Running Valgrind on multiple processors: a prototype
Valgrind and threads

- Valgrind runs properly multi-threaded applications
- But (mostly) runs them using a single CORE
- Valgrind needs a lot of CPU:
  - Depending on the tool, single-threaded applications are slowed down by a factor 4x to 100x or more
Valgrind and CPU consumption

- Significant development effort was and is spent to make Valgrind faster e.g.
  - Improvement of the JIT generated code
  - Self-modifying code detection
  - Translation chaining
  - Tool specific performance improvement
  - …
Improving Valgrind speed

• Improving 'sequential' speed is good for all applications
  • However, often, the last years, the gains are small typically around 5 .. 10%

• Multi-threaded CPU bounded applications would benefit a lot from parallelising Valgrind
  • But how hard is that?
Valgrind layers

TOOL
- Tool "runtime" code
- Tool instrument function

GUEST
- Generated/instrumented code
  (from program to run)

Valgrind CORE layer
- JIT decoder and compiler, malloc replacement, scheduler, ...
Valgrind layers typical control flow

1. CORE decodes guest code : instructions to IR
2. CORE calls TOOL instrument : IR to IR. Instrumented code typically contains many calls to TOOL runtime code or CORE code.
3. CORE translates instrumented code to executable code : IR to instructions
4. Instructions stored in the translation table
5. Valgrind scheduler calls the translation
(Most of) Valgrind code is non-reentrant/non thread-safe

- Translation is non thread-safe: VEX lib, tool instrument function, CORE translation framework, ...

- “Run time” is non thread-safe:
  - CORE scheduler, CORE malloc/free, CORE aspacemgr, CORE statistics, ...
  - TOOL runtime code, e.g. memcheck malloc/free, memcheck VA bits data structures, ...

- So, why is Valgrind able to run properly multi-threaded applications?
Valgrind “big lock” model

- Valgrind has a big lock
  - The big lock protects all Valgrind data structures/all Valgrind global variables/all tool data structures/...
  - Big lock implemented via a 'pipe based lock' (default) or via futex ('ticket lock'), cfr --fair-sched
- To execute JIT-ted guest or tool or core code, a thread first must acquire the big lock
- A thread releases the lock
  - After it has executed 100K basic blocks or
  - Before entering in a blocking syscall
To parallelise Valgrind

• We must
  • Remove the big lock

or

• At least decrease the use of the big lock
Parallelising Valgrind possible techniques

- Read/write locks
- (fine grained) mutex locks
- Atomic instructions
- Thread local storage instead of global variables
- Lock-less algorithms/data structures
- ....

- A prototype has used some of the above to parallelise some (small) parts of Valgrind
What to parallelise (first) ?

- A typical tool/application spends most of CPU in the generated JIT code, in the TOOL and CORE “runtime” code
- The time spent in TOOL instrument function is normally not a major part
- => First idea: ensure that the threads are running guest JIT-ted code in parallel
Running JIT-ted code in parallel

Basic idea

- Replace 'mutex Big lock' by 'read/write Big lock'
- A thread acquires the RW Big lock
  - In read mode to run guest JIT-ted code
  - In write mode to do anything else
- First implementation of basic idea:
  - Objective: ensure 'none' tool runs in parallel
  - How: RW lock implemented on top of 'pipe based locks'
Running JIT-ted code in parallel
First implementation expected results

• Of course, first implementation will be efficient
  • As the pipe based lock is efficient enough for current Valgrind, the rw lock will be efficient enough for parallel use

• Of course, first implementation will be correct
  • As “none” tool means no Valgrind data structure are accessed when running JIT-ted guest code

• Of course, all above
  • was shown WRONG !!!
Running JIT-ted code in parallel
First implementation problems

• Lack of efficiency when translating new code:
  • When new code to be translated, sequential valgrind just keeps the lock
  • Parallel Valgrind needs to (re-)acquire the lock in Write mode => a lot more (expensive) 'lock/unlock'

• Lack of correctness
  • What looks like a 'read-only' action (execute already translated code) is in fact doing many updates e.g.
    - statistical counters
    - fast cache associating guest code with JIT code
    - Translation chaining
    - ...
Running JIT-ted code in parallel
Fixing first implementation

• Better way to find non thread safe code
  • Valgrind and helgrind were improved to allow to run an 'inner parallel valgrind' under an outer helgrind
  • Improvements are now in Valgrind release: it is now easy/ier to run Valgrind under Valgrind
  • Helgrind was used to find race conditions in prototype parallel Valgrind

• Efficiency:
  • RW lock based on (slow) pipe based mutexes replaced by RW lock code copied/modified from glibc
Read the patch...

Prototype code accessible in SVN MTV branch
see also doc/internals/mtV.txt
Multi-threaded Valgrind : challenges

Valgrind core

● Make (more of) core parallel/thread-safe
  ● Prototype is far to be complete/correct
● Probably/maybe we need an option to have sequential run of parallel tools (e.g. to avoid memcheck false + or -) or avoid running non parallel tools in parallel
● Implement atomic ops for other arch
● What about Darwin and fast mutex ?
Multi-threaded Valgrind: challenges
Making Valgrind tools parallel

- At least memcheck (the most used tool)
- Keep cpu and/or memory efficiency is difficult (apart of trivial tools such as \texttt{--tool=none})
- No tool was made parallel (except \texttt{none})
  - Parallel memcheck somewhat discussed/ tried
  - Draft proposal of new VA-bits approach made by Julian Seward
Multi-threaded Valgrind: challenges
Memcheck VA-bits data structure

- Is currently highly optimised, CPU and memory
- No solution found that at the same time
  - Is efficient in CPU and memory
  - and has no false + and/or false -
- Maybe make 'VA-bits read' inline fast, 'VA-bits write' use mutex? (or an option to activate write mutex)
- Maybe we need tuning options such as
  --va-bits=sequential | parallel-cpu | parallel-memory | ...
Multi-threaded Valgrind: challenges

- Probably many challenges not known yet ...
  - Because not exercised by the prototype 'testing'
  - Many core modules not looked at
    e.g. Valgrind malloc, error mgr, stack unwind, ...
- Do all the above without slowing down the sequential case
  - Many optimisations to be redone/reworked!
Questions?