx(tx_call_t, iovec const*, long, int, sockaddr const*, unsigned int) 83.60 %
dst_entry_udp::fast_send(iovec const*, long, bool, bool, bool) 62.67 %
qp_mgr_eth_mlx5::send_to_wire(ibv_exp_send_wr*, vma_wr_tx_packet_attr, bool) 34.05 %
ring_simple::mem_buf_de…
lock_spin_recur…
ring_simple::mem_buf_tx_get(int, bo…
pthread_mutex_lock
pthread_mutex_unlock

Incl.	Self	Called	Function	Locat
99.94	0.00	(0)	0x00000000000011e0	ld-2.1
99.64	0.00	1	0x000000000046ba3d	sockp
99.64	0.00	1	(below main)	libc-2
99.64	0.00	1	main	sockp
96.98	0.00	1	do_test()	sockp
96.31	0.00	1	client_handler(handler_…	sockp
96.31	0.00	1	void client_handler<IoR…	sockp
96.31	0.00	1	void client_handler<IoR…	sockp
96.31	0.00	1	void client_handler<IoR…	sockp
95.35	8.61	1	Client<IoRecvfrom, Swi…	sockp
87.17	3.16	1 567 171	sendto	libvm
84.01	11.99	1 567 171	sockinfo_udp::tx(tx_call…	libvm
62.98	9.26	1 567 171	dst_entry_udp::fast_se…	libvm
50.55	3.92	1 567 171	ring_simple::send_ring…	libvm
42.70	3.94	1 567 171	qp_mgr::send(ibv_exp…	libvm
34.22	30.95	1 567 172	qp_mgr_eth_mlx5::sen…	libvm
3.67	0.08	24 494	cq_mgr_mlx5::poll_and…	libvm
3.59	0.03	24 487	cq_mgr::process_tx_b…	libvm
3.56	3.55	24 487	ring_simple::mem_buf…	libvm
3.27	3.27	3 135 220	__memcpy_ssse3_back	libc-2
3.17	3.05	195 898	ring_simple::mem_buf…	libvm
2.83	2.18	1 567 177	lock_spin_recursive::lo…	libvm
2.72	1.53	1 567 171	dst_entry::try_migrate_…	libvm
2.59	0.00	1	bringup(int const*)	sockp
2.59	0.00	1	socket	libvm
2.59	0.00	1	socket_internal(int, int, …	libvm
2.59	0.00	1	do_global_ctors()	libvm
2.29	2.29	1 568 454	pthread_mutex_lock	libpth
1.85	1.85	1 568 454	pthread_mutex_unlock	libpth
1.64	1.64	2	buffer_pool::buffer_poo…	libvm
1.20	1.20	1 567 171	ring_allocation_logic::s	libvm

Ir	Count	Callee
96.98	1	do_test() (sockperf: SockPerf.cpp)
2.59	1	bringup(int const*) (sockperf: SockPerf.cpp, ...)
0.05	1	exit (libc-2.17.so)
0.01	1	set_defaults() (sockperf: SockPerf.cpp)
0.00	1	proc_mode_throughput(int, int, char const**) (sockperf: SockPerf.cpp)
0.00	4	_dl_runtime_resolve (ld-2.17.so)
0.00	1	printf (libc-2.17.so)
0.00	1	App::App(user_params_t const&, mutable_params_t const&) (sockperf: Defs.h, ...)
0.00	1	getenv (libc-2.17.so)
0.00	12	__strcmp_sse42 (libc-2.17.so)
0.00	1	TicksBase::getCurrentTicks() (sockperf: Ticks.h, ...)
0.00	1	get_tsc_rate_per_second() (sockperf: Ticks.cpp, ...)

Cachegrind- cache and branch-prediction

- Cachegrind simulates how your program interacts with a machine's cache hierarchy and branch predictor.
- Cachegrind gathers the following statistics:
  - cache reads data and instructions
  - cache read misses data and instructions
  - cache writes data and instructions
  - cache write misses data and instructions
  - Conditional branches executed\mispredicted
  - Indirect branches executed\mispredicted
- You can compare different runs with cg_diff to understand how your changes influences the cache

Cachegrind

- Compile your code with regular optimizations and debug symbol
- use cg_merge to merge runs from different run

```
--------------------------------------------------------------------------------
Ir         I1mr ILmr Dr         D1mr  DLmr  Dw         D1mw   DLmw    file:function
--------------------------------------------------------------------------------
8,821,482    5    5 2,242,702 1,621    73 1,794,230      0      0  getc.c:_IO_getc
5,222,023    4    4 2,276,334    16    12   875,959      1      1  concord.c:get_word
2,649,248    2    2 1,344,810 7,326 1,385         .      .      .  vg_main.c:strcmp
2,521,927    2    2   591,215     0     0   179,398      0      0  concord.c:hash
2,242,740    2    2 1,046,612   568    22   448,548      0      0  ctype.c:tolower

Ir-instruction read , I1mr – read miss, ILmr  - instruction read miss
DR – memory read, D1mr – read miss, Dlmr- data read miss,
DW – memory write, D1wm – write miss, DLmw – data write miss
```

Cg_annotate – source file view

```
--------------------------------------------------------------------------------
-- User-annotated source: concord.c
--------------------------------------------------------------------------------
Ir         I1mr ILmr Dr      D1mr  DLmr  Dw      D1mw   DLmw

      .    .    .       .     .     .       .      .      .  void init_hash_table(char *file, Word_Node *table[])
      3    1    1       .     .     .       1      0      0  {
      .    .    .       .     .     .       .      .      .      FILE *file_ptr;
      .    .    .       .     .     .       .      .      .      Word_Info *data;
      1    0    0       .     .     .       1      1      1      int line = 1, i;
      .    .    .       .     .     .       .      .      .
      5    0    0       .     .     .       3      0      0      data = (Word_Info *) create(sizeof(Word_Info));
      .    .    .       .     .     .       .      .      .
  4,991    0    0   1,995     0     0     998      0      0      for (i = 0; i < TABLE_SIZE; i++)
  3,988    1    1   1,994     0     0     997     53     52          table[i] = NULL;
      .    .    .       .     .     .       .      .      .
      .    .    .       .     .     .       .      .      .      /* Open file, check it. */
      6    0    0       1     0     0       4      0      0      file_ptr = fopen(file, "r");
      2    0    0       1     0     0       .      .      .      if (!(file_ptr)) {
      .    .    .       .     .     .       .      .      .          fprintf(stderr, "Couldn't open '%s'.\n", file);
      1    1    1       .     .     .       .      .      .          exit(EXIT_FAILURE);
      .    .    .       .     .     .       .      .      .      }
      .    .    .       .     .     .       .      .      .
165,062    1    1  73,360     0     0  91,700      0      0      while ((line = get_word(data, line, file_ptr)) != EOF)
146,712    0    0  73,356     0     0  73,356      0      0          insert(data->;word, data->line, table);
      .    .    .       .     .     .       .      .      .
      4    0    0       1     0     0       2      0      0      free(data);
      4    0    0       1     0     0       2      0      0      fclose(file_ptr);
      3    0    0       2     0     0       .      .      .  }
```

cg_annotate <filename> concord.c

Cachegrind-improving my app

- The line-by-line source code annotations are very useful. The best place to start is by looking at the Ir numbers. They measure how many instructions were executed for each line, and very useful for identifying bottlenecks.
- LL misses are typically a much bigger source of slow-downs than L1 misses. So it's worth looking for any snippets of code with high DLmr or DLmw counts. (You can use --show=DLmr --sort=DLmr with cg_annotate to focus just on DLmr counts). If you find any, it's still not always easy to work out how to improve things. You need to have a reasonable understanding of how caches work, the principles of locality, and your program's data access patterns. Improving things may require redesigning a data structure.

Thanks