Tuning Valgrind for your Workload

Hints, tricks and tips to effectively use Valgrind on small or big applications

Philippe Waroquiers
FOSDEM 2015 valgrind devroom
Some rumours about Valgrind ...

- Valgrind burns all the CPU it can
- ... and it burns it on a single CORE
- Valgrind eats memory as much as it can
- Valgrind is powerful and sophisticated, it finds nasty bugs and gives you a lot of information about your bugs and your program.

- Last rumour is true
- First 3 rumours are also (somewhat) true
Valgrind resource consumption

- To give sophisticated functionalities, Valgrind is effectively a big resource consumer
- Can we do something about that?
- Yes we can!
  - Simple use: default tool and default options: `valgrind your_program`
  - Otherwise valgrind and all its tools have more than 150 user command line options to e.g. control
    - what kind of bugs to detect
    - which information to capture
    - ...
Valgrind resource consumption
what can we do?

• Use command line options to
  • consume even more CPU/memory
    and have more information/features
  • decrease (somewhat) CPU/memory consumption
    by reducing captured information

• What can be controlled can be
  • Tool independent, e.g. stacktrace size, use of debug information, …
  • Tool dependent e.g. uninitialised memory origin tracking for memcheck, detailed race condition history for helgrind, …
Tuning Valgrind malloc replacement

- Red zones useful to detect over/under-run
  - Configurable via \texttt{--redzone-size=xxxx}
- But are costly if many small blocks
  - => Reduce redzone size if short on memory
    - In particular for helgrind
  - => Increase redzone size if suspecting (big) over/under-run
- Use \texttt{--stats=yes -v -v} to have some useful info about the valgrind malloc arenas
Tuning Valgrind stacktrace capture

• Configure the nr of recorded program counters
  --num-callers=xx

• To merge recursive calls
  --merge-recursive-frames=x

• valgrind >= 3.10 shows inlined calls
  unless you give --read-inline-info=no

• To have stats about recorded stack traces:
  valgrind --stats=yes .... 2>&1 | grep exectxt:
  For full list, use gdb+vgdb monitor command:
  (gdb) monitor v.info exectxt
Tuning Valgrind stacktrace capture memcheck specific

• By default, one stack trace is referenced:
  • memcheck records both malloc and free stack trace
  • A block references the last recorded stack trace: the malloc stack trace, and when freed, the free stacktrace

• Use `--keep-stacktraces=....` to control what to record and reference
  --keep-stacktraces=alloc-and-free
  only one word overhead per block, compared to
  --keep-stacktraces=alloc-then-free
Tuning Valgrind stacktrace capture
helgrind specific

- By default, helgrind keeps a stacktrace (max 8 frames) for past memory accesses
- Use `--history-level=none|approx|full` to control what history stacktraces to record
- Use `--conflict-cache-size=N` to configure the size of the full history cache
Obtaining more info about your bugs

• **Default values** for Valgrind options are **chosen** to provide a good balance between **cost** (CPU and memory) and **provided functionality**

• Examples:  
  --read-inline-info=yes  
  --read-var-info=no  
  --track-origins=no  
  --history-level=full  
  (memcheck)  
  (helgrind)
Tuning Valgrind JIT

- You might (unlikely) gain a few % by using the VEX command line options
  - Use `valgrind --help-debug` for details
- If your application code is big
  - You might avoid re-translating code by increasing valgrind JIT code cache:
    `--num-transtab-sectors=NN` (impacts memory!)
- Use `--stats=yes` to see when a transtab sector is recycled
Getting Valgrind info/stats

- Use `valgrind --stats=yes (-v -v)` for general stats
- Use `valgrind --profile-heap=yes` for detailed internal valgrind memory use
- During run, you can use (from shell)
  - `vgdb v.info stats`
  - `vgdb v.info memory aspacemgr`
Optimising Valgrind for speed/CPU

- Set your CPU frequency to fixed high speed
  - e.g. using `cpufreq-selector -g performance`
- Tune stack recording (e.g. if heavy malloc use)
- If huge code, increase `--num-transtab-sectors`
- Disable some tool specific features
  - e.g. `--undef-value-errors=no` (memcheck)
    `--track-lockorders=no` (helgrind)
- Unlikely/limited gain using vex options
- ... (study `valgrind --help` and valgrind user manual)
Optimising Valgrind for memory

- Disable some tool specific features
e.g.  `--undef-value-errors=no` (memcheck)
   `--track-lockorders=no` (helgrind)

- Tune stack recording

- Decrease redzone size  `--redzone-size=N`

- Decrease `--num-transtab-sectors`

- ... (study `valgrind --help` and `valgrind user manual`)
Optimising Valgrind for functionality

- Enable optional tool functionalities e.g.
  --track-origin=yes (memcheck)
  --leak-check-heuristics=all (memcheck)

- Record more/all what you can, e.g. memcheck
  --freelist-vol=NNNNNN
  --keep-stacktraces=alloc-and-free

- ... (study `valgrind --help` and valgrind user manual)
Conclusions/guidelines

- Default options are ok for an average user
  - => automate your regression tests
  - => run them under Valgrind
    - and be patient
- Read Valgrind manual
  - You have nice optional features to activate
  - You can (somewhat) tune valgrind for your workload
Questions?